Trabajo Final
Objetos II
2022

Alumnos:

↦ Aguado, Thomas. 14245/6
↦ Ponte Ahon, Santiago Andres. 16123/6

Enunciado:

Tomar alguno de los ejercicios del final de la guia de ejercicios de OO1 (alguno que tenga un modelo de objetos medianamente interesante): https://docs.google.com/document/d/1HZc2nokWoRIHcSvYUD-o6kPImRrBBerdhDP247VsrDg/edit#heading=h.i5rz9w99kmv9

La idea es que construyas un servicio web (API) básico, y que persistas el modelo en una BD relacional (utilizando un ORM).

Y que pienses y escribas sobre lo siguiente:

Se eligió el ejercicio 19 " **Mercado de Objetos"**

Solución:

1. ***De qué manera y en qué momento afectó tu diseño (y la forma en la que lo encaraste) el saber que ibas a persistir?***

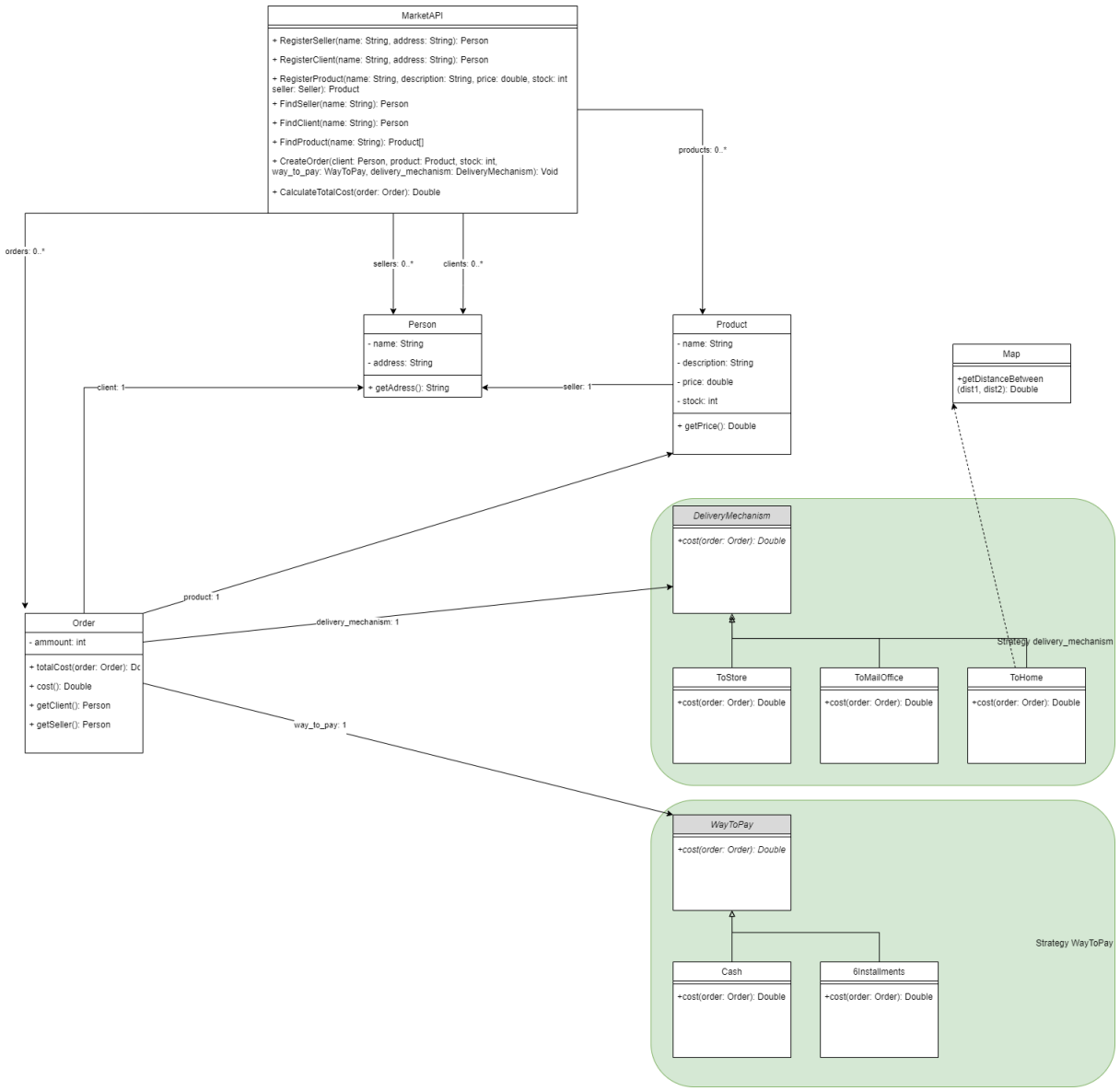

figura 1

En un principio. Se empezó diseñando sin tener en cuenta la persistencia. Para lograr un enfoque puramente pensado en objetos para luego hacer la transformación a partir de ahí (figura 1).

