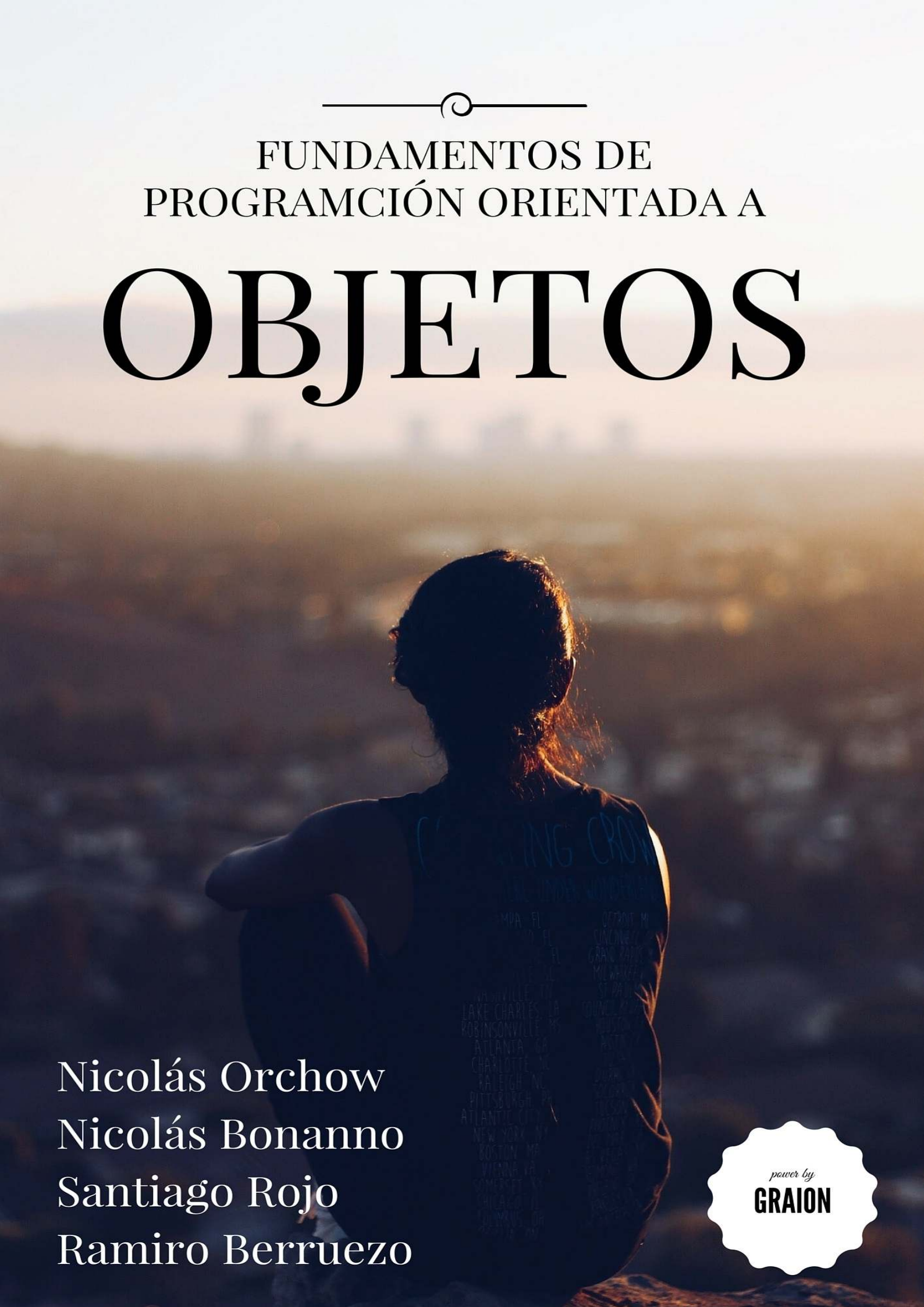
FUNDAMENTOS DE PROGRAMCIÓN ORIENTADA A

OBJETOS



Nicolás Orchow
Nicolás Bonanno
Santiago Rojo
Ramiro Berruezo

power by
GRAION

Tabla de contenido

Prefacio

Con el objetivo de generar un entendimiento compartido y un lenguaje con significados comunes entre los miembros de los equipos de desarrollo, en Graion nos hemos abocado a la tarea de recopilar, editar, escribir y curar el contenido de lo que consideramos necesario conocer y comprender para cualquier desarrollador o desarrolladora de software, que trabaje con lenguajes de programación, para lograr que siente sus bases sobre los conceptos de la programación orientada a objetos.

Así también, consideramos que uno de nuestros objetivos, al momento de escribir este texto, es que el mismo sirva como material de consulta para toda aquella persona que quiera comprender los conceptos fundamentales del paradigma.

Si bien explicaremos los conceptos del paradigma de objetos como tema central, también cubriremos una diversidad de conceptos que no son exclusivos de esta forma de diseño de software pero que sí consideramos importante entender para comprender conceptos relacionados.

Antes de empezar

Acerca del material

Como bien se sabe, en Internet existe una basta cantidad contenidos sobre POO, sin embargo, consideramos muy útil que las personas se sienten a revisar, mejorar y generar nuevos contenidos.

Es muy probable que se encuentren definiciones parecidas a las de otras bibliografías, ya que para generar este material consultamos muchos libros y apuntes de facultades y también, a escritores independientes de blogs.

Nuestro objetivo no es lucrar con este libro, sino simplemente devolver algo a la comunidad de desarrollo que tanto nos ha dado.

Acerca del lenguaje

A lo largo de este libro se muestran diferentes ejemplos de código, los mismos se encuentran en un lenguaje ficticio que decidimos crear por varios motivos.

- En primer lugar, para evitar inclinarnos por un lenguaje en particular, buscando con esto no expresar ninguna preferencia por parte de los autores.
- Por otro lado, también se busca generar abstracciones, sin preocuparnos por las implementaciones de cada lenguaje y mostrar un pseudocódigo que permita hacer foco sobre los conceptos tratados.
- Finalmente, el utilizar este lenguaje inventado, brinda más flexibilidad para generar los ejemplos.

Confiamos que luego de que el lector entienda los conceptos, busque la forma de implementarlos en el lenguaje que utiliza para trabajar o, simplemente, que es de su agrado.

Como leer este libro

Existen dos maneras que sugerimos usen para leer este libro.

A aquellos que se estén iniciando en el paradigma, les sugerimos leer el libro completo y sin saltar capítulos. Ya que para comprender un tema, primero se deben conocer los conceptos que se tocaron secciones anterior.

