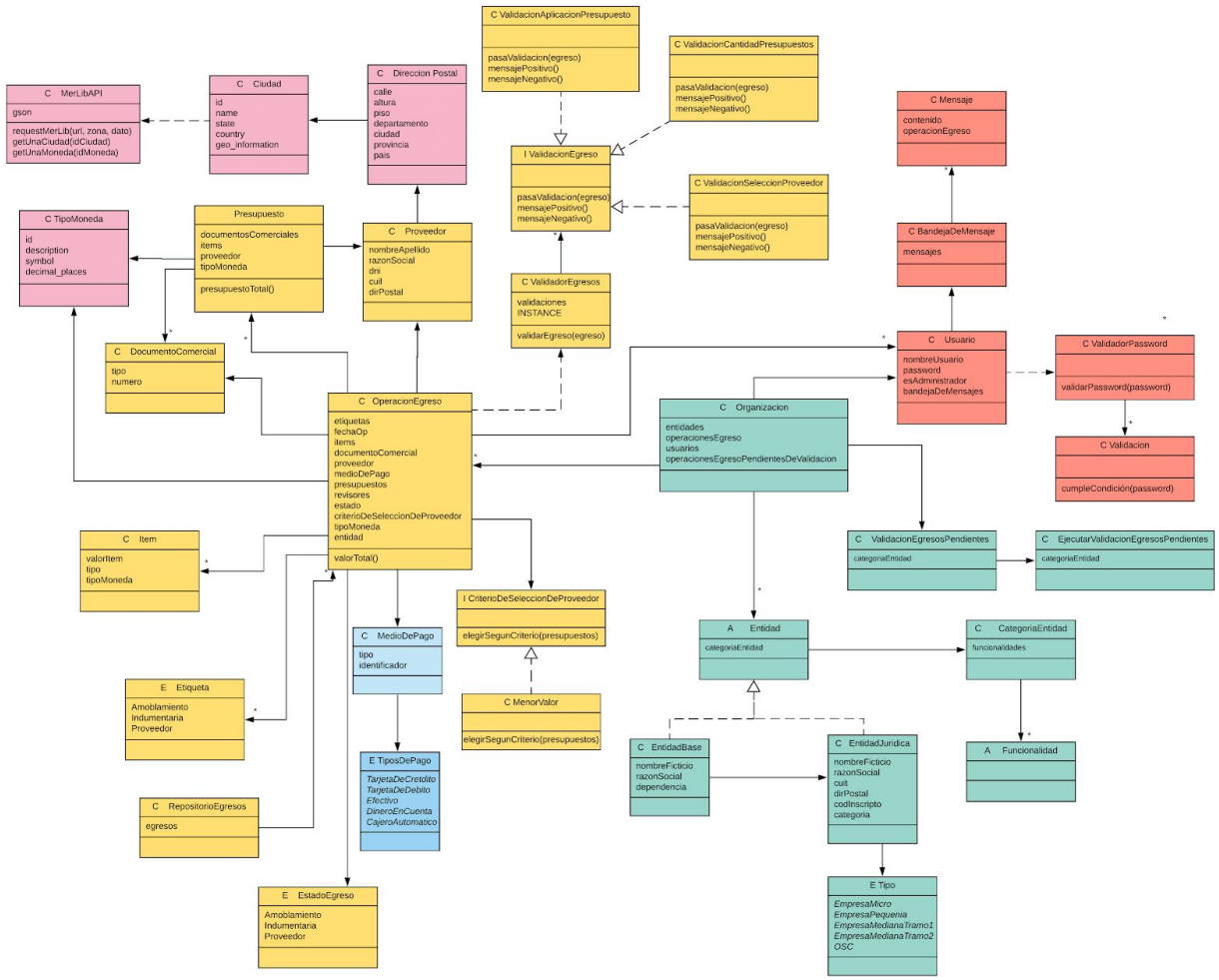
GeSoc DDS

Entrega 3 - Jueves Mañana - Grupo 4



Decisiones tomadas

- En la parte de medio de pago, nos inclinamos por usar un Enum para su tipo en vez de clases para cada medioDePago porque aún no tienen un comportamiento propio.
- Se podría haber utilizado un dictionary/map/hash en la operación de egreso para los ítems, pero se decide crear la abstracción Item porque nos pareció la opción más clara y que más se ajusta al dominio del negocio.
- Las entidades jurídicas y las base deben poder tratarse de manera indistinta, por lo que decidimos modelarlas empleando una interfaz Entidad. Ésta es implementada por las clases de ambos tipos de entidades.
- Como las entidades base son una convención informal, decidimos que sea la entidad base la que conoce la entidad jurídica a la que pertenece, y no al revés. Esto a su vez se asegura que

siempre se cumpla el requerimiento de que una entidad base pertenezca a una sola entidad jurídica sin tener que agregar código que verifique esto y evitando así agregar complejidad innecesaria.

