Template Method

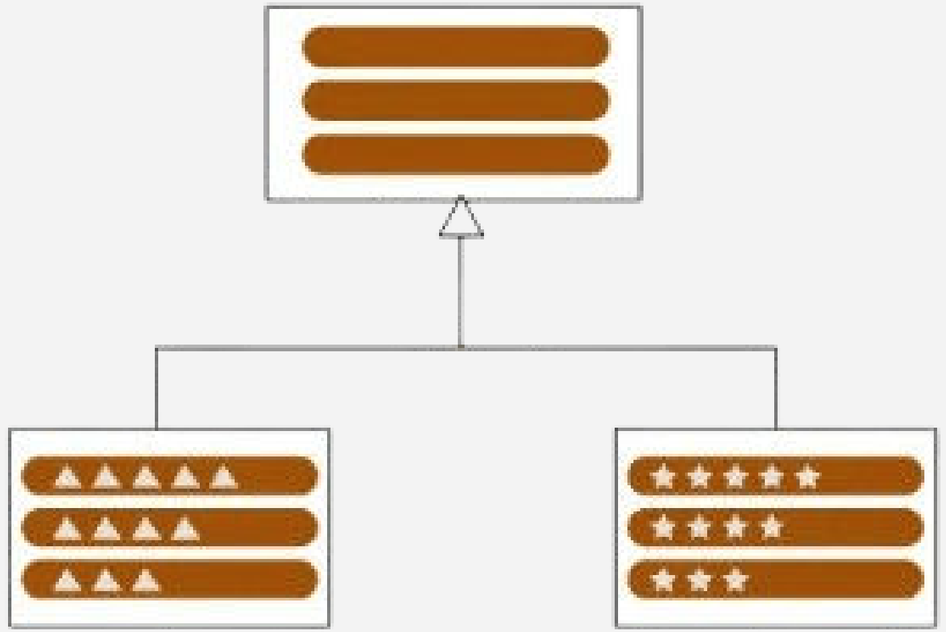
Module : **Java Avancée**

Encadré par : Pr. ELHAJJAMY Oussama

Réalisée par : Hayat Roubakhi

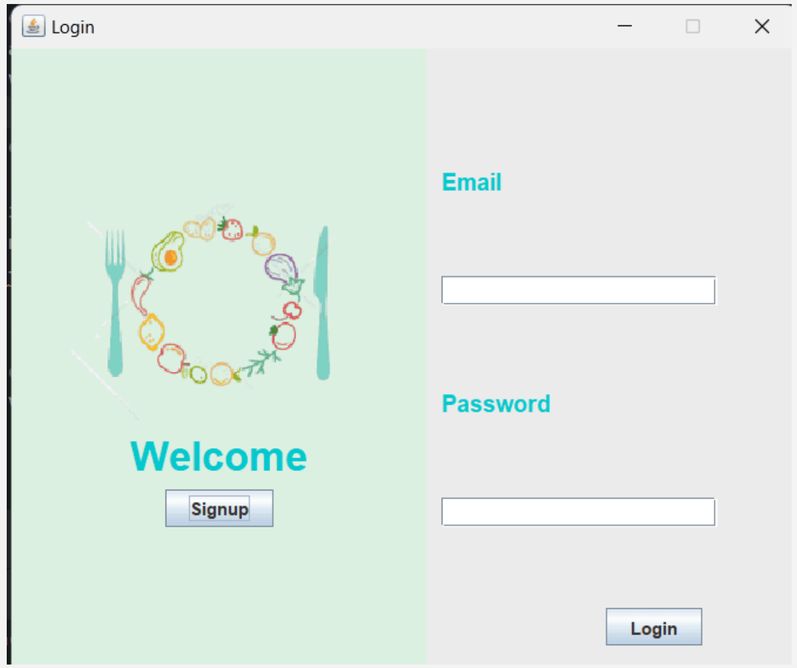
Introduction

Le design pattern **Template Method** est un modèle de conception comportemental dans lequel une classe abstraite définit la structure ou le squelette d'un algorithme avec des étapes spécifiques à implémenter par des sous-classes.



VITARES

L' application permet aux clients de commander des plats de restaurant **"VITARES"** en ligne et de les faire livrer à domicile. L'application donne accès à un menu complet que l'on peut réserver à distance. Pour l'utiliser, il faut d'abord se connecter ou créer un compte. Une fois cette étape franchie, les utilisateurs peuvent consulter les plats disponibles du jour et indiquer le nombre de plats qu'ils souhaitent commander.