A los lectores que ya tengan conocimiento sobre la programación orientada a objetos, pueden simplemente consultar los capítulos que contienen los temas que desean ver en particular.

Conceptos Básicos

Comenzemos este camino interminable de aprendizaje con los conceptos básicos de POO (Programación Orientada a Objetos) y a medida que avanzemos vamos a ir tocando temas más complejos.

Objetos

¿Qué es un objeto?

La definición más básica dice: "Es un ente computacional que puede contener datos y comportamientos".

EJ: un usuario, una persona o un número entero. Cada uno de estos objetos tienen un comportamiento propio.

EJ: una persona puede comer/caminar/programar/atacar. Por lo tanto, un objeto lo podemos definir formalmente como: *Ente computacional que exhibe un comportamiento*.

Mensaje

Todos los objetos de los cuales hablamos tienen comportamientos y se relacionan con otros objetos (se piden cosas entre ellos). Por cual podemos definir a un mensaje como:

Interacción entre dos objetos: un emisor E y un receptor R. Un emisor le envía un mensaje a un receptor. El emisor puede obtener o no una respuesta.

Ej: caminar, pelear, comer, atacar.

Ciclo de vida de un objeto

Todos los objetos tienen una vida, se puede decir que nacen cuando son instanciados y mueren cuando se los elimina de la memoria.

No obstante, existen dos formas de eliminar de memoria a un objeto dependiendo del lenguaje de programación. Los mecanismos se llaman Garbage Collector y los Destructores.

Garbage Collector: es un mecanismo que se encarga de borrar de la memoria las referencias a objetos y entidades que ya no se usan más, de manera que se pueda maximizar el uso del espacio en memoria.

Destructores: son métodos que se definen para cada objeto y cuyo principal objetivo es liberar los recursos que fueron adquiridos por el objeto a lo largo de su ciclo de vida y romper vínculos con otras entidades que puedan tener referencias a él.

Métodos

¿Que es un método? ¿Qué diferencia existe con un mensaje?

Un método es la sección de código que se ejecuta al enviar un mensaje. Se identifica con una firma, que es la misma firma del mensaje enviado. Entonces, cuando un objeto recibe un mensaje, se ejecuta un método cuya firma es la misma que la del mensaje.

Ejemplo:

```
method comer(Tarta tarta){
    calorias += tarta.calorias
}
```

La firma de un objeto se define con tres componentes:

1. El nombre del método.
2. Los parámetros que recibe el método.
3. Lo que devuelve el método (que puede ser nada u otro objeto).

Clases

Muchos lectores se preguntarán porque no empezamos con la definición de Clase. Esto se debe a que **las clases solo son una forma de implementar objetos**, pero no son la única manera de hacerlo (como veremos más adelante), por lo que fundamental que no pensemos automaticamente en clases cuando hablamos de objetos. Sin embargo, la mayor parte de los lenguajes de programación que usamos laboral y académicamente usan clases. Por eso vamos a tratarlas en este libro.

¿Qué es una clase? Buena pregunta.

Podemos definir a una clase utilizando dos definiciones complementarias: **Una clase es un molde, a partir del cual se crean los objetos**. Cuando instanciamos un objeto, el ambiente le pregunta a dicha clase que características y métodos tiene que tener el objeto

que vamos a instanciar.

La otra definición es: **Una clase es un ente que determina el comportamiento y el tipo al que pertenecen sus instancias**.

Ejemplo de código:

```
class Persona {
    public attribute calorias
    public method comer(Tarta tarta){
        self.calorias += tarta.calorias
    }
}

class Tarta{
    public attribute calorias
}

class ComidasTests{
    method ComerTartaTest (){
        var caloriasDespuesDeAlmorzar = 600
        Persona alan = new Persona()
        alan.calorias = 500
        Tarta tartaJamonYQueso = new Tarta()
        tartaJamonYQueso.calorias = 100
        alan.comer(tartaJamonYQueso)
        Assert.AreEqual(caloriasDespuesDeAlmorzar,alan.calorias)
    }
}
```

Prototipos

La orientación a objetos basada en prototipos es un estilo de reutilización de comportamiento (herencia*) que se logra por medio de la clonación de un objeto ya existente, que sirve como prototipo. Javascript, es el lenguaje orientado a objetos basado en prototipos más conocido.

Autoreferencia

Así como un objeto puede conocer a otro objeto teniendo una referencia hacia este, también se puede conocer a sí mismo. Cualquier objeto tiene una auto-referencia, denominada this o self (según el lenguaje) para poder mandarse algún mensaje a si mismo.

Así como el self, hay otra referencia, denominada super o parent la cual es equivalente al this o self, con la particularidad de que le dice al Method LookUp “empezá desde 1 más arriba”. Es decir, que no busca la implementación en la clase donde está ejecutando el método, sino en la inmediata superior. (puede que eso desemboque en que siga subiendo niveles) En el 99% de los casos, sólo deberíamos llamar a super/parent para ejecutar el mismo método que estamos ejecutando (En algunos casos, como Ruby, el lenguaje mismo nos lo restringe). Es decir:

```
class Gato {
    method caminar(){
        super.caminar() //No debería hacer, por ejemplo, super moverPiernaDerecha()
        self.lamerse()
    }
}
```

Características de POO

Los siguientes conceptos que vamos a ver, son características que tiene un buen diseño orientado a objetos, si bien este paradigma no garantiza dichas características, hace mucho más fácil poder lograrlas y en cierta manera nos obliga a usarlas.

Abstracción

La abstracción es la propiedad de los objetos que consiste en tener en cuenta sólo los aspectos más importantes desde un punto de vista determinado y no tener en cuenta otros aspectos. Durante el proceso de abstracción es cuando se decide qué características y comportamientos debe tener el modelo para asi reducir su complejidad. De este modo, las características complejas se hacen más manejables.

Ejemplo:

En POO, podemos considerar una Persona como un objeto que tiene propiedades (como nombre, altura, peso, color de pelo, color de ojos, etcétera) y métodos (como hablar, mirar, andar, correr, parar, etcétera). Gracias a la abstracción, otro objeto Tren puede manipular objetos del tipo Persona sin tener en cuenta sus propiedades ni métodos ya que sólo le interesa, por ejemplo, calcular la cantidad de personas que están viajando en él en ese momento, sin tener en cuenta ninguna otra información relacionada con dichas personas, tales como la altura, el nombre, el color de ojos, etcétera. Nuestro objeto Tren se abstrae de objetos del tipo Persona.

Ocultamiento de la información

Este concepto hace referencia a que los componentes se deben utilizar como si sólo se conocieran su interfaz y no se tuviera conocimiento de su implementación. En otras palabras, un objeto sabe que otro objeto entiende un determinado mensaje, el cual recibe ciertos parámetros y devuelve algo (o no).