Una vez terminado el diseño de objetos se tuvo que decidir acerca de que persistir y que no. Por un lado se supo que toda Persona, Producto y Orden tenía que tener un lugar en la base de datos. Un problema que surgió fue a la hora de enfrentar los *strategy*, si bien persistirlos era la idea, lo único que importaba de estos era su abstracción respecto a las distintas formas de calcular sus costos. El persistir estas clases evidentemente no podría guardar su funcionalidad (por lo menos no de forma sencilla), todo lo que se podría persistir son atributos de diferenciación de estos métodos. En un inicio se trató de contemplar esta solución, se escribió la jerarquía de los *strategy* como clases java y se las mapeo haciendo uso de las estrategia de herencia SINGLE_TABLE provista por *hibernate*. Cada clase

sobrescribió el método *cost* y se hizo que los objetos de tipo Order conozcan las estrategias concretas que le corresponden. No obstante, no pareció adecuado gastar espacio de la DB en estas clases, sobre todo al considerar que lo importante de estas clases (su implementación concreta del método *cost*) seguía dependiendo del código. ¿Estaba bien hacer que la DB quede enlazada permanentemente a una implementación Java concreta? ¿Qué pasaría si alguien quisiera hacer uso de la BD mediante otro lenguaje? ¿Le sería útil la diferenciación persistida en una tabla separada sin los métodos que representaban a cada una? Llegamos a la conclusión que no valía la pena persistir estas clases, esta persistencia no valía de nada si no se tienen los métodos *cost* adecuados para cada clase java y la diferenciación se podía lograr perfectamente haciendo uso de unas meras etiquetas de tipo String en vez de enlaces a entidades separadas totalmente persistidas. Las cuales serían relativamente fijas y en el futuro es improbable que adquieran atributos para persistir (debido a la naturaleza del patrón *strategy*). Además, al eliminar la persistencia de estas entidades también se reducía la complejidad de la BD.

Por todo lo anterior, se decidió no mantener estas clases en la BD. En su lugar se optó por guardar identificadores tipo String en las Entidades Order para así diferenciar qué estrategias concretas les corresponden.

Por lo tanto, los strategy correspondientes a **DeliveryMechanism** (figura 1.1) y **WayToPay** (figura 1.2) fueron rediseñados de la siguiente forma:

