UNIDAD TEMÁTICA 5 – Patrones de diseño– Trabajo de Aplicación 4

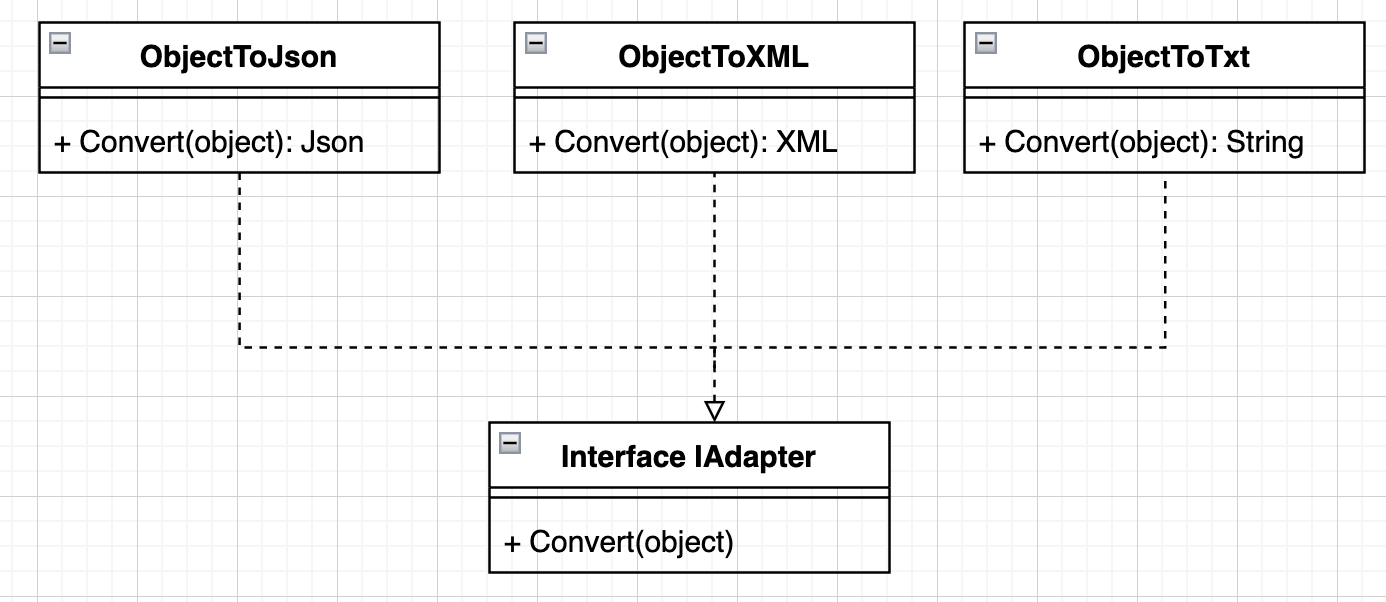
Para cada uno de los siguientes ejercicios, en equipo:

1. Determine que patrón puede resolver el problema de una forma más eficiente.
2. Agregue las clases, interfaces, métodos que considere necesarios para remediar la situación.

EJERCICIO 1



El patrón para resolver sería Adapter ya que se requiere una interfaz para cada convertidor.