Ejemplo:

Yo como objecto médico quiero saber cuantas cirugías tuvo un paciente, y entonces le envía un mensaje para que el paciente me devuelva el número de cirugías que tuvo. El objeto médico no sabe como devolvió ese número el Paciente, solo sabe que tiene que enviarle el mensaje y recibir un número.

Encapsulamiento

Encapsulamiento es la capacidad de diferenciar qué partes de un objeto son parte de la interfaz y cuales permanecerán inaccesibles por el usuario. Son los lenguajes de programación los cuales, por medio los modificadores de acceso, permiten indicar el modo de accesibilidad de un componente.

En POO, a la conjunción de abstracción y ocultamiento de implementación se la llama **encapsulamiento**.

Según Booch, encapsulamiento “... es el proceso de almacenar en un mismo comportamiento los elementos de una abstracción que constituyen su estructura y su comportamiento; sirve para separar la interfaz contractual de una abstracción y su implementación”.

Veamos si se entiende con un ejemplo:

```
class Guerrero{
    attribute vitalidad
}
class Monstruo {
    method atacar(Guerrero barbaro){
        barbaro.vitalidad = barbaro.vitalidad - 10
    }
}
```

En este caso, el monstruo cuando ataca al guerrero lo debilita, cambiando el valor de la propiedad vitalidad directamente. Obviamente si escribimos una implementación como esta va a funcionar, pero existen diferentes razones por las cuales no nos conviene hacerlo así.

A nivel conceptual, nuestro objeto monstruo, no tiene porque conocer como el bárbaro maneja su vida, este podría hacerlo simplemente con un valor numérico o mismo con otro tipo de objeto. El monstruo solo tiene que mandar el mensaje y es ya cuestión del bárbaro entenderlo y accionar en base a ello.

Viéndolo desde el punto de vista de la escalabilidad y de un lenguaje de programación, si el día de mañana tenemos diferentes objetos que tiene que modificar dicha propiedad, nuestro código se puede volver difícil de cambiar debido a que en varias partes se usar la propiedad vitalidad (por ejemplo supongamos que ahora manejamos la vitalidad con un objeto Vida), si tenemos encapsulado ese comportamiento, solo vamos a tener que hacer un cambio en un solo lado.

Una forma más amigable de hacer esto sería:

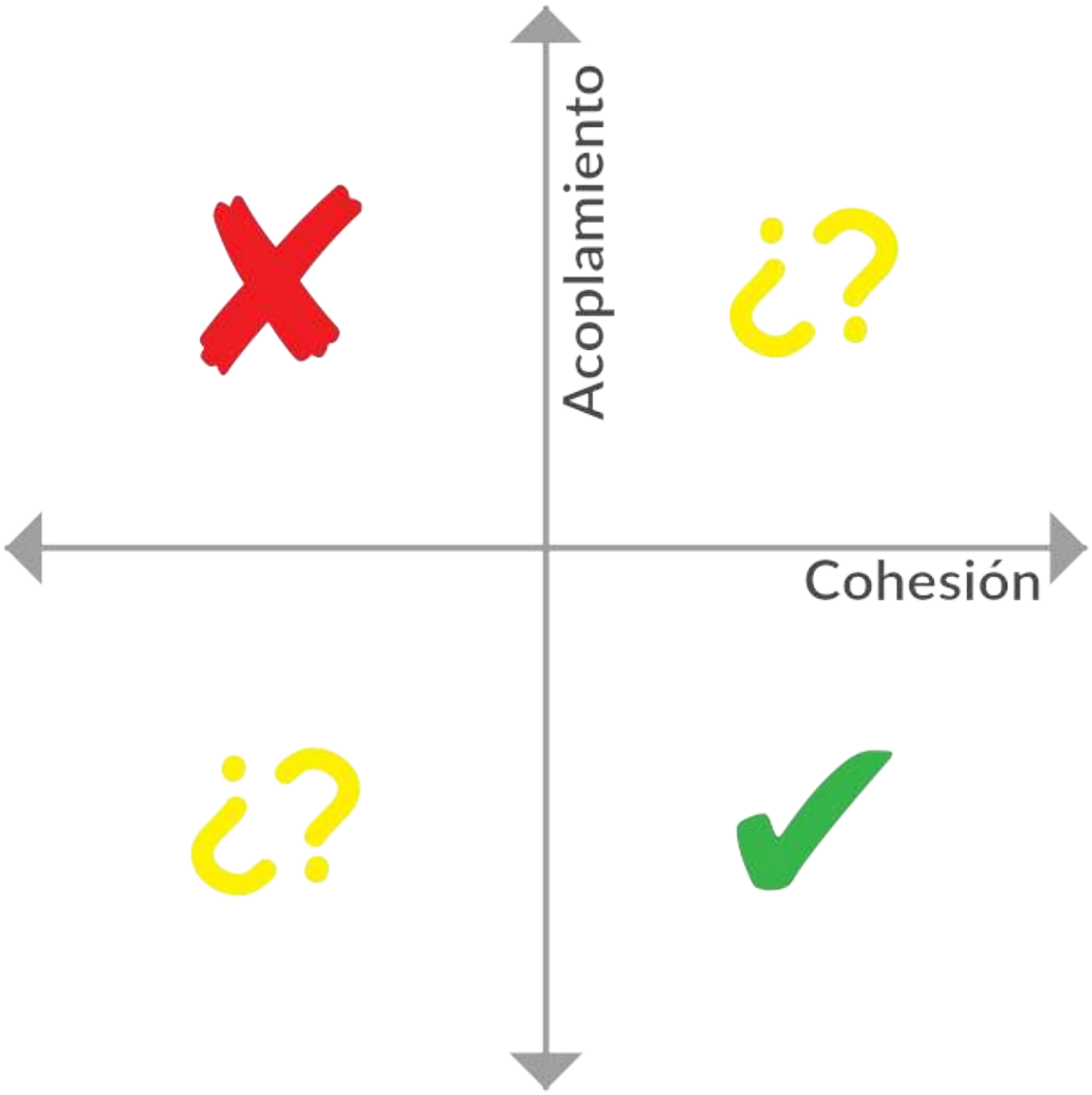
```
class Guerrero {
    attribute vitalidad
    method recibirAtaque(int puntuacionDeAtaque){
        self.vitalidad = self.vitalidad - puntuacionDeAtaque
    }
}

class Monstruo {
    attribute puntuacionDeAtaque
    method atacar(Guerrero barbaro){
        barbaro.recibirAtaque(puntuacionDeAtaque)
    }
}
```

Cohesión y Acoplamiento

La cohesión mide la relación entre el concepto que deseamos modelar y las responsabilidades del componente que lo representan. El acoplamiento mide qué tan relacionados están los componentes del sistema entre sí y cómo esa dependencia provoca un diseño.