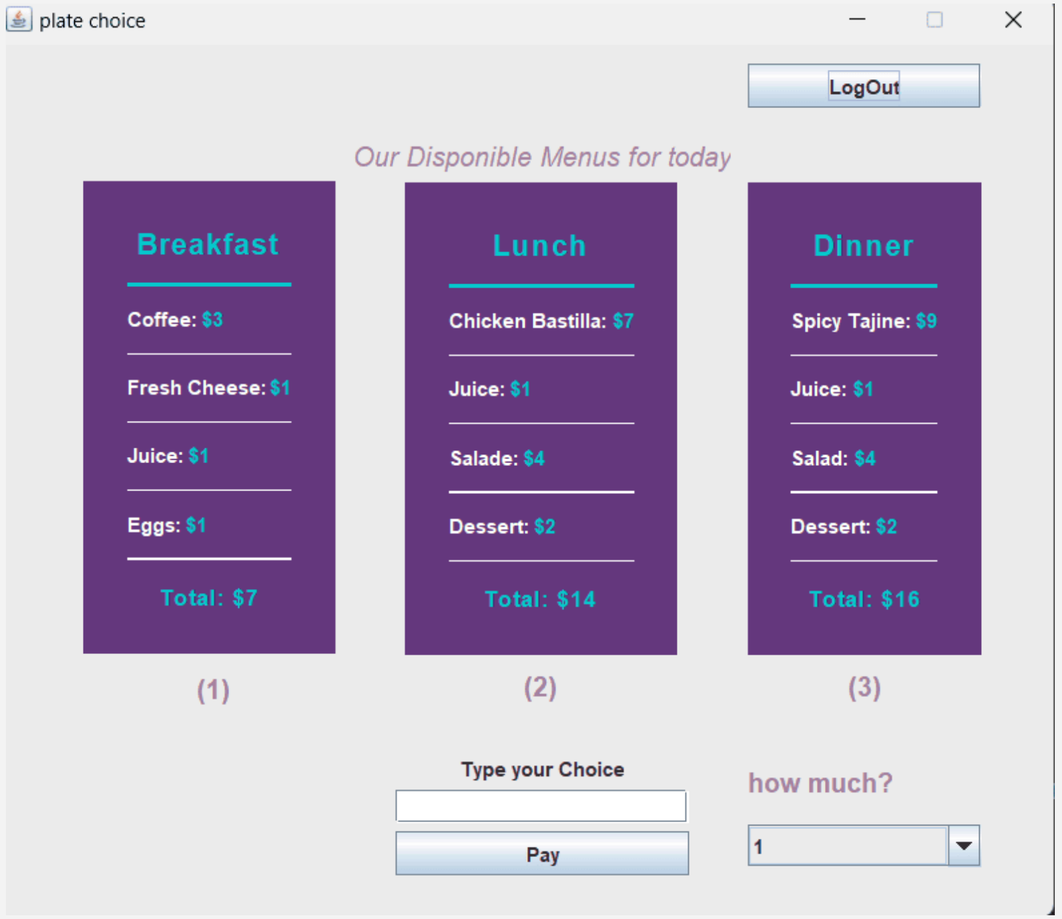


Diagramme de classes de l'application

Utilisation de trois design patterns :

