Herencia

Bertrand Meyer en su libro *Touch of class* brinda un ejemplo muy claro y simple: analiza cómo la botánica necesitó clasificar de la mano de Linneo las especies vivientes. Esto muestra cómo "objetos" de la naturaleza obtienen una taxonomía artificial con el fin de poder estudiarlos.

Por otro lado, los objetos de la matemática (números, funciones, series) son creaciones humanas. Con la venida de Cantor y sus teorías de conjuntos y grupos, se organizaron estos objetos artificiales.

Como programadores nosotros también lidiamos con entidades artificiales, objetos salidos de nuestra imaginación. Para evitar el desorden que fácilmente podemos ocasionar, tenemos a la mano la posibilidad de utilizar una taxonomía.

Es así como organizamos nuestras clases de acuerdo a una relación jerárquica denominada **herencia**. Del mismo modo en que un perro *es un* mamífero y un mamífero *es un* vertebrado, nuestro objeto estudiante *es una* persona.

> La herencia nos habilitará a razonar sobre una relación "es un" y utilizar las taxonomías resultantes para estructurar nuestro software.
> Bertrand Meyer [MEYER-2009]

Puesto en esos términos, la herencia es un mecanismo que nos proveen *algunos* lenguajes orientados a objetos para definir relaciones jerárquicas entre clases, y que nos permiten estructurarlas de acuerdo a un sentido de genericidad / especialización.

Breve ejemplo: **Un Estudiante es una Persona**. Estudiante tiene características específicas que una Persona no tiene. La relación entre Estudiante y Persona será:

- *generalización*, si se lee como "un Estudiante es una Persona"
- *especialización*, si la vemos como "algunas Personas son Estudiantes"

Consecuencia de esta relación de especialización/generalización y de la estructuración de las clases obtendremos la posibilidad de compartir comportamiento, ya sea idéntico, modificado, aumentado o totalmente diferente.

Herencia "clásica"

> **Nota:** Se denomina herencia clásica por ser la variante de herencia que se efectúa *entre clases*, y no en el sentido de clásica como "convencional".

En un lenguaje con herencia de clases, son éstas las que definen de qué otra u otras clases heredan, dejando establecido de un modo explícito (y generando un acoplamiento jerárquico) cuál será su estructura y su posición en la taxonomía

de tipos que se está generando. Esto no significa ni más ni menos que al hacer lo siguiente...

```
class Estudiante extends Persona { ... }
```

... estaremos definiendo una clase `Estudiante` que declara extender el tipo `Persona`, brindando una especialización del mismo.

Cuándo utilizar la herencia

La herencia, como toda relación que deseemos establecer entre nuestras clases con el fin de resolver un problema, debe utilizarse cuando ésta sea la solución natural y no una estrategia forzada. Obvio como parece, es muy difícil de cumplir dado que se requiere de cierta experiencia y de una visión global del sistema para notar que no se están introduciendo acoplamientos ni rigideces innecesarias que sólamente dificultarán nuestro trabajo a futuro.

Para bajar este concepto a algo práctico tomaremos un enfoque basado en el comportamiento de nuestros tipos.

Ejemplo: descubriendo herencias

Comenzaremos con un sistema que nos pide administrar el transporte y la entrega de paquetes. Estamos encargados de modelar el desgaste del medio de transporte utilizado para los envíos, y la evidencia obtenida de las charlas con los usuarios indica que sólamente se entregan paquetes en bicicleta. Nuestro código será similar a esto:

```
class Transporte {
  // omitiremos los atributos
  Double calcularIndiceDesgasteTotal() { ... }
  Boolean debeArreglarse() { ... }
  void agregarPaquete(Paquete paquete) { ... }
  Double capacidadRestante() { ... }
}
```

Como puede verse, preferimos evitar el modelado del medio de transporte específico (tenemos un cierto sentido de los posibles ejes de cambio del sistema), aunque no introdujimos clases adicionales al mismo.

