

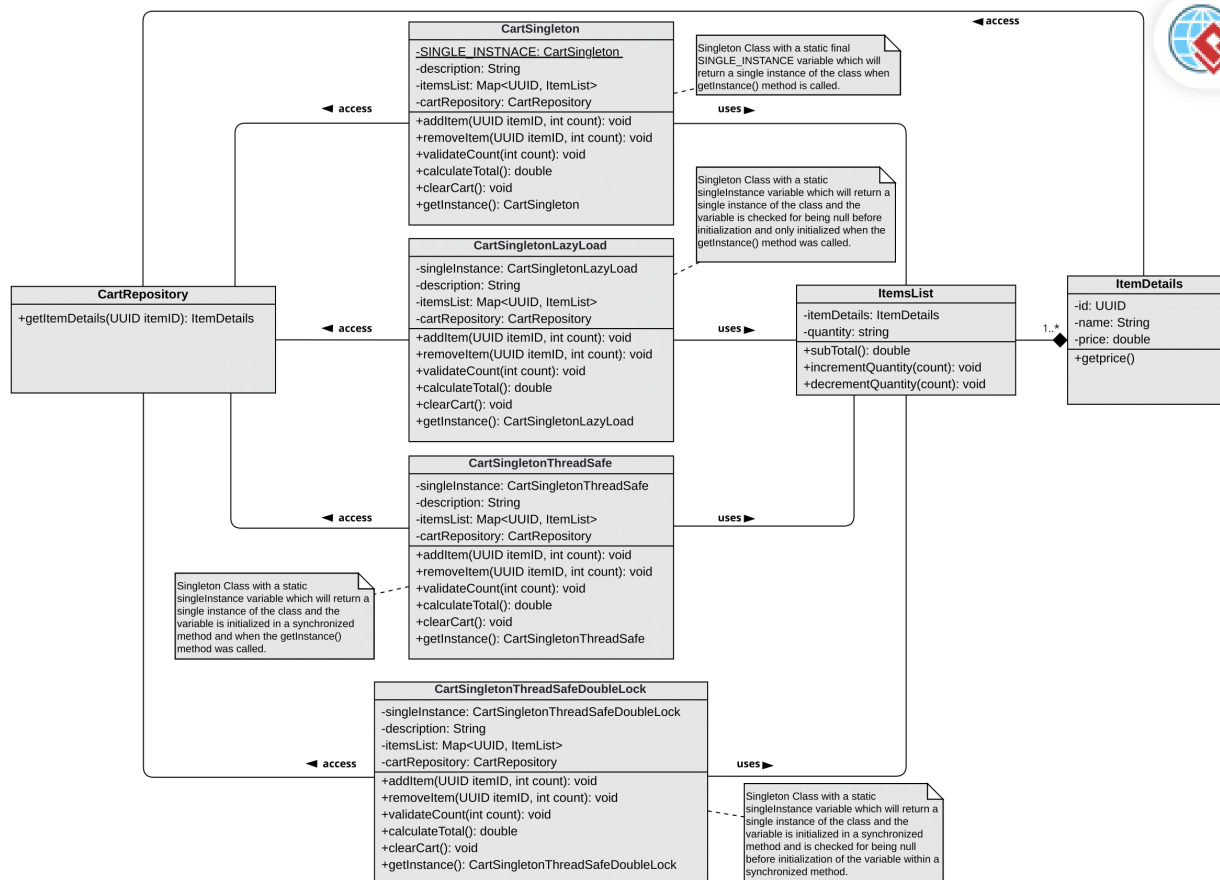
Name: Aravind Anand
Student ID: 030821269
Title: Assignment Unit 2

Question 1: Find a compelling application of Singleton design pattern for the food delivery system defined in assignment1 and draw its corresponding class diagram and a collaboration diagram and implement it in either Java or Python. Explain why singleton should be applied in that scenario (30 points).