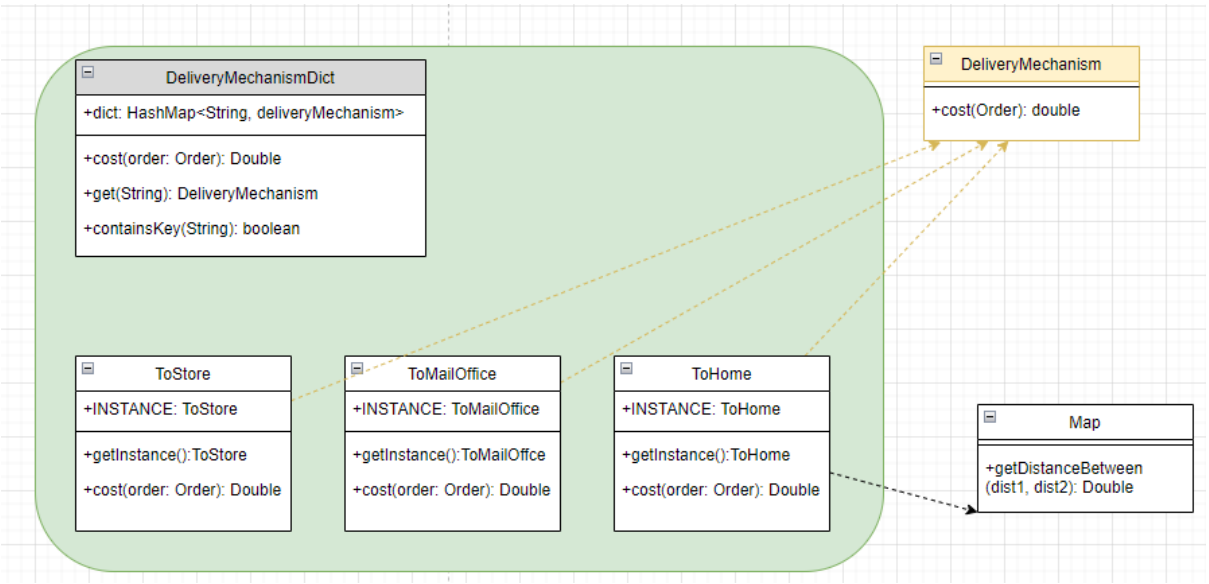

figura 1.1

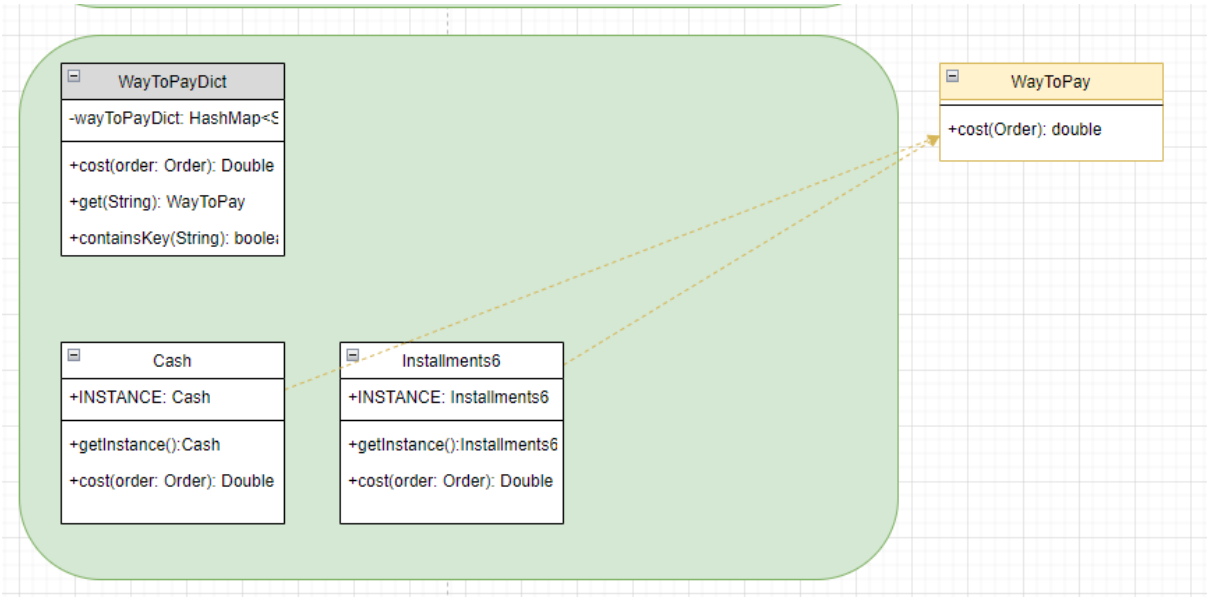

figura 1.2

Ambas clases: **WayToPay,** y **DeliveryMechanism** ya no son mapeadas a entidades de la base de datos, los *strategy* ahora derivan de una interfaz en vez de una clase abstracta y para cada jerarquía se mantiene un diccionario que permite mapear los identificadores de cada Orden con *sole-instances* correspondientes a las estrategias concretas que representen. Estos diccionarios nacieron como un medio de adaptar el modelo a estas nuevas condiciones, no es tan directo como el uso de un *strategy* puro, pero cumple el propósito de evitar que el usuario de la clase se complique decidiendo qué estrategia concreta debe usar, el diccionario devuelve la instancia que tenga que usar y cómo las distintas clases que puede devolver comparten siempre la misma interfaz entonces el usuario puede seguir aprovechando el polimorfismo de los métodos, sin que tenga que identificar de forma directa qué estrategia tiene que usar.

La base de datos quedó de la siguiente manera:

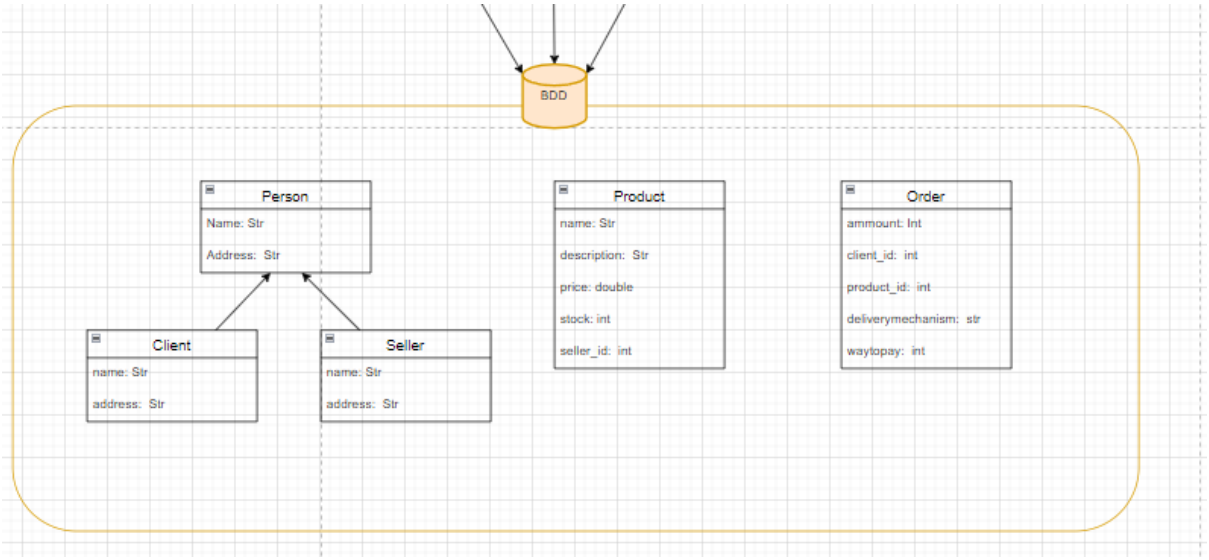

figura 1.3

De esta manera sólo estamos persistiendo lo justo y necesario para no guardar tablas de más. Y mantener la coherencia.

Otro problema que surgió es que en un modelado puramente orientado a objetos. Las relaciones entre los mismos pueden funcionar. A la hora de persistirlos esto no es así. Entonces se tuvo que agregar identificadores que permitan relacionar tablas entre ellas. Por ejemplo, un producto, para conocer a su vendedor lo sabe por su campo **seller_id** que este almacena el ID de persona. Entre otros.

Estas claves que se usan para relacionar tablas se denominan claves foráneas. Como en este caso particular las relaciones entre tablas no son de NxN no se llegó a crear una tabla intermedia para administrar esa relación. Dejando así solo tablas con claves foráneas. Afortunadamente, en el mapeo de las clases estas características se pudieron resolver de forma relativamente sencilla haciendo uso de las anotaciones @OneToOne, @OneToMany y @ManyToMany de provistas por *hibernate jpa*.