Un pequeño reconto de los métodos nos permite saber que:

- `calcularIndiceDesgasteTotal():Double` devolverá un número entre 0 y 1 que nos indicará el desgaste del vehículo.
- `debeArreglarse():Boolean` se basará en cuán desgastado está para indicar si debe o no pasar por el taller.
- `agregarPaquete(Paquete):void` permitirá agregar carga.

- `capacidadRestante():Double` nos informará cuántos kilogramos pueden cargarse aún.

Con esto cumplimos las funcionalidades pedidas por el cliente. Entregamos el sistema, y todo funciona de maravillas... hasta que un tiempo después decide que también desea registrar los envíos que hacen a distancias ligeramente mayores. Y por supuesto, ya no se hacen en bicicleta, sino con automóviles.

Sin entrar en pánico, procedemos a estimar el cambio y a efectuarlo. Lo que haremos, de un modo conservador, es **esperar evidencias suficientes antes de introducir un acoplamiento excesivo**. Por ello, comenzamos a introducir nuestra herramienta básica: *el if*.

```
class Transporte {
  String tipo = "bicicleta";
  Transporte () { }
  Transporte (String tipo) { this.tipo = tipo; }
  Boolean debeArreglarse() {
    Double valorLimiteDesgaste
    if ("bicicleta".equals(this.tipo)) {
      valorLimiteDesgaste = 0.3;
    } else {
      valorLimiteDesgaste = 0.5;
    }
    return this.desgasteActual < valorLimiteDesgaste;
  }
  // se omiten los otros métodos para simplificar el ejemplo
}
```

De este modo, introduciendo un discriminante, podemos salvar la situación de un modo simple y sin cambiar la interfaz de los métodos. Aunque si necesitamos crear un `Transporte` que represente a un automóvil, deberemos indicárselo por el constructor. Esto ya nos está *oliendo raro* pero el sistema funciona según lo requerido.

No se hace esperar el cambio: los envíos de larga distancia se realizan por medio de camiones. Éstos tienen mayor capacidad y resistencia al desgaste.

No podemos seguir con nuestro modelo de discriminantes ya que tenemos evidencia suficiente de que los tipos de `Transporte` son varios, y su comportamiento es diferente según cuál sea. Adicionalmente, las clases que queremos introducir pasan la prueba del "es un":

- Una `Bicicleta` es un `Transporte`
- Un `Automovil` es un `Transporte`
- Un `Camion` es un `Transporte`

  Nota: Es importante que tengamos en cuenta tanto la taxonomía al establecer una herencia, como los comportamientos a los que estarán suscribiéndose nuestras clases mediante esa relación.

Ahora podemos modificar nuestro código, permitiendo ese eje de cambio:

```
class Transporte {
  Double calcularIndiceDesgasteTotal() { ... }
  Boolean debeArreglarse() { ... }
  void agregarPaquete(Paquete paquete) { ... }
  Double capacidadRestante() { ... }
}

class Bicicleta extends Transporte {
  Double calcularIndiceDesgasteTotal() {
    // implementación específica de la bicicleta
  }
  // ...
}

class Automovil extends Transporte {
  Double calcularIndiceDesgasteTotal() {
    // implementación específica del automóvil
  }
  // ...
}

class Camion extends Transporte {
  Double calcularIndiceDesgasteTotal() {
    // implementación específica del camión
  }
  // ...
}
```

Ahora bien, nos quedan algunos arreglos que hacer en este código. Quedarán en manos del lector, pero plantearemos una serie de preguntas que proporcionan una guía en ese sentido:

- ¿Es correcto que existan ejemplares de `Transporte`? En otras palabras, no sería necesaria la posibilidad de instanciar `Transporte`: estamos utilizando el concepto para poder derivar la jerarquía especializada. Podemos solucionar el problema declarándola como *clase abstracta*.
- Notamos que mucho código podría repetirse en cada subclase. Por ejemplo, el método `debeArreglarse():Boolean` es siempre igual, aunque toma otro valor de referencia. En este caso necesitamos comenzar a reutilizar comportamiento. Si el método se definiese en forma suficientemente genérica a nivel `Transporte`, y se dejasen los "huecos" de la implementación específica (como ser el `valorLimiteDesgaste`) no necesitaríamos reescribir líneas de código idénticas, aumentando la mantenibilidad de nuestro software.
- El paso intermedio utilizando el discriminante ya no es necesario, por lo que debemos removerlo. Esto trae cambios en módulos cliente, debiendo quitar

líneas del tipo `new Transporte('automovil')` y reemplazándolas por `new Automovil()`. Si hubiésemos querido evitar ese problema tendríamos que haber aplicado un patrón de diseño llamado **Abstract Factory**.

