

Trabajo Práctico Anual Integrador

**para**

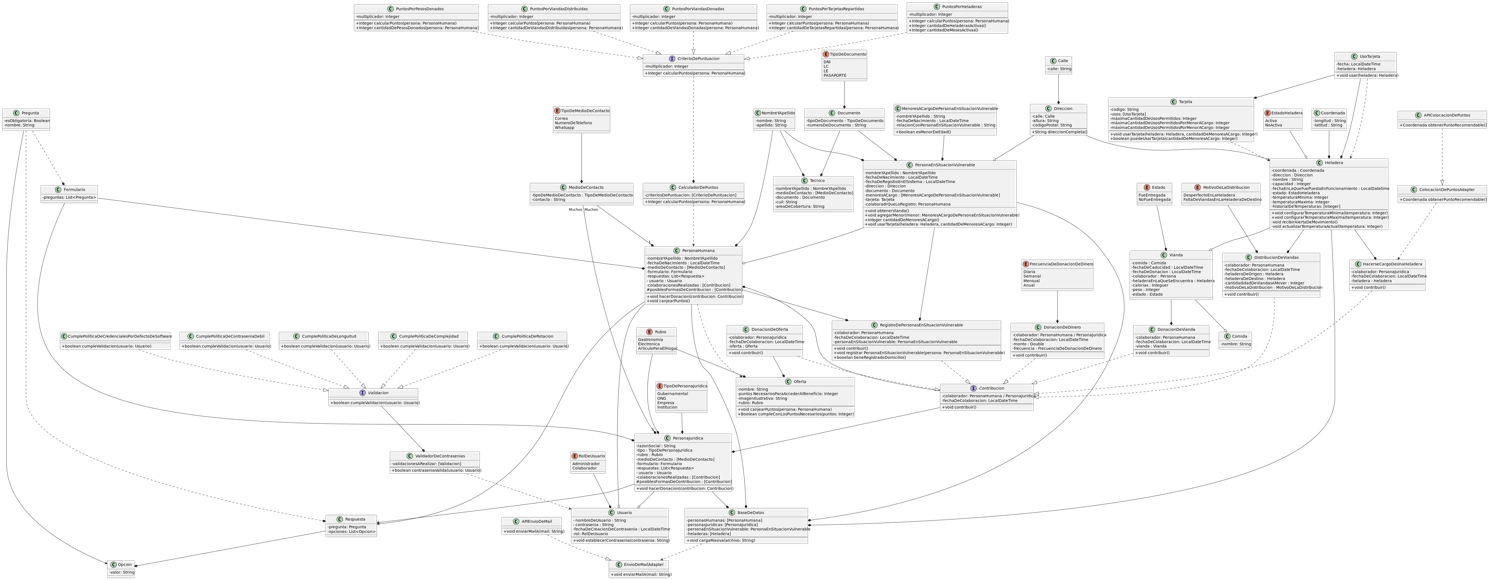
Diseño de Sistemas

**Preparado por**

|  |  |
| --- | --- |
| **GRUPO N° 14** | |
| **Nombre y Apellido** | **Legajo** |
| Alvarez Kalustian, Sahira Aylén. | 175.584-5 |
| Deza, Alexia. | 203.562-5 |
| Guardines, Brenda. | 172.658-4 |
| Juncos, Fernando. | 152.342-9 |
| Porcel, Gustavo. | 202.760-4 |

**ENTREGA 2: Modelado en Objetos Parte II – Integración de Servicio**

**Diagrama de Clases:**



**Decisiones de Diseño:**

Tarjetas:

Creamos la clase Tarjeta, a pesar de que inicialmente consideramos que fuese un método, ya que tendríamos múltiples instancias de la misma, así como su código y una lista de usos. Para manejar dicha lista, creamos la clase UsoTarjeta, que cuenta con una fecha de uso y la heladera en la que se utilizó (originalmente pensamos hacer dos listas, pero luego decantamos por crear una clase, ya que nos permitía asociarla fecha con la heladera).

En Tarjeta, manejamos el límite de usos de la tarjeta como un atributo (uno para la persona en situación vulnerable y otro para los menores a cargo, multiplicándolo por la cantidad de dichos menores), y comparándolo con las fechas en lista de Usos ver si cumple o no con el máximo establecido, ejecutando excepciones si no se cumple con la cuenta.

Abstracciones:

Creamos numerosas abstracciones para el manejo de datos repetitivo, como Direccion, NombreYApellido, Documeno, Rubro y Comida.

Técnico:

Creamos la clase Técnico a la que le agregamos el nombreYApellido, medioDeContacto, documento, cuil y areaDeCobertura (que por ahora manejamos como String, ya que luego podría convertirse en país o región, según su alcance).

Sistema de puntos para reconocer a los colaboradores:

En cuanto al requerimiento del reconocimiento a los colaboradores, nos basamos en el diseño del validador de contraseñas para su diseño. Se creó una interfaz “CriterioDePuntuación” que tendrá un método que retorna los puntos según el tipo de contribución. Luego, una clase por cada criterio de determinación de puntos (Puntos por pesos donados, por viandas distribuidas, viandas donadas, etc.), cada clase tiene un atributo “multiplicador” que es el valor por el que se multiplica al valor de los puntos obtenidos dependiendo de cada clase (valor que se obtiene en un método específico para cada criterio). Entonces el método calcularPuntos() retorna el producto entre la cantidad de contribuciones que se especifique y el “multiplicador”. La clase “CalculadorDePuntos” sería la entidad que tiene como responsabilidad realizar la sumatoria total de todos los puntos de una persona en base a las contribuciones que realizó, por eso tiene una lista de criterios de puntuación. Así la solución queda extensible por si se quieren agregar nuevos criterios de puntuación o si se quiere modificar el factor “multiplicador” por otro.