méthode template

médiateur

stratégie

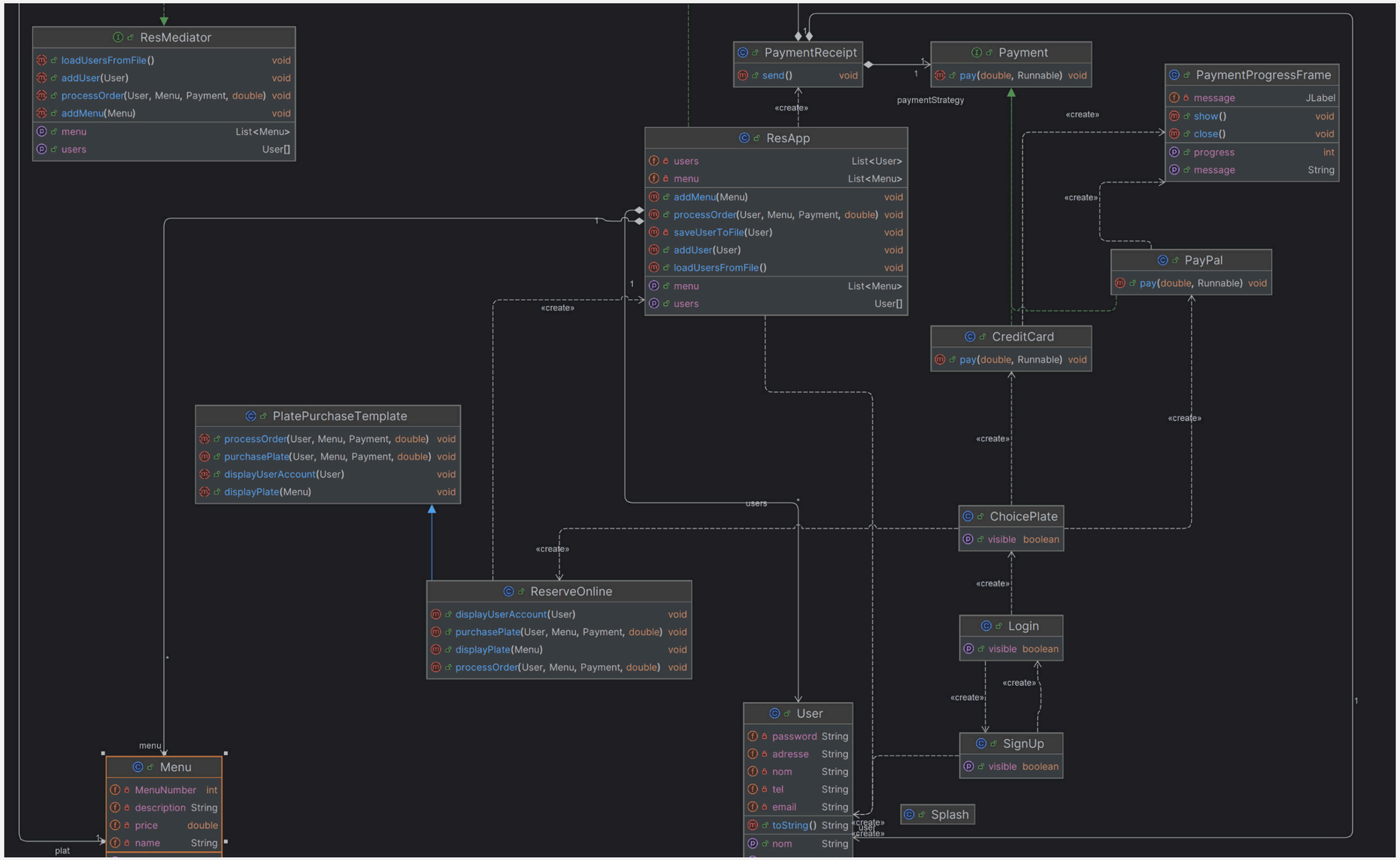
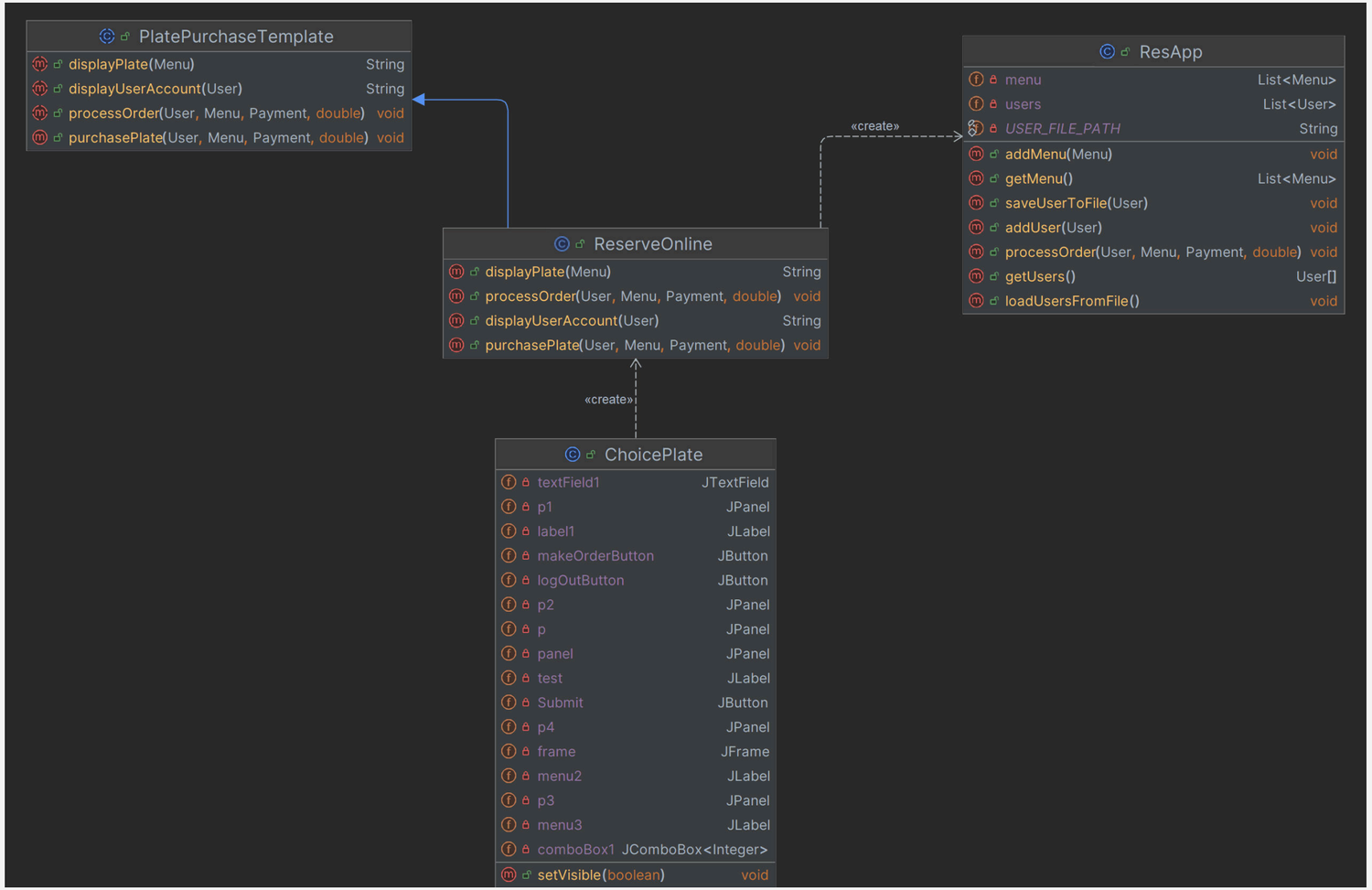