2. *Qué patrones, específicos a aspectos de persistencia y mapeo ORM, tuviste que aprender y aplicar?*

Se investigó acerca de patrones de persistencia y usamos 2 patrones por un lado el patrón **Repository**, por otro lado el **Singleton**.

El patrón **Repository** se utilizó a la hora del controlador interactuar con la BDD, se implementó una interfaz llamada *"MarketRepository"* que tenía los métodos definidos para interactuar con la BDD (además de los creados con *Jpa*). De esta manera. Podemos interactuar con cualquier tipo de BDD solo tenemos que implementar estos métodos de cómo accederla. En este caso se creó la clase "**MarketRepositoryConcrete**".

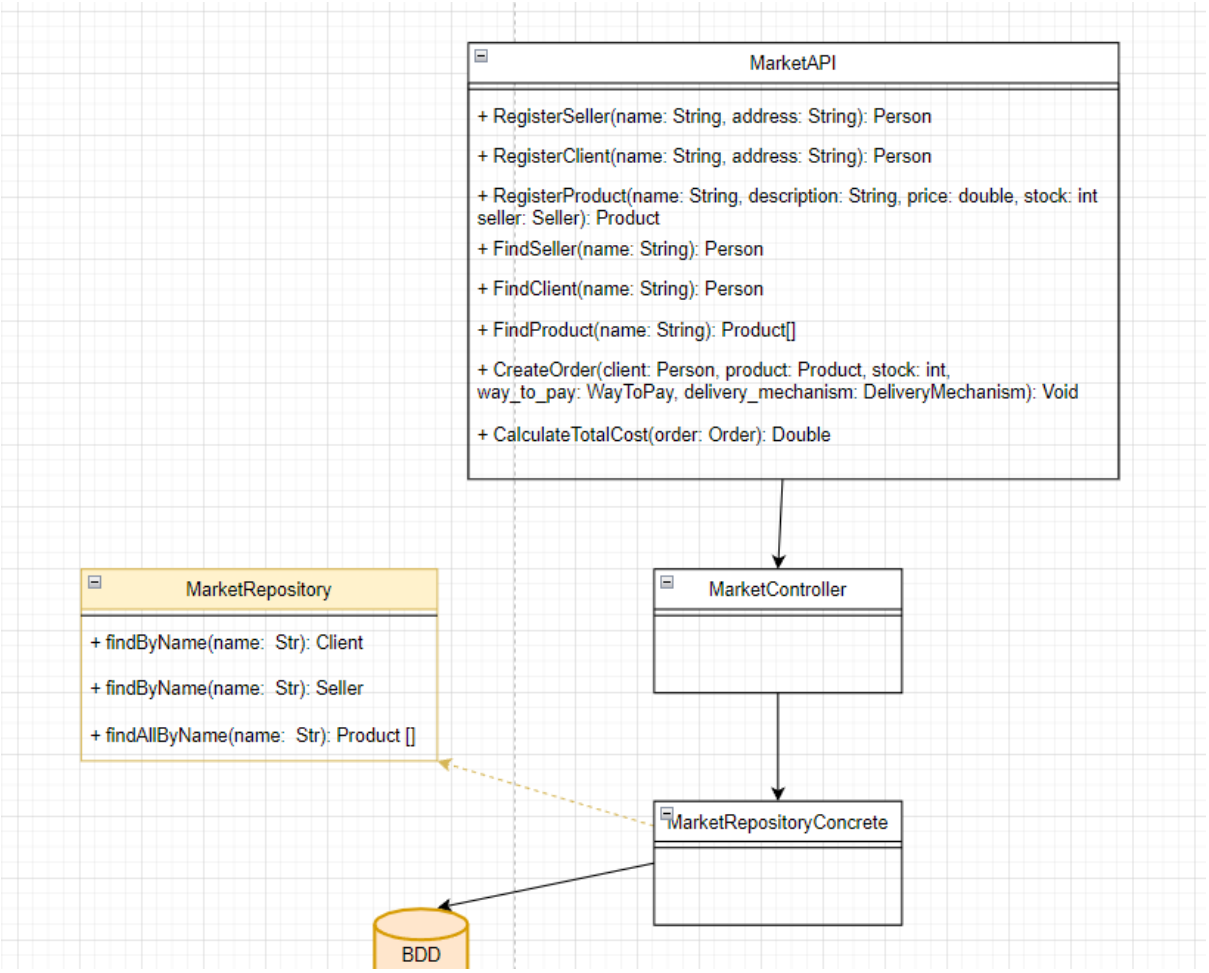

figura 1.4 **En el enunciado 2 se verá que se sigue manteniendo el patrón repository pero dividido en los distintos services.**

El otro patrón utilizado es el **Singleton** este nos permite regular la cantidad de instancias sobre un objeto. Se aplicó particularmente a aquellas clases que solo nos interesaban sus métodos, que no tenían información para persistir en la BDD y que a su vez no podían declararse como *static* debido a que de hacerlo él *strategy* que implementan perdería su sentido, ya que no se pueden sobreescribir métodos heredados marcados como *static*. de las mismas. Este es el caso de las clases que calculaban el costo

En términos generales se adoptó el patrón de diseño MVC. En este caso solo se pedía una API entonces las vistas no fueron implementadas. Pero se tuvieron los puntos de entrada de la API. Dónde está llamaba al controlador para que este trabaje sobre el modelo y devuelva los datos pedidos. Además de sobre algunas clases estáticas de nuestro proyecto. Este patrón de desarrollo de arquitectura de software es uno de los más ampliamente utilizados. Separando el modelo (estructuras de datos), de las vistas (archivos estáticos, .html, .css, .js) que es desde donde interactúan nuestros clientes y en el medio están los controladores. Encargados de manejar la lógica de la aplicación.