- El uso de condicionales repetidos a lo largo de un tipo es lo que se denomina un *smell*. Nosotros lo hicimos sabiendo que terminaríamos cambiando esa pieza si era necesario, y comprobamos que así fue ¿Podríamos haber modelado la herencia desde un principio? La respuesta está en la definición de ingeniería: llevar adelante una solución que surja como la mejor a raíz del balance de los costos y los beneficios de diversas alternativas. Introducir el primer *if* fue simple (en este caso). Sin embargo, mantener esa política ante la potencial crecida del número de alternativas hubiese sido poco profesional.

  Nota: La herencia utilizada en forma prematura genera acoplamientos innecesarios y relentiza el desarrollo. La herencia descubierta en forma tardía provoca cambios masivos a lo largo de los clientes de nuestras clases (salvo que se hayan empleado los recaudos necesarios) y puede ser muy tediosa de implementar.

Herencia y composición

Hay ocasiones en que perdemos la orientación y utilizamos la herencia como simple mecanismo de reutilización de código. Por supuesto **la herencia no es simplemente un mecanismo de reutilización de código**, aunque lo es en parte. Recordemos que la relación que impone la herencia es jerárquica, acoplando nuestras clases a una taxonomía determinada que debe honrar al compartir la semántica del tipo y el comportamiento del mismo.

Un buen contraejemplo

Miremos un ejemplo que roza el ridículo:

```
class Boca {
  void comer(Alimento alimento) { ... }
}
```

Supongamos que necesitamos ese comportamiento en una persona, por lo que utilizamos nuestra herramienta de reutilización: la herencia.

```
class Persona extends Boca {
}
```

¡Listo! Obtuvimos el comportamiento deseado mediante la extensión del tipo adecuado.

Cómo debimos haberlo hecho

Por supuesto no hemos hecho nada más que ganarnos el odio de nuestros colegas y que nuestro "futuro yo" desee golpear al "yo actual".

Simplemente necesitábamos reutilizar el comportamiento, pero no generar esa relación. Podríamos haberlo resuelto de la siguiente manera:

```
class Persona {
  private Boca boca;
  Persona() { ... }
  void comer(Alimento alimento) {
    this.boca.comer(alimento);
  }
}
```

Y mediante esa simple delegación, evitamos hacer una herencia que no es natural. La composición nos ahorró una relación innecesaria. Adicionalmente, esta composición cumple con la regla del "tiene un", ya que es correcto enunciar que una `Persona` tiene una `Boca` [que le permite `comer(Alimento)`].

> **Nota:** Siempre puede reemplazarse la herencia por composición. Lamentablemente la sentencia opuesta también es verdadera. Deberemos estar atentos a las implicaciones semánticas que cada una de las relaciones conlleva para elegir sabiamente.

Análisis de la composición sobre la herencia

Ventajas

Puede decirse a favor de la composición que este tipo de diseño proporciona **mayor flexibilidad** y otorga un **dominio más estable a largo plazo** (los métodos pueden seguir llamándose del mismo modo a pesar de cambiar su comportamiento interno dejando de delegar, por ejemplo). Una relación del tipo "tiene un" es menos demandante que una relación del tipo "es un".

Adicionalmente, es más **simple** diseñar agregando comportamientos que sean delegados en otros objetos que atando herencias que puedan proporcionar una rigidez innecesaria para futuros cambios.

Por último **se evitan los problemas derivados** del cambio en los supertipos, que acarrean cambios encadenados en todos sus subtipos.