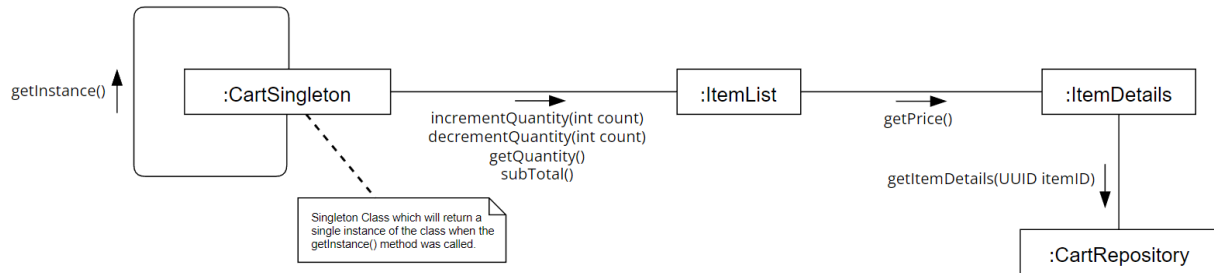
Singleton:

Using Singleton will ensure that a class only has one instance and provide a global point of access to it. In the online food delivery application, the operation of creating a cart can have only one instance as we can have only one cart at any given point in time. An implementation of cart creation using different singleton operations with their respective collaboration diagram, class diagram, and code snippets are as follows:

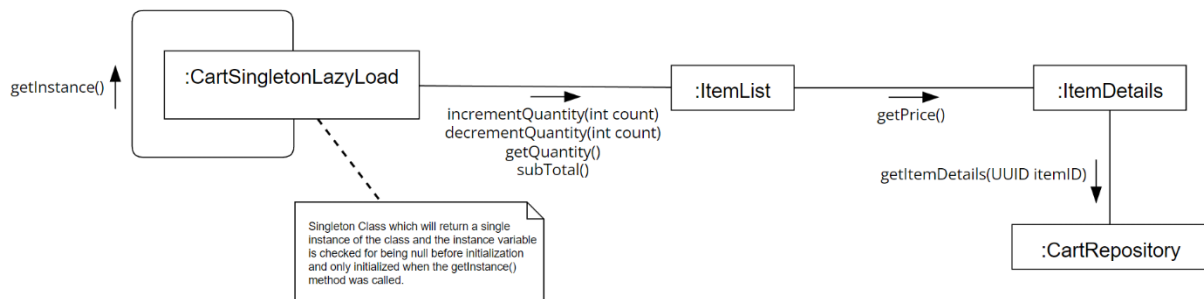
Class Diagram:



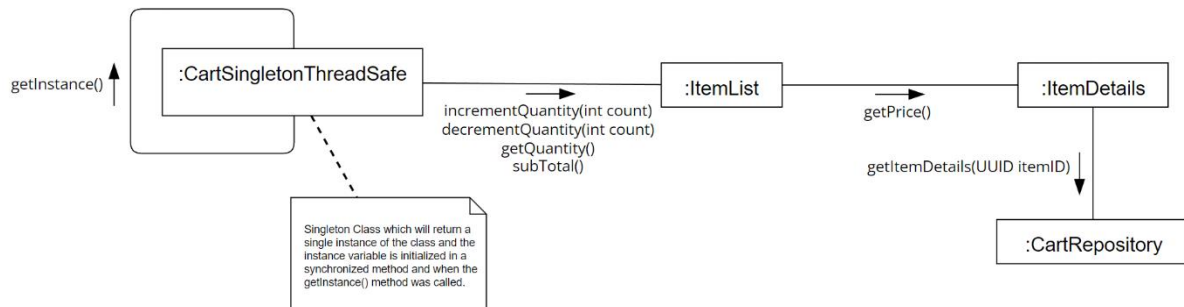
Collaboration Diagrams: Singleton:



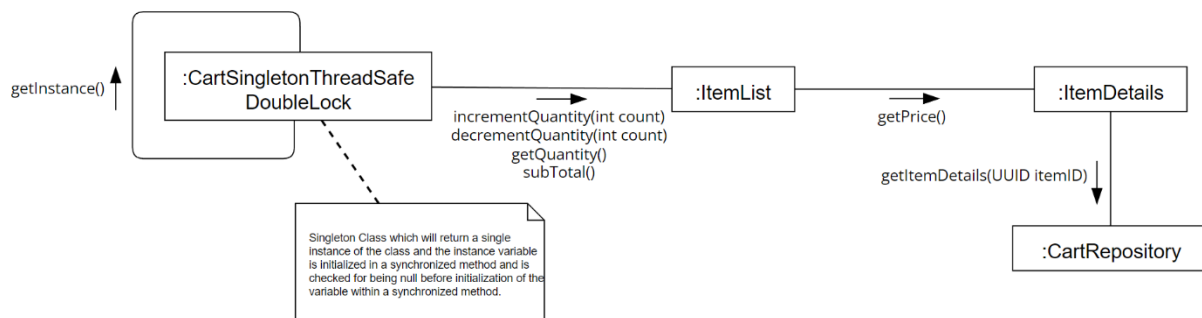
Singleton Lazy Load:



Singleton with Thread Safety:



Singleton with Thread Safety Double Locking:



Singleton – CartSingleton class:

```
public class CartSingleton {  
    1 usage  
    private static final CartSingleton SINGLE_INSTANCE = new CartSingleton(new CartRepository());  
}
```

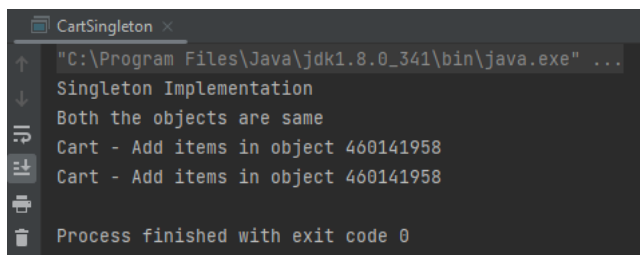
- Instance variable created/initialized.

```
2 usages  
public static CartSingleton getInstance() { return SINGLE_INSTANCE; }
```

- Definition of the getInstance() method

```
no usages  
public static void main(String[] args) {  
    CartSingleton cart1 = CartSingleton.getInstance();  
    CartSingleton cart2 = CartSingleton.getInstance();  
  
    if (cart1 == cart2) {  
        System.out.println("Both the objects are same");  
        cart1.printMessage();  
        cart2.printMessage();  
    }  
}
```

- Two objects are created for the CartSingleton class with getInstance() and checked if they are similar.



```
CartSingleton x  
"C:\Program Files\Java\jdk1.8.0_341\bin\java.exe" ...  
Singleton Implementation  
Both the objects are same  
Cart - Add items in object 460141958  
Cart - Add items in object 460141958  
Process finished with exit code 0
```

- Two objects were created with same hash code which infers that the class is returning only single instance for the class.

Singleton Lazy Load - CartSingletonLazyLoad:

```
public class CartSingletonLazyLoad {  
    3 usages  
    private static CartSingletonLazyLoad singleInstance = null;  
}
```

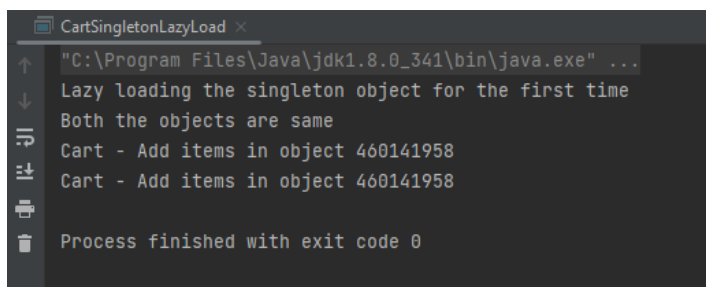
- Instance variable declared.

```
2 usages  
public static CartSingletonLazyLoad getInstance() {  
    if (singleInstance == null) {  
        singleInstance = new CartSingletonLazyLoad(new CartRepository());  
        System.out.println("Lazy loading the singleton object for the first time");  
    }  
    return singleInstance;  
}
```

- Definition of the getInstance() method – the instance variable is initialized only when the getInstance() method was called and the singleInstance variable is null.

```
public static void main(String[] args) {
    CartSingletonLazyLoad cart1 = CartSingletonLazyLoad.getInstance();
    CartSingletonLazyLoad cart2 = CartSingletonLazyLoad.getInstance();
    if (cart1 == cart2) {
        System.out.println("Both the objects are same");
        cart1.printMessage();
        cart2.printMessage();
    }
}
```

- Two objects are created for the CartSingletonLazyLoad class with getInstance() and checked if they are similar.



```
CartSingletonLazyLoad x
"C:\Program Files\Java\jdk1.8.0_341\bin\java.exe" ...
Lazy loading the singleton object for the first time
Both the objects are same
Cart - Add items in object 460141958
Cart - Add items in object 460141958
Process finished with exit code 0
```

- Two objects were created with same hash code which infers that the class is returning only single instance for the class.