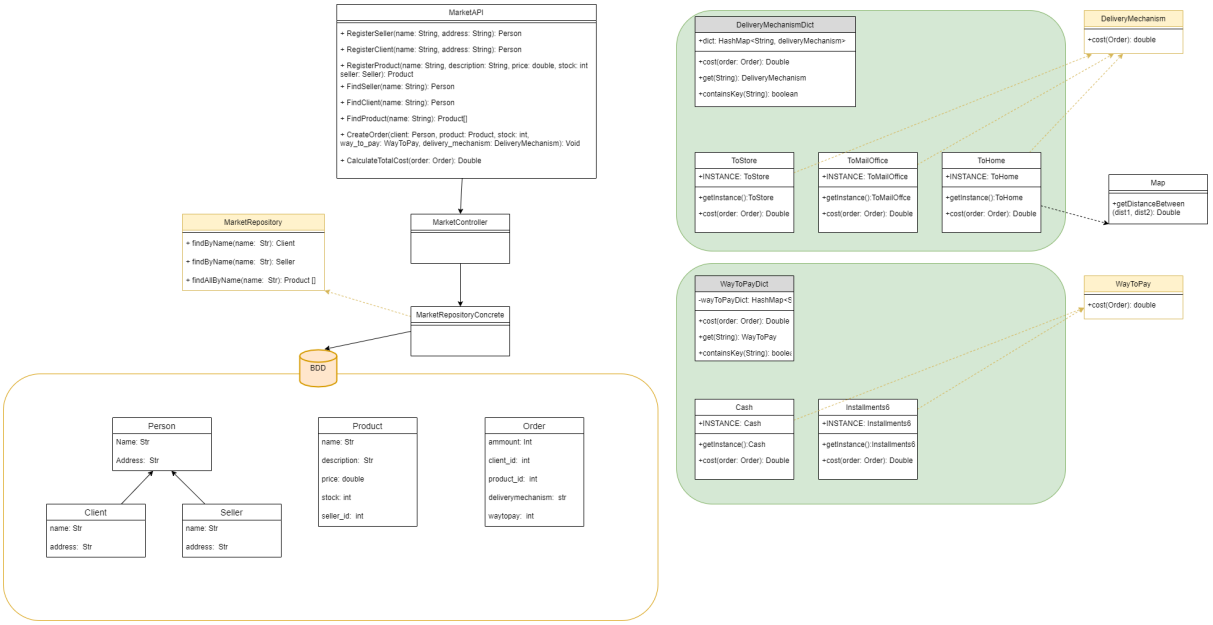

figura 1.5

3. ***Si en lugar de persistir en una BD relacional, fueses por algo tipo Mongo (donde la impedancia entre el modelo de objetos y el modelo persistido es otro), que harías diferente?***

En una BDD relacional se guardan relaciones. Valga la redundancia, es decir. Yo no puedo guardar un objeto entero dentro de otro. Lo que tengo que lograr es un mecanismo por el cual me permite relacionar dos o más tablas entre sí. Las claves foráneas son un ejemplo para manejar relaciones. Si en vez de utilizar una BD relacional se utiliza Mongo cambiaría el diseño. Porque este si permite que se relacionen objetos completos. Mongo utiliza algo llamado documentos. Esto permite a los diseñar una BDD puramente orientada a objetos. Sin la necesidad de tomar estas decisiones de cómo relacionar las tablas entre sí (como un modelo SQL).

Enunciado 2:

Nuevamente, la idea sería tomar alguno de los ejercicios simples de OO1 (que tenga casos de uso que uno naturalmente partiría en distintos servicios). Implementarlo (desde la persistencia al servicio web, sin interfaz) en microservicios aplicando los criterios/patrones que creas convenientes. Y luego responder ...