DIAGRAMME DE CLASSES : DESIGN PATTERN MÉTHODE TEMPLATE

PLATEPURCHASETEMPLATE

CLASSE ABSTRAITE DÉFINISSANT LA STRUCTURE GÉNÉRALE POUR L'ACHAT D'UN PLAT

RESERVEONLINE: **SOUS-CLASSE DE PLATEPURCHASETEMPLATE**



UTILISATION DE TEMPLATE METHOD

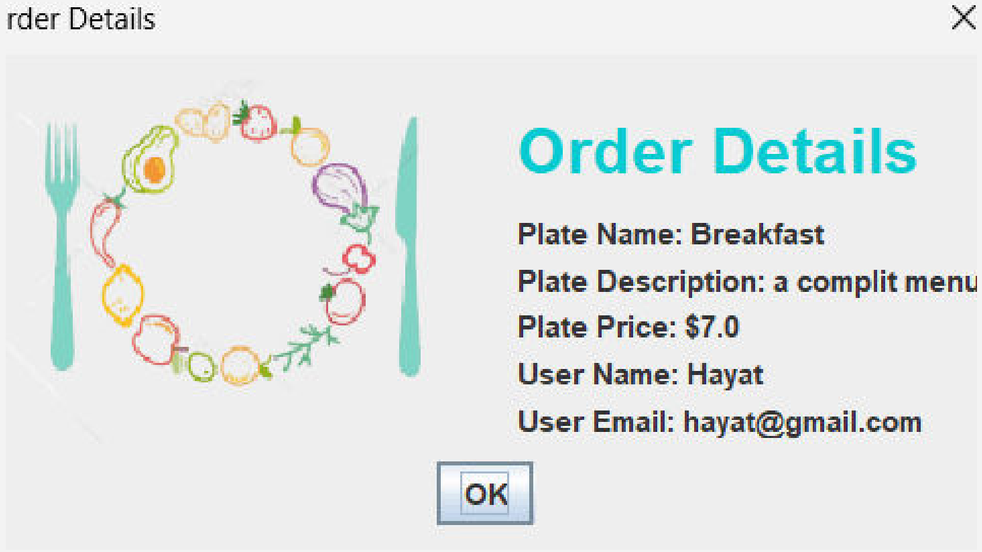
pour garantir une expérience utilisateur cohérente lors de l'achat de plats, la gestion des comptes utilisateurs et le traitement des paiements

```
    public void purchasePlate(User user, Menu plat, Payment paymentStrategy, double Price) {  2 usages
        displayPlate(plat);
        displayUserAccount(user);
        processOrder(user, plat, paymentStrategy,Price);
    }

    public abstract void displayPlate(Menu plate);  1 usage  1 implementation
    public abstract void displayUserAccount(User user);  1 usage  1 implementation
    public abstract void processOrder(User user, Menu plate, Payment paymentStrategy, double Price);
```

Utilisation de modèle Template Method

displayPlate(plat), displayUserAccount(user): **code Source**

```
public String displayPlate(Menu plate) {
    // Display plate information in a message dialog
    return "Plate Name: " + plate.getName() + "\n"
            + "Plate Description: " + plate.getDescription()
            + "Plate Price: $" + plate.getPrice();
}

@Override  2 usages
public String displayUserAccount(User user) {
    // Display user account information in a message dialog
    return "User Name: " + user.getNom() + "\n"
            + "User Email: " + user.getEmail();
```



Utilisation de modèle Template Method

Methode processOrder : code Source

```java
public void processOrder(User user, Menu plate, Payment paymentStrategy, double price) {
    // Assuming ResMediator exists and is valid
    ResMediator mediator = new ResApp();
    mediator.addUser(user);
    mediator.addMenu(plate);
    mediator.processOrder(user, plate, paymentStrategy, price);
```

Order Confirmation

Order for plate: Breakfast processed successfully!
Payment amount: $7.0

You're going to receive your order at the provided address:
123BesideMe
Regards, VitaRes's Restaurant

OK

Utilisation de modèle Template Method

Création d'une instance de la sous-classe ReserveOnline en utilisant le type abstrait PlatePurchaseTemplate