Singleton Thread Safe - CartSingletonThreadSafe:

```
public class CartSingletonThreadSafe {
    3 usages
    private static CartSingletonThreadSafe singleInstance = null;
}
```

- Instance variable declared.

```
2 usages
synchronized public static CartSingletonThreadSafe getInstance()
{
    if (singleInstance == null){
        singleInstance = new CartSingletonThreadSafe(new CartRepository());
        System.out.println("Singleton with Thread Safe");
    }
    return singleInstance;
}
```

- Definition of the synchronized getInstance() method – the synchronized instance variable is initialized only when the getInstance() method was called and the singleInstance variable is null.

```

public static void main(String[] args) {
    CartSingletonThreadSafe cart1 = CartSingletonThreadSafe.getInstance();
    CartSingletonThreadSafe cart2 = CartSingletonThreadSafe.getInstance();

    if (cart1==cart2){
        System.out.println("Both the objects are same");
        cart1.printMessage();
        cart2.printMessage();
    }
}

```

- Two objects are created for the CartSingletonThreadSafe class with getInstance() and checked if they are similar.

```

CartSingletonThreadSafe x
"C:\Program Files\Java\jdk1.8.0_341\bin\java.exe" ...
Singleton with Thread Safe
Both the objects are same
Cart - Add items in object 460141958
Cart - Add items in object 460141958
Process finished with exit code 0

```

- Two objects were created with same hash code which infers that the class is returning only single instance for the class.

Singleton Thread Safe - CartSingletonThreadSafeDoubleLock:

```

public class CartSingletonThreadSafeDoubleLock {
    4 usages
    private static CartSingletonThreadSafeDoubleLock singleInstance = null;
}

```

- Instance variable declared.

```

2 usages
public static CartSingletonThreadSafeDoubleLock getInstance() {
    if (singleInstance == null) {
        //synchronized block to remove overhead
        synchronized (CartSingletonThreadSafeDoubleLock.class) {
            if(singleInstance ==null){
                singleInstance = new CartSingletonThreadSafeDoubleLock(new CartRepository());
                System.out.println("Singleton with Thread Safe Double Lock");
            }
        }
    }
    return singleInstance;
}
}

```

- Definition of the getInstance() method – when the getInstance() method was called, the instance variable is checked for null and instance variable is initialized within a synchronized method and only when the singleInstance variable is null.

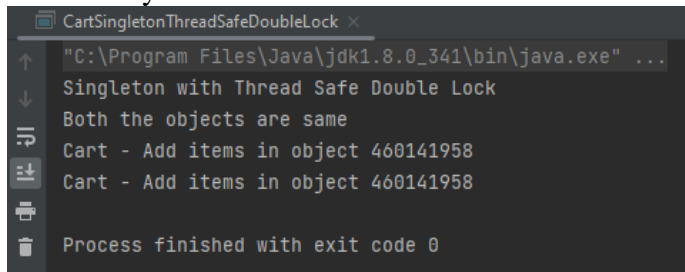
```

public static void main(String[] args) {
    CartSingletonThreadSafeDoubleLock cart1 = CartSingletonThreadSafeDoubleLock.getInstance();
    CartSingletonThreadSafeDoubleLock cart2 = CartSingletonThreadSafeDoubleLock.getInstance();

    if (cart1==cart2){
        System.out.println("Both the objects are same");
        cart1.printMessage();
        cart2.printMessage();
    }
}

```

- Two objects are created for the CartSingletonThreadSafeDoubleLock class and checked if they are similar.



```

"C:\Program Files\Java\jdk1.8.0_341\bin\java.exe" ...
Singleton with Thread Safe Double Lock
Both the objects are same
Cart - Add items in object 460141958
Cart - Add items in object 460141958
Process finished with exit code 0

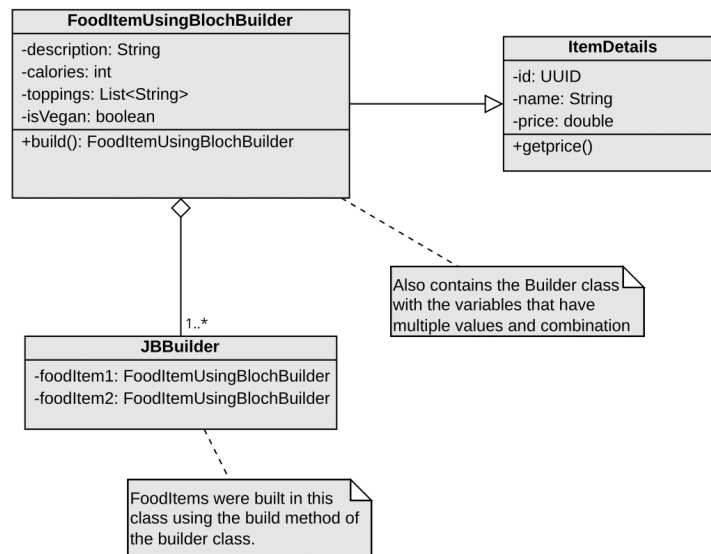
```

- Two objects were created with same hash code which infers that the class is returning only single instance for the class.