Siempre buscaremos tener un bajo acoplamiento y que los objetos sean lo más cohesivos posible.

Claramente se puede ver que ambos conceptos están inversamente relacionados, nuestros diseños tienen que tener una alta cohesión y un bajo acoplamiento. El paradigma de objetos se basa en esta regla para generar diseños más sencillos, y al mismo tiempo fáciles de programar, probar y mantener.



Declaratividad y Expresividad

Estos conceptos no son particulares de POO, pero aun asi se nos olvidan, por eso les vamos a hacer un repaso.

La **expresividad** tiene que ver con qué tan claro es el código, qué tanto expresa lo que tenías en la cabeza cuando lo escribiste, es decir, que tan bien expresadas están las ideas del programador. Usar variables y métodos con nombres que representan lo que son o hacen, es forma fácil de aplicar expresividad que no tiene nada que ver con el paradigma sino con el programador.

Por lo general la **declaratividad** ayuda a que el código sea más expresivo, porque no se mezcla el algoritmo con lo que querés que haga más a alto nivel. Cuando tenemos un código bien declarativo, podemos entender que es lo que hace sin ver en detalle el algoritmo implementado. Es importante recalcar que la declaratividad es contrario a la imperatividad (osea, detallar linea a linea los pasos que hace el algoritmo).

Entonces, la declaratividad indica que se hace y la expresividad muestra la intención de lo que el código que se va hacer.

Recordemos una frase de Martin Fowler que vale la pena comentar: "*Cuando sientas la necesidad de escribir un comentario, intenta primero refactorizar el código de manera que cualquier comentario se convierte en innecesario.*"

Ejercicio

¿Qué les parece que hace este método?

```
method ElementoNuevo (var parametro1){
    if(parametro1.tipo = "R"){
        var numero = coleccion1.count()
        self.coleccion1[numero +1] = parametro1
    }else{
        var numero = coleccion2.count()
        self.coleccion2[numero +1] = parametro1
    }
}
```

¿Qué les parece que hace este otro método?

```
method Agregar(Soldado soldado){
    if(soldado.esDeReserva()){
        self.reservaDeEjercito.Add(soldado)
    }else{
        self.ejercito.Add(soldado)
    }
}
```

Claramente en el segundo ejemplo se entiende bien que estamos manejando un ejército y que nuestro método quiere agregar un nuevo soldado dependiendo de su tipo al ejército que luchara o al ejército de reserva.

Si tuviésemos que manejar modificar el primer método, tendríamos que hacer un análisis más profundo, revisando en el resto de nuestra aplicación qué significa la letra "R" en el tipo de soldado y para qué sirve la coleccion1 y coleccion2.

La expresividad y declaratividad son muy importantes para poder generar un código autodocumentado, ya que nunca sabemos en el futuro quienes van a tener que revisar nuestro código. Si un compañero que tiene que modificar nuestro código el día de mañana y no logra comprender fácilmente que quisimos hacer, entonces ahi tenemos un problema. Ahora, si nosotros mismos volvemos a un código que escribimos y no lo entendemos, entonces tenemos un grave problema.

Contratos

Diseño por contratos

El diseño por contratos asume que todos los componentes del cliente que invocan una operación en un componente del servidor van a encontrar las precondiciones (y postcondiciones) especificadas como obligatorias para esa operación.

Muchas veces podemos tener algunas cosas para validar en nuestro sistema. Esto es:

- Pre-condiciones
- Post-condiciones
- Condiciones “permanentes” o invariantes

Por ejemplo, yo siempre que alimente a mi mascota, debo garantizarme que no esté llena. Por lo que quisiera tener algo como:

```
public class Mascota implements Domesticable {
    public method alimentarse(){
        requires tengoHambre()
        //ejecutar método
    }
}
```

Lo mismo puede suceder con las post condiciones, o condiciones que se deben dar en todo momento.

```
public class Mascota implements Domesticable{
    method alimentarse(){
        requires tengoHambre()
        //ejecutar método
        guarantees estoyLleno()
    }
}
```

Para hacer cumplir estos contratos, hay 2 maneras:

1. Validación manual y lanzamiento de excepciónes - Custom y fácil
2. Integrado por el lenguaje (Eiffel) o algún framework (para PHP, Java, Ruby, etc) - Beneficios en cuanto a meta-data

Lo importante es que se entienda el concepto del contrato, de ver que yo para poder interactuar de cierta manera con un componente, debo cumplir ciertos requisitos, tanto antes, durante o después de la interacción.

Contratos & Herencia

Básicamente, aplican las mismas ideas y conceptos que en la herencia de comportamiento. Las precondiciones, postcondiciones e invariantes se heredan de clase a subclase.

Sin embargo existen algunos condicionamientos, para garantizar el principio de intercambiabilidad. Que se pueden resumir en la siguiente frase:

Require no more, and promise no less

Muy relacionado con la L (Liskov Substitution Principle) de SOLID, la cual plantea que donde uso una clase, debería poder usar cualquier subclase de ella y que siga funcionando todo correctamente.

Tipos

¿Qué es un tipo? Un tipo describe un conjunto de valores.

La idea de tipo nos permite relacionar:

- un conjunto de valores que tienen ese tipo o son de ese tipo,
- con las operaciones que pueden ser realizadas sobre esos valores.

Los objetivos de un sistema de tipos son:

- Ayudar a detectar errores al programar.
- Guiar al programador sobre las operaciones válidas en un determinado contexto, tanto en cuanto a documentación como en cuanto a ayudas automáticas que puede proveer por ejemplo un IDE.
- En algunos casos el comportamiento de una operación puede variar en función del tipo de los elementos involucrados en la misma. Polimorfismo, sobrecarga (veremos más adelante), multimethods, etc.