- Decidimos usar un Enum para las distintas categorías de las Entidades Jurídicas, en las que incluimos las OSC, pues aún no tienen un comportamiento propio como para ser clases independientes.

- Para los tipos de usuario por el momento hemos decidido usar un booleano para saber si se trata de un usuario estándar o un administrador, ya que no tienen comportamiento propio para separarlos en clases diferentes.

- Optamos por delegar la validación de contraseñas a una nueva clase, dado que no consideramos que esta responsabilidad corresponde al usuario. Además, implementamos las validaciones como una lista ya que beneficia la extensibilidad, simplificando la adición de nuevas validaciones.

- Para validar que no hay contraseñas con caracteres repetidos utilizamos un regex. Descartamos utilizar un archivo con caracteres repetidos, dado que podría perjudicar su extensibilidad y dar lugar a posibles errores como la omisión de algún caso particular.

- Se decidió poner métodos que permitan ajustar los egresos a las reglas establecidas en lugar de usar un builder para este fin. El builder nos permitía que en un principio no se le puedan mandar mensajes a prendas que aún no fueron validadas, sin embargo una vez construida la operación de egreso (al no lanzar excepciones en las validaciones, sino solamente informarle a los revisores el resultado) nada nos garantiza que no se le puedan mandar mensajes a egresos inválidos. Por esta razón, consideramos que el builder no aporta demasiado al sistema y solo agrega complejidad innecesaria.

- Se utiliza una lista de egresosAsociados en la clase Item. Se descarta el uso de un booleano, debido a que cuando se quiere quitar un ítem de un egreso (para ajustar el mismo a las validaciones), no se sabría con exactitud si se debería cambiar este booleano a false o debería permanecer en true debido a que podría estar asociado a otro egreso.

- Para la selección de un criterio del proveedor decidimos utilizar una interfaz que es conocida por la por la clase clase OperacionEgreso, el cual implementa un comportamiento conocido por la interfaz. En caso de necesitar un nuevo de criterio para el proveedor podrá ser implementado mediante la interfaz, facilitando su implementación.

- Creamos un validador de egreso que instanciamos una única vez, ya que se utiliza un objeto de manera global. Además el validador de egreso conoce a una interfaz que declara tres comportamientos que luego serán implementados por las clase que implemente dicha interfaz.

- Decidimos implementar una clase BandejaDeMensaje para notificar a los interesados en cada egreso para que puedan ser notificados cuando uno de sus egresos requiere ser revisado.

- Se decidió poner una lista de funcionalidades en la CategoríaEntidad, lo cual permitirá agregar nuevos comportamientos.

- Se descartó lanzar las excepciones en las entidades ya que consideramos que eran responsabilidad de las funcionalidades.

- Decidimos usar un Enum para las etiquetas ya que por el momento no tienen comportamiento propio.

- Decidimos implementar el JOINED para herencia de la clase Entidad, pues en el SINGLE_TABLE quedaba en un caso (en las entidades base) casi la mitad de los campos en null. Por otro lado, el TABLE_PER_CLASS no es el más eficiente para consultas polimórficas y en nuestro sistema tendría bastante sentido querer conocer todas las entidades.

- Se tuvo en cuenta que era conveniente **desnormalizar** la dirección postal que teníamos como clase aparte, considerando que proveedores es la única clase que utilizaba una dirección postal. De esta manera, la tabla proveedores incorporaría nuevos atributos/campos: calle, altura, piso, departamento y un id_ciudad, **con el fin de aumentar el rendimiento**.

  Si tuviéramos una tabla dedicada a las direcciones postales, tendríamos que hacer un JOIN entre ambas tablas, a pesar de que un proveedor tiene una única dirección postal.

- Para evitar redundancia de datos o duplicación de los mismos, decidimos **normalizar** en tres tablas distintas: Ciudad, Provincia y País. Si tuviéramos todo esto dentro de una única tabla (en nuestro caso Ciudad), vamos a tener una gran duplicación de datos en alguna columna.

  Podemos tener X cantidad de ciudades, y en todos su registros repetir la misma Provincia como País. Si cambia alguna Provincia/País, deberíamos hacerlo en todos los registros. En cambio si lo delegamos a una tabla distinta, con tan solo modificar un registro nos alcanza.

- Optamos por no persistir la organización porque nos quedaba una tabla con un único dato que es el ID, y considerando que es la única organización de la cual estamos hablando, lo sacamos.