UNIVERSIDAD AUTÓNOMA DE OCCIDENTE

FACULTAD DE INGENIERÍA

Entrega parcial II

Juan David Muñoz Olave – 2226531

Juan Pablo Zuluaga Diaz – 2226319

Ingeniería de software II

Periodo 2025-01

Docente

Jonathan López Londoño

Santiago de Cali
11 de abril de 2025

Diagrama de la arquitectura usada:

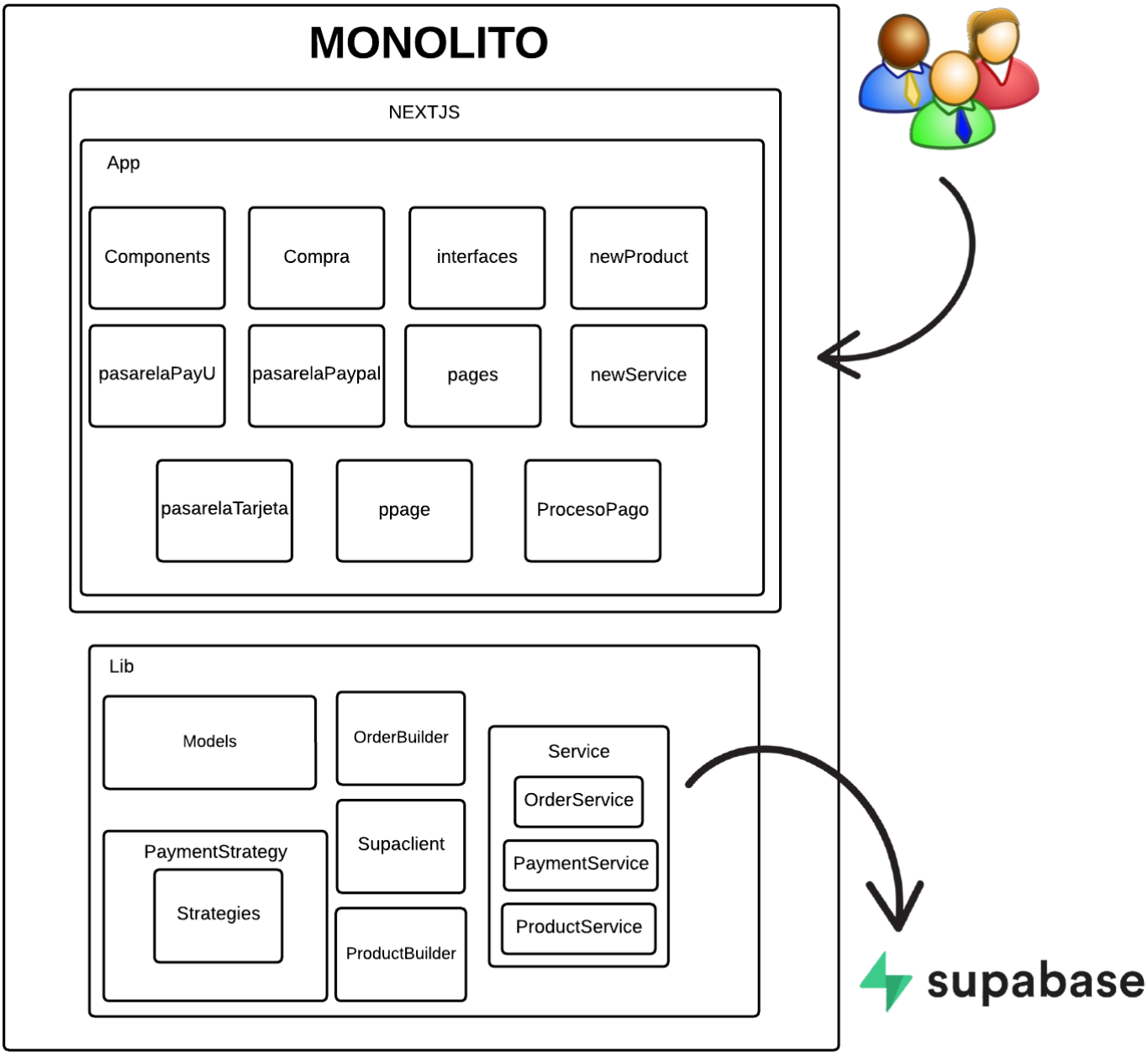
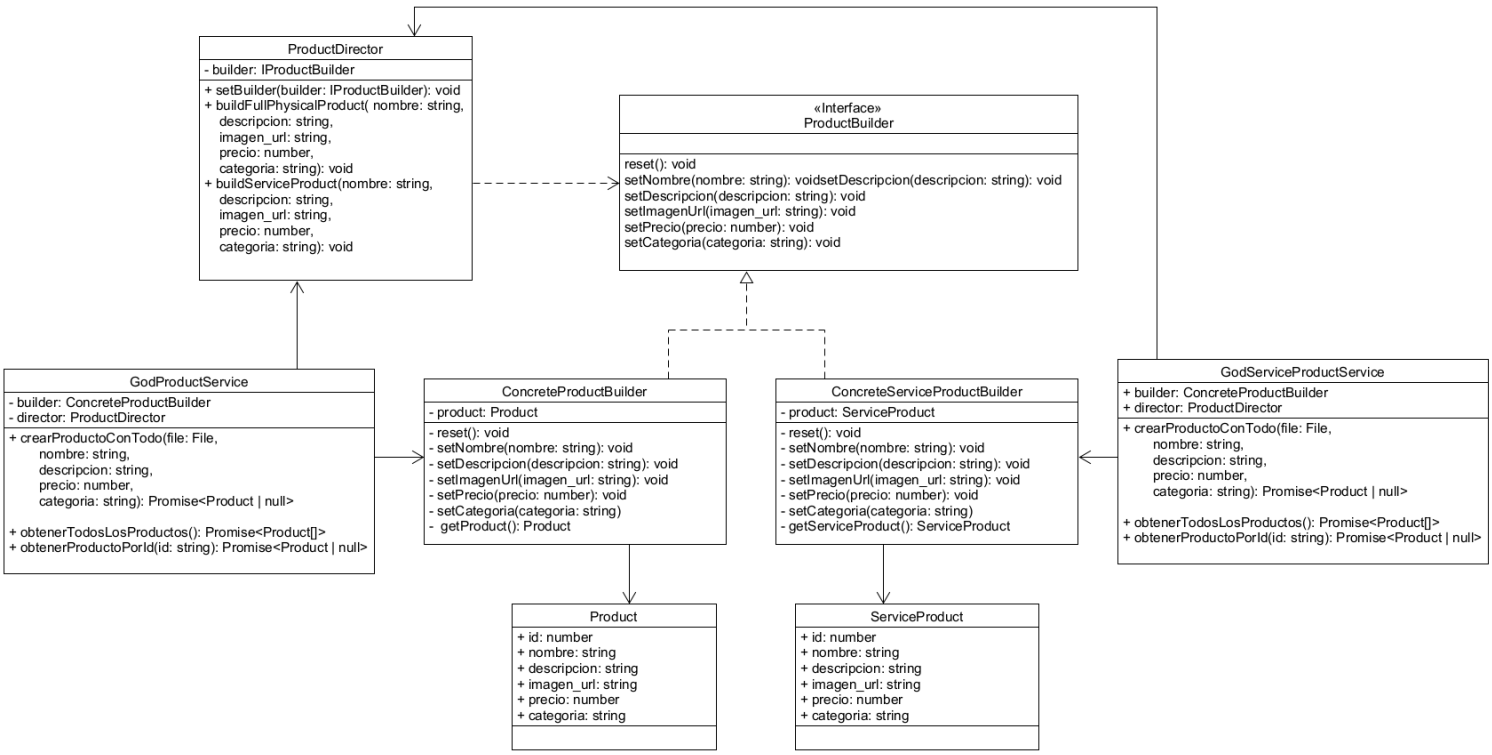
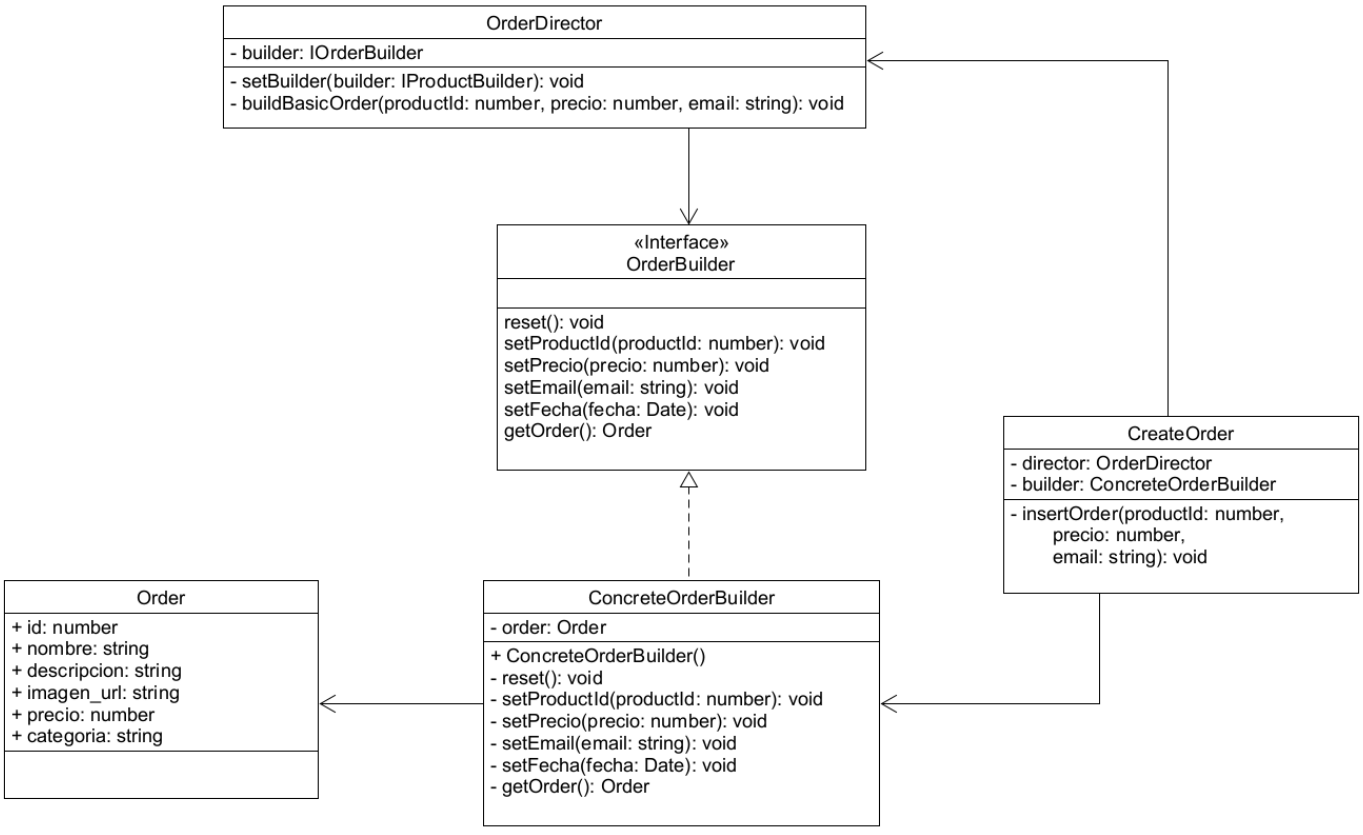
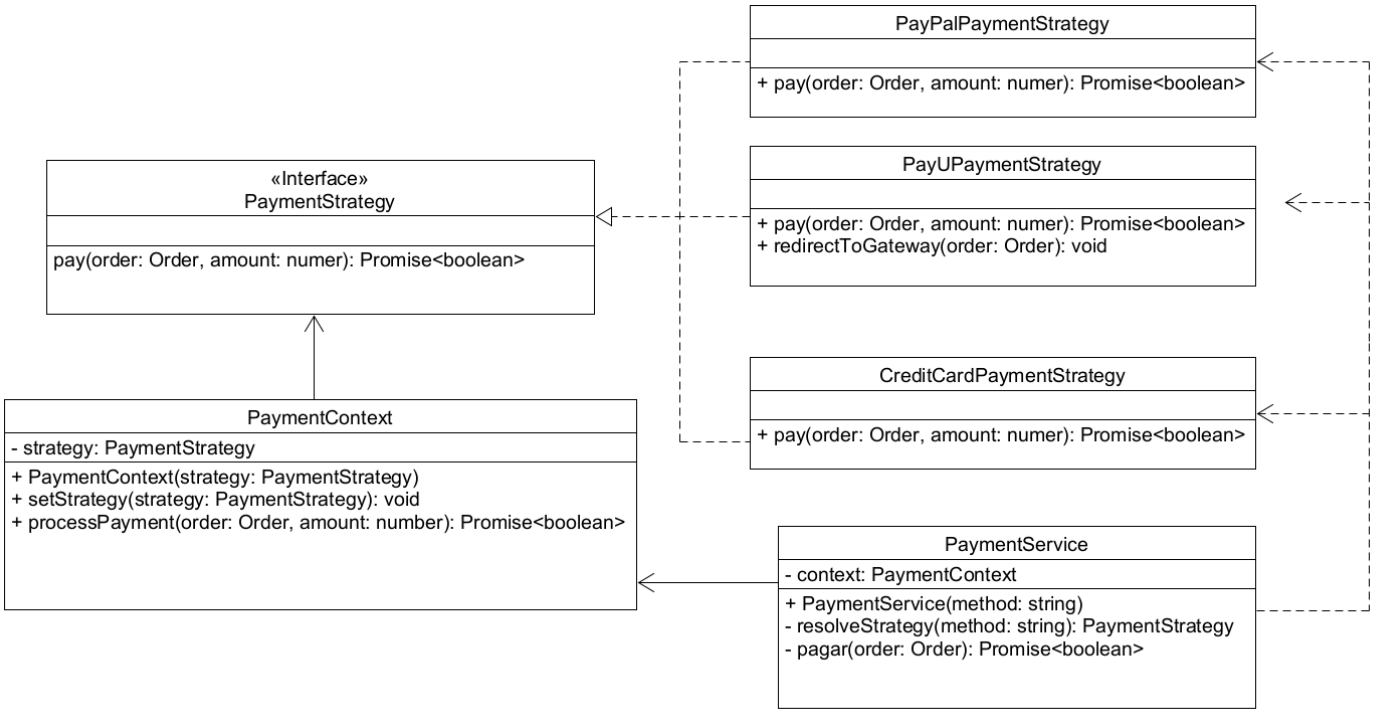


