UNIVERSIDAD DE MURCIA

Tecnología de la Programación

Sesión 4 de Prácticas

Relación entre clases

Dpto. de Ingeniería de la Información y las Comunicaciones

Curso 2025-2026

Última modificación:
29 de septiembre de 2025

Sesión 4: Relación entre clases

Índice

4.1 Tipos de relaciones en POO

Cuando trabajamos con múltiples clases en un programa, es común que establezcan vínculos entre ellas. Estos vínculos se denominan **relaciones** y pueden variar en su **nivel de dependencia**, desde relaciones simples donde un objeto utiliza a otro de manera puntual, hasta relaciones complejas que afectan al ciclo de vida de los objetos. A continuación, se describen los principales tipos de relaciones en programación orientada a objetos.

4.1.1. Relación de uso

Una **relación de uso** ocurre cuando una clase utiliza los servicios o funcionalidades de otra de forma **esporádica**. Esto significa que el vínculo entre ambas clases **no es constante ni duradero en el tiempo**, sino que ocurre puntualmente cuando una clase necesita interactuar con la otra. El caso más típico es cuando una clase A invoca un método de una clase B para ejecutar una acción, sin que A almacene una referencia a B. Es decir, la clase B es utilizada temporalmente para cumplir una función específica. Un ejemplo cotidiano de este tipo de relación es el caso de una persona que utiliza un cajero automático para realizar una transacción, sin necesidad de ser cliente del banco. En el código, este tipo de relación se manifiesta al pasar un objeto de la clase B como parámetro a un método de la clase A.

Para comprender mejor todos los tipos de relaciones, retomaremos el código de la sesión de prácticas anterior, que hacía uso de la clase `Persona`. Sin embargo, para trabajar mejor con relaciones vamos a necesitar una clase adicional: `Libro`. La definición de un libro, de forma muy simplificada, podría venir determinada por su título, su autor, y su número de páginas. Es importante destacar que no ofrecemos métodos set para el título y el autor, pues no tiene sentido modificar esta información una vez creado. Sin embargo, si podríamos necesitar cambiar el número de páginas, por ejemplo para un libro que está escribiéndose (por ejemplo, pensemos en un diario personal).

```python
# Módulo libro.py
class Libro:
    def __init__(self, titulo: str, autor: str, paginas: int):
        self._titulo: str = titulo
        self._autor: str = autor
        self._paginas: int = paginas

    # Método getter para el título
    def get_titulo(self) -> str:
        return self._titulo

    # Método getter para el autor
    def get_autor(self) -> str:
        return self._autor

    # Métodos getter y setter para las páginas
    def get_paginas(self) -> int:
        return self._paginas

    def set_paginas(self, nuevas_paginas: int) -> bool:
        if nuevas_paginas > 0:
            self._paginas = nuevas_paginas
            return True
        return False
```

Por su parte, la clase `Persona` establece una **relación de uso** con la clase Libro haciendo uso del método *hojear_libro*. Como podemos observar, este método simplemente muestra un mensaje por pantalla en base a la información del libro que se ha pasado por parámetro, simulando un uso puntual de un objeto con el que no mantenemos ninguna vinculación como tal (no lo poseemos, no lo hemos leído, etc). Este es un ejemplo sencillo, aunque clarificador, de utilización de relación de uso: la clase Persona no tiene una relación a nivel de atributos con la clase Libro y se usa directamente el libro recibido como parámetro.

```python
# Módulo persona.py
from app.tdas.libro import Libro

class Persona:
    especie: str = "Humano"

    def __init__(self, nombre: str, edad: int, altura: float = None, activo: bool = None):
        self._nombre: str = nombre
        self._edad: int = edad
        self._altura: float = altura
        self._activo: bool = activo
        self._hobbies: list = []

        if hobbies is None:
            self._hobbies: list = []
        else:
            self._hobbies: list = hobbies

    # (...) Métodos get/set no mostrados por simplicidad

    # Método para hojear un libro (relación de uso)
    def hojear_libro(self, libro: Libro) -> None:
        print(f"{self.get_nombre()} está hojeando {libro}")
```

Gracias a la sesión de prácticas previa, sabemos que el código de calidad debe estar correctamente organizado en **paquetes y módulos**. Así, en nuestro ejemplo tendremos, dentro del paquete *tdas*, dos módulos principales: Persona y Libro. En el código anterior hemos indicado la importación de la clase Libro desde del módulo *libro* que está, a su vez, contenido en los paquetes *tdas* y *app*. No obstante, por simplicidad, nos abstraeremos de estos aspectos en este guión, para así centrarnos simplemente en las modificaciones de código. Los ejemplos completos de código están disponibles en el proyecto de GitHub proporcionado por los profesores.

