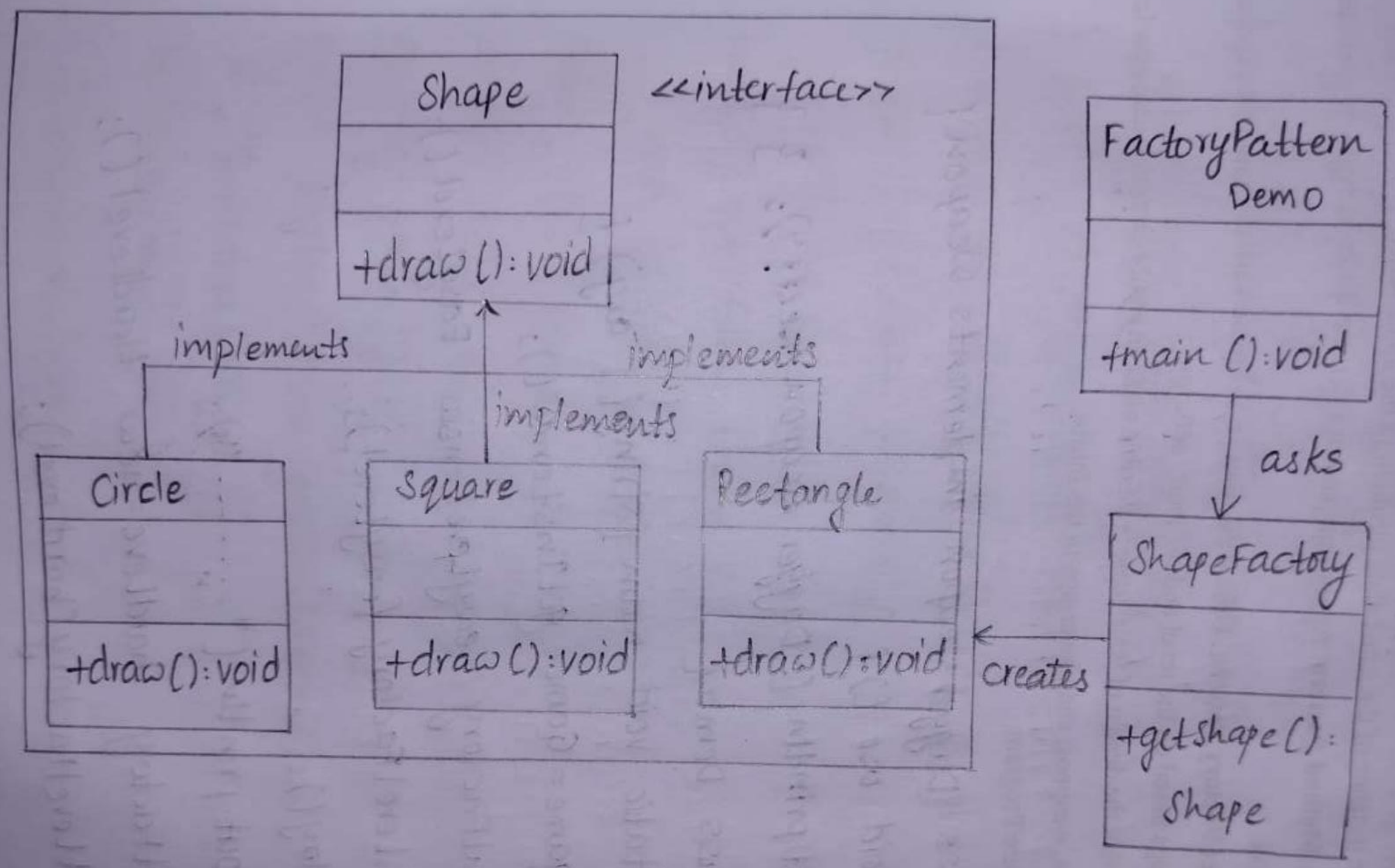
Experiment#		Student ID	
Date		Student Name	

2) Draw the UML Relationship Diagram for Factory Design Pattern for any customized scenario.



In-Lab:

1) Develop a game application with multiple levels and varying difficulty se
Implement the following design patterns to manage different aspects of the game

Singleton Pattern: Use this pattern to manage the game state, ensuring that th
only one instance of the game state throughout the application.

Factory Method Pattern: Apply this pattern to create different types of enemi
each level.

