CONCEPTO

En las clases se definen operaciones. Estas operaciones se definen considerando el nombre de la clase, el nombre de la propia operación, los parámetros que reciben y su valor de retorno. Estos 3 últimos elementos constituyen "La firma de la operación".

El conjunto de todas las firmas definidas en una clase (y por consecuencia en el objeto) se denomina "Interfaz del objeto". La interfaz de un objeto engloba al conjunto de solicitudes que se le puede enviar al objeto (Del libro Design Patterns: Elements of Reusable Object-Oriented Software).

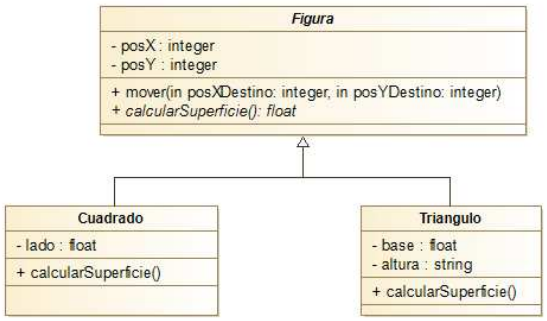
En criollo lo anterior indica: defina las operaciones disponibles en la clase. Bien…y ahora ¿Cómo pueden acceder a esas operaciones OTROS objetos?

1) Hay que definir un mecanismo para que esas operaciones estén disponibles (lo que se denomina Interfaz del objeto).
2) Esa interfaz del objeto es la propia semántica (el nombre del método) y firma de la operación (el conjunto de parámetros del método); es decir para acceder a la operación se debe hacer referencia al nombre de la operación, sus parámetros y tipo de retorno.
3) La forma en que un objeto se comunica con la interfaz de otro objeto es a través del protocolo de mensajes, el cual según nuestro lenguaje de programación (Processing) se define como: nombreDelObjeto.nombreDelMétodo(parametros);

REALIZANDO UNA APROXIMACIÓN DESDE LA EXPERIENCIA DEL MODELADO

En la programación orientada a objetos, a veces es útil definir **QUÉ** debe hacer una clase, pero no **CÓMO** lo hará. Para realizar lo anterior podemos recurrir a dos estructuras:

1) Por un lado, las clases abstractas, las cuales vimos cómo se definen. Estas clases tienen la particularidad de que en ellas se pueden definir tanto operaciones "normales o concretas" como las denominadas operaciones abstractas. Al momento de llevar un diagrama de clases modelado a la etapa de programación, las operaciones concretas poseen implementación. Es decir, el desarrollador deberá codificar el algoritmo de esa operación. En cambio, las operaciones abstractas no poseen implementación. Esto significa que alguna subclase de la clase abstracta deberá proporcionar la implementación de las operaciones abstractas definidas en la superclase abstracta.
   En pocas palabras una clase abstracta puede poseer operaciones abstractas (aquellas en la que especifica la semántica y la firma, pero no su implementación)
   Un ejemplo clásico de la bibliografía es el siguiente



Tenemos una superclase abstracta. En esta clase se pueden definir atributos como en cualquier clase. También posee una operación concreta (`mover()`), que recibe dos parámetros para cambiar la posición de la figura en un espacio 2D. La operación `calcularSuperficie()` es abstracta. Resulta claro que no posee implementación porque sin saber el tipo de figura no es posible utilizar una "formula" para calcular esta operación. Se hace necesario definir subclases, y que cada una de ellas sobre escriba esta operación para establecer la implementación correcta.