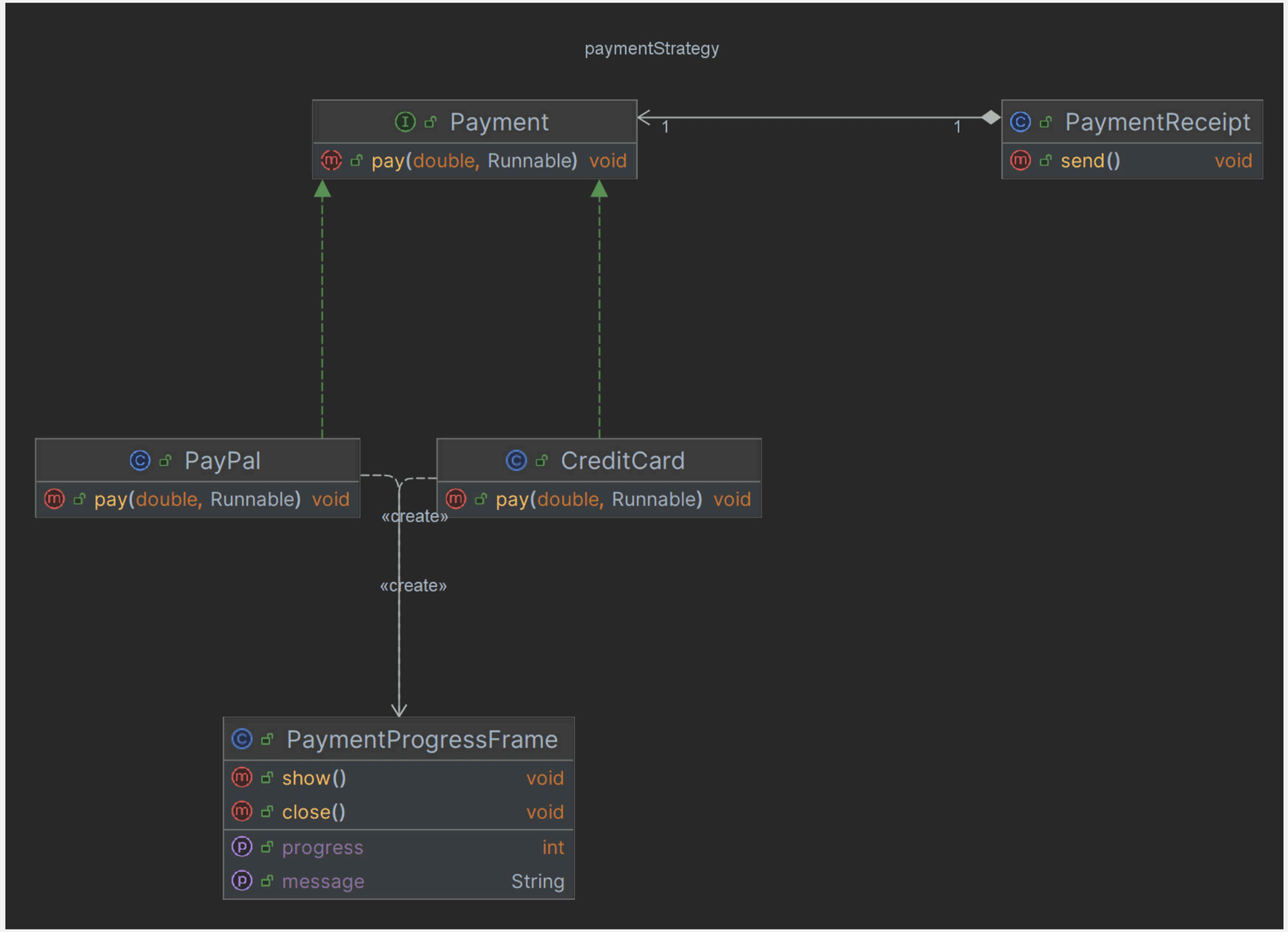
```
PlatePurchaseTemplate ReserveOnline = new ReserveOnline();
```

Appel de la méthode purchasePlate pour initier l'achat d'un plat, On passe en paramètre : l'utilisateur qui effectue le demande - le plat sélectionné à partir du menu , la stratégie de paiement à utiliser et le prix total basé sur le prix unitaire multiplié par le nombre d'articles sélectionnés

```
ReserveOnline.purchasePlate(user, matchingMenu, paymentStrategy, Price: matchingMenu.getPrice()* selectedItem);
frame.dispose();
```

Diagramme de classes Pattern Strategy

Le modèle Strategy facilite la flexibilité dans la sélection des méthodes de paiement