Podemos hacer 3 clasificaciones de tipado:

- Implícito o explícito. Un lenguaje va a tener tipado explícito si:
    - Todos los elementos (variables, métodos, etc) tienen un tipo definido.
    - Para que dos objetos sean polimórficos, debo explicitarlo (por una interfaz o por heredar de la misma clase).
- Chequeo dinámico o estático.
    - La diferencia está en el momento en que es ejecutado el chequeo de tipos. En los lenguajes de chequeo estático (como Java o C#) se chequea en tiempo de compilación, mientras que en los de chequeo dinámico (como Ruby, PHP o JS) se realiza en tiempo de ejecución.
    - El chequeo dinámico da lugar a un concepto famoso denominado Duck Typing. El cual se basa en que “si algo tiene pico de pato, camina como pato, y hace cuack, es un pato”. Es decir, toma sentido el tipo al que pertenece un objeto según qué mensaje puedo mandarle y cómo responde, no se me especifica desde antes.
- Estructural o nominal
    - No nos vamos a detener mucho en este. Hace referencia a si se identifica un tipo por su nombre o por su estructura. Es muy común en los lenguajes del paradigma funcional (otro paradigma).

Combinaciones más frecuentes:

- Explícito, estático y nominal
- Implícito y dinámico

Casteo

Es un mecanismo utilizado en los lenguajes de chequeo estático, por el cual nosotros le “aseguramos” al compilador que cierto objeto pertenece al tipo especificado.

Ejemplo:

```
//Suponemos que los métodos de la API siempre reciben un Object
method recibirPerroDeApi(Object obj){
    var perro = (Perro) obj
    perro.ladrar()
}
```

Hay que ser muy cuidadosos al usar el casteo, porque podría llevar a “confundir” al compilador y hacer que rompa en tiempo de ejecución (perdiendo el beneficio que nos da el chequeo estático), por ejemplo si en Object que se recibe es un Gato en vez de un perro, al llamar al método "ladrar" nuestro programa lanzará una excepción.

Herencia y Composición

Sobrecarga de métodos

La sobrecarga es la capacidad de un lenguaje de programación, que permite nombrar con el mismo identificador diferentes variables u operaciones.

La sobrecarga de métodos se refiere a la posibilidad de tener dos o más métodos con el mismo nombre pero diferente firma. El compilador usará una u otra dependiendo de los parámetros usados.

El mismo método dentro de una clase permite hacer cosas distintas en función de los parámetros.

Ejemplo

```
class Articulo {
    private attribute precio
    public method setPrecio() {
        precio = 3.50
    }
    public method setPrecio(Currency nuevoPrecio) {
        precio = nuevoPrecio
    }
}
```

En este ejemplo que vemos, uno podría invocar al método "setPrecio" enviando un precio o sin parámetros, dependiendo como se invoque en nuestro código, el compilador decidirá si llama a la primer implementación o a la segunda.

En tiempo de compilación, se buscan todas las llamadas a este método y según el tipo de los parámetros con los que se esté invocando y el objeto que puede o no devolver (necesariamente son lenguajes de chequeo estático), se determina a qué implementación llamará. Si la combinación de parámetros no es excluyente, el compilador falla.

Los multimethods, a diferencia de la sobrecarga, son un conjunto de métodos con la misma firma, pero que se pueden solapar, y se decide cuál ejecutar en tiempo de ejecución.

Composición

La composición se refiere a la combinación de objetos simples para hacer objetos más complejos.

Los objetos a menudo pueden dividirse en tipos compuestos y componentes, y la composición puede considerarse como una relación entre estos tipos: un objeto de un tipo compuesto (ej.: auto) "tiene un" objeto de un tipo simple (ej.: rueda).

Considere la relación de un automóvil con sus partes, a saber: el automóvil tiene o se compone de objetos como el volante, asientos, caja de cambios y el motor. Esta relación podría definirse como una relación de composición.

Los objetos compuestos generalmente se expresan por medio de referencias de un objeto a otro. Tales referencias pueden ser conocidas como atributos, campos, miembros o propiedades y la composición resultante como tipo compuesto. Sin embargo, tener esas referencias no necesariamente significa que un objeto es compuesto. Sólo se llama compuesto, si los objetos de los que se refiere son realmente sus partes, es decir, no tienen existencia independiente.

Herencia

Es el mecanismo por el cual un objeto se basa en otro objeto o clase, extendiendo la implementación para reutilizar el comportamiento.

Supongamos que los gatos y los perros, por ser cuadrúpedos, caminan de la misma manera:

```
class Perro{
    public method caminar(){
        moverPataDelanteraDerecha()
        moverPataDelanteraIzquierda()
        moverPataTraseraDerecha()
        moverPataTraseraIzquierda()
    }
}

class Gato{
    method caminar(){
        moverPataDelanteraDerecha()
        moverPataDelanteraIzquierda()
        moverPataTraseraDerecha()
        moverPataTraseraIzquierda()
    }
}
```

Esto puede traer varias complicaciones:

- Cada vez que quiero agregar un cuadrúpedo, tengo que copiar las mismas 6 líneas de código (y todos sabemos que cada 2 copy paste de código, muere un gatito).
- Si por alguna razón todos los cuadrúpedos empiezan a caminar primero con la izquierda, tenemos que cambiarlo en N lugares.

Quisiera entonces, poner todo este comportamiento en algún otro lado, y poder llamarlo desde todos los cuadrúpedos. Esta es la verdadera motivación de la herencia. La de reutilizar código. Para eso, vamos a implementar el método en una superclase y todos los que hereden de esa superclase, podrán usarlo también.

```
class Cuadrupedo{
    method caminar(){
        moverPataDelanteraDerecha()
        moverPataDelanteraIzquierda()
        moverPataTraseraDerecha()
        moverPataTraseraIzquierda()
    }
}

class Gato extends Cuadrúpedo{
    ...
}
```

¿Cómo funciona?

El compilador, cuando le llega un método a un objeto, hace lo siguiente:

1. Busco la implementación en la clase del objeto al que le llegó el mensaje (Gato)
2. Si no la encontré, busco en su padre (en nuestro ejemplo Cuadrupedo)
3. Repetir el paso anterior hasta que no haya más padres.
4. Si no encuentra ninguna implementación en todos los niveles, lanza una excepción.

A este mecanismo se lo conoce como **Method LookUp**.

Algo a destacar de la herencia es que las clases hijas van a heredar TODO lo que haya definido su superclase, tanto atributos como métodos. No puede heredarse sólo una parte.