Abstract Factory Pattern: Utilize this pattern to create various types of weapon
power-ups based on the level and difficulty settings.

Ensure that the design of your game is flexible and can easily accommodate new l
enemies, weapons, and power-ups in the future.

Procedure/Program:

```
public class DaggerWeapon implements Weapon{
  public void use (){
  System.out.println ("Dagger weapon used!"); } }
public class Demo{
  public static void main (String[] args) {
   Game game = Game. getInstance ();
GameElementFactory easyLevel = new EasyLevel ();
  game.setLevelFactory (easyLevel);
    game.play();
  System.out.println ("-------");
GameElementFactory hardLevel=new HardLevel ();
  game.setLevelFactory(hardLevel);
```

```
game.play(); } }
public class EasyLevel implements GameElementFactory {
  public Enemy createEnemy() {
    return new GoblinEnemy();
  } public Weapon createweapon() {
    return new DaggerWeapon();
  } public PowerUp createPowerUp() {
    return new HealthPowerUp(); } }
  public interface Weapon {
      void use(); }
Public interface Enemy {
      void attack(); }
Public class Game {
private static Game instance;
private game() { }
Public static Game getInstance() {
 if (instance==null) {
instance=new Game();
} return instance; }
```

```
public void setLevelFactory (GameElementFactory
                                factory) {
    this.factory = factory;
} public void play() {
Enemy enemy = factory.createEnemy();
  Weapon weapon = factory.createWeapon();
  PowerUp powerup = factory.createPowerUp();
    enemy.attack();
    weapon.use();
    powerup.activate(); } }
public interface GameElementFactory {
    Enemy createEnemy();
    Weapon createWeapon();
    Powerup createPowerUp(); }
public class GoblinEnemy implements Enemy {
 public void attack() {
System.out.println("Goblin enemy attack!");
    }
}
```



```
public class
  public En
    return
} public
    return
} public P
    return
public clas
  public v
System.out.p
public class
 public voi
 system.out.p
public inter
  void.act
} public
      void
}
```

```
public class HardLevel implements GameElementFactory {
  public Enemy createEnemy() {
    return new OrcEnemy();
} public Weapon createWeapon() {
    return new SwordWeapon();
} public PowerUp createPowerUp() {
     return new ShieldPowerUp(); } }
public class HealthPowerUp implements PowerUp {
   public void activate() {
System.out.println("Health power-up activated"); } }
public class OrcEnemy implements Enemy {
 public void attack() {
System.out.println("Orc enemy attack!"); } }
Public interface PowerUp {
   void activate();
 } public interface Weapon {
      void use();
 }
```

Experiment#		Student ID	
Date		Student Name	

✓ **Data and Results:**

Enemy created: Easy Enemy
Weapon created: Gun
Power-up created: Health Power-up

✓ **Analysis and Inferences:**

Both the Factory and Abstract Factory design patterns focus on encapsulating object creation and enhancing flexibility in object instantiation. Factory pattern is suitable when you want to create different instances of a single type.

Course Title	Advanced Object-Oriented Programming	ACADEMIC YEAR: 20
Course Code	23CS2103A & 23CS2103E	

Experiment#		Student ID	
Date		Student Name	

VIVA-VOCE Questions (In-Lab):

1) Which classes are candidates of Singleton? Which kind of class do you make Singleton in Java?

classes that manage global state, resources or provide utility functions are good candidates for singleton pattern. Singleton pattern ensures a single instance with controlled access, making it suitable for managing shared resources and configurations.

2) Discuss the difference between Factory and Abstract Factory design patterns.

Both patterns abstract the object creation process, Factory Method pattern is best for creating a single type of object with varying implementations, whereas Abstract Factory pattern is ideal for creating families of related objects that need to work together.

3) Mention the pros and cons of Factory Design pattern

It offers improved encapsulation, flexibility and reduced code duplication. It introduces complexities such as increased codebase complexity and potential overuse.

Post-Lab:

1) Design and implement a ride-sharing application that allows users to request ride various types of vehicles (cars, bikes, scooters). Utilize the Factory Method pa create vehicle instances, the Abstract Factory pattern to implement different pa methods (credit card, PayPal, cash), and the Singleton pattern to manage authentication securely. Provide a detailed example demonstrating the interacti these patterns within the application.