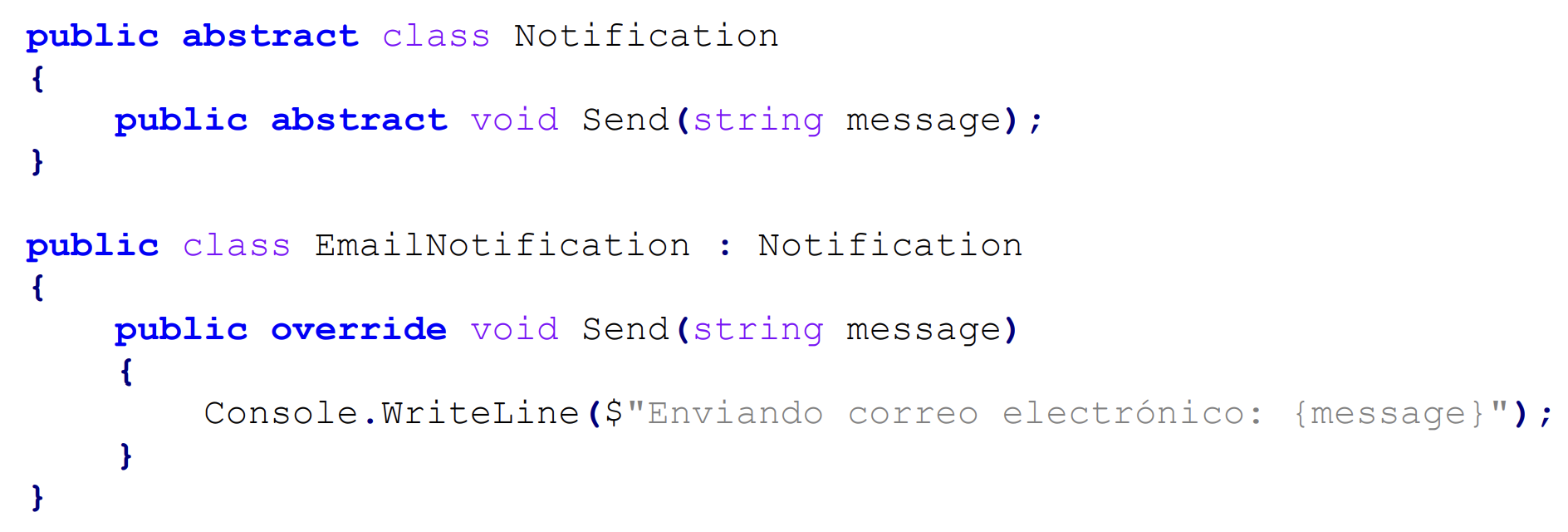


EJERCICIO 2



Dado que el objetivo es hacer que la tienda en línea sea compatible con "SafePay" sin cambiar el código existente de las clases que procesan pagos con "QuickPay" el patrón que soluciona es Adapter.

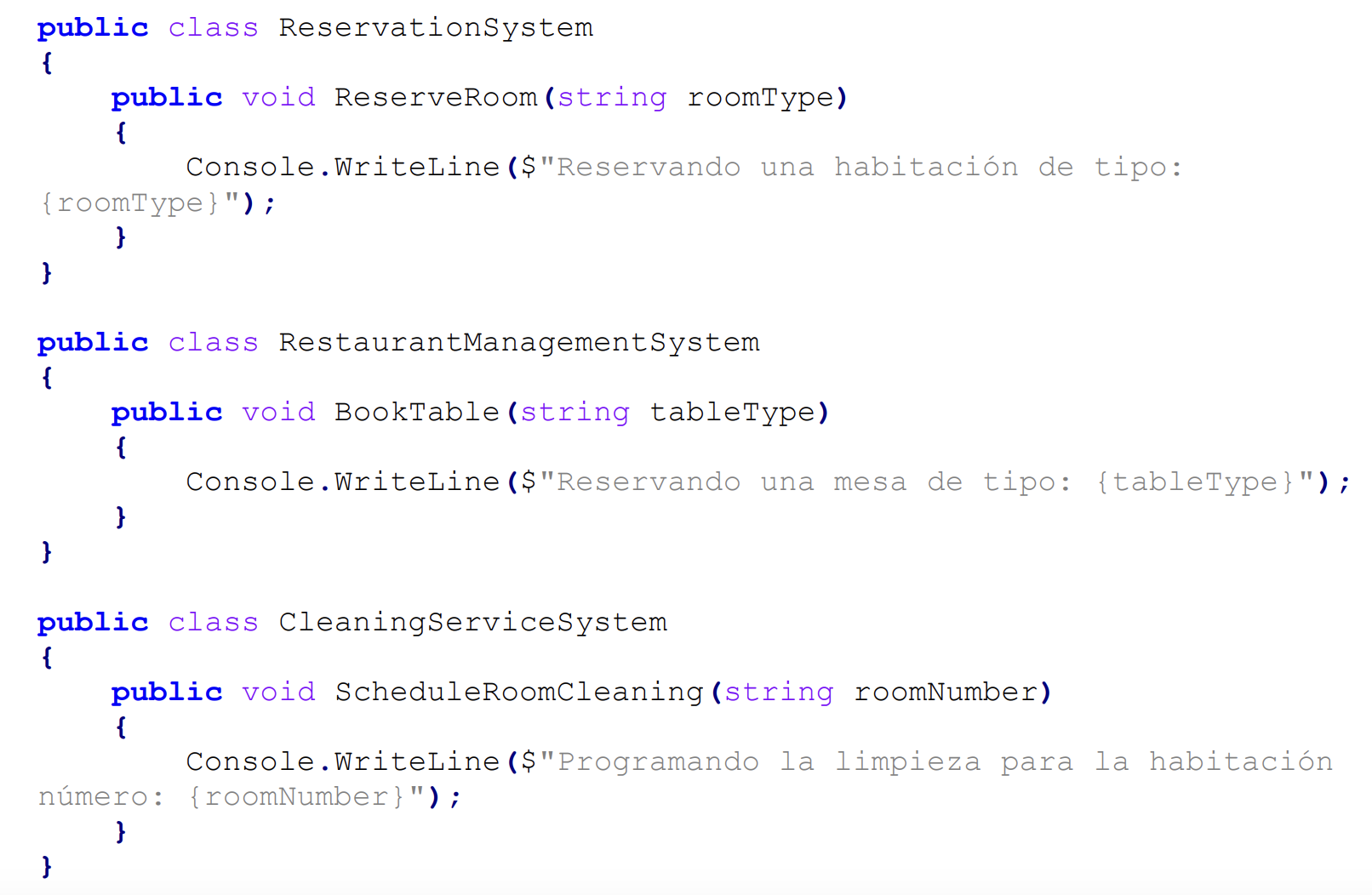
| public interface IQuickPay  {  bool MakePayment(double amount, string currency);  }  public class QuickPayService : IQuickPay  {  public bool MakePayment(double amount, string currency)  {  Console.WriteLine($"Pagado {amount} {currency} usando QuickPay.");  return true; // Simular éxito  }  }  public class OnlineStore  {  private IQuickPay \_paymentService;  public OnlineStore(IQuickPay paymentService)  {  \_paymentService = paymentService;  }  public void Checkout(double amount, string currency)  {  if (\_paymentService.MakePayment(amount, currency))  {  Console.WriteLine("Pago exitoso!");  }  else  {  Console.WriteLine("El pago ha fallado.");  }  }  }  public class SafePayService  {  public void Transact(string fromAccount, string toAccount, string currencyType, double amount)  {  Console.WriteLine($"Transfiriendo {amount} {currencyType} de {fromAccount} a {toAccount} usando SafePay.");  }  }  public class SafePayAdapter : IQuickPay  {  private SafePayService \_safePayService;  public SafePayAdapter(SafePayService safePayService)  {  \_safePayService = safePayService;  }  public bool MakePayment(double amount, string currency)  {  \_safePayService.Transact("Mi cuenta", "Cuenta de la tienda", currency, amount);  return true;  }  } |
| --- |

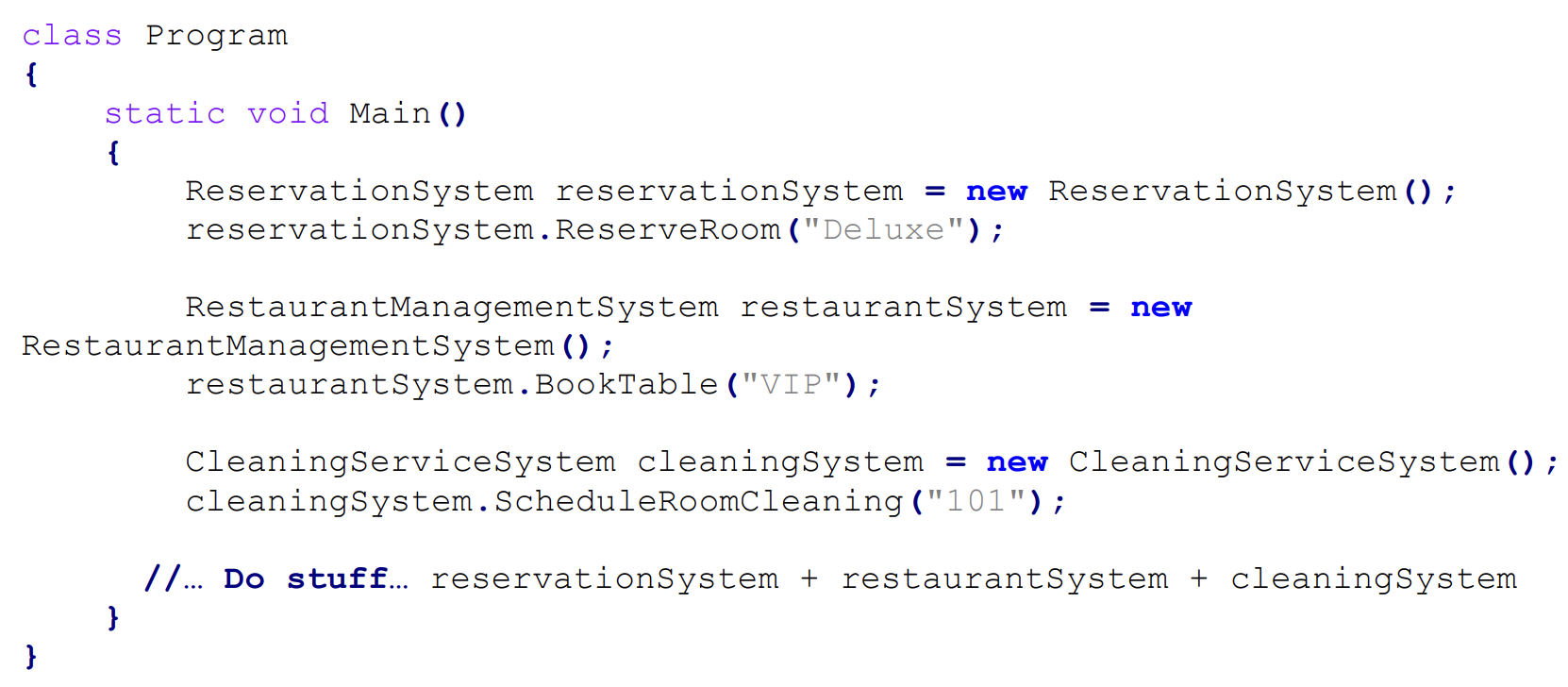
EJERCICIO 3

Es Decorator

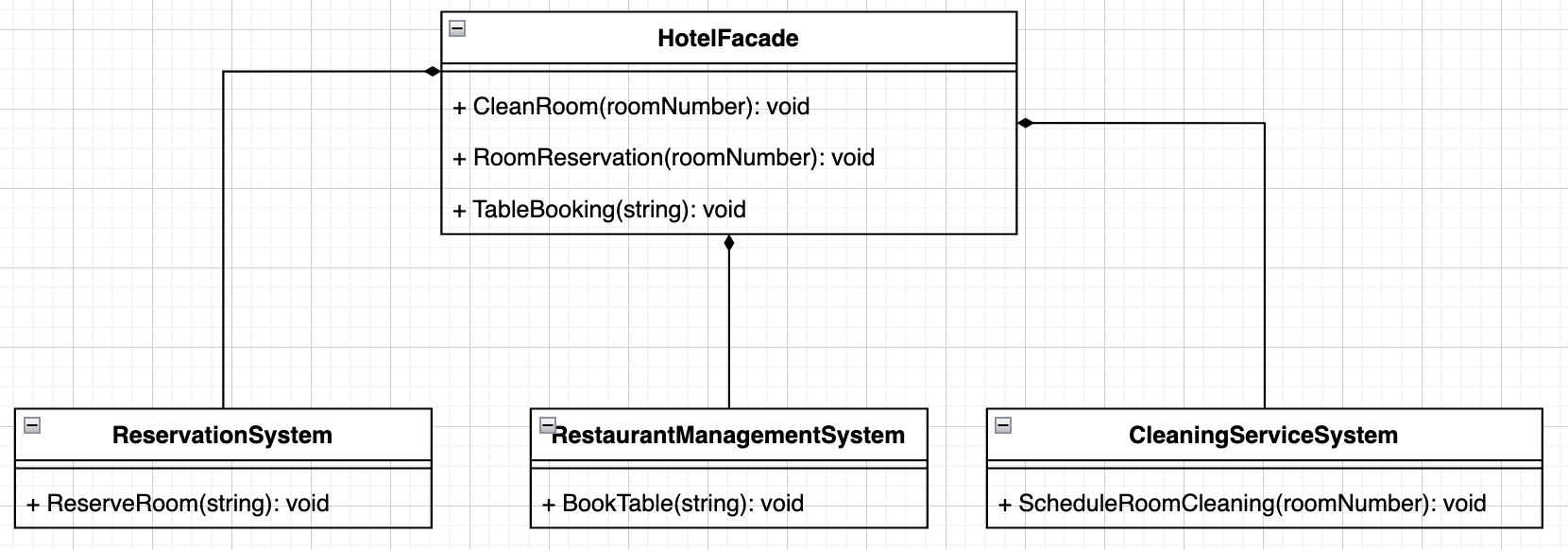
| using System;  public interface INotification  {  void Send(string message);  }  public class NotificationDecorator : INotification  {  protected INotification \_notification;  public NotificationDecorator(INotification notification)  {  \_notification = notification;  }  public virtual void Send(string message)  {  Console.WriteLine($"Enviando: {message}");  }  }  public class EmailNotification : NotificationDecorator  {  public EmailNotification(INotification notification) : base(notification) { }  public override void Send(string message)  {  Console.WriteLine($"Enviando EmailNotification: {message}");  if (\_notification != null)  {  \_notification.Send(message);  }  }  }  public class SMSNotification : NotificationDecorator  {  public SMSNotification(INotification notification) : base(notification) { }  public override void Send(string message)  {  Console.WriteLine($"Enviando SMS: {message}");  if (\_notification != null)  {  \_notification.Send(message);  }  }  }  public class TwitterNotification : NotificationDecorator  {  public TwitterNotification(INotification notification) : base(notification) { }  public override void Send(string message)  {  Console.WriteLine($"Enviando Mensaje Directo en Twitter: {message}");  if (\_notification != null)  {  \_notification.Send(message);  }  }  }  public class FacebookNotification : NotificationDecorator  {  public FacebookNotification(INotification notification) : base(notification) { }  public override void Send(string message)  {  Console.WriteLine($"Enviando Mensaje en Facebook: {message}");  if (\_notification != null)  {  \_notification.Send(message);  }  }  }  class Program  {  static void Main(string[] args)  {  INotification notification = new EmailNotification(null);  notification = new SMSNotification(notification);  notification = new TwitterNotification(notification);  notification = new FacebookNotification(notification);  notification.Send("¡Hola mundo!");  }  } |
| --- |