Las motivaciones de utilizar herencia van a ser entonces:

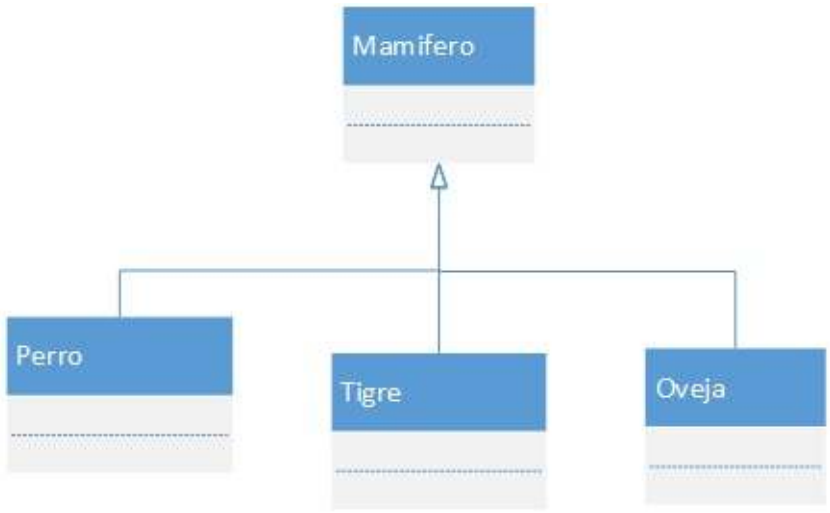
- Reutilizar código.
- Generar una abstracción.
- Proveer un Tipo (para lenguajes con tipado estático).

Herencia Vs Composicion

Uno de los objetivos que buscamos cuando programamos es reutilizar métodos y funcionalidades, para lograr una mayor mantenibilidad. Dentro de los lenguajes convencionales orientados a objetos existen varias formas de hacer esto, las dos más conocidas son: **herencia de clases** y **composición**.

Herencia de clases

También conocido como “reutilización de caja blanca”, debido a la visibilidad que dan ya que mediante la herencia la implementaciones de las clases padres se hacen visible a las clases hijas.



Ventajas

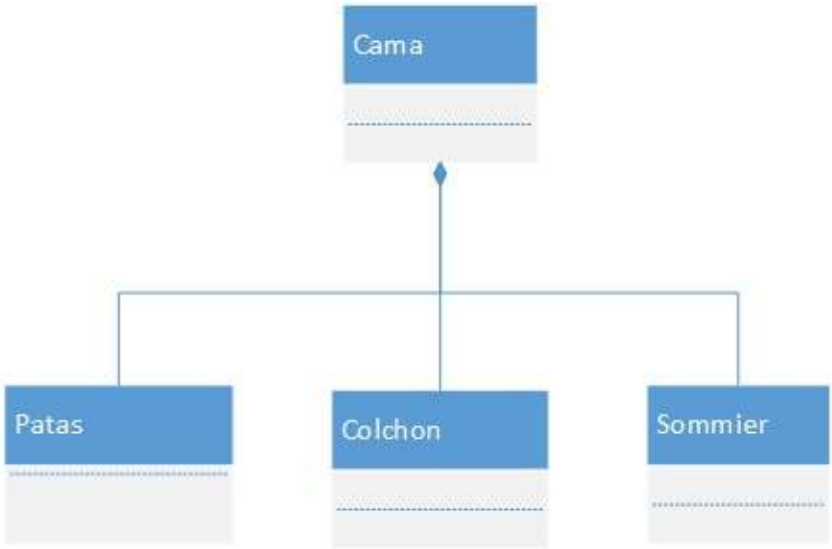
- Se define en tiempo de compilación y no tiempo de ejecución, es decir que conocemos con seguridad que se va a ejecutar.
- Es más sencillo de modificar la implementación que está siendo reutilizada.

Desventajas

- Como dijimos antes, se definen en tiempo de compilación, por lo que no se puede cambiar el comportamiento de un objeto en tiempo de ejecución, lo cual puede ser un requisito de nuestro diseño.
- Se rompe el concepto de encapsulación al exponer los detalles de la implementación en la clase padre.

Composición

Conocida como “reutilización de caja negra”, a diferencia del caso anterior nosotros no conocemos los detalles de la implementación, ya que la misma se encuentra encapsulada en el objeto al cual estamos invocando.



En el ejemplo, la cama está compuesta por patas, un colchón y sommier.

Ventajas

- Al tener objetos que hacen referencia a otros objetos, el vínculo se define en tiempo de ejecución y como a dichos objetos se accede mediante su interface no se rompe el principio de encapsulación (entonces podríamos reemplazar al objeto por otro cuando corre nuestro programa y tener un comportamiento totalmente distinto!).
- Permite tener clases más centradas y encapsuladas, haciendo que nuestro diseño tenga más objetos (menos clases) de menor tamaño y con menos responsabilidades.

Conclusión

Se podría decir que conviene favorecer la composición por sobre la herencia, sin embargo esta respuesta no es definitiva, cada uno tiene que identificar la necesidades y el dominio en el cual está trabajando para identificar en casos conviene usar cada metodología.

Algunas recomendaciones

- En lenguajes que no soportan herencia múltiple, puede resultar ventajoso la composición.
- La composición genera un diseño más desacoplado, que puede ayudar a hacer el testing (o TDD) mas fácil.
- Muchos patrones de diseño favorecen la composición.

Polimorfismo

Es la capacidad que tiene un objeto de poder tratar indistintamente a otros que sean potencialmente distintos, es decir, es la capacidad que tienen distintos objetos de entender un mismo mensaje.

Ejemplo

Hay una persona, que siempre que llega a su casa interactúa con su mascota. Como su mascota es un perro, lo que hace es jugarle.

```
class Persona {
    public attribute mascota
    public method llegarACasa(){
        mascota.ladrar()
    }
}

class Perro {
    public method ladrar(){
        ...
    }
}
```

¿Qué pasa si ahora si la persona vende al perro y se compra un gato? Debemos saber interactuar con ambos tipos de animales.

```
class Persona {
    //Método no polimórfico
    public method llegarACasa(){
        if(mascota isA Perro)
            mascota.ladrar()
        else if(mascota isA Gato)
            mascota.maullar()
    }

    //Método polimórfico
    public method llegarACasa(){
        mascota.saludar()
    }
}
```

Este concepto se conoce como **polimorfismo**. Es la capacidad de intercambiar un objeto con otro, abstrayendo al que lo conoce de su implementación.

¿Por qué es importante diseñar polimórficamente?

Para ganar extensibilidad. Sino, cada vez que aceptemos otro animal como mascota, vamos a tener que modificar la clase. (Rompemos con la O de S.O.L.I.D.) En cambio, si lo mantenemos, cuando agregue una mascota, es suficiente con crearle una implementación de estos métodos para que encaje perfectamente en el sistema.

Entonces, para hacerlo genérico, se establece un contrato: "Todas las mascotas deben entender el mensaje interactuar". Agregamos algo más, no queremos interactuar con nuestra mascota si está dormida, por lo que: "Todas las mascotas deben evitar interactuar si están dormidas".