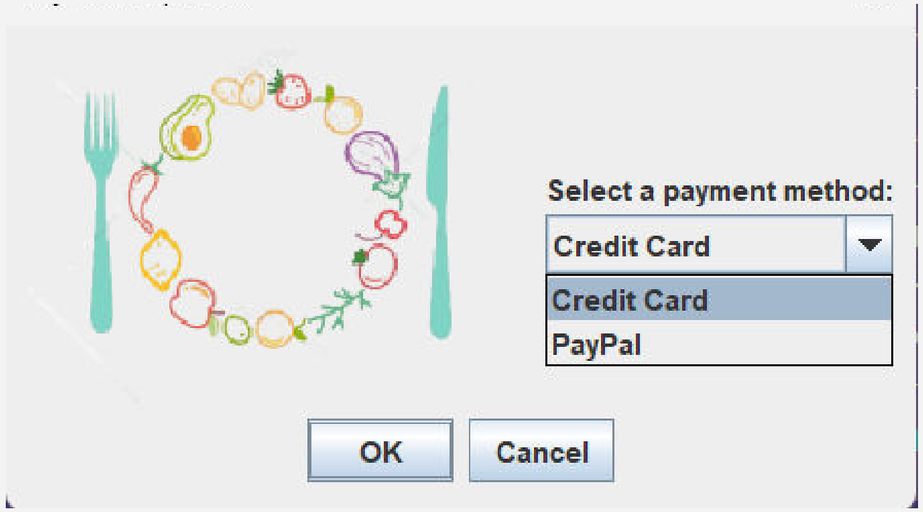
Utilisation de Pattern Strategy

Interface Payement: code Source

```
public interface Payment {

        1 usage   2 implementations
        public void pay(double amount, Runnable onComplete);


}
```

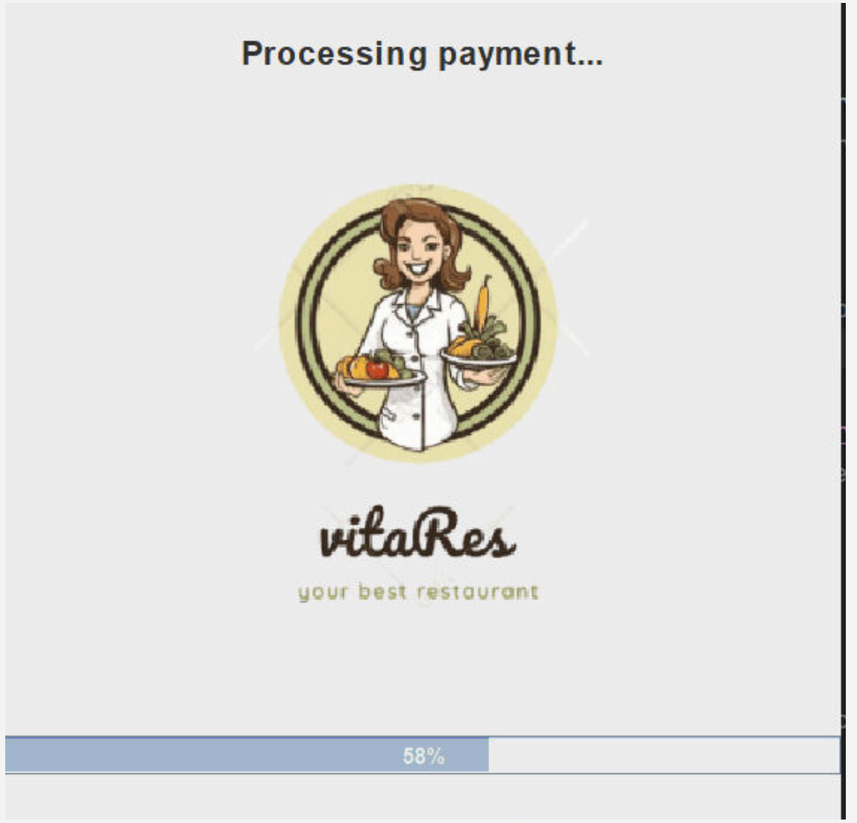
```
Payment paymentCard = new CreditCard();
Payment paymentPayPal = new PayPal();
Payment paymentStrategy = null;
```

Choix entre les méthodes de paiement



Methode Pay : code Source

```java
public void pay(double amount, Runnable onComplete) {
    PaymentProgressFrame progressFrame = new PaymentProgressFrame();
    progressFrame.show();

    Thread paymentThread = new Thread(() -> {
        try {
            for (int i = 0; i <= 100; i++) {
                final int progress = i;
                SwingUtilities.invokeLater(() -> progressFrame.setProgress(progress));
                Thread.sleep( millis: 20);
            }


            Thread.sleep( millis: 500);
            SwingUtilities.invokeLater(() -> {
                progressFrame.close();
                onComplete.run();
            });

        } catch (InterruptedException e) {
            SwingUtilities.invokeLater(() -> progressFrame.setMessage("Payment process interrupted."));
        }
    });
```



class PaymentRecepient : code Source

```
2 usages
public class PaymentReceipt {
    2 usages
    private User user;
    2 usages
    private Menu plat;
    2 usages
    private double amount;
    3 usages
    private Payment paymentStrategy;

    1 usage
    public PaymentReceipt(User user, Menu plat, double amount, Payment paymentStrategy) {
        this.user = user;
        this.plat = plat;
        this.amount = amount;
        this.paymentStrategy = paymentStrategy;
    }

    1 usage
    public void send() {
        // Determine the payment method
        String paymentMethod = paymentStrategy instanceof CreditCard ? "Credit Card" :
                paymentStrategy instanceof PayPal ? "PayPal" : "Unknown";

        // Build the receipt message
```