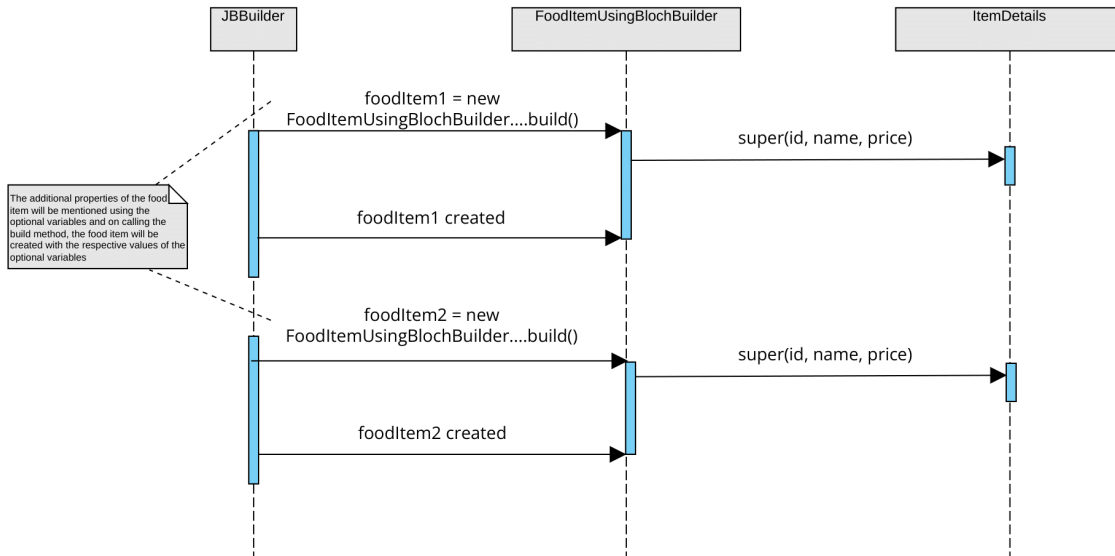
Question 2: Find a compelling scenario where you can apply Joshua Bloch's builder design pattern to the food delivery system. Draw the corresponding class diagram and sequence diagram and implement it in Java or Python (35 points)

In this Food Delivery System, adding an item to the cart by itself has additional options to be selected by the customer of their choice, such as spice level, vegetarian, etc. Providing multiple constructors (telescoping) for different combinations will not be manageable/maintainable and extendable. So, using Joshua Bloch's Builder Pattern we can create Builder class that can provide variables which can be built with different options/values. An implementation of building food item using Joshua Bloch's Builder Pattern with its sequence diagram, class diagram, and code snippets are as follows:

Class Diagram:



Sequence Diagram:



Joshua Bloch's Builder Pattern:

```
public class FoodItemUsingBlochBuilder extends ItemDetails {
    3 usages
    private final String description;
    3 usages
    private final int calories;
    3 usages
    private final List<String> toppings;
    3 usages
    private final boolean isVegan;

    1 usage
    private FoodItemUsingBlochBuilder(Builder builder) {
        super(builder.id, builder.name, builder.price);
        this.description = builder.description;
        this.calories = builder.calories;
        this.toppings = builder.toppings;
        this.isVegan = builder.isVegan;
    }
}
```

- FoodItemUsingBlochBuilder class with variables that will have multiple options and combinations.


```

    public FoodItemUsingBlochBuilder build() {
        return new FoodItemUsingBlochBuilder(this);
    }
}

@Override
public String toString() {
    return "FoodItemUsingBlochBuilder{" +
        "description='" + description + '\'' +
        ", calories=" + calories +
        ", toppings=" + toppings +
        ", isVegan=" + isVegan +
        '}';
}