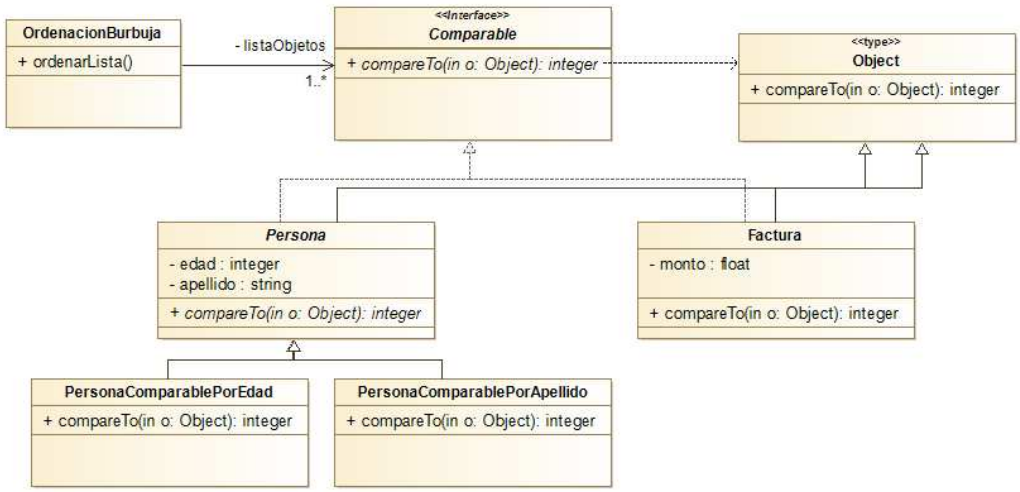
2) Si bien las clases y operaciones abstractas son útiles, es posible llevar este concepto un paso más allá. Tanto en UML como en los principales lenguajes de programación, es posible separar por completo la interfaz de una clase de su implementación, utilizando interfaces.

   ¿Por qué querríamos hacer esto?

   Las interfaces sirven para representar "contratos". Establecen que operaciones deberán implementar las clases que adhieran a este contrato. De esta manera, si una clase adhiere a este contrato, debe respetarlo, por ende, deberá implementar el código de todas las operaciones que están definidas en ese contrato.

   ¿Y esto para que nos sirve?

   Separar la interfaz de su implementación en términos de ingeniería del software implica minimizar el acoplamiento y maximizar la cohesión. Para no entrar en detalles de estos dos conceptos, suponga que debe desarrollar una aplicación que ordena una lista de cualquier tipo objetos. Ud podría crear algo como esto:



Observe varias cosas:

Por empezar se define una interface denominada Comparable. Toda clase que implemente esta interfaz establece un contrato por el cual esa clase debe tener implementado una operación `compareTo()` que compara ese objeto contra otro objeto, devolviendo un valor que indica el resultado de la comparación (por ejemplo podría ser: negativo, 0 o positivo si el objeto actual es menor, igual o mayor respectivamente al objeto comparado).

Probablemente si Ud define una clase Persona no piense que deba poseer una operación `compareTo()` como algo común que se le pediría a una persona. Pero el contrato establece claramente que debe implementar esa operación. De allí la primera utilidad de una interface.

En segundo lugar, si observa la clase `OrdenacionBurbuja` notará que posee un atributo que es una lista o un arreglo de objetos comparables, y además posee una operación `ordenarLista()`. Ud. podría agregar otra clase que ordene utilizando otra técnica de ordenación (por ejemplo, debido a un requisito no funcional, respecto del tiempo de respuesta del método de ordenación, es decir que se necesita un método más rápido de ordenación que el de burbuja) sin casi tocar todo el resto del modelo. También podría agregar nuevos mecanismos de comparación para la persona, lo cual agrega versatilidad; algo que no posee `Factura`, ya que solo posee un mecanismo de comparación, que sería por el monto. Todo gracias a la combinación del uso de interfaces y clases abstractas.

En tercer lugar, aparece una nueva relación, que es la dependencia. La operación `compareTo()` no puede llevarse a cabo si no recibe por parámetro un objeto de tipo `Object`. Esta relación se utiliza para evitar que una clase deba tener un atributo de otra clase, así en este caso `Comparable` no está ligado a `Object` (no conoce a `Object`, o no tiene referencia a `Object`, o no apunta a `Object`). Además, como `Comparable` es una interfaz, por definición no posee atributos.

SOBRE LA DEFINICIÓN DEL CONTRATO

En este contrato, la interfaz DEFINE:

- Los nombres de las operaciones
- Su tipo de retorno
- El tipo y cantidad de parámetros que reciben las operaciones

En este contrato, la interfaz NO DEFINE:

- Implementaciones (en una interfaz todas las operaciones son abstractas)
- Atributos

Al igual que las clases abstractas, las interfaces no se pueden instanciar.

Generalmente se las confunde con clases abstractas, aunque la diferencia principal reside en que para que una clase herede de otra abstracta debe seguir cumpliéndose la frase semántica “es un tipo de”, y además las operaciones que se definan tienen un sentido asociado al nombre de la clase; mientras que una clase puede implementar una o varias interfaces, simplemente con el objetivo de indicar el contrato de operaciones que debe cumplir.

En adición a lo indicado en el párrafo anterior, mientras que la forma en que se lee la relación de una clase cualquiera con su clase abstracta es “**es un tipo de**”, en las interfaces la lectura es “**se comporta como**”.

Por último, hay que recordar que las interfaces no son clases.

REPRESENTACIÓN EN UML

Vamos a aprovechar este espacio para introducir más ejemplos. Tenemos que diseñar el sistema de una Clínica de terapias alternativas antiestrés (viene bien luego de ver este tema).

La clínica tiene un método innovador que disminuye el estrés a través de acariciar distintos objetos. La clínica tiene osos de peluches, gatos, y retazos de alfombras.

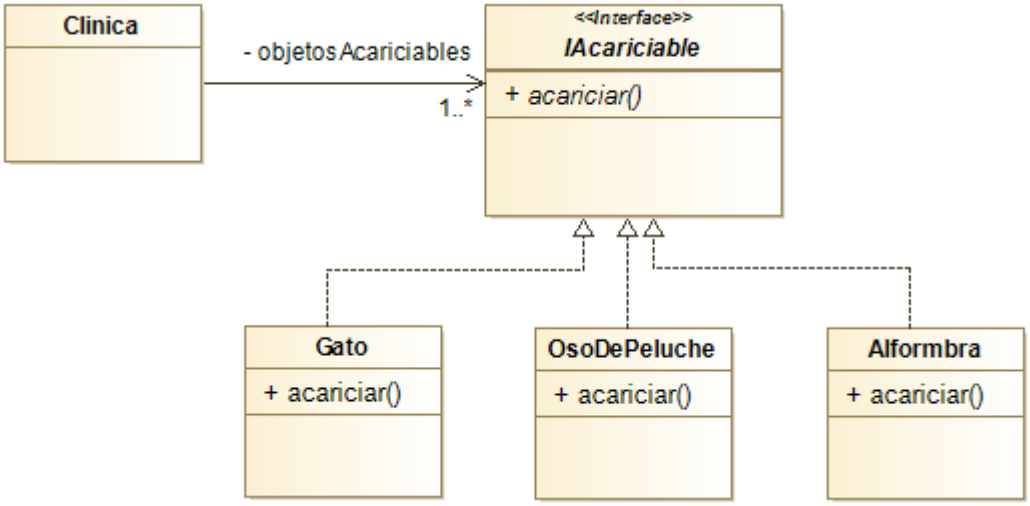
Podemos identificar estas clases como `OsoDePeluche`, `Gato`, `Alformbra`.

Entonces podríamos definir una interfaz `IAcariciable` y que posea una operación `acariciar()`.

Todas las clases que implementen esta operación tienen que respetar el contrato establecido y por lo tanto están obligadas a implementar la operación `acariciar()`.

La clínica desea mantener una lista de todos los objetos acariciables.

Un diagrama de clases que represente esta situación sería el siguiente



Podemos notar varias cosas (que también están reflejadas en el primer ejemplo):

- El estereotipo <<interface>> se antepone al nombre de la interfaz
- Se suele denotar el nombre de la interfaz con la letra I, no siendo obligatorio, pero es comúnmente aceptado dentro del desarrollo de software. Sobre todo, entre los programadores.
- El nombre de la operación está en cursiva para resaltar que es abstracto, aunque algunos autores no lo ponen, por cuanto se sobre entiende al estar definido dentro de una interface.
- La relación es similar al de la generalización, pero la línea es punteada, y tiene por nombre realización o implementación.

Podemos observar la implementación del diagrama de clases. En Processing se define una interface usando la palabra reservada interface:

```
1 interface IAcariciable{
2   void acariciar();
3 }
```

Como se indicó anteriormente, las interfaces no poseen atributos y todos sus métodos son abstractos.

Para indicar que una clase "implementa" esta interface se debe agregar a su definición la palabra reservada implements de la siguiente manera:

```
1 class Alfombra implements IAcariciable{
2
3 }
```

Notará que Processing le indicará que hay un error. Esto se debe a que le está indicando que debe "cumplir el contrato con IAcariciable". En la interface se definió el método abstracto acariciar(); por lo tanto Alfombra está obligado a implementarlo. Por ese motivo la definición de la clase quedaría de la siguiente manera:

```
1 class Alfombra implements IAcariciable{
2   public void acariciar(){
3     println("Me tiro encima de la alfombra y la acaricio");
4   }
5 }
```

De manera similar debe definir las clases Gato y OsoDePeluche.

Ahora veamos la definición de la clase Clinica

```
1  class Clinica{
2    private ArrayList<IAcariciable> objetosAcariciables;
3
4    public Clinica(ArrayList<IAcariciable> objetosAcariciables){
5      this.objetosAcariciables=objetosAcariciables;
6      for(IAcariciable o:objetosAcariciables){
7        o.acariciar();
8      }
9    }
10 }
```

Se ha creado un ArrayList a partir de la interface, y en el constructor se recibe como parámetros la lista de objetos acariciables que se asignarán al atrbuto, y luego simplemente por comodidad se recorre esa lista para indicarle a cada elemento que realice su método acariciar().

Finalmente, observemos el punto de partida:

```
1  Clinica clinica;
2
3  public void setup(){
4    ArrayList<IAcariciable> objetos = new ArrayList();
5    objetos.add(new Gato());
6    objetos.add(new OsoDePeluche());
7    objetos.add(new Alfombra());
8
9    clinica = new Clinica(objetos);
10 }
```

En este caso, observe que se instancia una lista de objetos de tipo IAcariciable, y luego se agregan tres objetos diferentes. ¿Por qué no indica error? Debido a que esas tres clases

"implementan" la interfaz IAcariciable. Es decir, esas tres clases "se comportan" como objetos acariciables, por lo tanto, se pueden agregar.

Técnicamente hablando, ***las interfaces permiten la definición de variables con comportamiento polimórfico***.

Esto significa que cada objeto IAcariciable responderá de manera diferente al método acariciar() dependiendo del tipo de objeto acariciable que sea. Sobre el polimorfismo veremos más detalles en posteriores entregas.

Finalmente se agrega la instanciación de la clínica y por ese motivo se obtiene el siguiente resultado:

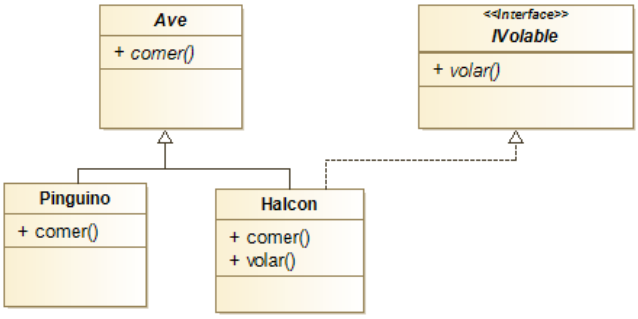
```
Acariciando un gato. Que buena terapia
Acariciando un oso de peluche. Que tierno
Me tiro encima de la alfombra y la acaricio
```

OTRAS UTILIDADES DE LAS INTERFACES

Dado que las interfaces permiten definir contratos únicamente sobre el comportamiento de las clases pueden resultar útiles para resolver de manera elegante problemas de diseño.

Por ejemplo, suponga que crea un videojuego donde utilizará diferentes personajes que son animales. Suponga que utiliza aves como personajes, aquí sabe que todas las aves comen, pero no todas las aves vuelan, y justo uno de sus personajes es un pingüino. En términos de diseño, podrías decidir crear una clase abstracta Ave con el método comer() y luego crear una subclase AveVoladora y que esta posea el método volar(). Así por ejemplo un Halcon heredará de AveVoladora, mientras que Pingüino de Ave.

Pero, si reflexiona un poco más, esta decisión se realiza sobre el comportamiento de las aves. Por lo cual una solución más elegante sería la siguiente:



Con esto establecemos un contrato únicamente con las aves que son voladoras, o dicho de otra manera que tienen comportamiento de voladoras.