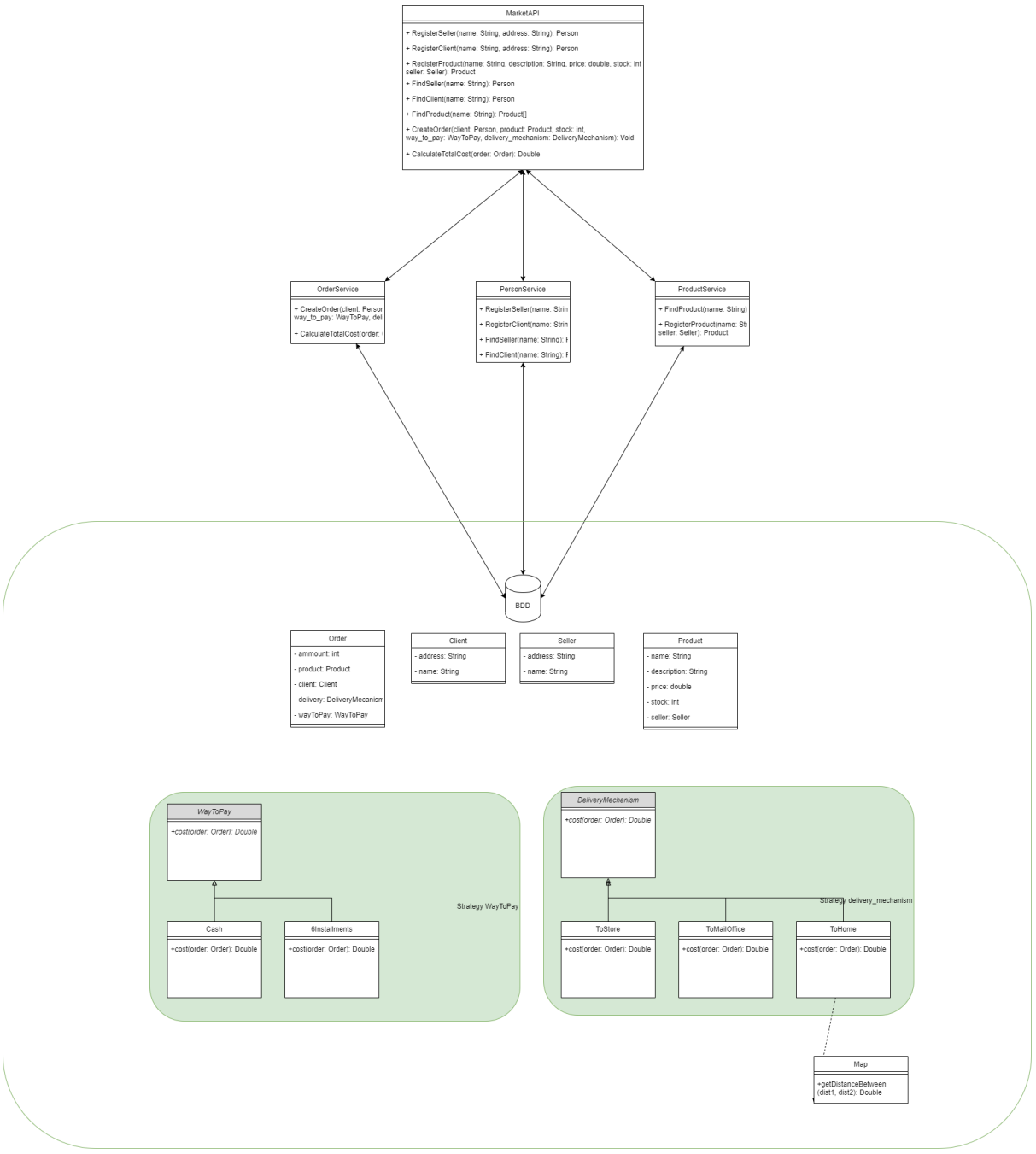

figura 2

1. De qué manera y en qué momento afectó tu diseño (y la forma en la que lo encaraste) el saber que estabas hablando de varios servicios**?**

Se realizó la misma metodología que el punto anterior de tal manera de diseñar primero, puramente en objetos y luego pasarlo a un patrón de desarrollo de microservicios.

Recordemos que la idea de los microservicios es separar las funcionalidades de la aplicación de manera que queden lo más independientes y autónomas posibles,

cada *service* cuenta con su propia conexión, de tal forma que se pueda crear microservicios y conectarlos rápidamente para agregar funcionalidad.

Sobre el ejercicio, se decidió separarlo en tres servicios distintos. Por un lado el ***personService*** encargado del manejo de usuarios, como segundo el ***productService*** encargado del manejo de productos y por último el ***orderService*** encargado del manejo órdenes. Como se ve en el gráfico (figura 2)

También se creó una interfaz para cada uno de los services. Esta decisión fue porque tanto la clase MarketAPI llamaba a los métodos de los services, y los services llaman a la base de datos para interactuar. Ambos métodos son llamados de la misma forma. Al implementar la interfaz podemos soportar la herencia múltiple y además damos seguridad obligando a que la clase que implementa la interfaz, desarrolle los métodos. En este caso cada *service* y la clase MarketAPI.

2. Qué patrones, específicos a aspectos de arquitecturas de microservicios, tuviste que aprender y aplicar?

Se investigó un poco sobre patrones de microservicios y se encontró que un patrón Agreggator podría mejorar nuestro diseño. Ya que en el caso actual tenemos la clase MarketAPI que tiene que conocer a cada service y lo ideal seria que conozca a 1 Aggregator y este se encargue de llamar al service correspondiente. De esta manera se abstrae a MarketAPI de los *service* concretos que debe usar, a la par que fomentamos el fácil crecimiento, la independencia y seguridad. En cierta forma este patrón es similar al *facade*, ya que centraliza todas las peticiones en una sola clase, la cual de forma interna sabe como resolver cada una y permite al usuario abstraerse de los distintos componentes del sistema. Con la particularidad de que acá nuestro usuario sería MarketAPI y nuestro sistema corresponde a los distintos *services* con los que este debe interactuar.