```

- build() method which can be used to create Food Item with different specification.

```

public class JBBuilder {
    no usages
    public static void main(String[] args) {
        FoodItemUsingBlochBuilder foodItem1 = new FoodItemUsingBlochBuilder
            .Builder(UUID.randomUUID(), name: "Margherita Pizza")
            .description("Tomato sauce, mozzarella cheese, and basil")
            .price(10.99)
            .calories(600)
            .toppings(Arrays.asList("Tomato sauce", "Mozzarella cheese", "Basil"))
            .isVegan(false)
            .build();

        System.out.println("Food item 1: " + foodItem1);

        FoodItemUsingBlochBuilder foodItem2 = new FoodItemUsingBlochBuilder
            .Builder(UUID.randomUUID(), name: "Plain Margherita Pizza")
            .price(8.99)
            .calories(400)
            .build();

        System.out.println("Food item 2: " + foodItem2);
    }
}

```

```

JBBuilder
"C:\Program Files\Java\jdk1.8.0_341\bin\java.exe" ...
Food item 1: FoodItemUsingBlochBuilder{description='Tomato sauce, mozzarella cheese, and basil', calories=600, toppings=[Tomato sauce, Mozzarella cheese, Basil], isVegan=false}
Food item 2: FoodItemUsingBlochBuilder{description='null', calories=400, toppings=null, isVegan=false}
Process finished with exit code 0

```

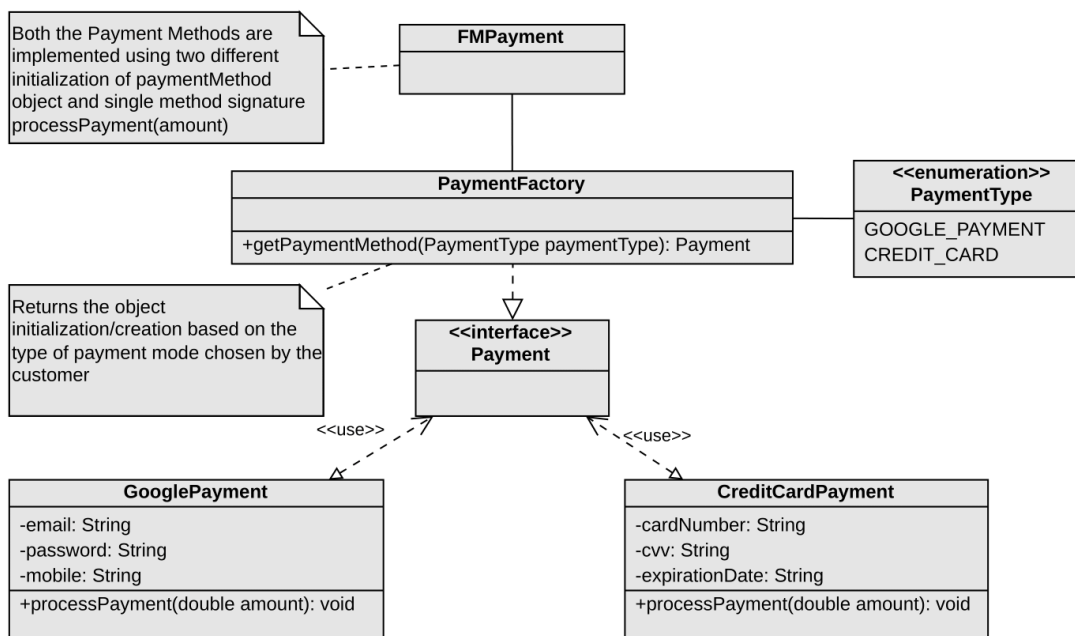
- Two food items were built, using build() method of FoodItemUsingBlochBuilder class, with two different combinations/options.

Question 3: Find a compelling scenario where you can apply either the Abstract Factory design, or factory method, or prototype, or GoF builder pattern to the food delivery system (pick one). Draw the corresponding class diagram and sequence diagram and implement in either Java or Python (35 points).