Desventajas

La clase compuesta siempre **debe implementar las interfaces de llamada a sus métodos delegados**, aún cuando sólo sea una simple redirección. En

contraste, la herencia tiene un mecanismo automático de redirección de llamadas implícito: si no la posee la clase, se buscará en alguno de sus ancestros.

De hecho es tan fuerte este concepto, que Sandi Metz define la herencia de forma pragmática con estas palabras:

> La herencia es, en esencia, un mecanismo para la delegación automática de mensajes. Define un camino de reenvío para los mensajes no entendidos.
> Sandi Metz [METZ-2012]

Algunos conceptos prácticos sobre la herencia

Hasta el momento hemos brindado un pantallazo de los conceptos centrales de la herencia: cuándo utilizarla, cómo descubrir un mal uso de la misma y alguna alternativa para evitar el acoplamiento que ésta genera. Sin embargo aún nos quedan muchos conceptos prácticos que debemos tener en cuenta, y esta sección busca ponerlos de manifiesto.

Especialización/generalización

Es necesario recordar que mediante la herencia estamos definiendo una relación de generalización/especialización entre clases, dependiendo del sentido de lectura de la misma. Esto implica que los métodos de los subtipos deben ser más específicos que los del supertipo.

La herencia resuelve el problema de los tipos íntimamente relacionados que comparten muchos aspectos pero que difieren en algunos de un modo particular. Es por ello que las herramientas que nos son dadas sirven a estos fines: todo subtipo se comportará como su supertipo salvo que se indique lo contrario. Y esto sucederá mediante la inclusión de nuevas responsabilidades, o de la modificación del modo en que se cumple con las suscriptas por su antecesor. Sin embargo **no se debería** dejar de cumplir con responsabilidades heredadas (ver Refused Bequest, en el Principio de Sustitución de Liskov).

Sobreescritura de métodos

El mecanismo más simple para cambiar el comportamiento de una subclase es la sobreescritura de los métodos de la clase antecesora. Esto significa simplemente ocultar una implementación, proporcionando una más adecuada para el caso. Por ejemplo:

```
class Transporte {
  Double calcularDesgaste() {
    return kilometrosRecorridos / kilometrosTotalesPosibles;
```

```
  }
}

class Camion extends Transporte {
  @Override Double calcularDesgaste() {
    return kilometrosRecorridos / kilometrosTotalesPosibles + cargaTransportada / cargaTotal
  }
}
```

En ese ejemplo vemos cómo la forma de calcular el desgaste de un camión es diferente a la de un transporte estándar. Es por ello que sobreescribimos el método, eclipsando al del ancestro y dejando vigente la nueva implementación.

La forma de hacerlo es simple: escribimos un método con la misma firma que el método de su ancestro. La forma de verificarlo es agregar la anotación semántica `@Overrides` al método, lo que le indica al compilador que nuestra intención es sobreescribir un método.

> La anotación no hará más que verificar que así sea, sin forzarlo. Si escribimos mal la firma del método simplemente fallará en tiempo de compilación, sin corregirlo ni hacer nada en particular en favor de nuestra intención.

En otro escenario también podríamos haber reutilizado una parte del método preexistente:

```
class Transporte {
  Boolean debeArreglarse() {
    return calcularDesgaste() > calcularDesgastePermitido();
  }
}

class Camion extends Transporte {
  @Override Boolean debeArreglarse() {
    return super.debeArreglarse() && estaActivo;
  }
}
```

Con el código proporcionado estamos enfatizando que el camión se debe arreglar bajo las mismas condiciones que todo transporte, pero adicionalmente debe estar activo.

Notemos la utilización de la palabra reservada **super**: la misma nos permite referirnos al ancestro directo de una clase dada. En este caso y por estar dentro de la clase `Camion` se refiere a la clase `Transporte`. Al escribir `super.debeArreglarse()` está invocando la implementación de `debeArreglarse():Boolean` presente en la superclase, y utilizando ese resultado para calcular su propio valor.

Debemos ser cuidadosos con el uso de esta palabra reservada, ya que aunque

es una herramienta poderosa de reutilización ésta comienza a generar contradependencias y suma puntos de acoplamiento para con la implementación del supertipo.

Clases abstractas

Ciertos conceptos de nuestro software no tienen sentido más que taxonómico: sólo nos sirven para formar jerarquías. Es por ello que surgen las **clases abstractas**, las cuales son una abstracción de conceptos necesarios para el modelado de una solución pero que no tienen sentido de existencia por sí mismos.

> Las clases abstractas existen para ser extendidas. Éste es su único propósito. Proveen un repositorio para el comportamiento que es compartido entre un conjunto de subclases (subclases que a su vez proporcionan especialización).
> Sandi Metz [METZ-2012]

Como consecuencia de ésto, las clases abstractas no necesitan ser instanciadas. Es de principal importancia comprender que la forma de ver estos hechos es la expuesta en este apartado: *dado que las clases abstractas son conceptos que no tienen sentido por sí mismos más que como parte de una taxonomía, no es necesario instanciarlas.*

En el ejemplo anterior, la clase `Transporte` bien podría ser abstracta. Y eso le permite (pero no obliga) tener métodos sin definición. Por ejemplo:

```
abstract class Transporte {
  abstract void cargarPaquete(Paquete paquete);
}
```

Ese método está declarado pero no definido (tiene firma pero no comportamiento). Eso implica que *todas las clases que se declaren como extensiones de `Transporte` deben definir ese método para poder utilizarlo.*

El análisis es evidente: mientras no existan objetos del tipo abstracto, no importa que sus métodos estén definidos. Sin embargo, si necesitamos objetos de ese tipo, deberemos definir las funcionalidades que hasta el momento son abstractas para poder enviarle esos mensajes una vez tengamos el objeto propiamente dicho.

El contrato es simple: quien extiende una clase abstracta deberá dar comportamiento a los métodos abstractos, o ser abstracto a su vez.

Constructores y herencia

Al construir una subclase, implícitamente se está construyendo una superclase. El detalle carece de importancia hasta el momento en que sufrimos las consecuencias de redefinir constructores:

```
class MiClase {
  MiClase(String a, String b) { ... }
}

class MiSubclase extends MiClase {
}
```

Ese código no compila. Para construir una clase del tipo `MiClase` necesitamos dos parámetros. Eso genera que el constructor por defecto, implícito, que nos proporciona el lenguaje deje de tener vigencia. Al momento de extender, `MiSubclase` no requiere parámetros, por lo que posee el constructor implícito. Sin embargo, como se requiere construír el subtipo sobre el supertipo, el compilador anuncia que no sabe cómo construír un objeto del tipo `MiClase` si no se le proporcionan los parámetros necesarios.

Solución 1: Nuevo contructor en la superclase

```
class MiClase {
  MiClase(String a, String b) { ... }
  MiClase() { ... }
}

class MiSubclase extends MiClase {
}
```

Ahora el código compila, ya que el constructor implícito de la subclase invoca al constructor explícito (sin parámetros) de la superclase en forma automática. Sin embargo puede que este código no sea completamente correcto: ¿tiene sentido instanciar un objeto del tipo `MiClase` sin esos parámetros?

Solución 2: Declarar un constructor idéntico en la subclase

```
class MiClase {
  MiClase(String a, String b) { ... }
}

class MiSubclase extends MiClase {
  MiSubclase(String a, String b) {
    super(a, b);
  }
}
```

Nótese la invocación al constructor de la clase antecesora en forma explícita: empleando la palabra reservada **super**.

En este caso, estamos acoplando nuestra subclase al constructor de la superclase. Puede que tampoco sea correcto, pero al menos no estamos introduciendo posibles

bugs al no proporcionar los parámetros necesarios para la superclase desde la subclase.

Solución 3: Declarar un constructor sin parámetros en la subclase

```
class MiClase {
  MiClase(String a, String b) { ... }
}

class MiSubclase extends MiClase {
  MiSubclase() {
    super("default", "default");
  }
}
```

Este puede ser el modo más inocuo si es que deseamos que las instancias de `MiSubclase` puedan instanciarse sin proporcionar parámetros, y conocemos los parámetros necesarios por defecto para `MiClase`. Los clientes de la subclase no deberán arrastrar dependencias, y en forma encapsulada estamos controlando la creación de nuestro tipo y la jerarquía involucrada.

Conclusión

En general hay que tener cuidado al momento de definir constructores en las superclases: si son necesarios, habrá que utilizarlos. Si no, deberemos evitarlos ya que generan un acoplamiento que se arrastra a las generaciones venideras.

Reglas de visibilidad

Cuando declaramos una relación jerárquica nos encontramos con que las subclases no siempre deberán acceder a todo lo que la superclase posee: debemos preservar el encapsulamiento y el ocultamiento de la información de la mejor forma que podamos. Es por ello que existen reglas de visibilidad específicas, en las que según sea la declaración de la superclase, así será la de la subclase:

- Si la superclase define un miembro como **privado**, la subclase no podrá accederlo bajo ningún concepto. Las clases ajenas a la jerarquía, como es de esperar, tampoco.
- Si la superclase define un miembro como **protegido**, la subclase podrá accederlo como si fuera propio pero privado. Las clases ajenas a la jerarquía no podrán accederlo.
- Si la superclase define un miembro como **público**, la subclase lo poseerá con la misma visibilidad. Las clases ajenas a la jerarquía podrán acceder al mismo.

> **Nota:** No introduciremos el modificador de visibilidad *package* dado que excede el interés de este capítulo.

final

Cuando una clase declara un método como **final** esto implica que las subclases que esta posea no podrán sobreescribirlo. Esto facilita que ciertos patrones de diseño puedan implementarse adecuadamente (como ser el Template Method).

Adicionalmente puede declararse una clase completa como **final**, lo que generará que no pueda extenderse (la clase `String` proporcionada por la API de Java es un buen ejemplo de una clase que no puede extenderse). Esto protege de manera terminante la clase construida, restando flexibilidad en favor de algún tipo de mantenibilidad.

Herencia simple vs. herencia múltiple

Java sólo permite la herencia simple, lo que significa que una clase puede tener tantas derivadas como sea necesario, pero sólamente un ancestro. El ancestro originario de todas las clases es la clase `Object`, de la cual heredamos implícitamente si no declaramos otra relación.

La herencia múltiple permite la obtención de comportamiento de varios orígenes, lo que lleva a un diseño generalmente más enredado y con acoplamientos más marcados. Adicionalmente, si un método de una subclase se encuentra definido en varias superclases, sería complejo identificar a cuál pertenece la implementación que deseamos utilizar.

En Java la solución a la necesidad de la herencia múltiple viene dada por la utilización de **interfaces** o de la ya descripta **composición/delegación** de tipos. Profundizaremos en este sentido en próximos capítulos.

Recursos

Bibliografía

- [MEYER-09] **Touch of class: learning to program well with objects and contracts**, *Bertrand Meyer* - Springer - 2009
- [METZ-2012] **Practical Object-oriented Design in Ruby: An Agile Primer** *Sandi Metz* - Addison-Wesley - 2012
- [SHARP-97] **Smalltalk by example: the developer's guide** *Alec Sharp* - McGraw Hill - 1997
- [FREEMAN-2004] **Head First Design Patterns** *Eric Freeman et al* - O'Reilly - 2004

Links de interés

- "Composite reuse principle", *Wikipedia, The Free Encyclopedia.* Wikimedia Foundation, 21 de marzo de 2013 (accedido el 1 de abril de 2013)
- "Inheritance". *docs.oracle.com* (accedido el 1 de abril de 2013)
- "Composition instead of inheritance". *c2.com* (accedido el 1 de abril de 2013)
- "OOP - Inheritance & Polymorphism". *ntu.edu.sg* (accedido el 1 de abril de 2013)

Por Lucas Videla. Versión original disponible en delucas/notas