PaymentRecepient

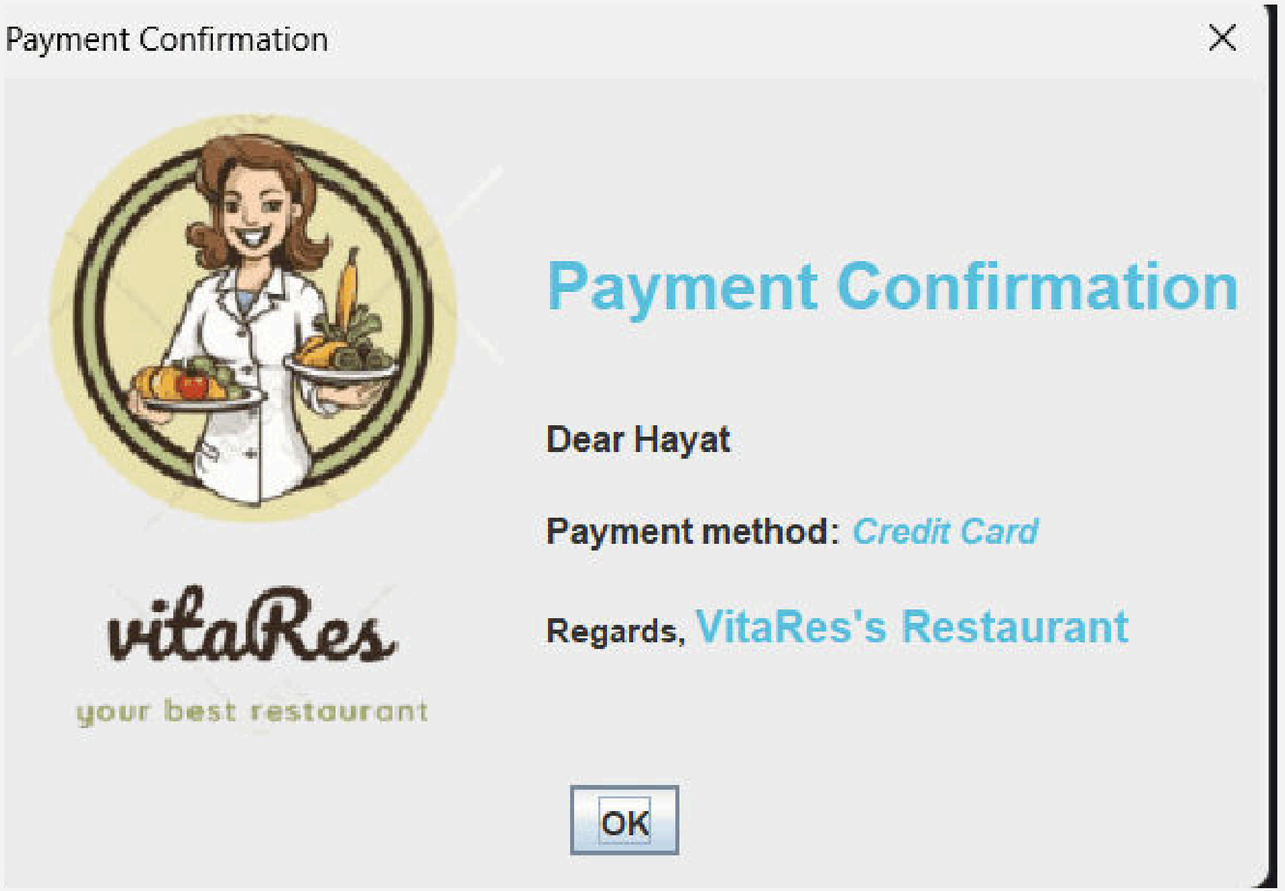
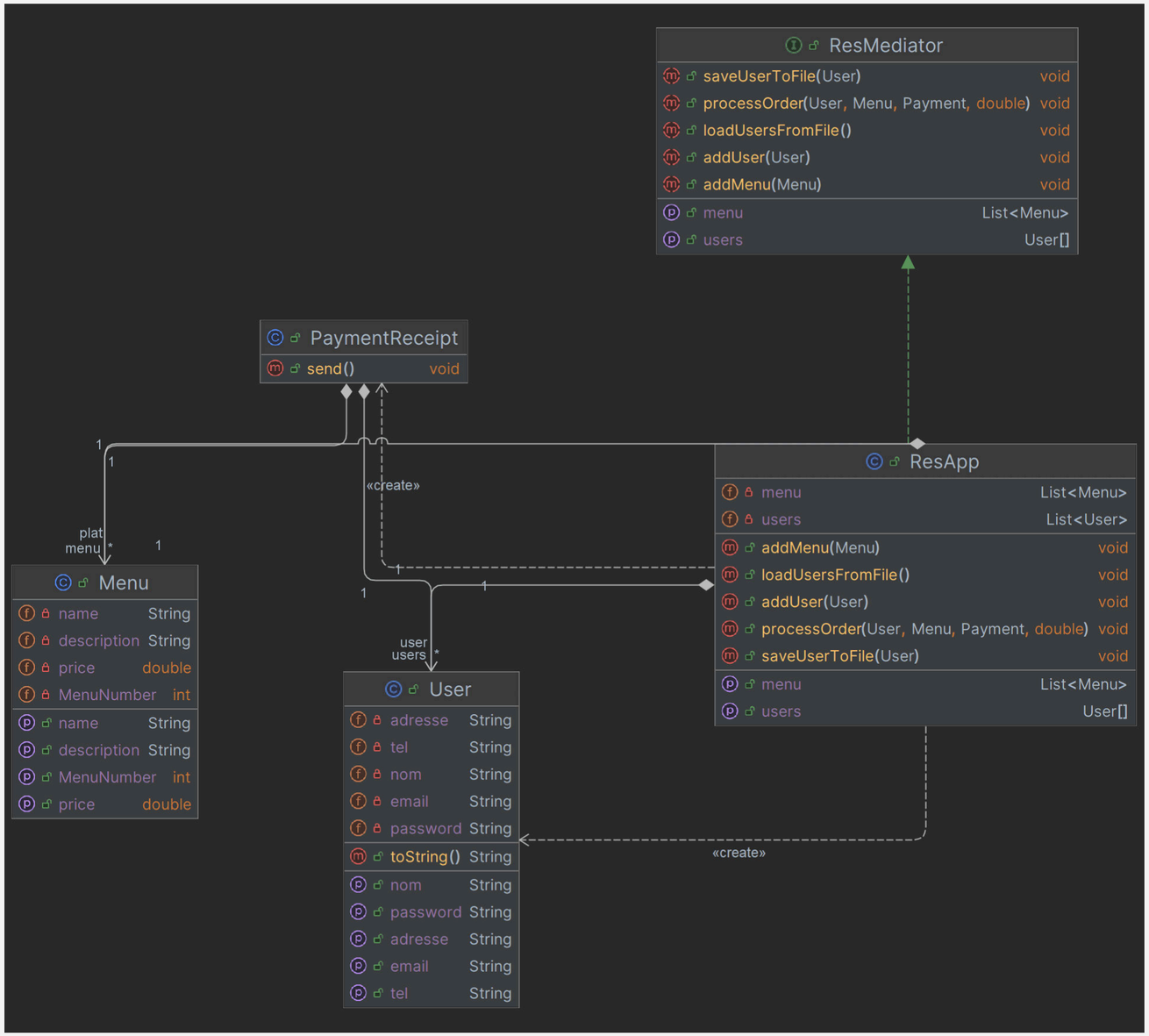