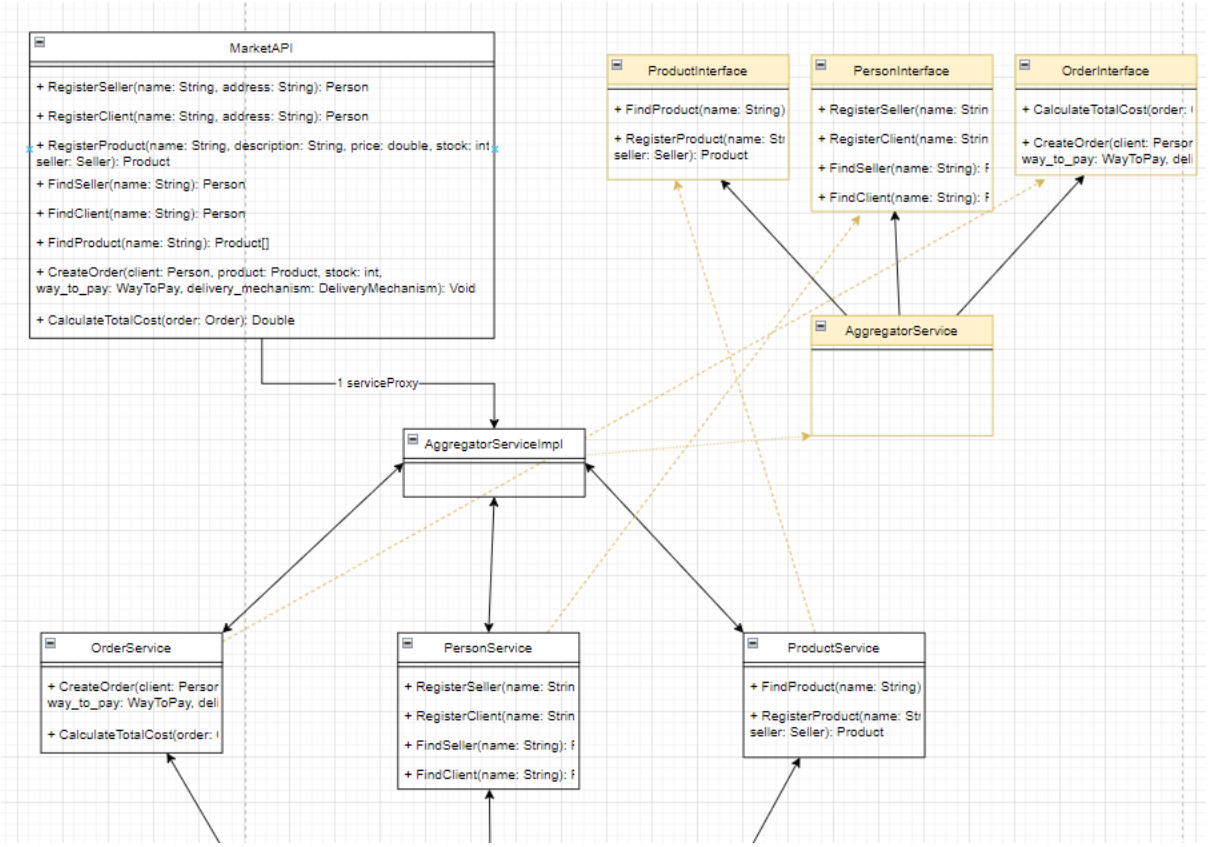

figura 2.1

quedando así. Los recuadros amarillos son interfaces implementadas tanto por los *Services*, como el *AgregatorService* ya que todas utilizaban el metodo pero de distintas formas. MarketAPI llama al Aggregator. El Aggregator ve que service es el necesario y lo llama. Por último, el servicio hace la consulta necesaria con la base de datos y devuelve el resultado. Está diseñado con jerarquías de Interfaces para fomentar el crecimiento. Si se quiere agregar un nuevo service basta con crear, la interfaz, el service y luego las llamadas en *aggregatorService* y *marketAPI*.

Como resultado final. Del ejercicio 1 y 2, quedó el diseño como lo vemos ahora:

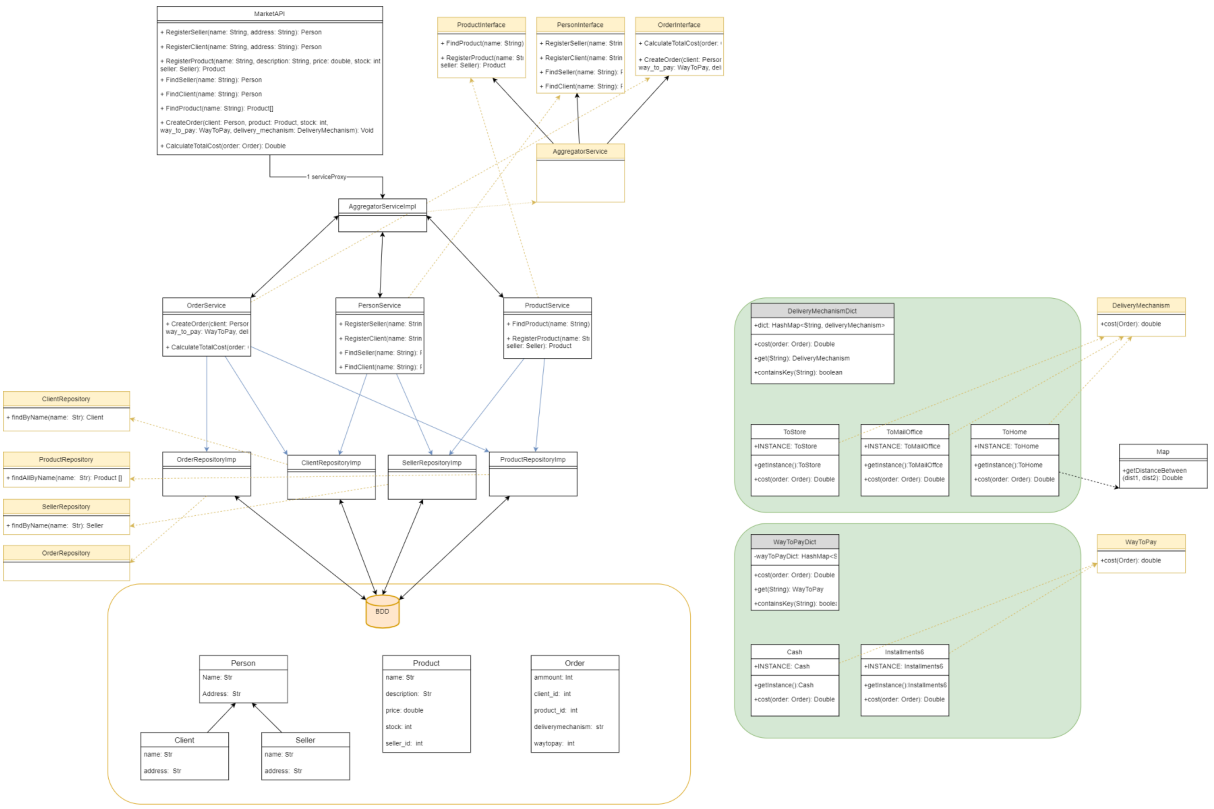

figura 2.2

3. De qué manera se relaciona el diseño y la implementación (asumamos que todos están hechos con el mismo stack tecnológico) de los distintos microservicios? Pros y contras?

Claramente hay diferencias entre el diseño y la implementación. De todas formas, al usar un framework para Java, un lenguaje puramente orientado a objetos, tanto el diseño como implementación no variaron tanto. Esto se considera una ventaja. También fue difícil pensar un diseño que avale el crecimiento progresivo de la aplicación. Y qué datos eran mejor persistirlos.

Respecto al diseño UML inicial debido a la naturaleza del framework utilizado para persistir los datos (*hibernate jpa*) y nuestras propias consideraciones de que valía la pena persistir y que no se tuvo que realizar modificaciones que desacople ciertas clases del resto del diagrama (WayToPay y DeliveryMechanism). Esto naturalmente fue reflejado en unas segundas instancias del diseño, el cual a su vez continuó siendo desarrollado en pos de hacer lo más desacopladamente posible las forma en la que los microservicios eran consumidos de como estaban implementados (Todos los services junto a sus distintas interfaces). De igual forma también se agregó un intermediario más para que el usuario se pueda abstraer de que servicio concreto debe usar, para simplemente limitarse a utilizarlo (AggregatorService).

Naturalmente a la hora de hacer la implementación los service no funcionaron por arte de forma mágica, hubo que establecer todos los mecanismos necesarios que el

framework *hibernate jpa* exige para la interacción con la base de datos. Esta tarea puede ser particularmente difícil, si bien hay múltiples guías online el que el proceso involucre a tantas clases hace difícil seguir el hilo y es fácil olvidarse de algún detalle aquí o por allá, además siempre están los problemas derivados del IDE concreto que se esté utilizando. Naturalmente después de varios intentos reconstruyendo el proyecto con distintas metodologías se logró una versión funcional, utilizando las la anotación @Entity (entre otras) para establecer cómo se realizará el mapeo de las clases de la carpeta *model* respecto a las tablas de la DB. La anotación @Repository para definir las interfaces de los repositorios que serían utilizados por los *services* para acceder a los datos. Naturalmente la implementación concreta de cada *repository* no fue construida manualmente, esta se ve facilitada por *hibernate jpa*, siendo que uno simplemente tiene que usar la etiqueta @Autowired en la variable desde la que quiera usar un repositorio y el framework inyectara implementaciones concretas de estas clases en esas variables para que puedan ser usadas en tiempo de ejecución. Naturalmente los repositorios únicamente eran usados por los *services* los cuales eran especificados por medio de la anotación @Service.

Resuelto esto junto a con el respectivo código de toda la lógica de negocio en los servicios y los demás métodos propios del problema. Ya se podía disponer de una aplicación totalmente implementada. En perspectiva, hay ciertos pros y contras que vale la pena destacar, del lado negativo tenemos la importante barrera de acceso a la tecnología hibernate, no es simple de comprender y tiende a romperse por defectos mínimos, los cuales a veces son problemas del propio IDE. Por otro lado, no son pocas las ventajas que trae, facilita muchísimo la implementación de aplicaciones que interactúan con bases de datos, permitiendo un mapeo claro y fácilmente editable, además su uso implica (de forma indirecta) el uso del patrón repository, el cual es particularmente útil a la hora de interactuar con otros tipos de bases de datos. Si se quisiera cambiar de tipo de DB gracias al patrón bastará con redefinir los repositorios de interés respecto a la base de datos en cuestión y sustituir la variable correspondiente. Además, nada impide tener múltiples repositorios para múltiples tipos de DB, de forma que uno pueda interactuar en los *services* de múltiples DB al mismo tiempo. Por otro lado la implementación de los endpoints de la MarketAPI fueron echos por medio de herramientas proveídas por paquetes derivados de spring-boot, su implementación desde luego no fue complicada y online se la puede encontrar muy bien documentada, por lo que desde luego es otro punto a favor.