EJERCICIO 4



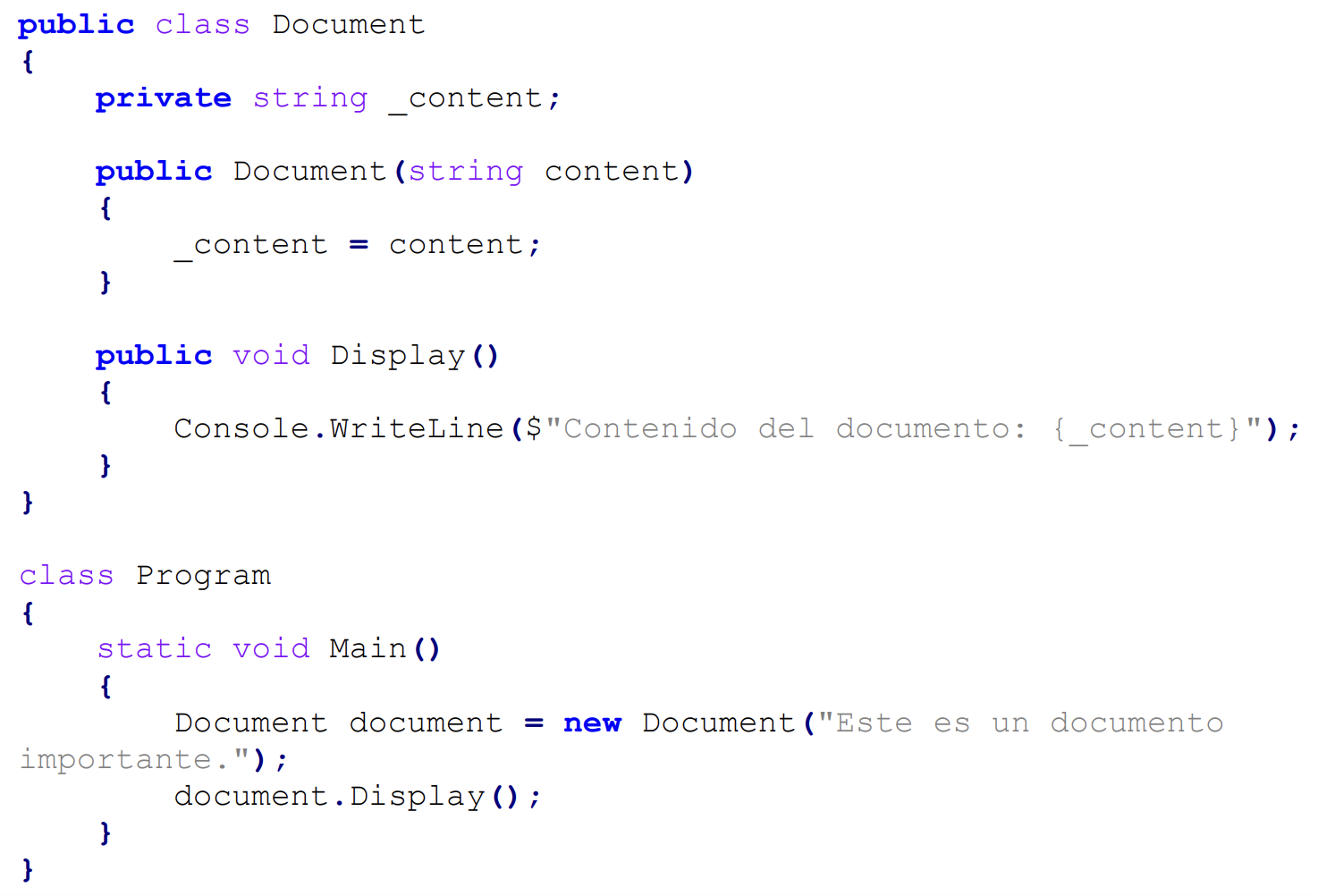


Esto es tremendo facade, porque se tienen muchos subsistemas y conviene hacer una clase facade que centralice esos subsistemas para tener mayor control.

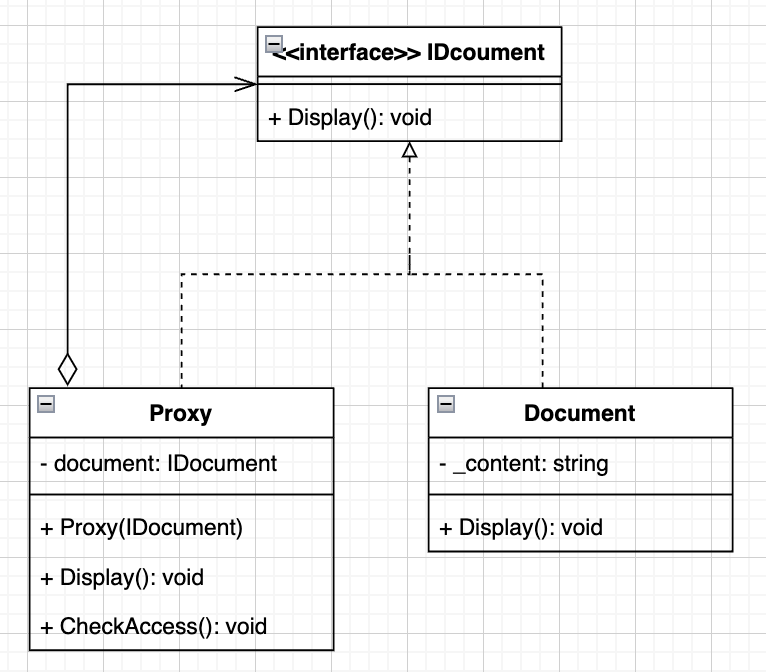


| public class HotelFacade  {  private ReservationSystem \_roomBooking;  private CleaningServiceSystem \_roomCleaning;  private RoomRestaurantManagementSystem \_roomService;  public HotelFacade()  {  \_roomBooking = new ReservationSystem();  \_roomCleaning = new CleaningServiceSystem();  \_roomService = new RoomRestaurantManagementSystem();  }  public void RoomReservation(string roomType)  {  \_roomBooking.ReserveRoom();  }  public void CleanRoom(string roomNumber)  {  \_roomCleaning.ScheduleRoomCleaning(roomNumber);  }  public void TableBooking(string tableType)  {  \_roomService.BookTable(tableType);  }  }  public class ReservationSystem  {  public void ReserveRoom()  {  Console.WriteLine("Reserving room...");  // Logic to reserve the room  }  }  public class RoomRestaurantManagementSystem  {  public void BookTable(string tableType)  {  Console.WriteLine("Booking table...");  // Logic to book the table  }  }  public class CleaningServiceSystem  {  public void ScheduleRoomCleaning(string roomNumber)  {  Console.WriteLine($"Scheduling cleaning for room {roomNumber}...");  // Logic to schedule room cleaning  }  }  public class Program  {  public static void Main()  {  var hotelFacade = new HotelFacade();  string roomType = "Deluxe";  string roomNumber = "101";  string tableType = "Window";  hotelFacade.RoomReservation(roomType);  hotelFacade.CleanRoom(roomNumber);  hotelFacade.TableBooking(tableType);  }  } |
| --- |

EJERCICIO 5

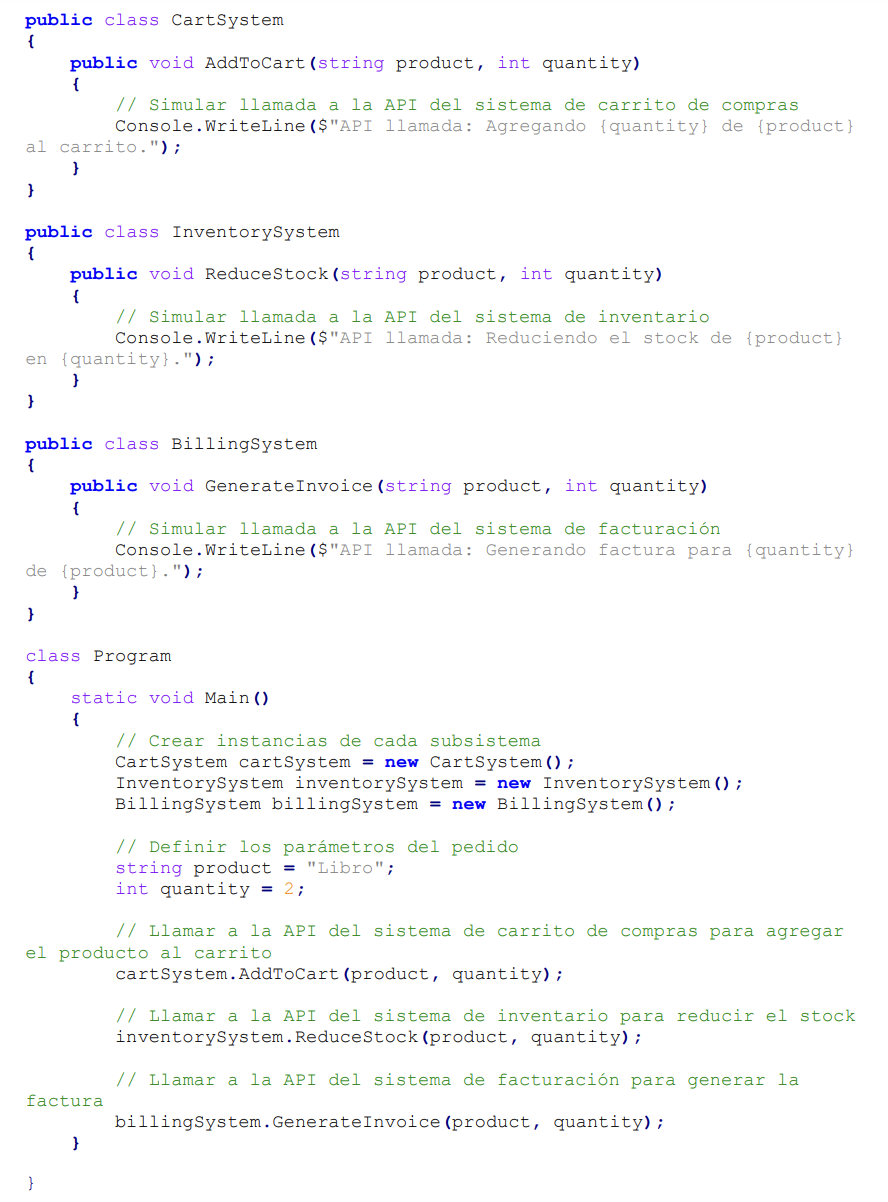


El patrón que resuelve el problema es Proxy debido a que este provee un control de acceso al objeto original, en este caso se desea un mecanismo de control de acceso a los documentos almacenados para evitar que todos puedan acceder a él.



| public interface IDocument {  void Display();  }  public class Proxy implements IDocument {  private IDocument document;  public Proxy(IDocument document) {  this.document = document;  }  public boolean CheckAccess() {  // Some real checks should be here.  return true;  }  @Override  public void Display() {  if (CheckAccess()) {  document.Display();  }  }  }  public class Document implements IDocument {  private String content;  public Document(String content) {  this.content = content;  }  @Override  public void Display() {  System.out.println("Displaying " + content);  }  } |
| --- |

EJERCICIO 6



Esto es tremendo facade, porque se tienen muchos subsistemas y conviene hacer una clase facade que centralice esos subsistemas para tener mayor control.

EJERCICIO 7

El patrón que resuelve el problema es Proxy debido a que este provee un control de acceso al objeto original, en este caso se desea un mecanismo de control de acceso a los documentos almacenados para evitar que todos puedan acceder a él.

EJERCICIO 8

Bridge o builder