Diagramme de classes Pattern Mediator

le modèle Mediator facilite la communication entre les différents composants de l'application, comme le menu des plats, et l'utilisateur.



code Source

1. Créer une instance de ResMediator pour la gestion des ressources.
2. Charger les utilisateurs depuis un fichier.
3. Créer une instance de l'écran de connexion avec le médiateur.
4. Créer des instances de menus pour les différents repas. 5. Ajouter les menus au médiateur.

```
ResMediator mediator = new ResApp();
 mediator.loadUsersFromFile();
Login loginScreen = new Login(mediator);
loginScreen.setVisible(true);

Menu menu1 = new Menu( name: "Breakfast",  description: "a complit menu",  price: 7, menuNumber: 1);
Menu menu2 = new Menu( name: "Lunch",  description: "a complit menu",  price: 14, menuNumber: 2);
Menu menu3 = new Menu( name: "Dinner",  description: "a complit menu",  price: 17, menuNumber: 3);
mediator.addMenu(menu1);
mediator.addMenu(menu2);
mediator.addMenu(menu3);
```

```
@Override  1 usage
public void processOrder(User user, Menu plat, Payment paymentStrategy, double Price) {
    if (menu.contains(plat) && users.contains(user)) {
        PaymentReceipt paymentReceipt = new PaymentReceipt(user, plat, Price, paymentStrategy);
        paymentReceipt.send(); // Ensure this runs on the EDT
    } else {
        System.out.println("Sorry, we could not process your order. Please try again later.");
    }
}

@Override  2 usages
```

Création d'un nouvel utilisateur et ajout à la liste des médiateurs.

```
// Create a new user and add to the mediator
User newUser = new User(name, email, address, phone, password);
mediator.addUser(newUser);
mediator.saveUserToFile(newUser);

// Provide feedback to the user
```

CONCLUSION

En conclusion, le modèle de conception Template Method offre une approche puissante pour **optimiser les structures de programme** en favorisant la **réutilisation du code** et la **flexibilité du design**. Il est essentiel de **comprendre ses principes fondamentaux** pour une utilisation efficace.

MERCI!