Diagrama de clases:







Patrones de diseño escogidos:

1. Patrón creacional: Builder

Este patrón de diseño permite construir objetos paso a paso. Permite producir distintos tipos y representaciones de un objeto empleando el mismo código de construcción[1].

Teniendo en cuenta que el aplicativo desarrollado para el ejercicio parcial es una simulación de un comercio electrónico (e-commerce), el patrón de diseño Builder nos da la posibilidad de construir los productos y órdenes por medio de una serie de pasos. Es aplicable al contexto debido a que los objetos de servicio se pueden construir con distintas configuraciones, además, se separa su creación paso a paso. En el aplicativo existen, tanto productos físicos como productos de servicio. Los pasos que se requieren para crearlos son los mismos entre ellos, pero con diferencias clave en su implementación. “El patrón Builder se puede aplicar cuando la construcción de varias representaciones de un producto requiera de pasos similares que sólo varían en los detalles” (Refactoring.Guru). Dentro del ejercicio propuesto se puede observar cómo los tipos de productos concuerdan en sus atributos y métodos; sin embargo, la implementación de su ConcreteBuilder difiere.

En suma, en el caso de las órdenes, si se requiriera más adelante, por ejemplo, agregar en el aplicativo un “recibo de la orden”, este podría ser creado por medio del Builder de la orden, ya que sus atributos y métodos serían similares y adaptables a este. Esta característica hace que el e-commerce sea escalable y ordenado.

2. **Patrón de comportamiento: Strategy**

Da la posibilidad de definir una familia de algoritmos, colocar cada uno de ellos en una clase separada y hacer sus objetos intercambiables[1].

En el aplicativo, el patrón Strategy está siendo usado para manipular la simulación de los distintos métodos de pago. Así pues, recurriendo a la aplicabilidad de este patrón, en lugar de hacer uso de condicionales y bloques de código grandes para manejar los procesos de pago, se extrajo el algoritmo que deben implementar las “estrategias”, para ubicar cada una de ellas en una clase separada y que todas implementen la misma interfaz. Dentro del contexto, es aplicable de igual manera, ya que los métodos de pago (al ser reducidos) pueden estar abiertos a extensión. El patrón Strategy permite introducir cuantos métodos de pago sean necesarios sin necesidad de cambiar el contexto. Por último, es clara la separación de responsabilidades entre las estrategias: cada una de ella se hace cargo de su lógica correspondiente, y no por otras partes de la aplicación. Esto, en esencia, permite generar código de manera limpia y que este sea mantenible.

Antipatrón:

1. **The God Object:**

Este antipatrón ocurre cuando una clase o módulo concentra muchas responsabilidades en relación con la aplicación.

Dentro del ejercicio del comercio electrónico, se implementó el objeto dios teniendo en cuenta el manejo de productos. Se creó la clase GodProductService y GodServiceProductService, con el objetivo de que en cada una de estas se encargue de: subir la imagen correspondiente a la base de datos, construir el producto haciendo uso del Builder, realizar inserciones en la base de datos y realizar

consultas a esta misma. Por la naturaleza de los antipatrones, no es una buena práctica que una clase como esta se encargue de cubrir la lógica de todo un objeto de servicio, además de hacer llamados directos a la base de datos. Sin embargo, la implementación de este aplica en el ejercicio propuesto debido a que por medio de la clase dios de cada producto es posible encargase de las funcionalidades que debe cumplir cada uno con una sola clase. El hecho de que únicamente se esté creando y consultando, tanto todos, como un producto individualmente hace posible hacer uso de este antipatrón. Por supuesto, en caso de querer escalar el aplicativo, sería necesario descomponer esas clases, aplicando el principio de responsabilidad única y dividir la lógica en múltiples clases con funciones específicas para cada una.

Bibliografía:

1. Refactoring.Guru. Catálogo de patrones diseño.
https://refactoring.guru/es/design-patterns/builder