En general, un contrato establece un acuerdo entre dos (o más) partes. Si lo cumplimos el sistema va a tener una funcionalidad dada, o un comportamiento. De alguna forma regula la interacción entre dos módulos de nuestra aplicación. Entonces intercambiando un módulo por otro que cumple el mismo contrato debería ser transparente para el otro módulo.

La parte del contrato más frecuentemente usada, va a ser lo que denominamos interfaz. La interfaz de un objeto es el conjunto de mensajes que entiende.

Adicionalmente, están las construcciones específicas denominadas interfaces. Estas, no son más que una especificación del conjunto de mensajes que tienen que cumplir aquellos que la implementen.

Va de nuevo, supongamos que todos los animales “Domesticables” son aquellos a los que se puede alimentar o se puede jugar con. Podríamos definir la interfaz Domesticable de la siguiente manera:

```
interface Domesticable{
    method alimentarse()
    method jugar()
}
```

Nótese que no le definimos ningún comportamiento o implementación. Lo único que establecemos es “Todos aquellos que quieran ser domesticables, tendrán que poder alimentarse y jugar”.

Una buena thumb rule para identificar la necesidad de una interfaz es establecer un adjetivo común a los objetos que quiero que la implementen (Domesticable/Amable/Convertible), a diferencia de una superclase (veremos más adelante), las cuales suelen ser sustantivos (Animal/Persona/Auto)

Clase Abstracta

Cuando vimos lo de las pre-condiciones, apareció la clase Mascota..

```
class Mascota implements Domesticable{
    method alimentarse(){
        requires noEstoyLleno()
        //ejecutar método
    }
}
```

Pero, ¿Va a haber algún objeto Mascota o todos van a ser Perros/Gatos/Iguanas/etc?

En este caso, la clase Mascota no debería ser instanciable, solo quiero utilizarla para que las clases que heredan de esta puedan usar el código definida en ella.

A esto se lo llama una **clase abstracta**. Al contrario de las abstractas, las clases que sí son instanciables, se las denomina concretas.

Una clase abstracta define sólo la interfaz o firma de algunos de sus métodos. Podemos tener:

- clases abstractas puras (¿interfaces*?).
- clases parcialmente abstractas.

Manteniendo el concepto, una clase abstracta puede obligar a que sus subclases implementen cierto método, pero sin definir ningún comportamiento por default. A estos métodos, naturalmente, se los denomina métodos abstractos.

Si una clase tiene al menos UN método abstracto, entonces debe ser abstracta, porque no tiene sentido que haya un objeto que sea instancia de ella.

A fines de comprensión de concepto, podría decirse que una interfaz es una clase abstracta con todos sus métodos abstractos.

Interfaces

Una interfaz define un **tipo** y es una colección de **métodos abstractos**. De forma similar a una clase abstracta, obliga a quienes implementen la interfaz a implementar los métodos. Las interfaces no pueden definir la implementación de los métodos de los objetos que las van a utilizar, pero si obligan a dichos objetos a definir esa implementación.

Las interfaces sirven para solventar la limitación de muchos lenguajes, que no soportan la herencia múltiple.

Clases abstractas puras vs interfaces

- Solo se puede extender de una clase abstracta, pero se pueden implementar más de una interfaz en una clase.
- Un interfaz no puede definir atributos.
- Así como no se puede crear instancias de clases abstractas, tampoco es posible crear instancias de interfaces.

Bonus Tracks

Los siguientes temas, ya no están tan relacionados con POO, pero también nos parecieron interesantes y decidimos mencionarlos brevemente para que el lector se lleve un poco más.

Excepciones

¿Cuántas veces recibimos un mail de un cliente diciendo que apareció una pantalla con un mensaje de error poco descriptivo?

Esto probablemente se deba a un bajo nivel de detalle en el manejo de excepciones dentro de nuestra aplicación. El manejo de excepciones consiste en controlar los errores que surjan dentro de nuestro sistema para poder tratarlos debidamente.

Tratarlos debidamente implica:

- Mostrar un mensaje más amigable a los usuarios (la más común).
- Hacer un rollback de alguna transacción.
- Escribir en un log el error para analizarlo.
- Si el error no afecta el flujo de trabajo: contenerlo para que el flujo siga.

Y así podemos seguir durante un buen rato...

Ejemplo básico:

```
public method GuardarInformacionImportante(Factura factura) {
    try {
        // Se ejecuta algo que puede producir una excepción
        // Un error en la base, la famosa división por cero, una referencia a un null, et
    } catch (DividedByZeroException e) {
         // manejo de una excepción de una división por cero
    } catch (Exception e) {
         // manejo de una excepción cualquiera
    } finally {
         // código a ejecutar haya o no excepción
    }
}
```

La mayoría de los lenguajes de programación soportan estos tipos de sentencias. Vale la pena resaltar la línea “catch (Exception e)”, esta sentencia es casi obligatoria, ya que su función es atrapar cualquier tipo de excepción que no allá sido atrapada previamente. En nuestro ejemplo anterior solamente “atrapamos” las excepciones que sean del tipo “división

por cero”, si dentro de la sección try se hubiese lanzado otro tipo de excepción como un error de conexión a la base de datos, este error habría sido atrapado en la sección de excepciones genéricas.

Forzando una excepción

Muchas veces existen ocasiones donde existen errores que si bien no hacen que nuestra aplicación explote por los aires, si afectan nuestro flujo de negocios, es por eso que muchos lenguajes permiten lanzar excepciones manualmente y nuestro programa las va a ver como si fueran excepciones comunes. Ejemplo:

```
public method void IrATrabajar(Dia dia, Trabajador trabajador) {
    try {
        if(dia.EsLunes() && trabajador.SalioElFinde())
            throw new Exception()
    } catch (Exception e) {
         Interface.MostrarError(“El trabajador está enfermo”);
    } finally {
         // código a ejecutar haya o no excepción
    }
}
```

Creando nuestras propias excepciones

Así como podemos forzar excepciones, también podemos crear nuestros propios tipos de excepciones, conocidas como “User-Defined-Exceptions”. ¿Para que? Para poder definir comportamientos propios de excepciones dentro de nuestro dominio. Es decir, supongamos que estamos haciendo un método que registre a personas físicas en nuestro sistema, pero que no soporte personas jurídicas!

Primero definimos nuestro tipo de excepción:

```
public clase EsPersonaJuridicaException: Exception
{
    public EsPersonaJuridicaException()
    {
    }
}
```

Luego la usamos:

```
public metodo RegistrarPersona(Persona persona) {
    try {
        if(persona.EsJuridica())
        throw new EsPersonaJuridicaException()
    } catch (EsPersonaJuridicaException e) {
         Interface.MostrarError(“No se pueden registrar personas jurídicas”)
    }catch (Exception e) {
         Interface.MostrarError(e.Message)
    } finally {
         // código a ejecutar haya o no excepción
    }
}
```

En este caso parece medio trivial definir un tipo excepción para algo tan trivial, pero cuando el tipo de excepción se da en varios casos y tiene un comportamiento más elaborado, se justifica crear nuestro propio tipo de excepción.

