**Trabajo Practico: Proceso de Desarrollo de Software**

* **Integrante:**
  + Fernandez Nicolas
* **Docente:** 
  + Martinez Ana Carolina
* **Universidad:**
  + UADE
* **Asignatura:** 
  + Proceso de Desarrollo de Software
* **Curso:** 
  + Turno Mañana – viernes - Primer Cuatrimestre 2025
* Fecha de entrega:
  + 16/05/25

## **Introducción**

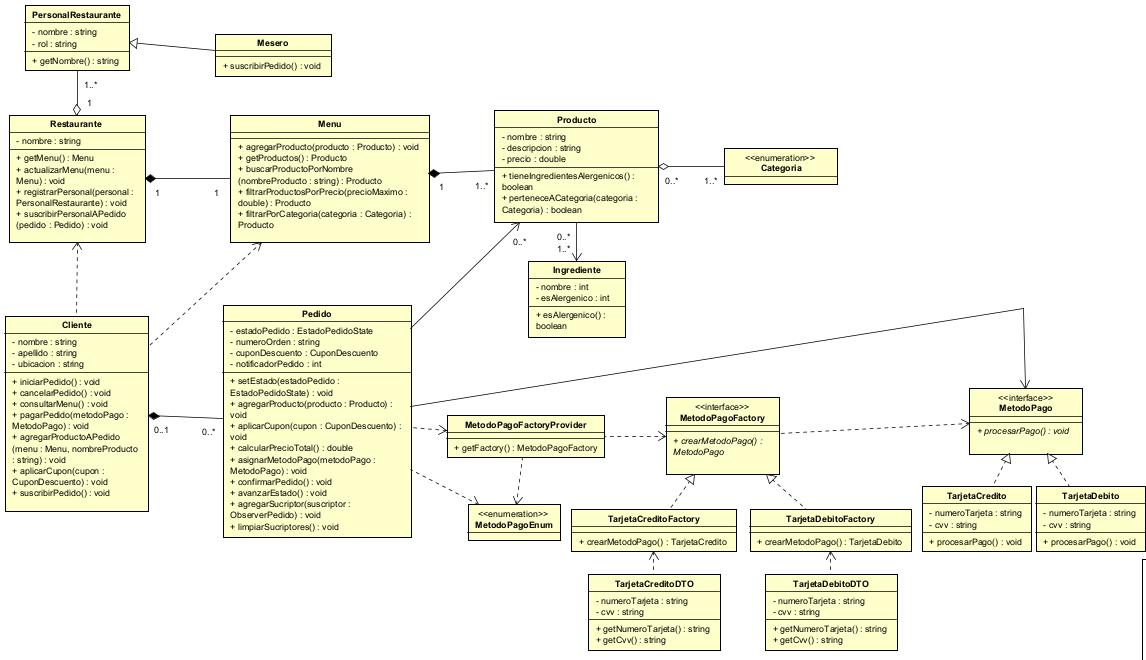
El presente informe documenta las decisiones de diseño tomadas en la construcción de una aplicación para la gestión de pedidos en un restaurante, siguiendo principios de programación orientada a objetos (POO), buenas prácticas de diseño de software y aplicación de patrones de diseño. A continuación se justifica la estructura del modelo, la elección de patrones y la responsabilidad de cada componente.

## **Modelado del Dominio**

A continuación, se detallan las entidades y sus relaciones:

* **Restaurante y Menu:** el Restaurante tiene una relación de composición con Menu, ya que el menú está totalmente contenido dentro del restaurante. Si el restaurante deja de existir, su menú también. Además, el restaurante tiene una relación de agregación con PersonalRestaurante (ej. meseros), ya que estos pueden existir independientemente del restaurante.
* **Menu, Producto y Categoría:** el Menu contiene múltiples Producto, en una relación de agregación. Cada Producto puede pertenecer a una o más Categoría, modelado como una relación de muchos a muchos. Esto permite que un producto esté categorizado como "Entrada" y también como "Vegetariano", por ejemplo.
* **Producto e Ingrediente:** Producto tiene una composición con Ingrediente, ya que los ingredientes están completamente definidos dentro del contexto del producto. Cada ingrediente puede indicar si es alérgeno o no.
* **Cliente y Pedido:** un Cliente puede tener múltiples Pedido, y mantiene una relación de agregación. El cliente puede iniciar, cancelar o pagar su pedido activo. Además, mediante su método suscribirPedido(), recibira notificaciones automáticamente ante los cambios de estado del pedido.
* **Pedido y CuponDescuento:** el Pedido mantiene una referencia a un objeto de tipo CuponDescuento. Por defecto, este será un objeto CuponVacio (patrón Objeto Nulo). También puede usar una instancia de PorcentajeDescuento, que aplica descuentos sobre el total si es válido.
* **Pedido y MetodoPago:** el Pedido se le puede asignar un método de pago dinámicamente, el cual será generado por una fábrica especializada (MetodoPagoFactory) y su proveedor (MetodoPagoFactoryProvider) basado en un DTO correspondiente.
* **Pedido y EstadoPedidoState:** Pedido implementa el patrón State con estados como EnEsperaState, EnPreparacionState, ListoParaEntregarState y EntregadoState. Cada clase de estado define su comportamiento y transiciones válidas.
* **Observer y NotificadorPedido:** el Pedido tiene una relación de agregación con NotificadorPedido, quien mantiene una lista de ObserverPedido. Estos pueden ser ObserverCliente, ObserverMesero u ObserverFacturaCliente. Cada uno responde de forma diferente según el cambio de estado: el cliente es notificado por la app, el mesero si el pedido está listo, y el observer de factura genera y simula el envío por correo.

Los diagramas ilustran cómo se distribuyen responsabilidades de forma clara y coherente, usando agregación, composición y herencia donde corresponde, alineados con una arquitectura orientada a objetos.



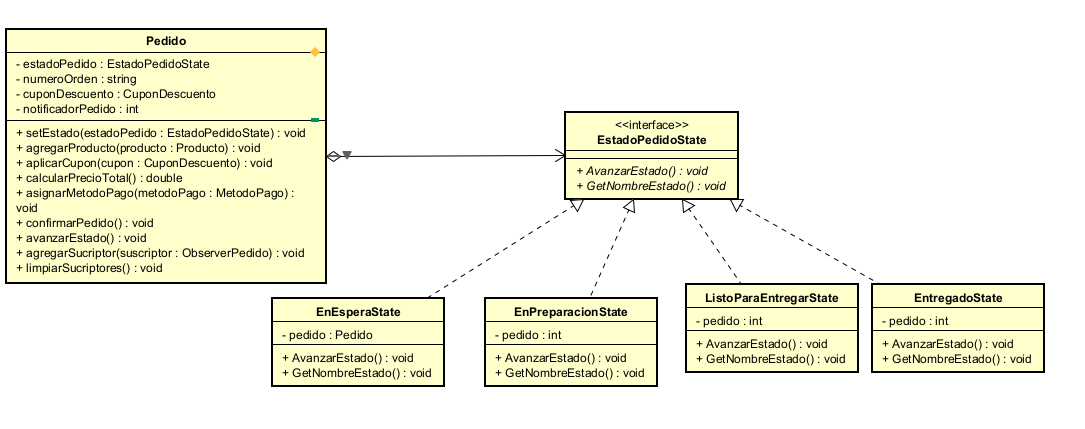
## **Patrones de Diseño Utilizados**

### **1. Patrón State**

Este patrón permite modelar comportamientos que dependen del estado interno del objeto. Al encapsular cada estado como una clase, se reduce el acoplamiento y se incrementa la cohesión. También se mejora la extensibilidad, ya que agregar un nuevo estado implica simplemente crear una nueva clase.

El ciclo de vida de un pedido implica una secuencia de estados mutuamente excluyentes: "En espera", "En preparación", "Listo para entregar" y "Entregado". Usar condicionales para manejar estas transiciones haría que la lógica de Pedido fuera difícil de mantener.

Cada estado es una clase que implementa la interfaz EstadoPedidoState, la cual define métodos como AvanzarEstadoPedido() y GetNombreEstado(). La clase Pedido mantiene una referencia al estado actual y delega en él el comportamiento. Cada estado conoce el Pedido al que pertenece y decide cuál será su siguiente estado.



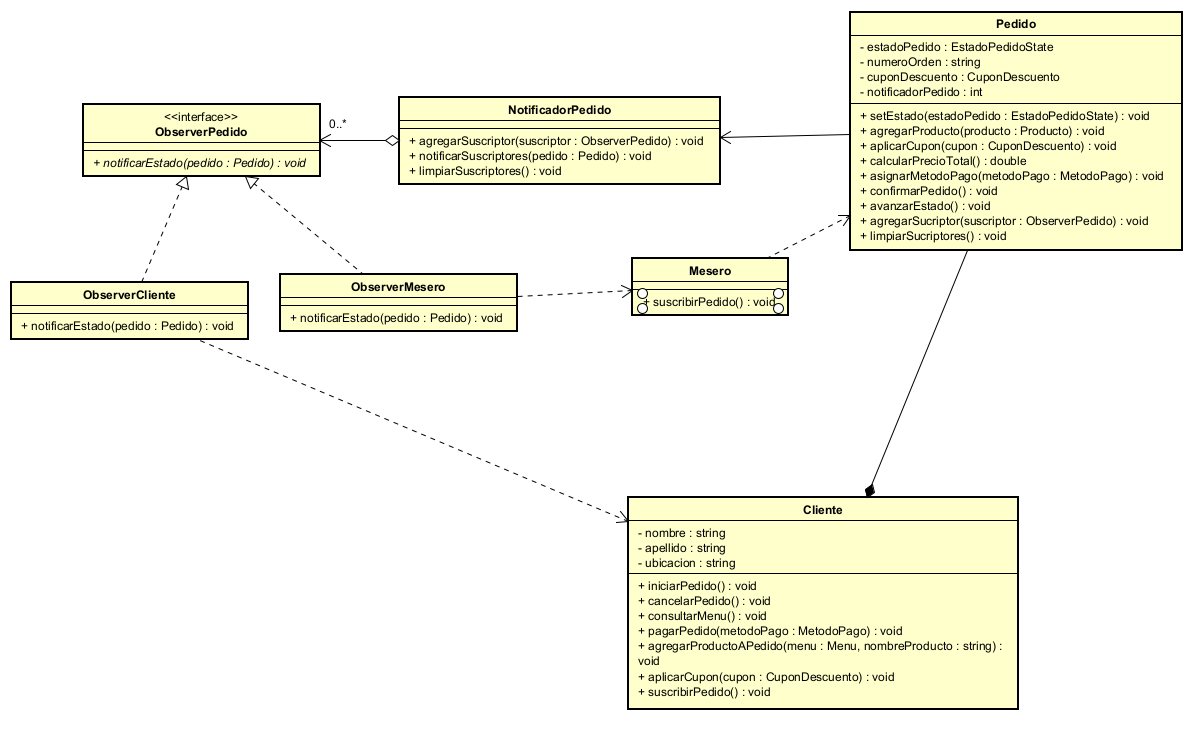
### **2. Patrón Observer**

El patrón Observer promueve el principio de inversión de dependencias. El objeto observado (Pedido) no conoce los detalles de sus observadores, solo sabe que puede notificarles. Esto permite una gran flexibilidad, ya que se pueden agregar nuevos observadores sin alterar el código del emisor.

El sistema debe notificar automáticamente a diferentes actores cuando un pedido cambia de estado. Por ejemplo, los clientes deben saber cuando su pedido está listo o entregado, y los meseros deben ser alertados cuando un pedido está listo para entregar.

**Implementación:**

* La clase Pedido contiene un objeto NotificadorPedido que mantiene una lista de ObserverPedido.
* Las clases ObserverCliente y ObserverMesero implementan la lógica de reacción ante los cambios de estado.
* Cada Observer tiene una referencia a la clase que quiere ser notificada
* La suscripción se realiza automáticamente al confirmar un pedido, conectando al cliente y al personal necesario.



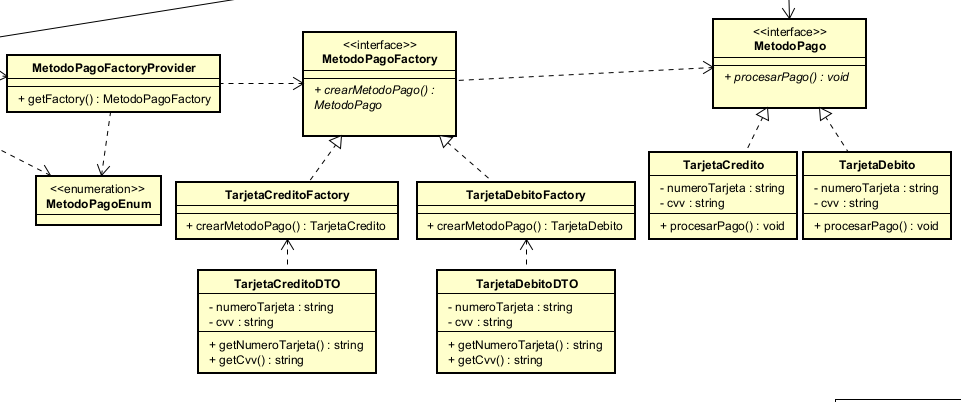
### **3. Patrón Factory Method**

El patrón Factory Method permite delegar la creación de objetos a subclases, promoviendo el principio de abierto/cerrado. En nuestro caso, la creación de objetos de pago queda encapsulada en fábricas especializadas, mejorando la modularidad del sistema.

La creación de métodos de pago como tarjetas de crédito o débito puede requerir validaciones distintas. Queremos abstraer esa lógica de creación para mantener un código limpio y desacoplado.

**Implementación:**

* Se define una interfaz genérica MetodoPagoFactory<T> con un método crearPago(T datos).
* Cada forma de pago tiene su propia implementación de la fábrica: TarjetaCreditoFactory, TarjetaDebitoFactory, etc.
* El cliente selecciona el tipo de pago, y el sistema usa la fábrica correspondiente.
* Los datos necesarios para crear los métodos de pago se encapsulan en DTOs como TarjetaCreditoDTO, que contienen campos específicos (ej. número de tarjeta, CVV).
* Se implementa una clase MetodoPagoFactoryProvider que actúa como **punto de entrada único** para obtener la fábrica apropiada en función del tipo de pago (TipoMetodoPago).



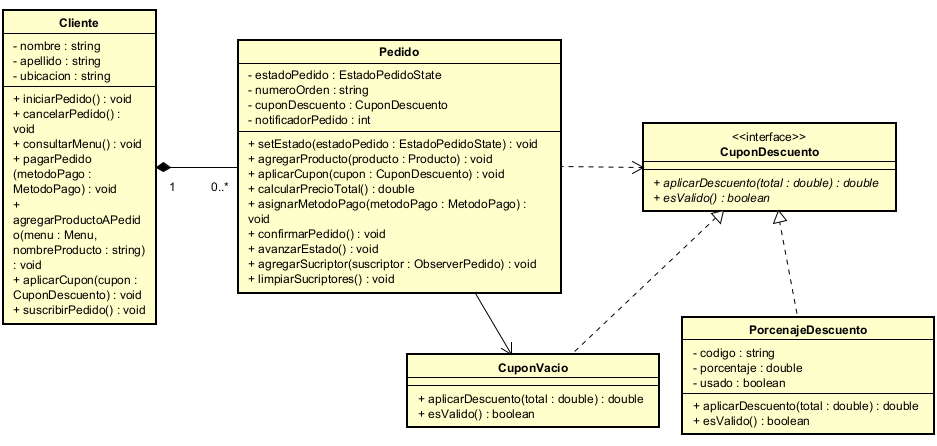
### **4. Patrón Objeto Nulo (Null Object Pattern)**

Este patrón permite evitar verificaciones explícitas de null, ya que se proporciona un objeto válido con comportamiento "nulo" por defecto. Mejora la legibilidad del código y evita errores de puntero nulo. Además, mantiene la coherencia de la interfaz, ya que todos los objetos CuponDescuento son tratados de forma uniforme, independientemente de si realizan o no un descuento.

Esto evita validaciones repetitivas y condicionales innecesarios al aplicar cupones de descuento en los pedidos. Por ejemplo, si no hay cupón asignado, evitar constantemente comprobar if (cupon != null).

**Implementación:**

* Se define una interfaz CuponDescuento con los métodos aplicarDescuento(total) y esValido().
* Se crean implementaciones concretas como PorcentajeDescuento y CuponVacio.
* CuponVacio implementa la lógica nula: no modifica el precio y siempre es válido.
* Pedido siempre tiene una instancia de CuponDescuento, que por defecto es un CuponVacio.



## **Estrategia de Tests Unitarios**

Se desarrolló una batería de tests unitarios utilizando JUnit con enfoque TDD (Test-Driven Development). Los tests verifican:

* Creación, modificación y filtrado de productos en el Menu.
* Comportamiento del ciclo de vida del Pedido con el patrón State.
* Que ObserverCliente y ObserverMesero reciban notificaciones según el estado correcto.
* Que ObserverFacturaCliente genere y simule el envío de factura al llegar al estado "Entregado".
* Que el cliente pueda iniciar, pagar, y cancelar pedidos correctamente.
* Que el sistema maneje correctamente los cupones y el cálculo del total con descuentos.
* Que las fábricas de métodos de pago creen instancias válidas y manejen errores en datos.
* Creación y modificación en Restaurante
* Asignación de Mesero en Restaurante

Los tests cubren tanto caminos felices como casos de error. Se valida el comportamiento esperado de cada clase en aislamiento y también la interacción entre componentes clave.

## **Conclusión**

El diseño de la aplicación se guió por principios de responsabilidad única, bajo acoplamiento y alto encapsulamiento. Los patrones de diseño aplicados ayudaron a reducir la complejidad, mejorar la mantenibilidad y facilitar la extensión del sistema. Este enfoque permite que el sistema crezca en funcionalidad sin comprometer su estructura o claridad. La estrategia de pruebas unitarias asegura que cada módulo cumpla su contrato y que el sistema se mantenga robusto frente a cambios.