Procedure/Program:

```
public class Bike implements Vehicle {
 public void requestRide () {
System.out.println ("Requesting a Bike ride!!!");
 } }
public class BikeFactory implements VehicleFactory {
 public Vehicle createVehicle () {
      return new Bike ();
```

```
}} public class Car implements Vehicle {
    public void requestRide(){
    System.out.println("Requesting a Car Ride!!!");
}} public class CarFactory implements VehicleFactory{
    public Vehicle createVehicle(){
        return new Car();
    }}
public class CreditCardFactory implements PaymentMethod {
    public void pay(double amount) {
System.out.println("Paid $" + amount + " using a credit
                                    card..!");
    }}
    public class DemoMain {
public static void main(String[] args) {
UserAuthentication authentication = UserAuthentication.
                                    getInstance();
boolean isAuthenticate = authentication.authenticateUser
                ("username", "password");
    if (isAuthenticate) {
```

```
VehicleFactory carFactory = new CarFactory();
Vehicle car = carFactory.createVehicle();
car.requestRide();
VehicleFactory bikeFactory=new BikeFactory();
Vehicle bike= bikeFactory.createVehicle();
bike.requestRide();
VehicleFactory scooterFactory=new ScooterFactory();
Vehicle scooter= scooterFactory.createVehicle();
scooter.requestRide();
PaymentMethodFactory creditCardFactory=new
                     CreditCardFactory();
PaymentMethod creditCardPayment= creditCardFactory.
                     createPaymentMethod();
creditCardPayment.pay(20.0);
PaymentMethodFactory payPalFactory=new
                     PayPalFactory();
PaymentMethod payPalPayment=payPalFactory.
                     createPaymentMethod();
payPalPayment.pay(15.0);
}else{
```



```
System.ou
request a
    }
  }
} public
    void
} public
Payment
}
public class
  public Pa
    return
  }
}
public class
  Public v
System.out.pri
  }
}
```

```
System.out.println ("Authentication failed. Unable to
request a ride or make a payment.");
        }
    }
} public interface PaymentMethod {
    void pay(double amount);
} public interface PaymentMethodFactory {
 PaymentMethod createPaymentMethod();
}
public class PayPalFactory implements PaymentMethod
                                      Factory {
    public PaymentMethod createPaymentMethod() {
      return new PayPalPayment();
        }
    }
public class PayPalFactory implements PaymentMethod{
      Public void pay (double amount) {
System.out.println ("Paid $" + amount + "using PayPal..!");
    }
}
```

```
public class Scooter implements Vehicle {
public void requestRide () {
System.out.println ("Requesting a scooter ride!!!");
    }
  }
public class ScooterFactory implements VehicleFactory {
 public Vehicle create_Vehicle () {
     return new Scooter ();
   }
 } public class UserAuthentication {
private static UserAuthentication instance;
  private UserAuthentication () { }
public static UserAuthentication getInstance () {
    if (instance == null) {
    instance = new UserAuthentication ();
  } return instance;
} public boolean authenticateUser (String username,
                    String password) {
return true; } }
```



```
public int
   void req
} public i
   Vehicle
}
```

Experiment#		Student ID	
Date		Student Name	

```
public interface Vehicle {
    void requestRide();
} public interface VehicleFactory {
    Vehicle create_Vehicle();
}
```

Experiment#		Student ID	
Date		Student Name	

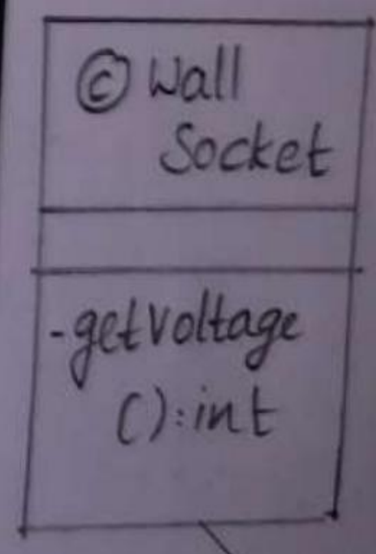
✓ **Data and Results:**

Riding a car.

Paying $10.0 with credit card

✓ **Analysis and Inferences:**

Factory Method pattern separate the process of creating different types of vehicles from client code. Abstract Factory pattern separates the creation of payment methods from the rest of application logic.



Evaluator Remark (if Any):	Marks Secured: _____out of 50
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment