


POO Tema 4. Herencia. Jerarquías de clases, composición y abstracción

Martín González Dios 

25 de noviembre de 2024

1. Herencia

Es la segunda de las características principales de la Programación Orientada a Objetos. Es el mecanismo en virtud del cual **una clase derivada reutiliza los atributos y los métodos que pertenecen a una clase base** (o superior).

Los **constructores de una clase no serán heredados por las clases derivadas**, ya que se consideran que son específicos de la clase a la que pertenecen.

Entre una clase derivada (p.e., CartaDeMision) y una clase base (p.e., Carta) se dice que existe una **relación del tipo “es-un/a”** (p.e., una CartaDeMision es una Carta)

Tipos de herencia:

1. Una clase B hereda todos los atributos y métodos de una clase A. Es el tipo de herencia más básico, en la que están basados los demás tipos de herencia.

2. Una clase C hereda los atributos y métodos de una clase B que, a su vez, hereda de una clase A, de modo que la clase C hereda los métodos y atributos de la clase A.

3. Las clases B y C heredan los atributos y los métodos de la clase A, de modo que ambas clases se diferencian en los atributos y métodos específicos de cada una de ellas.

4. La clase C hereda los métodos y atributos de las clases A y B, de modo que es necesario definir políticas de herencia cuando A y B tienen métodos comunes.

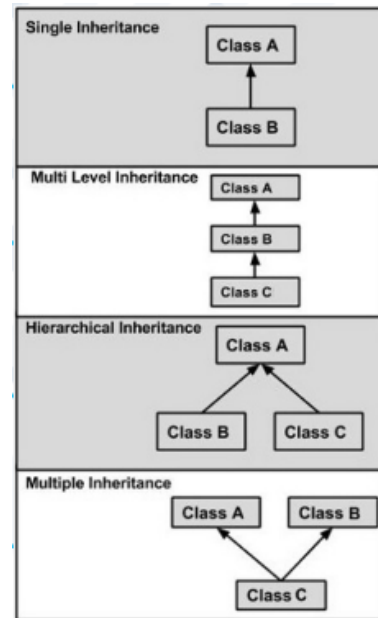


Figura 1: Tipos de herencia

1.1. Características de la herencia

Ventajas de la herencia:

- Reutilización de código
- Simplifica el código
- Facilita el mantenimiento (si se hace correctamente)
- Facilita la extensibilidad de los programas (si se hace correctamente)

Desventajas de la herencia:

- Si el nivel de jerarquización es muy profundo el **código es más difícil de entender**.
- Si hay cambios en la clase base estos podrían ser **inconsistentes** en las clases derivadas.
- Es posible que los atributos deban tener un modificador de acceso diferente al privado que, por tanto, sería **contrario al concepto de encapsulación**.

2. Composición

Mecanismo mediante el cual una clase contiene objetos de otras clases a los que delega ciertas operaciones para conseguir una funcionalidad dada.

Se dice que la composición entre dos clases es una **relación de tipo “tiene-un/a”**.

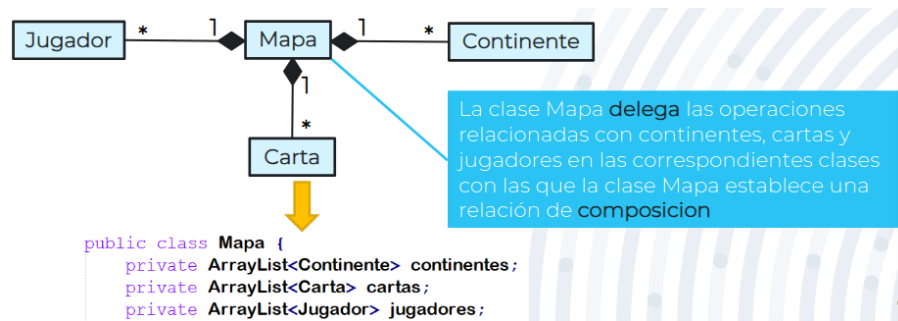


Figura 2: Ejemplo de composición

2.1. Características de la composición

Ventajas de la composición:

- Reparte responsabilidades entre objetos
- Facilita el mantenimiento del programa
- Facilita la extensibilidad
- Las clases están desacopladas entre sí, de modo que un cambio en una de ellas tiene un impacto reducido en las otras

Desventajas de la composición:

- Tiende a **generar mucho más código y de mayor complejidad**, para soportar la misma funcionalidad que se consigue cuando se aplica herencia.
- Más tiempo de desarrollo en comparación con la herencia, ya que al no estar basadas unas clases en otras hay que **implementar los métodos en todas las clases**.

3. Herencia vs Composición

3.1. Ejemplo

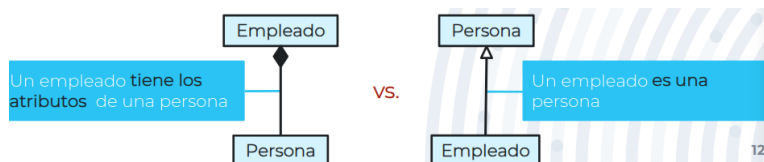


Figura 3: Ejemplo sobre la elección de herencia o de composición

La herencia en programación implica que un empleado es una persona, lo que puede ser problemático, ya que si un empleado deja de serlo, también dejaría de ser persona. En cambio, es más adecuado considerar que un empleado tiene atributos de una persona, lo que permite que la clase **Empleado** delegue las operaciones relacionadas con personas a la clase **Persona**, manteniendo así una separación clara entre las funcionalidades de ambas clases.

3.2. Composición sobre herencia

Cuando no se debería de utilizar herencia:

- Si las clases no están relacionadas desde un punto de vista lógico.
- Si una clase base tiene una sola clase derivada.
- Si las clases derivadas heredan código que no necesitan.
- Si existe la posibilidad de que las clases base cambien (en este caso la herencia complicaría el desarrollo del programa)
- Si es necesario sobrescribir muchos métodos de las clases base (en este caso la herencia complicaría el desarrollo del programa)

Caso	Diseño basado en herencia	Diseño basado en composición
Inicio del desarrollo	Más rápido	Más lento
Diseño del software	Más sencillo	Más complejo
Efectos no deseados	Ocurren con más frecuencia, sobre todo en cuando se trata de jerarquías profundas	Se reducen y están más localizados, en los métodos delegados
Adaptación a cambios	Para jerarquías profundas y con múltiples sobreescritura es más complicado	Más sencillo de cambiar, puesto que solamente afectan a las clases delegadas
Validación	Es difícil de realizar, sobre todo en jerarquías profundas y con múltiples sobreescrituras	Más sencillo de validar al estar el código muy localizado en las clases delegadas
Extensibilidad	Es sencillo, aunque se complica en jerarquías profundas	Tiene lugar de forma más sencilla a través de la composición de clases

Figura 4: Composición sobre herencia

4. Herencia en Java

En Java no existe la herencia múltiple, es decir, una clase derivada solo puede tener una única clase base. Solamente se heredan los atributos y métodos de la clase base que son públicos para la clase derivada, depende de los tipos de acceso.

4.1. Tipos de acceso para la herencia

- **private**: la clase derivada **nunca heredará** los atributos y los métodos, independientemente del paquete en el que esté la clase base en relación a la clase derivada.
- **public**: la clase derivada **siempre heredará** los atributos y los métodos, independientemente del paquete en el que esté la clase base en relación a la clase derivada.
- **default**(no poner nada): la clase derivada **solamente heredará** los atributos y los métodos de la clase base si ambas clases **se encuentran en el mismo paquete**.
- **protected**: la clase derivada **siempre heredará** los atributos y los métodos, independientemente del paquete en el que esté la clase base en relación a la clase derivada.

Los atributos deben ser privados, ya que se considera que la encapsulación es una característica que ofrece mayores beneficios que la herencia.

La **clase derivada hereda todos los atributos**, independientemente de su tipo de acceso. El tipo de acceso especifica el nivel de visibilidad/accesibilidad que los métodos de la clase derivada tienen sobre los atributos y métodos de la clase base, es decir, **si son privados, se heredan pero no se puede acceder a ellos**.

La **clase derivada hereda todos los métodos**, independientemente de su tipo de acceso. El tipo de acceso no evita o limita la herencia de los métodos, sino que **impone restricciones sobre la visibilidad que tienen los métodos de la derivada sobre los métodos que se heredan de la clase base**.

En una jerarquía de clases, idealmente los **métodos** deberían de estar **implementados en la clase a la que pertenecen los atributos que se utilizan en su implementación**, evitando hacerlo en las clases derivadas de dicha clase.

5. Constructores

Los **constructores de la clase base no se heredan**. Los constructores tienen que reservar memoria e inicializar los atributos de la clase a la que pertenecen, pero el constructor de la clase base no tiene acceso a los atributos de la clase derivada, con lo cual si se hereda el constructor de la clase base, no podría ser invocado para crear objetos de la clase derivada.

- Si el **constructor de la clase base no tiene argumentos**:
 - Cuando se invoca al constructor de la clase derivada, se invocará automáticamente el constructor de la clase base.
 - Si se implementa un constructor sin args para la clase derivada, la clase base deberá tener obligatoriamente un constructor sin args.
- Si el **constructor de la clase base tiene argumentos**:
 - Solamente se invocará a dicho constructor, puesto que no se puede garantizar que en la clase base exista un constructor con los mismos args.
 - Se invoca manualmente al constructor de la clase base con args que se considere oportuno en cada caso. Se hace uso de `super`

super permite acceder desde la clase derivada a los atributos, métodos y constructores de la clase base sobre los que existe cierto nivel de visibilidad. Únicamente puede acceder a los elementos de la clase base que es **inmediatamente superior** a la clase derivada.

Buenas prácticas:

- Se deben reservar e inicializar los atributos en los constructores de las clases a las que pertenecen dichos atributos, evitando hacerlo en las clases derivadas de dicha clase.
- Se debe hacer uso de `super` para invocar a los constructores de las clases base para reservar memoria en inicializar los atributos (privados) que estén definidos en ellas.

6. Métodos y sobreescritura

Un método heredado de una clase base se implementa nuevamente en la clase derivada. La firma del método debe ser la misma, y el tipo de acceso el mismo o superior.

Una forma de sobrescribir los métodos en las clases derivadas es hacer uso de `super` para reutilizar la implementación del método que está en la clase base.

- Desde el método de la clase derivada `super` invoca al mismo método de la clase base, es decir, al método que se sobrescribe y que tiene el mismo nombre y argumentos.
- Se usa `super` porque la clase base no es capaz de distinguir entre la implementación de la clase base y la implementación de la clase derivada, con lo cual la única forma de reutilizarlo es a través de `super`. (El método de la clase base se hereda, pero al sobrescribirse no se puede acceder a su implementación de la clase base)

7. Clases abstractas

Son clases que **no se pueden instanciar**. Pueden tener constructores, pero no se pueden invocar a través de new, de modo que únicamente se pueden invocar mediante super cuando son las clases base de otras clases. Pueden tener atributos y métodos. Se suelen usar como **clases base de otras clases** de las que sí se pueden crear objetos, **facilitando reutilización**.

Las clases abstractas deben tener implementadas la **mayor cantidad posible de métodos para que las clases derivadas puedan heredarlos** y así favorecer la reutilización de código. Una clase debería ser abstracta cuando todos los objetos pertenecen a cualquiera de las otras clases de la jerarquía, no siendo necesario crear un objeto de la clase abstracta.

Las clases abstractas pueden tener métodos abstractos, no tienen cuerpo, es decir, son métodos que no están implementados y en los que solamente se especifica su firma (nombre, lo que devuelve y los argumentos de entrada). Si una clase tiene al menos un método abstracto entonces debe ser **necesariamente** una clase abstracta. Los métodos abstractos deben ser implementados en las clases derivadas de la clase abstracta a la que pertenecen, **siempre que dichas clases no sean abstractas**.

8. Clases, atributos y métodos finales

Si una clase es final significa que **no se pueden crear clases derivadas a partir de esta clase**. Solamente debería ser final cuando se garantiza que no será necesario crear clases derivadas de ella.

Si un **método** es marcado como final, si este método pertenece a una clase base significa que no podrá ser sobrescrito en una clase derivada.

Si un **atributo** es marcado como final, una vez que el atributo haya tomado un valor no podrá ser modificado. (no pueden tener setter). Se usan para implementar **constantes** (se suelen hacer final static, para que se pueda usar sin que sea necesario crear un objeto de la clase).

Buenas prácticas:

- Las **constantes** del programa se deberían definir en **una o varias clases abstractas** que son accesibles por todas las clases en las que se hace uso de esas constantes.
- Las **clases y métodos finales** se deberían usar cuando se **garantice** que **no será necesarios crear clases derivadas o sobrescribir métodos**, respectivamente