Informe para TP final agronomía, OO2

Autor: Manuel Martinez, 14255/8

Introducción

Para el tp final de objetos 2, se planteó presentar un trabajo relacionado al encuentro / taller de agronomía e informática, titulado *“Cuando hablamos de Informática, agroecología y agricultura familiar, ¿de qué hablamos?”*, dictado en octubre de 2021.

Luego de cursar el taller, y realizar el encuentro en persona con una familia agricultora de la zona, se planteó diseñar un programa que resuelva alguna problemática que se nos ocurriera.

Así, el planteo del tp es un modelo simple para una aplicación de tipo e-commerce, en donde se tienen usuarios (tanto clientes como productores), productos y compras que se pueden realizar.

El modelo

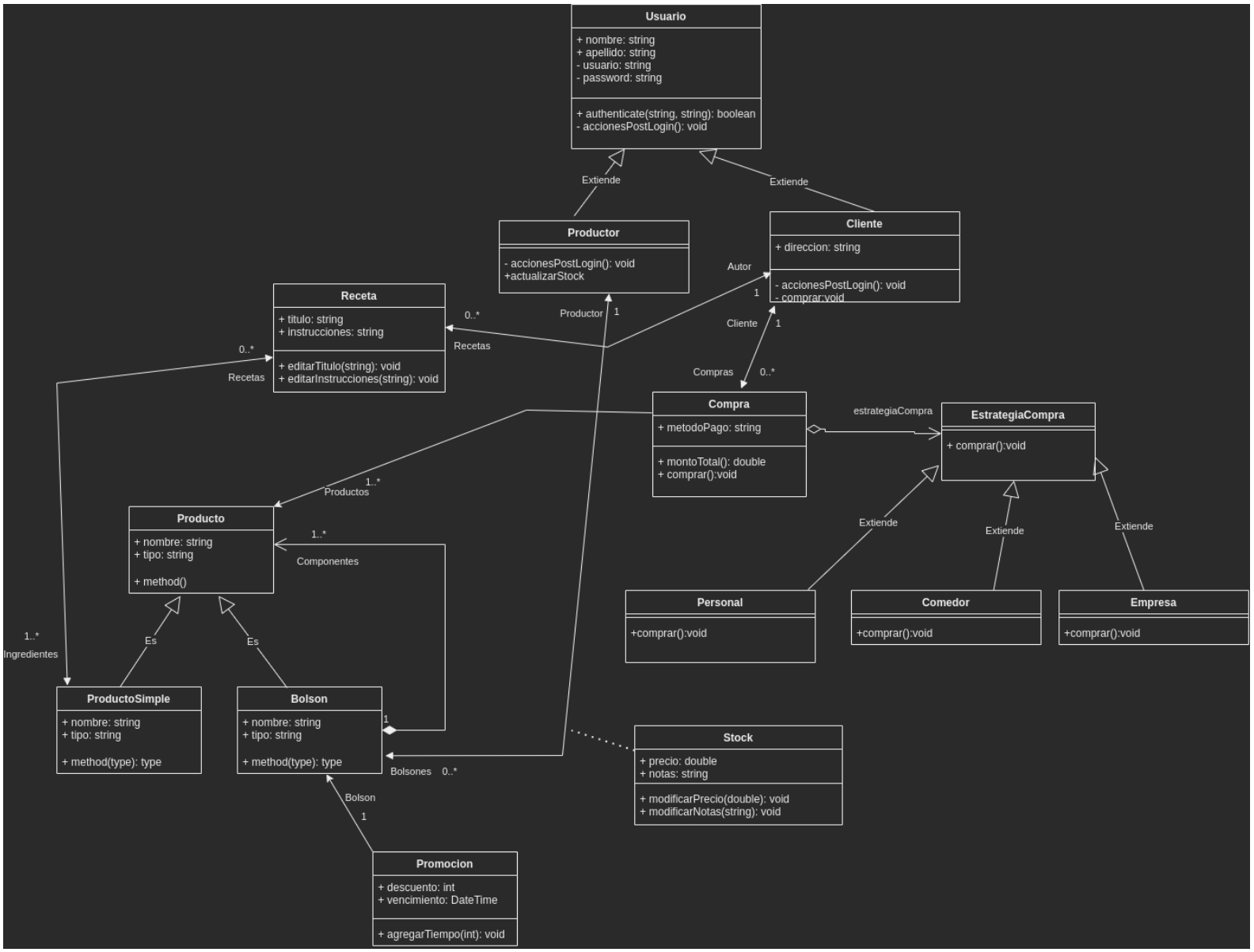
El modelo del planteo original, era el que se ve en el siguiente diagrama UML de la siguiente página.

