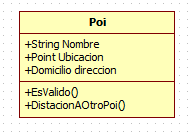
**Entrega 1 – Grupo 7**

Análisis y decisiones de diseño

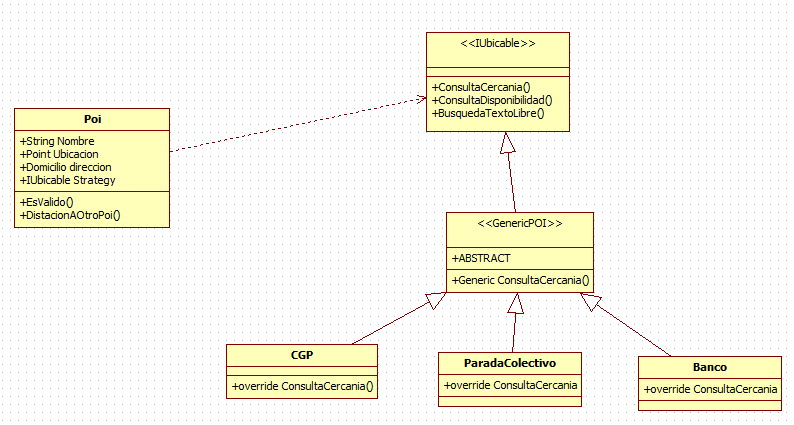
Primero comenzamos con la clase POI para la entrega 0B:



Decidimos abstraer el domicilio en una clase diferente para no sobrecargar la clase Poi.

El método DistanciaAOtroPoi se delega en la clase **Haversine** que realiza el cálculo.

Luego, para la entrega 1, el primer approach que utilizamos fue un patrón Strategy, de esta forma



Este patrón nos permite seleccionar diferentes estrategias a través de la variable de tipo IUbicable (interface), para poder elegir el comportamiento implementado en las clases concretas CGP, Banco, etc.

La ventaja que tiene es que podemos cambiar dinámicamente ese comportamiento, reapuntando la referencia de la estrategia.

En contrapartida, encontramos que los objetos contretos creados, además de comportamiento redefinido, agregaban otras **características de identidad**, como por ejemplo:

CGP: Agrega una lista de Servicios y una referencia a una comuna.

Banco: Agrega una lista de Servicios

Locales Comerciales: Agregan un rubro y una lista de horarios de atención.

Este modelo, generaba que tuviésemos que “configurar” cada objeto, haciendo por ejemplo:

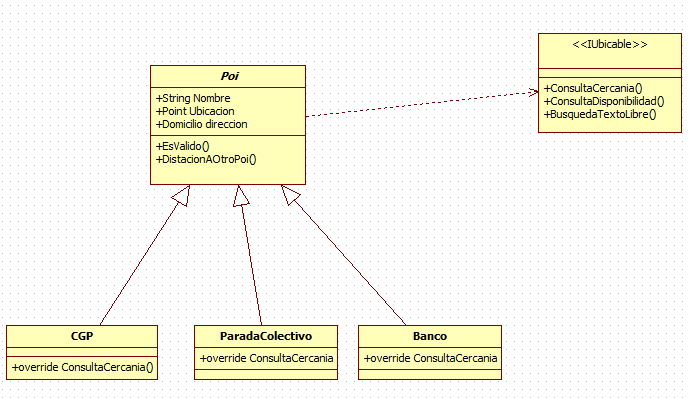
POI poi = new POI(); 🡪 *Al crear un POI, no se sabe a ciencia cierta que identidad tiene.*

poi.estrategia = new CGP() 🡪 *Se manifiesta la identidad del CGP a través de la estrategia. Eso nos pareció que la identidad del objeto quedaba escondida.*

*Además…*

poi.estrategia = new Banco() 🡪 *No solo se le agrega un comportamiento diferente, sino que ahora nuestro POI cambió de identidad. Ahora es un banco…*

Para mejorar esta situación, cambiamos el modelo radicalmente, quedando de esta manera:



La clase POI es abstracta y ahora implementa la interface (en vez de usarla, como sucedía anteriormente).

Las clases concretas heredan de POI, y agregan sus propias características (Rubro, Servicios, horarios, etc). Como se implementa la interface en la clase abstracta, no es necesario implementar los métodos allí mismos, podemos esperar a implementarlos en alguna clase concreta que herede de la abstracta. Esto nos permite como antes, agrupar comportamiento en super clases, y redefinir comportamiento en subclases.

La ventaja que obtenemos, es que ahora los objetos manifiestan mejor su identidad.

Banco banco = new Banco() 🡪 Es un banco ☺ (y también un POI por la herencia)

CGP cgp = new CGP() 🡪 Es un CGP ☺ (y también un POI por la herencia)

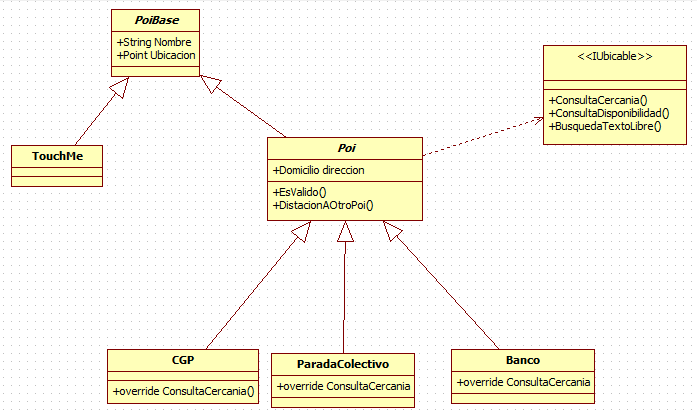
Como contrapartida, ya no se puede elegir una estrategia y luego modificarla, una vez creada la entidad, siempre será esa misma.

Por ultimo, decidimos modelar el objeto TouchMe.

Al analizarlo, lo pensamos como primero como un POI, heredando de la clase abstracta POI.

Pero, esto hacia que el objeto TouchMe exhiba comportamiento que no le queríamos otorgar (buscar por criterio, consultar disponibilidad, consultar cercanía). Decidimos que los objetos TouchMe se representen como puntos con nombre y quizás con una referencia a un mapa (para mas adelante).

De esta forma, modificamos el modelo para continuar con la jerarquía y mantener polimorfismo entre los POI y los TouchMe.



Creamos la clase PoiBase, también abstracta, y llevamos los atributos comunes de los POI y los TouchMe. De allí, hereda la clase TouchMe. Es cierto que esta clase no agrega comportamiento ni características nuevas, pero nos parece útil que exista como abstracción para mayor claridad en el código y porque quizás luego pueda agregársele comportamiento. Al estar separada de la jerarquía de la Clase POI, no implementa la interface.