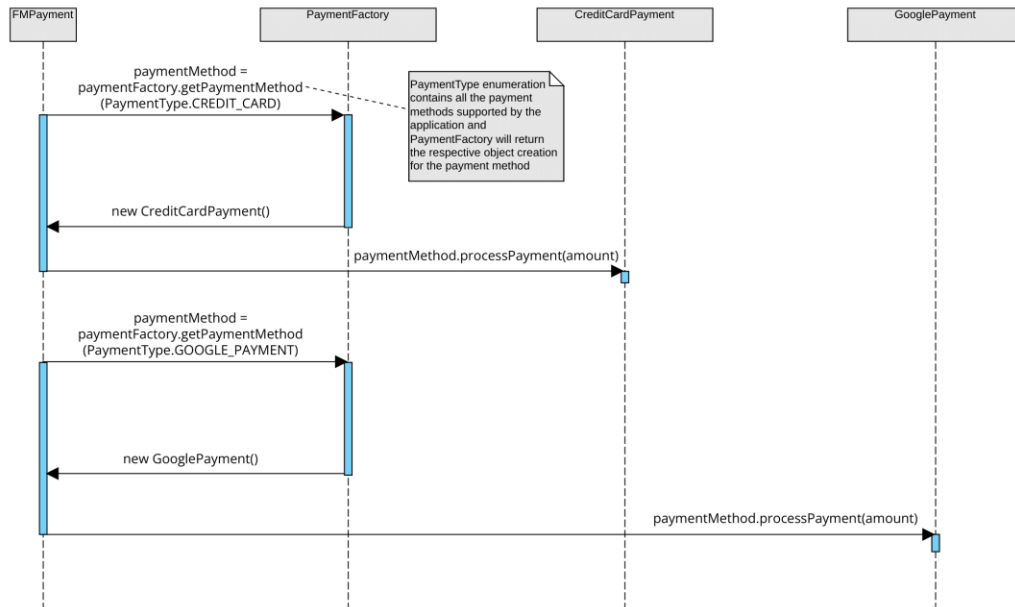
Factory Method:

In an Online Food Delivery, customers can use different payment methods. The factory method pattern can be used to create different types of payment methods based on the user's preferences. The payment processing methods can be described with multiple definitions and can be accessed by a single method signature. An implementation of multiple payment methods using Factory methods with its sequence diagram, class diagram, and code snippets are as follows:

Class Diagram:



Sequence Diagram:



Factory Method:

```
public interface Payment {
    2 usages 2 implementations
    void processPayment(double amount);
}
```

- Payment Interface

```
public class CreditCardPayment implements Payment {
    2 usages
    private String cardNumber;
    2 usages
    private String cvv;
    2 usages
    private String expirationDate;

    2 usages
    @Override
    public void processPayment(double amount) {
        // validate necessary info is provided
        // implementation of credit card payment processing
        System.out.println("Processing payment using credit card");
    }
}
```

- CreditCardPayment class with processPayment(double amount) method

```

public class GooglePayment implements Payment {
    2 usages
    private String email;
    2 usages
    private String password;
    2 usages
    private String mobile;
    2 usages
    @Override
    public void processPayment(double amount) {
        // validate necessary info is provided
        // implementation of Google payment processing
        System.out.println("Processing payment using GooglePay");
    }
}

```

- GooglePayment class with processPayment(double amount) method

```

public class PaymentFactory {
    2 usages
    public Payment getPaymentMethod(PaymentType paymentType) {
        if (paymentType == PaymentType.CREDIT_CARD) {
            return new CreditCardPayment();
        } else if (paymentType == PaymentType.GOOGLE_PAYMENT) {
            return new GooglePayment();
        } else {
            throw new IllegalArgumentException("Invalid payment method.");
        }
    }
}

```

- PaymentFactory class with getPaymentMethods(PaymentType paymentType) which will create/initialize and return the appropriate payment method objects.

```

public class FMPayment {
    no usages
    public static void main(String[] args) {
        PaymentFactory paymentFactory = new PaymentFactory();

        Payment paymentMethod = paymentFactory.getPaymentMethod(PaymentType.CREDIT_CARD);
        System.out.println("Payment method for type " + PaymentType.CREDIT_CARD);
        paymentMethod.processPayment( amount: 100);

        paymentMethod = paymentFactory.getPaymentMethod(PaymentType.GOOGLE_PAYMENT);
        System.out.println("Payment method for type " + PaymentType.GOOGLE_PAYMENT);
        paymentMethod.processPayment( amount: 100);
    }
}

```

FMPayment

```

"C:\Program Files\Java\jdk1.8.0_341\bin\java.exe" ...
Payment method for type CREDIT_CARD
Processing payment using credit card
Payment method for type GOOGLE_PAYMENT
Processing payment using GooglePay
Process finished with exit code 0

```

- Using Factory Method, both the payment modes were implemented with single method signature.