Por fines prácticos y de tiempo, se recortaron algunas clases del modelo, que son:

- Receta
- Promoción
- Stock

El resto de clases quedaron hechas en el código. Las clases recortadas no planteaban ningún patrón de diseño a aplicar, por lo cual se decidió que dejarlas de lado no era de gran impacto.

MODELO UML: (link: [UML TP final oo2.drawio](#))



Patrones utilizados

Referente a clases

En lo que respecta a las clases, se utilizaron 2 patrones de diseño:

- **Composite**: En la clase Producto. Sus dos clases, ProductoSimple y Bolson, representan al objeto individual y al compuesto respectivamente. Es decir, un Bolson está compuesto por uno o más ProductoSimple. Dentro de esta herencia, se aprovechó el **polimorfismo** para definir métodos de Producto que se implementan diferente en cada una de las subclases. Más específicamente, los métodos

getPrice() y getAmountProducts(), actúan diferente según sean llamados desde un ProductoSimple o un Bolson.

- **Strategy**: Se implementó una clase Strategy para cada compra, la cual tiene 3 sub estrategias: *PersonalStrategy, ComedorStrategy y CompanyStrategy*. Cada una de estas, resuelve cómo calcular el monto de la compra, por ejemplo, aplicando cargos extra si se trata de una compañía o comedor. **IMPORTANTE**: este patrón quedó planteado en el código pero NO implementado, porque tuve problemas haciéndolo funcionar con JPA.

Referente a diseño general

Para diseñar la aplicación, se decidió usar el patrón MVC (model, view, controller), que plantea separar la lógica de la aplicación en distintas capas , cada una con su responsabilidad. En mi caso, la aplicación no cuenta con capa de **view**, pero si de **model (donde se modelan las entidades persistentes del sistema) y controller (la capa que atiende y responde a peticiones http)**. Además, se utilizaron los patrones de **service** (capa que realiza la lógica intermedia entre las peticiones externas y la comunicación con los repositorios) **y repository** (la capa que se encarga de comunicarse con la base de datos).

En particular, se aprovechó el patrón repository para crear repositorios para las jerarquias de clases, ya que se puede crear un repositorio base (para ejecutar las acciones genéricas, como guardar, recuperar todos, etc) y otros sub-repositorios que extienden al base, e implementan las queries específicas de cada subclase.

Uso de framework

Los framework facilitan abstraerse de las complicaciones de bajo nivel de las tecnologías, para concentrarse en programar la solución de forma eficiente y rápida. Se decidió usar el framework web de Java **Spring**, que provee muchas facilidades a la hora de crear aplicaciones con patrón MVC. Por ejemplo, resuelve muchas cosas con **anotaciones** en clases y variables. Además, se decidió usar las librerías oficiales de **Spring Data y Spring Boot**. Spring data facilita mucho la creación de los repositorios, que son las clases que se conectan a la base de datos para persistir y recuperar datos. Por otro lado, Spring Boot facilita mucho la creación de un servidor Spring listo para ser usado, y provee cosas como un servidor Tomcat listo y configurado, configuración http para las peticiones y respuestas, entre otras cosas.

También se utilizó el framework **JUnit** para realizar los tests de la aplicación. Este también provee una serie de anotaciones útiles para marcar diversas cosas de los tests.

Test driven development

En el proyecto se utilizó test driven development, es decir, se programaban los tests antes o a la par que el código. Así, los tests se iban corriendo y, al fallar, se programaba el

código que los haría pasar exitosamente. Esto se hizo para cada funcionalidad nueva que se agregaba.

Otra cosa que se hizo fue aplicar el refactor **extract method**, al notar que los tests contenían mucho código repetido. Asi, se extrajeron las instanciaciones de objetos repetidos para hacer cada test, a la función BeforeEach provista por el framework de testing JUnit, de forma que antes de cada test, se creaban los mismos objetos con unas pocas líneas de código.