Justificaciones de Diseño TPA – Entrega 2

Grupo 1

Diagrama de clases

Personas en situación vulnerable

Se modeló **Tarjeta** como *clase concreta*. Por conveniencia para el chequeo de usos por día, **Tarjeta** conoce a su **PersonaVulnerable** asociada pero la recíproca no es cierta, ya que generaría alto acoplamiento innecesario entre ambas clases. Sin embargo, **Tarjeta** conoce sus **RetiroDeVianda** y viceversa: esto debe a que nos favorece tanto para facilitar los cálculos de usos máximos como para asegurar la trazabilidad de las **Tarjetas** y **RetiroDeVianda**.

Se extendió el *enum* **FormasDeColaboracion**, agregando el campo *DISTRIBUCION_DE_TARJETAS* para representar una de las nuevas formas de colaboración.

Se agregó el atributo *dirección* en **PersonaFisica** (es decir, ya no es dinámico) ya que es importante la carga de un domicilio para quienes elijan colaborar con la *DISTRIBUCION_DE_TARJETAS*.

Se creó la clase **DistribucionDeTarjetas** en representación de dicha forma de colaboración.

Heladeras

Se modeló el método *estaActiva()* de tipo *Boolean* para saber, justamente, si una **Heladera** está activa en un momento dado. No se optó por un atributo dado que la actividad de una **Heladera** es algo calculable.

Se agregaron atributos como *fechasYHorasDejoDeEstarActiva*, *fechasYHorasVolvioAEstarActiva*, *fechasYHorasReubicada* para asegurar la trazabilidad de los estados de las **Heladera**.

Se modeló la *clase concreta* **ModeloDeHeladera** para rescatar las *temperaturaMaxima* y *temperaturaMinima* permitidas por el mismo. Además, se creó la *clase concreta* **MedicionDeTemperatura** para controlar la temperatura real de **Heladera**.

Se diseñó la *clase concreta* **ConfiguracionDeTemperatura** para permitirle al usuario configurar la *temperaturaMaxima* y *temperaturaMinima* deseada.

En **Heladera**, se agregó el atributo *alertaActual* para guardar la última alerta que llegó y que no fue atendida. También se agregó el historial de alertas *alertasHistorico*.

Técnicos

Se modeló la *clase concreta* **AreaDeCobertura**, que se interpretó como “ciudad de cobertura”.

Se incluyó el atributo *cuestionario* que refiere al cuestionario respondido por el técnico, especialmente para establecer sus medios de contacto.

Recomendación de puntos de colocación

Se aplicó el patrón de diseño *Adapter* para implementar la API REST. Esto nos brinda mayor mantenibilidad y cohesión.

Se realizó un *mockeo* de dicha API de la que aún no disponemos por motivos prácticos; nos otorga mayor facilidad para el testeo del componente.

Reconocimiento a los colaboradores

Se modeló la *clase concreta* **CalculadoraDePuntos** la cual posee los atributos configurables: *coeficientePesosDonados*, *coeficienteViandasDistribuidas*, *coeficienteViandasDonadas*, *coeficienteTarjetasRepartidas* y *coeficienteHeladerasActivas*.

También, posee los métodos necesarios para realizar los cálculos requeridos: *calcularPuntos(PersonaFisica)* y *calcularPuntos(PersonaJuridica)*. Implementamos estos métodos como stateless para poder utilizarlos las veces que sean necesarias.

Se extendió el *enum* **FormasDeColaboracion**, agregando el campo *OFRECER_PRODUCTOS_Y_SERVICIOS* para representar una de las nuevas formas de colaboración.

Se modeló la *clase concreta* **OfrecerProductosYServicios** la cual contiene los atributos: *formaDeColaboracion*, *personaJuridica*, *ofertas*. Representa la forma de colaboración en sí.

Se diseñó la *clase concreta* **Oferta** que contiene los atributos: *nombre*, *pathImagen*, *cantidadDePuntosNecesariosParaAccederAlBeneficio* y *rubro*. Se modeló como *clase concreta* debido a que la queremos tratar como un *value object*, es decir, una instancia desechable. Por lo que, en caso de querer cambiar algún atributo, simplemente la desechamos e instanciamos una nueva. Esto mismo es válido para la clase **Canje**.

Rubro se modeló como una *clase concreta* dado que consideramos necesario que sea extensible en tiempo de ejecución.

Carga masiva de colaboraciones

Por el principio de responsabilidad única, decidimos que el **Instanciador** sólo se encargue de instanciar las diferentes formas de colaboración y colaboradores.

Utilizamos inyección de dependencia, haciendo que reciba el **Lector**, **Notificador** y **Conversor** por parámetro.

Decidimos modelar el **Lector**, **Conversor** y **Notificador** siguiendo el principio de responsabilidad única. Es decir, el **Lector** sólo se encarga de leer, el **Notificador** de notificar y el **Conversor** de convertir.

Para la notificación vía email, decidimos utilizar el patrón de diseño *Adapter* para disminuir el acoplamiento con "**ApacheEmail**". De esta manera, si en un futuro la biblioteca quedase deprecada, sería fácilmente reemplazable por otra clase que implemente la interface **AdapterNotificador**. Además, el patrón nos brinda:

- Mayor mantenibilidad debido al bajo acoplamiento entre la clase **Notificador** y la biblioteca **ApacheEmail**.
- Mayor cohesión de la clase **Notificador** debido a la delegación de cierto comportamiento en el adapter.
- Mayor facilidad de testeo ya que se podría implementar un mock de **AdapterConcretoApacheEmail**.

También utilizamos inyección de dependencia, haciendo que reciba el **Mensaje** por parámetro.
