ESCUELA SUPERIOR POLITÉCNICA DEL LITORAL

Facultad de Ingeniería en Electricidad y Computación

TALLER PATRONES DE DISEÑO
DISEÑO DE SOFTWARE

Integrantes

Araujo Ortega Diego, dienarau@espol.edu.ec

Barzola Rodríguez Leidy, ljbarzol@espol.edu.ec

Borbor Gutiérrez Víctor, vicbguti@espol.edu.ec

Criollo Córdova Isaac, ivcrioll@espol.edu.ec

Índice

https://github.com/DiegoA00/TallerPatronDiseno1

Herramienta utilizada para modelar los diagramas:

1. Lucid Chart.
2. Visual paradigm.

Sección A

Elabore un reporte en el que identifique los patrones de diseño que son aplicables para el sistema de software descrito. Para cada patrón que considere pertinente, debe indicar la motivación de su uso y las consecuencias de su decisión (ej.: pros, contras, relación con principios SOLID). Indique cualquier asunción que realice.

Párrafo 1:

"Considere el sistema de software de una tienda minorista que se especializa en venta de ropa, productos para el hogar y productos alimenticios. La tienda ofrece afiliaciones a sus clientes por medio de la emisión de tarjetas de crédito, las cuales tienen un costo anual y un límite de crédito. De acuerdo con las características del cliente, la tienda le puede otorgar una de las siguientes tarjetas de crédito: Básica, Premium o VIP. En donde cada una tiene diferentes costos y límites crediticios; siendo Básica la más elemental y VIP la más exclusiva."

Patrón de diseño seleccionado: Factory Method.

Motivación de uso:

Basándose en la sección del párrafo "la tienda le puede otorgar una de las siguientes tarjetas de crédito: Básica, Premium o VIP. En donde cada una tiene diferentes costos y límites crediticios; siendo Básica la más elemental y VIP la más exclusiva."

Dicha sección es comprendida y puede ser abarcada desde el patrón de diseño Factory Method. Patrón de diseño que se utiliza principalmente para proporcionar una interfaz para la creación de objetos (tarjetas) en una superclase, pero que permite a las subclases alterar el tipo de objetos que se crearán.

El Factory Method permite que la clase cliente que utiliza el objeto no necesite conocer la clase específica de los objetos que crea. Esto reduce el acoplamiento entre la clase cliente y las clases concretas (tarjetas), facilitando la extensión y mantenimiento del código.

Este patrón proporciona un mejor control entre clases dado que, en lugar de que la clase cliente cree directamente objetos, la responsabilidad de la creación se delega a las subclases de las tarjetas. Esto invierte el control del flujo del programa, proporcionando mayor flexibilidad y facilitando la extensión del sistema.

Consecuencias:

El patrón de diseño Factory Method proporciona flexibilidad en la creación de objetos y desacopla la clase cliente de las clases concretas que crea, pero trae consigo un mayor número de clases en el sistema, ya que conlleva que cada producto y su correspondiente fábrica requieran de una clase adicional; en consecuencia:

- Se crea una jerarquía adicional ya que la necesidad de proporcionar un Factory Method en cada subclase resulta en una jerarquía de clases más profunda.
- Genera un acoplamiento indirecto puesto que, aunque el patrón busca desacoplar la clase cliente de las clases de tarjetas, todavía existe un acoplamiento a través de la interfaz del Factory Method.
- Aumenta la flexibilidad en la creación de objetos ya que, al utilizar el Factory Method, las subclases pueden proporcionar su propia implementación de la creación de objetos permitiendo adaptarse según las necesidades específicas de cada subclase sin modificar la lógica de la clase cliente.

Pros:

- Se evita un acoplamiento fuerte entre el cliente y los diferentes tipos de tarjetas concretas.
- Al invertir en la herencia, permite extender el comportamiento de las subclases, los tipos de tarjeta, según sea requerido.
- El patrón Factory Method desvincula la construcción del producto del código que utiliza dicho producto. Esto facilita la extensión del código de construcción del producto de manera independiente al resto del código.
- Reduce el código que construye componentes en todo el framework a un único patrón Factory Method y permitir que cualquiera sobrescriba este método además de extender el propio componente.

Contras:

- Se puede dar el caso de que el código se complique demasiado, ya que al incorporar muchas subclases al implementar el patrón se aumente la complejidad del código. Es decir, la creación de varios objetos vuelte dificultoso su entendimiento si es que hay muchas clases concretas y diferentes variantes de productos.

- La flexibilidad puede ser una desventaja ya que demasiada flexibilidad podría llevar a un diseño complejo y difícil de mantener si no se sigue un equilibrio adecuado que cumpla y no infrinja los principios SOLID.

Relación con principios SOLID:

- Single Responsibility Principle: Permite mover el código de creación de cada tarjeta a un lugar del programa, haciendo que el código sea más fácil de mantener puesto que cada clase cuenta con una única función de la cual es responsable.
- Open – Closed Principle: permite incorporar nuevos tipos de tarjetas en el programa sin descomponer el código del cliente existente y dando lugar a la extensión sin modificar código ya existente.

Párrafo 2:

“Debido a la antigüedad de su plataforma tecnológica, la tienda solo había venido ofreciendo sus servicios de forma presencial en sus locales. Pero ahora, sin haber migrado su plataforma, la tienda abrirá tres nuevos canales de atención virtual para sus clientes: Web, móvil y telefónico. No obstante, los subsistemas correspondientes a estos canalessolo ofrecerán un conjunto limitado de transacciones en comparación con aquellas que pueden realizarse presencialmente.”

Principio de diseño seleccionado: Facade.

Motivación de uso:

La motivación para usar el patrón Facade radica en proporcionar a los clientes una interfaz única y fácil de usar, sin importar qué canal virtual elijan. Facade actuará como una especie de "fachada" o interfaz principal que oculta la complejidad de los subsistemas subyacentes. De esta manera, cada cliente se conectará a través de su plataforma preferida (ya sea web, móvil o telefónica), y la fachada se encargará de dirigirlos al subsistema correspondiente según el canal de atención virtual seleccionado.

Este enfoque simplificado tiene como objetivo brindar una experiencia consistente para el usuario.

Consecuencias:

Pros:

Facade proporciona una interfaz única y fácil de usar para los clientes, independientemente de si eligen la plataforma web, móvil o telefónica. Esto simplifica la interacción del usuario.

La interfaz actúa como una "fachada" que oculta la complejidad de los subsistemas subyacentes. Esto beneficia a los clientes al brindarles una experiencia más intuitiva, sin acceder a la complejidad detrás de cada canal virtual.

Dado que los subsistemas virtuales ofrecen un conjunto limitado de transacciones en comparación con las opciones presenciales, Facade optimiza la gestión de estas transacciones. Esto permite una distribución eficiente de las funcionalidades en los canales virtuales.

Contras del uso del patrón Facade:

Cambios en los subsistemas pueden afectar la fachada y, por ende, a los clientes que utilizan la fachada. Esto puede resultar en una situación en la que pequeños cambios en los subsistemas requieren modificaciones en la fachada y, posiblemente, en varias partes del código del cliente:

La alta dependencia entre la fachada y los subsistemas podría dificultar la mantenibilidad del sistema a medida que evoluciona con el tiempo.en caso de querer añadir o borrar un subsistema se tendría que cambiar parte de la fachada.

Relación con Principios SOLID:

1. Principio de Responsabilidad Única (SRP): Facade, que actúa como el conductor central dirigiendo a los clientes hacia los nuevos canales virtuales (Web, móvil y telefónico). Facade se adapta a los cambios sin afectar la operación presencial existente, cumpliendo así con el SRP al tener una única razón para cambiar: las modificaciones en la gestión de los canales virtuales.

2. Principio de Abierto/Cerrado (OCP): Facade facilita la extensión al agregar nuevos canales virtuales sin modificar la fachada existente. La fachada está cerrada para modificaciones, pero abierta para extensiones, cumpliendo con el OCP.

Párrafo 3

“Una semana antes del vencimiento del plazo para pagar lo adeudado por compras con tarjeta de crédito, la tienda envía notificaciones a sus clientes. La notificación debe ser obligatoriamente por dos mecanismos: mensajes SMS y correos electrónicos. Sin embargo, si el cliente lo desea puede configurar dinámicamente mecanismos adicionales. Los cuales incluyen: WhatsApp, Signal, Wire o Telegram.”

Patrón de diseño seleccionado: Decorator

Motivación:

Decorator permite agregar dinámicamente nuevos comportamientos o funcionalidades a un objeto existente sin afectar su estructura base. En este caso, al permitir a los clientes configurar sus propios mecanismos de notificación adicionales (WhatsApp, Signal, Wire o Telegram), este patrón facilita la extensibilidad del sistema.

Además, a medida que surjan nuevas opciones de notificación o se necesiten más en el futuro, el patrón Decorator permite agregar estas funcionalidades de forma modular y escalable.

Consecuencias:

Pros:

- Permite agregar dinámicamente nuevos comportamientos al objeto.
- Extensibilidad sin modificar la estructura base del objeto.

Contras:

- Puede llegar a ser muy complejo si se utiliza múltiples capas de decoradores.

- Dificultad para eliminar una capa, ya que el patrón no proporciona una forma directa de quitar un funcionamiento después de agregarlo.

Relación con los principios SOLID:

- Cumple con el SRP al permitir que cada decorador tenga una responsabilidad única, como añadir un método de notificación específico.
- Cumple con el OCP al permitir la extensión del comportamiento sin modificar el código existente.

Sección B

Elabore los diagramas de clases y de secuencia para el sistema de software descrito en el que se reflejen los patrones de diseño escogidos y la escalabilidad de su solución. Presente el diagrama de clases de cada patrón en un paquete separado. Identifique herencias, multiplicidades, visibilidad de atributos y métodos relevantes para el patrón. Por medio de notación UML, muestre si las entidades corresponden a interfaces, clases abstractas o clases concretas. Además, muestre el uso de mensajes síncronos, asíncronos y de retorno que sean pertinentes. Indique cualquier asunción que realice.

https://lucid.app/lucidchart/ce6c8c46-3a7b-4b97-b5a7-2b864e46de53/edit?invitationId=inv_1a71b46f-7e7a-42ed-a87a-4a496ba3b3ad&page=0_0#

Factory Method

Diagrama de Clase:

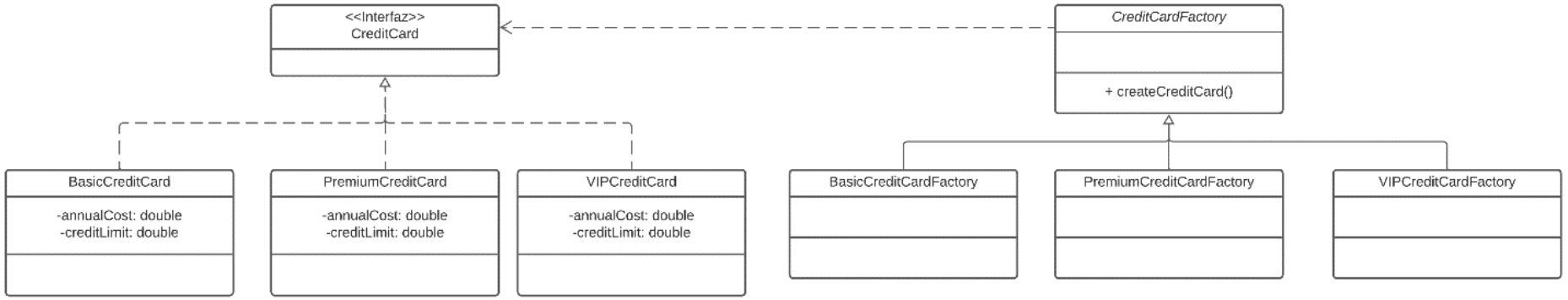
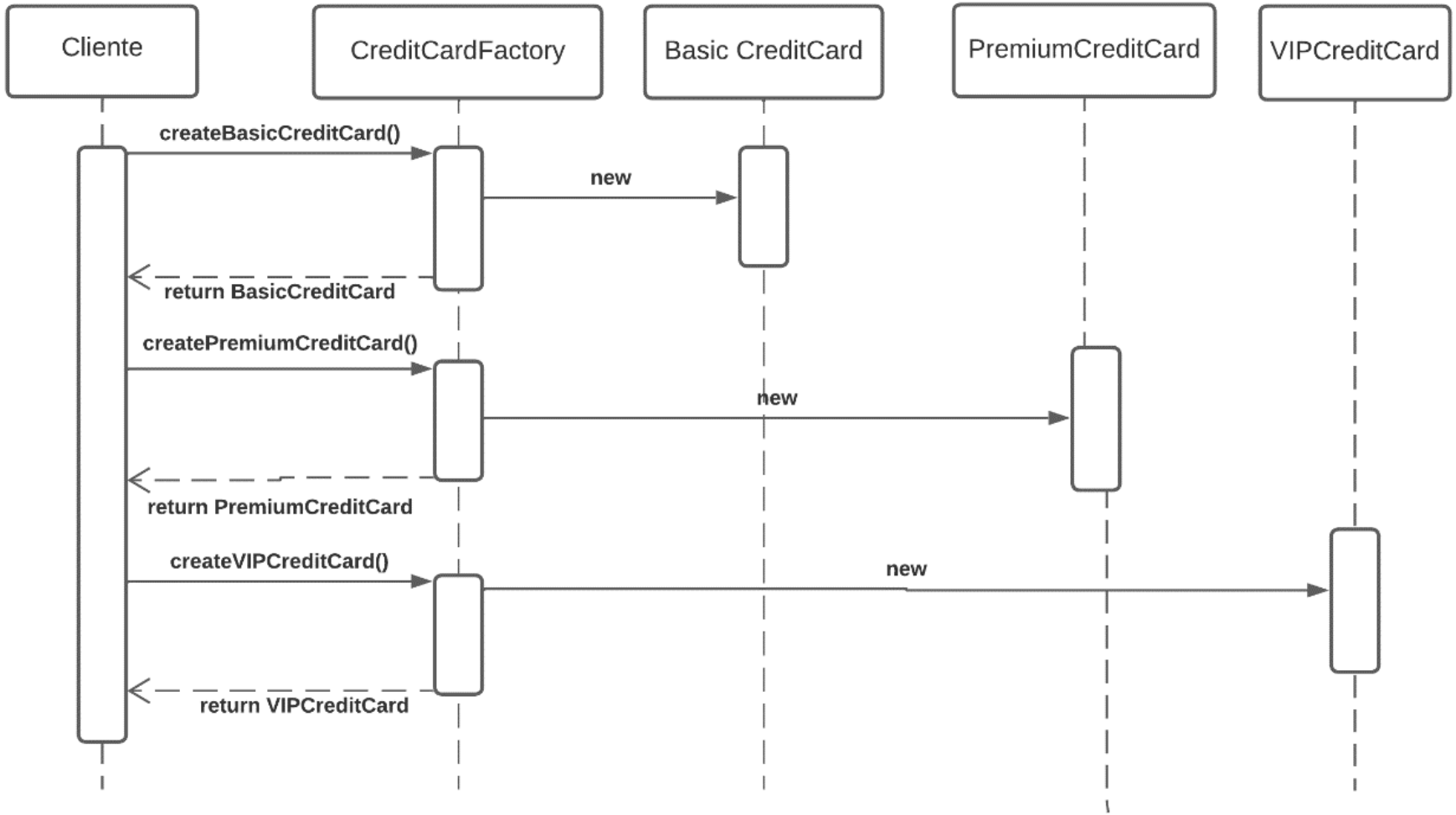


Diagrama Secuencia:



Facade

Diagrama de Clase:

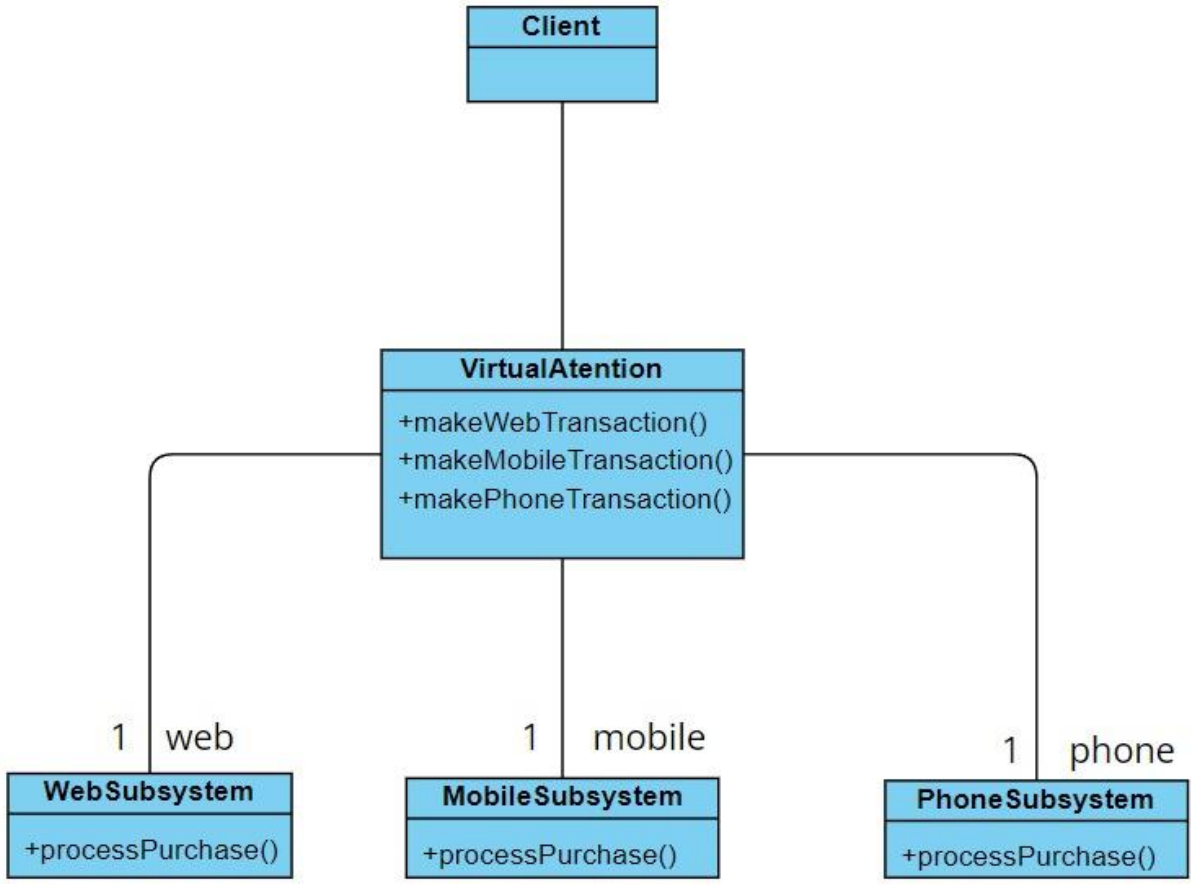
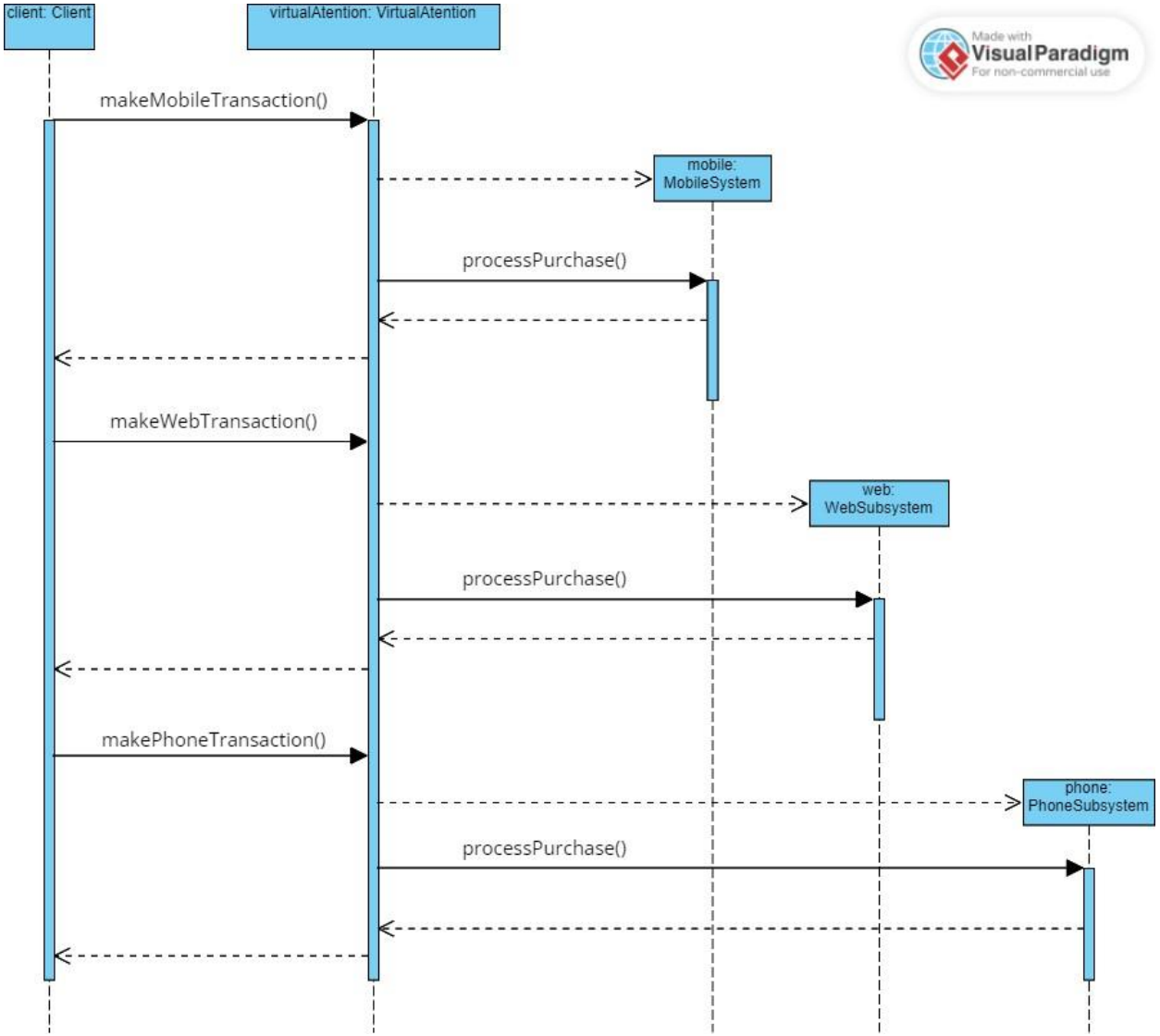


Diagrama Secuencia:



Decorator

Diagrama de Clase:

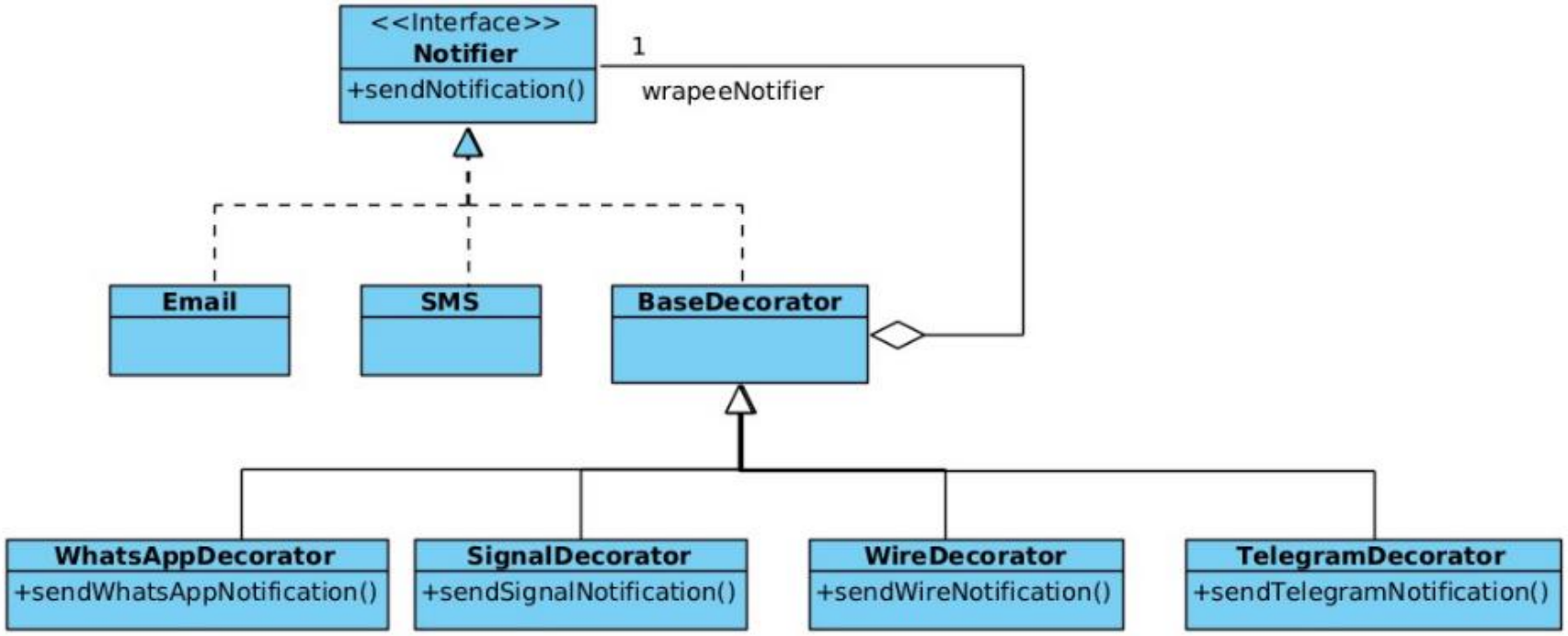
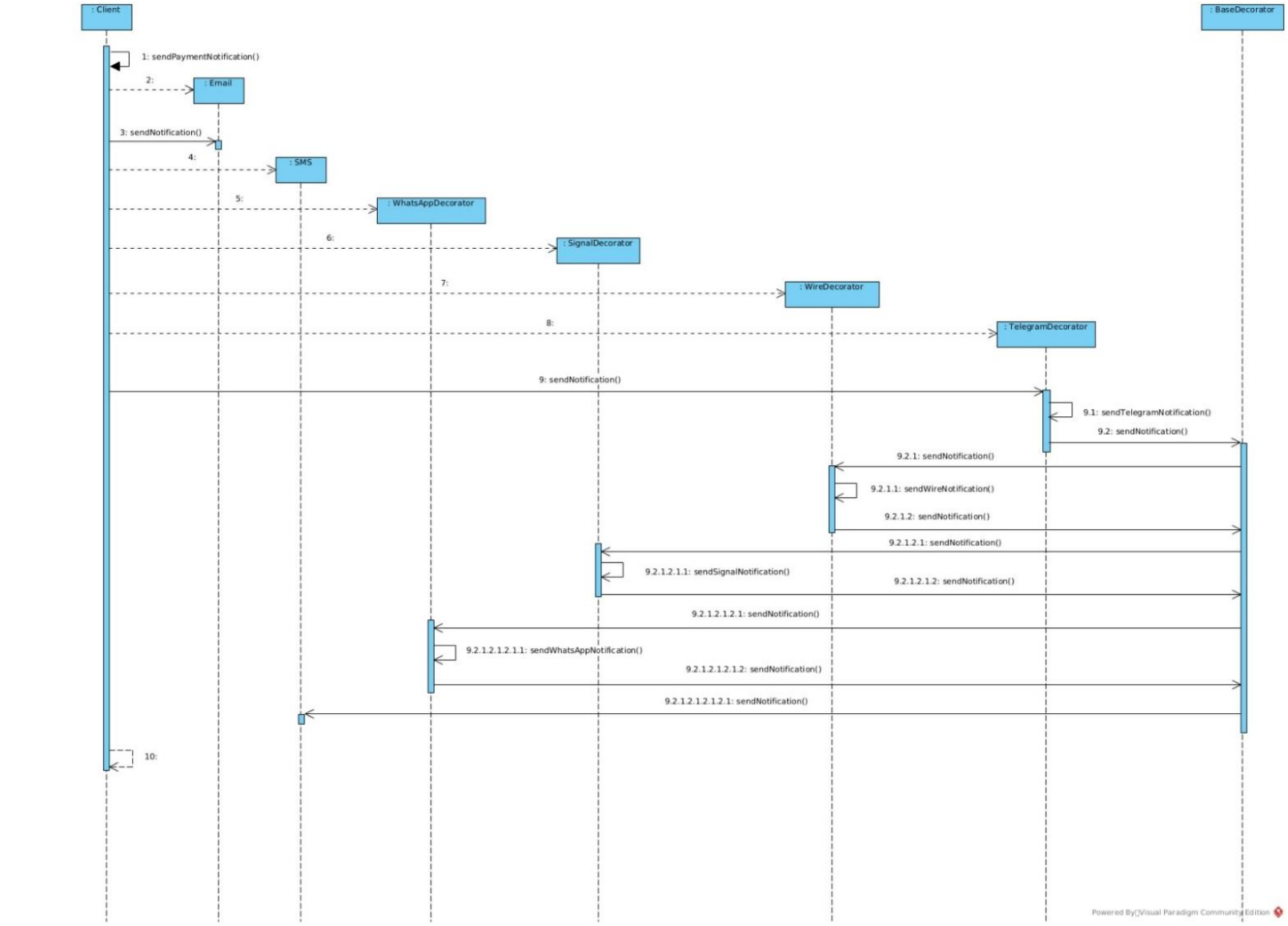


Diagrama Secuencia:


: Client
: BaseDecorator
1: sendPaymentNotification()
2:
: Email
3: sendNotification()
4:
: SMS
5:
: WhatsAppDecorator
6:
: SignalDecorator
7:
: WireDecorator
8:
: TelegramDecorator
9: sendNotification()
9.1: sendTelegramNotification()
9.2: sendNotification()
9.2.1: sendNotification()
9.2.1.1: sendWireNotification()
9.2.1.2: sendNotification()
9.2.1.2.1: sendNotification()
9.2.1.2.1.1: sendSignalNotification()
9.2.1.2.1.2: sendNotification()
9.2.1.2.1.2.1: sendNotification()
9.2.1.2.1.2.1.1: sendWhatsAppNotification()
9.2.1.2.1.2.1.2: sendNotification()
9.2.1.2.1.2.1.2.1: sendNotification()
10:
Powered By Visual Paradigm Community Edition

Sección C: Apéndice

Factory Method

```
public interface CreditCard {
    double getAnnualCost();
    double getCreditLimit();
}

public class BasicCreditCard implements CreditCard {
    private double annualCost;
    private double creditLimit;

    public BasicCreditCard(double annualCost, double creditLimit) {
        this.annualCost = annualCost;
        this.creditLimit = creditLimit;
    }

    @Override
    public double getAnnualCost() {
        return annualCost;
    }

    @Override
    public double getCreditLimit() {
        return creditLimit;
    }
}
```

```java
public class BasicCreditCardFactory extends CreditCardFactory {
    @Override
    public CreditCard createCreditCard() {
        return new BasicCreditCard(annualCost: 20.0, creditLimit:1000.0);
    }
}
```

```java
 */
public interface CreditCard {
    double getAnnualCost();
    double getCreditLimit();
}
```

```java
 */
public abstract class CreditCardFactory {
    public abstract CreditCard createCreditCard();
}
```

```java
 */
public class PremiumCreditCard implements CreditCard {
    private double annualCost;
    private double creditLimit;

    public PremiumCreditCard(double annualCost, double creditLimit) {
        this.annualCost = annualCost;
        this.creditLimit = creditLimit;
    }

    @Override
    public double getAnnualCost() {
        return annualCost;
    }

    @Override
    public double getCreditLimit() {
        return creditLimit;
    }
}
```

```java
public class PremiumCreditCardFactory extends CreditCardFactory {
    @Override
    public CreditCard createCreditCard() {
        return new PremiumCreditCard(annualCost: 50.0, creditLimit:5000.0);
    }
}
```

```java
 */
public class VipCreditCard implements CreditCard {
    private double annualCost;
    private double creditLimit;

    public VipCreditCard(double annualCost, double creditLimit) {
        this.annualCost = annualCost;
        this.creditLimit = creditLimit;
    }

    @Override
    public double getAnnualCost() {
        return annualCost;
    }

    @Override
    public double getCreditLimit() {
        return creditLimit;
    }
}
```

```java
 */
public class VipCreditCardFactory extends CreditCardFactory {
    @Override
    public CreditCard createCreditCard() {
        return new VipCreditCard(annualCost: 100.0, creditLimit:10000.0);
    }
}
```

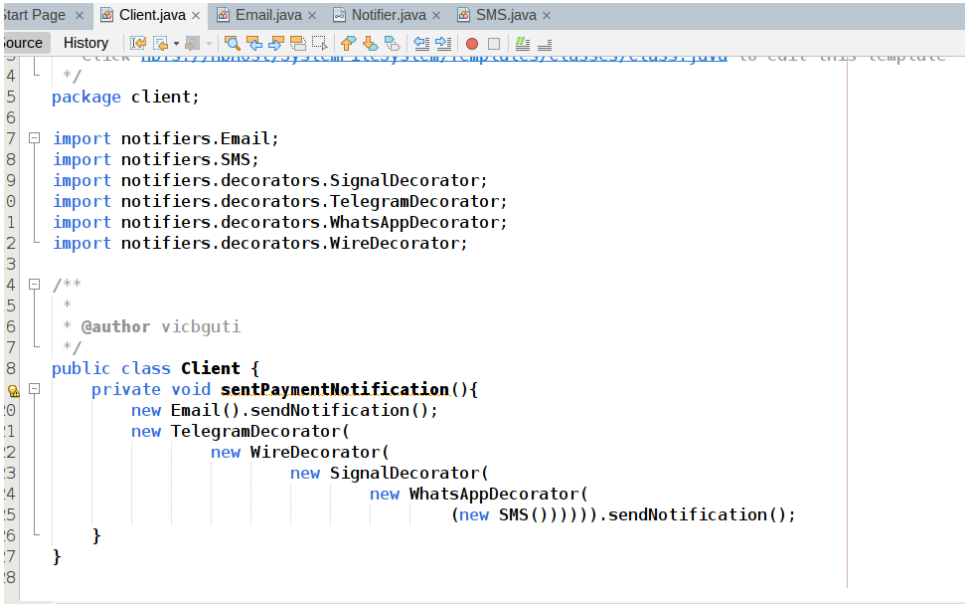
Facade

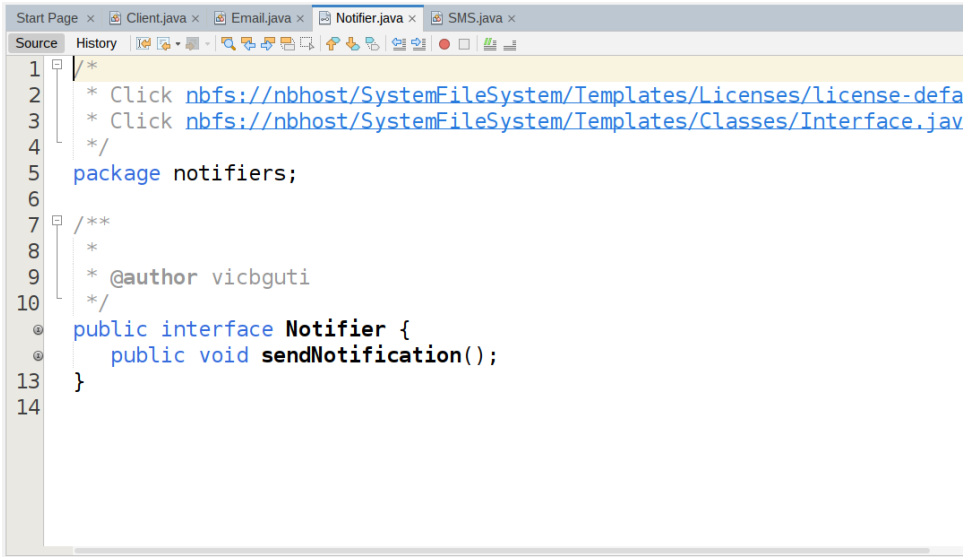
```java
public class Client {

    public static void main(String[] args) {

        VirtualAtention virtualAtention = new VirtualAtention();

        //Este cliente usa el servicio móvil
        System.out.println("Cliente 1");
        virtualAtention.makeMobileTransaction();

        //Otro cliente usa el servicio web
        System.out.println("Cliente 2");
        virtualAtention.makeWebTransaction();

        //Otro cliente usa el servicio telefónico
        System.out.println("Cliente 3");
        virtualAtention.makePhoneTransaction();
    }

}
```

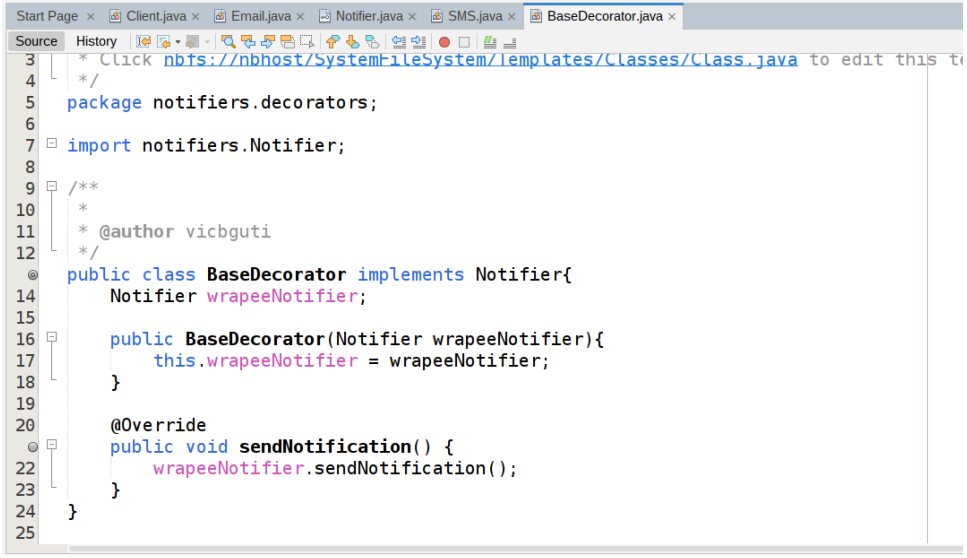
```java
public class VirtualAtention {
    private WebSubsystem web;
    private MobileSubsystem mobile;
    private PhoneSubsystem phone;

    public void makeWebTransaction(){
        web = new WebSubsystem();
        web.processPurchase();
    }

    public void makeMobileTransaction(){
        mobile = new MobileSubsystem();
        mobile.processPurchase();
    }

    public void makePhoneTransaction(){
        phone = new PhoneSubsystem();
        phone.processPurchase();
    }

}

public class WebSubsystem {
    public void processPurchase(){
        System.out.println("Procesando pago para web");
    }
}
```

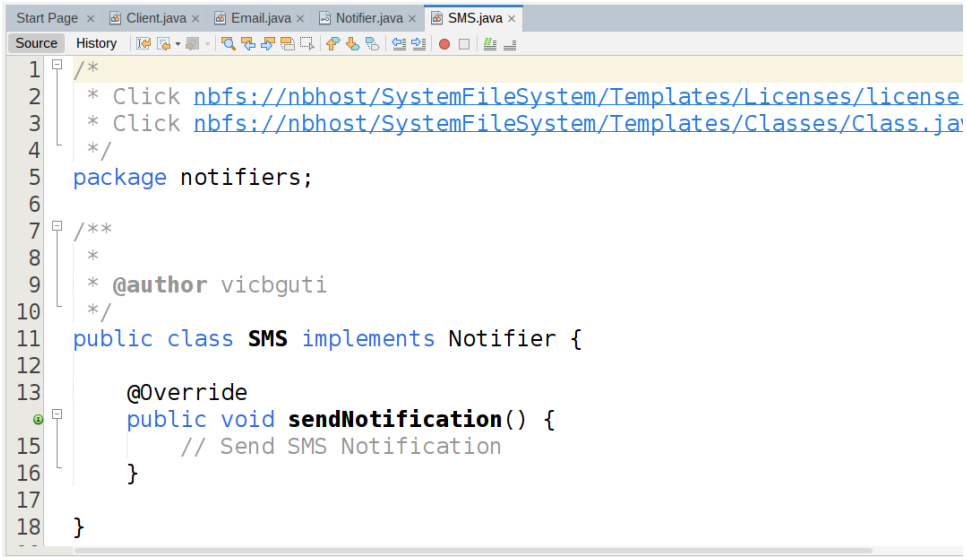
```java
public class MobileSubsystem {
    public void processPurchase(){
        System.out.println("Procesando pago para móvil");
    }
}

public class PhoneSubsystem {
    public void processPurchase(){
        System.out.println("Procesando pago para telefóno");
    }
}
```

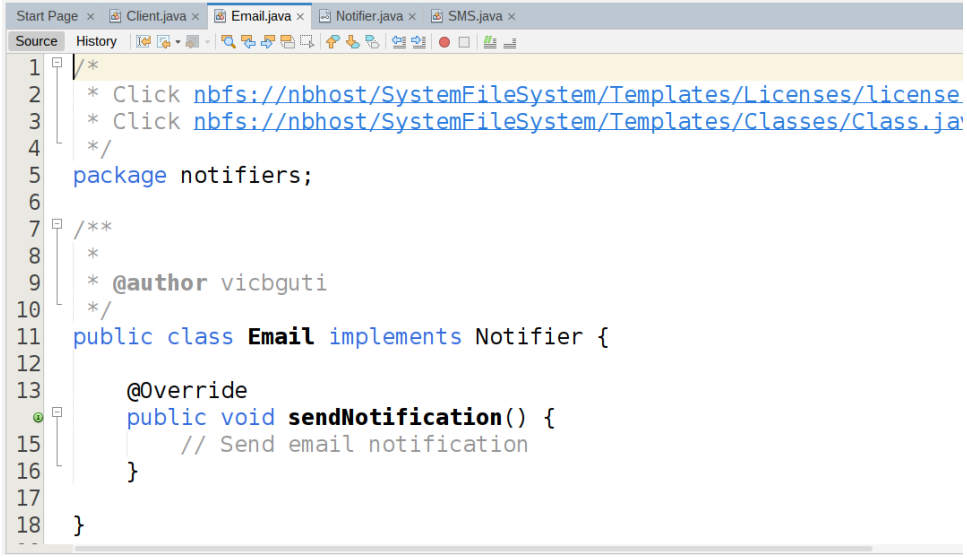
Wrapper

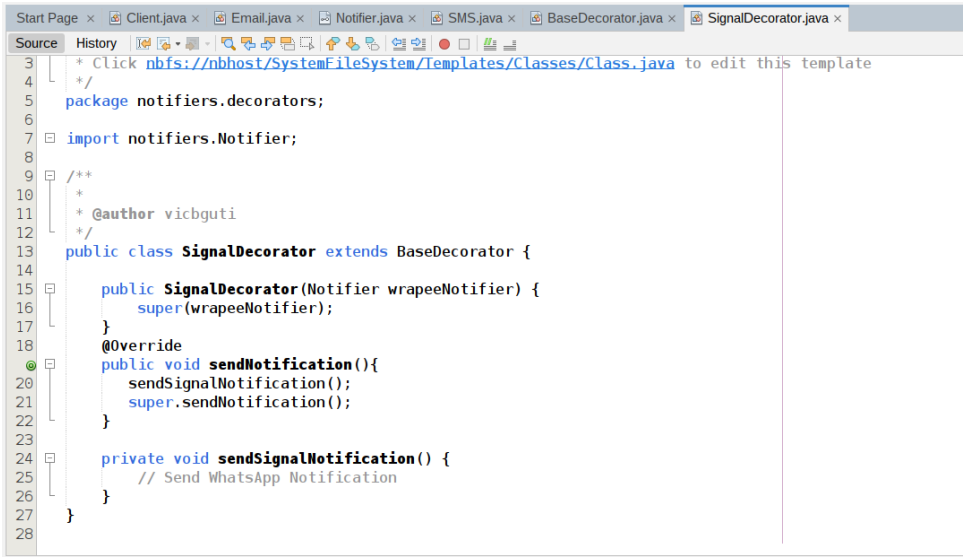

Start Page
Client.java
Email.java
Notifier.java
SMS.java
Source
History
*/
package client;
import notifiers.Email;
import notifiers.SMS;
import notifiers.decorators.SignalDecorator;
import notifiers.decorators.TelegramDecorator;
import notifiers.decorators.WhatsAppDecorator;
import notifiers.decorators.WireDecorator;
/**
*
* @author vicbguti
*/
public class Client {
private void sentPaymentNotification(){
new Email().sendNotification();
new TelegramDecorator(
new WireDecorator(
new SignalDecorator(
new WhatsAppDecorator(
(new SMS()))))).sendNotification();
}
}

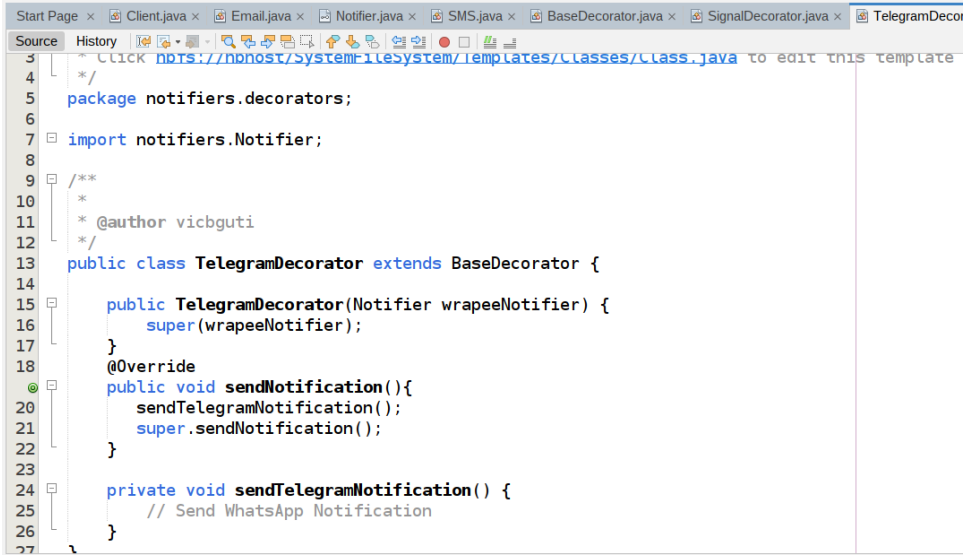

Start Page
Client.java
Email.java
Notifier.java
SMS.java
Source
History
/*
* Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-defa
* Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Interface.jav
*/
package notifiers;
/**
*
* @author vicbguti
*/
public interface Notifier {
public void sendNotification();
}


Start Page
Client.java
Email.java
Notifier.java
SMS.java
BaseDecorator.java
Source
History
* Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this te
*/
package notifiers.decorators;
import notifiers.Notifier;
/**
*
* @author vicbguti
*/
public class BaseDecorator implements Notifier{
Notifier wrapeeNotifier;
public BaseDecorator(Notifier wrapeeNotifier){
this.wrapeeNotifier = wrapeeNotifier;
}
@Override
public void sendNotification() {
wrapeeNotifier.sendNotification();
}
}

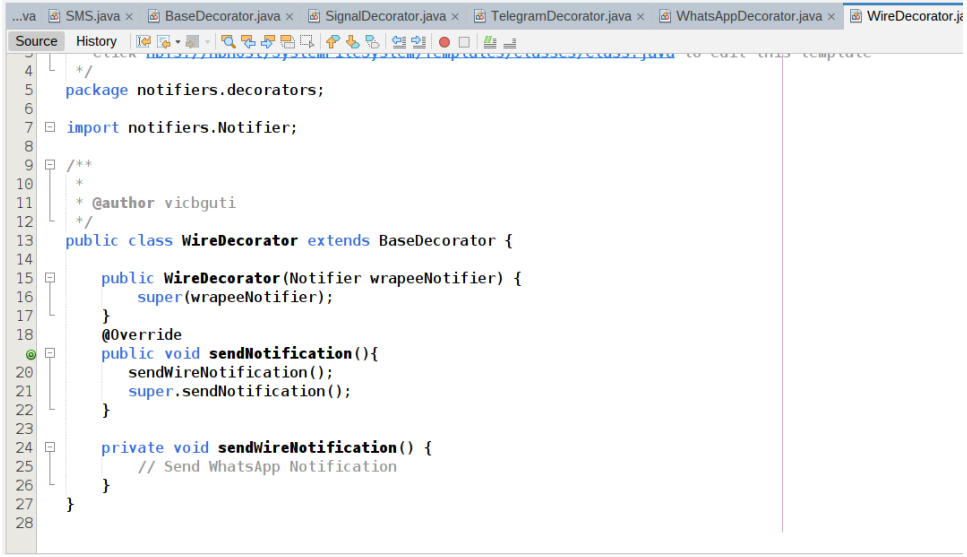

Start Page
Client.java
Email.java
Notifier.java
SMS.java
Source
History
/*
* Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license
* Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.jav
*/
package notifiers;
/**
*
* @author vicbguti
*/
public class SMS implements Notifier {
@Override
public void sendNotification() {
// Send SMS Notification
}
}


Start Page
Client.java
Email.java
Notifier.java
SMS.java
Source
History
/*
* Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license
* Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.jav
*/
package notifiers;
/**
*
* @author vicbguti
*/
public class Email implements Notifier {
@Override
public void sendNotification() {
// Send email notification
}
}


Start Page
Client.java
Email.java
Notifier.java
SMS.java
BaseDecorator.java
SignalDecorator.java
Source
History
* Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
*/
package notifiers.decorators;
import notifiers.Notifier;
/**
*
* @author vicbguti
*/
public class SignalDecorator extends BaseDecorator {
public SignalDecorator(Notifier wrapeeNotifier) {
super(wrapeeNotifier);
}
@Override
public void sendNotification(){
sendSignalNotification();
super.sendNotification();
}
private void sendSignalNotification() {
// Send WhatsApp Notification
}
}


Start Page
Client.java
Email.java
Notifier.java
SMS.java
BaseDecorator.java
SignalDecorator.java
TelegramDecor
Source
History
* Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
*/
package notifiers.decorators;
import notifiers.Notifier;
/**
*
* @author vicbguti
*/
public class TelegramDecorator extends BaseDecorator {
public TelegramDecorator(Notifier wrapeeNotifier) {
super(wrapeeNotifier);
}
@Override
public void sendNotification(){
sendTelegramNotification();
super.sendNotification();
}
private void sendTelegramNotification() {
// Send WhatsApp Notification
}


...va
Email.java
Notifier.java
SMS.java
BaseDecorator.java
SignalDecorator.java
TelegramDecorator.java
WhatsAppD
Source
History
* Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
*/
package notifiers.decorators;
import notifiers.Notifier;
/**
*
* @author vicbguti
*/
public class WhatsAppDecorator extends BaseDecorator {
public WhatsAppDecorator(Notifier wrapeeNotifier) {
super(wrapeeNotifier);
}
@Override
public void sendNotification(){
sendWhatsAppNotification();
super.sendNotification();
}
private void sendWhatsAppNotification() {
// Send WhatsApp Notification
}
}


...va
SMS.java
BaseDecorator.java
SignalDecorator.java
TelegramDecorator.java
WhatsAppDecorator.java
WireDecorator.ja
Source
History
*/
package notifiers.decorators;
import notifiers.Notifier;
/**
*
* @author vicbguti
*/
public class WireDecorator extends BaseDecorator {
public WireDecorator(Notifier wrapeeNotifier) {
super(wrapeeNotifier);
}
@Override
public void sendNotification(){
sendWireNotification();
super.sendNotification();
}
private void sendWireNotification() {
// Send WhatsApp Notification
}
}