En donde antes teníamos una clase Contraseña, ahora decidimos desglosarlo y creamos una clase ValidadorDeContraseñas la cual va a tener una lista de validacionesARealizar del tipo Validador, en donde esta última es una interfaz la cual tiene un método booleano cumpleValidacion()

Para cada validación distinta (CumplePoliticaDeLongitud, cumplePoliticaDeComplejidad, CumplePoliticaDeRotacion, CumplePoliticaDeContraseniaDebil, CumplePoliticaDeCredencialesPorDefectoDeSoftware) creamos clases que extienden de la interfaz Validador y las cuales implementan su método.

Colocación de heladeras:

Para la colocación de heladeras, el enunciado menciona que hay organizaciones que, aunque no posean un local a la calle, pueden hacerse cargo de una heladera, como forma de contribución. Para resolver este requerimiento, le sumamos a la clase Heladera el atributo coordenadas, que tendrá la longitud y latitud de cada una de ellas. Para una mayor modularidad y poder cambiar como estaría este a dato a futuro, decidimos que este atributo tenga una clase Coordenadas, el cual tendrá los datos de latitud y longitud, lo cuales serán String ya que, aunque tengan valores numéricos, serán atributos no calculables, y además representarlos como String es una mejor abstracción para a futuro pensaros como parámetros que se intercambiaran con otra capa de abstracción, como puede ser el servicio de la API REST.

Recordando como era esta forma de contribución para la entrega anterior, representamos esta forma de colaboración, de manera que la clase Heladera estará asociada con la clase HacerCargoDeUnaHeladera el cual tendrá el atributo de colaborador (la persona juridica), con la fecha de colaboracion y la heladera en cuestión.

Para la recomendación de puntos de las heladeras, utilizamos una APIColocacionDePuntos.

Nuevas formas de colaboración:

Nuestra implementación de las formas de contribución nos permitió poder añadir sin dificultad las dos nuevas formas de contribución identificadas.

* RegistroDePersonasEnSituaciónVulnerable: Abstrae la lógica de que ahora las personas humanas pueden distribuir tarjetas a las personas en situación vulnerable como nueva forma de colaboración. Esta nueva clase tiene como atributos, la fecha en la que se registró a la persona vulnerable o se entregó la tarjeta, a la persona que registró a la persona en situación vulnerable, a la persona en cuestión. De esta forma, ahora la persona en situación vulnerable tiene una tarjeta asignada la cual podrá usar según los usos máximos.
* DonaciónDeOferta: Las organizaciones pueden ofrecer ofertas con una “Oferta” en cuestión, la fecha en la que se publicó y la entidad que realiza la oferta.

Manejo de la carga masiva de datos:

Por ahora lo manejamos con una clase BaseDeDatos que cuenta con una lista de personasHumanas, otra de personasJuridicas, una de personasEnSituacionVulnerable y otra de heladeras, para poder realizar la búsqueda al realizar la carga masiva de datos. Desde esa clase llamamos al método “getColaboradores(archivo: String)” de la clase “CargaDeUsuarios” que abstrae toda la lógica de leer el archivo, recorrerlo, crear un colaborador por cada registro, añadirle las contribuciones que le corresponden, eliminar usuarios repetidos y el envío de mails con credenciales.

Envío de mails: Usamos el patrón “Adapter” para implementar este requerimiento ya que usamos una librería con métodos un poco “rebuscados” y el algoritmo ofrecido no era muy intuitivo. Por lo tanto, al implementar el patrón creamos la interface con los métodos que se van a utilizar en el resto del programa “enviarMail(destinatario: PersonaHumana, mensaje: MensajeMail)” y enviarMails(destinatarios: List<PersoanasHumana>, mensaje: MensajeMail) y la clase “SenderMailAdapter” es la que la implementa definiendo los métodos con ayuda de la clase concreta “MailSender” que es la clase con lógica de un tercero.

Contribuciones:

Se decidió quitar la interface “Contribución” ya que la idea de que las contribuciones existentes la implementen había sido producto de la intención de tratar a todas las contribuciones por igual (polimórficamente) y para eso habíamos creado la interface con un método intuitivo a una contribución: “void Contribuir ()”. Pero luego de repensar la solución se concluyó que el método no tenía razón de existir, no tenía funcionalidad alguna que sea destacable, o que tal vez pertenezca a otra capa del sistema..  
Luego de quitarla, surge la cuestión de cómo manipular a todas las contribuciones por igual o incluso si esto es necesario, la solución que se eligió fue la de crear listas como atributos en cada colaborador que representan las colaboraciones posibles que pueden hacer y en cada una las que realizó (si es que realizó alguna de ese tipo). De esta manera también se quita el atributo anterior “posiblesFormasDeContribución” que iba a ser útil para decidir qué contribuciones puede realizar un colaborador, pero con la nueva propuesta es más clara y sencilla.