Un detalle fundamental que no debemos olvidar, es que nuestro tipo de excepciones deben heredar de la clase Exception, ya para poder utilizar la sentencia try nuestras Excepciones deben heredar las propiedades de una excepción básica.

Colecciones

A veces necesitamos tener algo que no es un solo objeto, sino que en sí mismo contenga 0, 1 o varios objetos. Estos “elementos”, los vamos a llamar genéricamente Colecciones y también son objetos.

Eso significa que como cualquier otro, va a poder recibir mensajes y hasta le podríamos generar nuevos métodos para que entiendan (peligroso) y mejor aún, crear nuestro propio tipo de Colección que cumpla con lo que queramos.

Cabe destacar, que en los lenguajes tipados, todos los elementos de una colección deben ser del mismo tipo, es decir, deben (al menos) implementar la misma interfaz. Genéricamente, los elementos de una colección deben ser polimórficos, según lo que quiera hacer con ellos.

La ventaja de usar los métodos de colecciones por sobre la iteración por elemento, es ganar en declaratividad y expresividad, dejando un código más fácil de entender, modificar y escalar.

¿Qué cosas podemos querer hacer con una colección?

- Agregarle elementos.
- Saber el tamaño.

- Hacer un filtro o selección de los elementos según un criterio.
- Recorrerla y hacer algo con cada elemento.
- Preguntarle si contiene o no un determinado objeto.
- Ordenarla.

Agregar elementos

Este método es bastante trivial. Si bien depende del lenguaje, suele haber un método push/add o similares.

Saber el tamaño

También es algo bastante trivial. El método suele ser size o length. Un caso de uso bastante común es saber si una lista está vacía. Hay algunos lenguajes que tienen el mensaje isEmpty o similar, y sino, simplemente hacer if(!colec.length).

Filtrar

Acá empieza la ventaja de usar los métodos de las colecciones frente al paradigma imperativo. ¿Qué podríamos hacer de forma imperativa si queremos filtrar elementos según una condición?

```
public method algunFiltrado(){
   t_objeto colecaux[];
   for (int i = 0; i <= colec.length; i++) {
      if(colec[i].edad >= 18) colecaux[] = colec[i];
   }
   return colecaux;
}
```

Como vimos antes, esto tiene muchas bajas en cuanto a expresividad y declaratividad.

Por esta razón, las colecciones nos suelen proveer un método filter o similar que recibe una condición y devuelve los elementos de esa colección que la cumplan. Si bien se puede filtrar por un campo, también podría filtrarse por un método o un conjunto de condiciones que devuelvan un booleano.

El mismo caso de uso, utilizando el método filter, se vería de la siguiente manera

```
public method algunFiltrado(){
    return colec.filter(mayorDeEdad);
}

function mayorDeEdad(Persona persona){
    return persona.edad >= 18
}

public method algunFiltrado(){
    return colec.filter(\x -> x.edad >= 18); //Lambdas FTW
}
```

Recorrerla

Hay un método que denominamos genéricamente (y por ser usado por la mayoría de los lenguajes): map

Es un método que entienden las colecciones y que también recibe como parámetro un ejecutable. Léase bloque para Smalltalk/Ruby, función para JS, callable para php, etc. Este ejecutable, va a recibir un elemento de la colección, y puede o no retornar otro elemento. Devuelve otra colección con los cambios correspondientes según lo que se ejecute en el parámetro. (si no se devuelve nada, devuelve la misma colección)

Ejemplo de uso en JS:

```
[1,2,3,4].map(function(num) {
    return num*2;
});
```

Desde adentro de la función que recibe el map, uno podría querer cambiar alguna variable externa. Si bien en muchos lenguajes se permite, hay que tener cuidado con el scope.

Saber si un elemento pertenece a una colección Este método es muy similar al filter, pero en lugar de devolver una colección con los elementos que cumplen la condicíon dada, devuelve solo uno. Suele denominarse find.

Ordenarla

El método (genéricamente denominado sort) tiene como objetivo devolver una colección ordenada por un criterio. También es de orden superior, y en general recibe una función con 2 argumentos, los cuales representan a 2 elementos de la colección. La función debe determinar la condición para saber si una va antes que la otra (ya sea con un booleano o si un número es mayor o menor a 0).

```
products.sort( function(a, b) {
   return a.price - b.price; //ascending order
});

products.sort( function(a, b) {
   return a.price > b.price; //ascending order
});

products.sort('price'); //solo para atributos
```

Como mencionamos antes, la mayor potencia de todos estos métodos, además de su expresividad y declarativiad, es la facilidad de encadenarlos.

```
public method montañaRusa(){
    return colec
    .filter(midenMasDe(1,50))
    .sort('llegada')
    .map(function(jugador){ jugador.sacudir() });
}
```

Variables y métodos de clase

También se denominan métodos y variables estáticas. La implementación de los mismos varía dependiendo del lenguaje. En algunos (como Ruby o Smalltalk) las clases son objetos, en otros (como Java o PHP), las clases son entes particulares con una instancia asociada, que se pueden utilizar de determinada forma y enviarles mensajes y en algunos (JavaScript) no se utiliza ninguna de estas nociones, pero siguen teniendo el mismo concepto.

Las variables de clase nos sirven cuando queremos que nuestros objetos tengan alguna referencia a algún valor, que sea el mismo para todas las instancias de esa clase, y que además pueda cambiar (por eso usamos una variable y no hardcodeamos ese valor en el código del programa, asumiendo que es posible).

Si usáramos una variable de instancia con la intención de settear el mismo valor a todas las instancias de esa clase, y luego queremos cambiar el valor de modo de afectar a todas las instancias, tengo que poder encontrar todas las instancias ya existentes para poder mandarles el mensaje para que actualicen su referencia... Y si tengo muchos muchos objetos eso no está bueno, no sólo por la complejidad innecesaria del problema, sino también porque la performance puede verse afectada

Los métodos de clase, son métodos cuyo receptor / implementador va a ser una clase en lugar de un objeto. El caso más común es el del constructor, donde la clase recibe el “new” y ella sabe generar una nueva instancia e incluso llamar al constructor correspondiente.

Otro caso recurrente es el de querer tener una sola instancia por clase. Para profundizar más en este funcionamiento, ver el patrón de diseño Singleton.