4.1.2. Relación de asociación

Una relación de **asociación** se establece cuando una clase mantiene una **referencia a otra clase**, es decir, una instancia de una clase A tiene un atributo que es una instancia de la clase B. A diferencia de la relación de uso, donde la interacción entre las clases es temporal, en una relación de asociación el vínculo es **más estable en el tiempo**. Sin embargo, este vínculo **puede cambiar**, ya que un objeto de la clase A puede asociarse con diferentes objetos de la clase B a lo largo de su ciclo de vida.

Un aspecto importante de la asociación es que **la existencia de un objeto de la clase A no depende de la existencia del objeto de la clase B**. Si la relación termina, el objeto de la clase A puede continuar existiendo, aunque la referencia al objeto de la clase B sea eliminada o quede en un estado nulo. Un ejemplo común es la relación entre un **empleado** y un **proyecto**. El empleado está asignado a un proyecto y realiza tareas en él, pero este vínculo puede cambiar si el proyecto

finaliza o el empleado se reasigna a otro proyecto. El empleado sigue existiendo en la organización independientemente de la existencia de los proyectos con los que esté asociado.

4.1.2.1. Relación unidireccional

Llevemos estos conceptos al ejemplo de las clases `Libro` y `Persona`. En primer lugar, vamos a definir una **relación de asociación unidireccional**, en la que una `Persona` ha leído un listado de libros. Como podemos ver, la clase `Libro` no sufre ningún cambio, ya que es `Persona` la que establece la relación hacia `Libro` de forma unidireccional.

```python
# Módulo persona.py
class Persona:
    especie: str = "Humano"

    def __init__(self, nombre: str, edad: int, altura: float = None, activo: bool = None,
                 hobbies: list = None):
        self._nombre: str = nombre
        self._edad: int = edad
        self._altura: float = altura
        self._activo: bool = activo

        if hobbies is None:
            self._hobbies: list = []
        else:
            self._hobbies: list = hobbies

        self._libros_leidos: list[Libro] = []

    # (...) Métodos get/set no mostrados por simplicidad

    # Método para agregar un libro a la lista de libros leídos
    def leer_libro(self, libro: Libro) -> None:
        self.get_libros_leidos().append(libro)

    # Método para listar los libros leídos por la persona
    def listar_libros_leidos(self) -> None:
        print(f"Libros leídos por {self.get_nombre()}:")
        for libro in self.get_libros_leidos():
            print("> ", libro)
```

Así, para establecer esta relación unidireccional definiremos el listado de libros leídos como lista vacía ya que inicialmente una persona no ha leído libros. Además, definimos dos métodos adicionales: `leer_libro` y `listar_libros_leidos`. No tiene sentido en este contexto un método denominado `eliminar_libro_leido`, pues una vez que hemos leído un libro ya no podemos volver atrás en el tiempo y evitar leerlo (aunque a veces nos gustaría poder hacerlo). Así, también será necesario definir un método getter para consultar el nuevo atributo definido.

4.1.2.2. Relación bidireccional

Pasemos ahora a estudiar cómo podríamos definir una **relación de asociación bidireccional** entre las clases `Persona` y `Libro`. En este caso, ambas clases deben incluir un atributo que haga referencia a la clase opuesta. En concreto, la clase `Persona` contiene el **listado de libros leído** (ya definido en el apartado anterior), mientras que la clase `Libro` almacena el **listado de lectores** que han completado ese libro.

Por lo tanto, ahora se modifica la clase `Libro` para añadir un nuevo atributo, `listado_lectores`, que representa el listado de personas que han leído un libro concreto. En el constructor, dicha lista es vacía pues aún no habrá sido leído por nadie. Además, es interesante definir métodos de consulta y modificación: `get_listado_lectores` y `agregar_lector`:

```python
# Módulo libro.py
class Libro:
    def __init__(self, titulo: str, autor: str, paginas: int):
        self._titulo: str = titulo
        self._autor: str = autor
        self._paginas: int = paginas
        self._listado_lectores: list[Persona] = []  # Lista de personas que han leído el libro

    # (...) Métodos get/set no mostrados por simplicidad

    # Métodos getter y setter para los lectores del libro
    def get_listado_lectores(self) -> list['Persona']:
        return self._listado_lectores

    def agregar_lector(self, persona: Persona) -> None:
        self.get_listado_lectores().append(persona)
```

Por otro lado, es necesario modificar el método `leer_libro` en la clase Persona para actualizar el listado de lectores del libro antes de ser añadido a la lista de libros leídos de la persona. De esta forma establecemos la relación bidireccional cuando vamos añadiendo libros leídos manualmente a una persona que ya existe.

```python
# Módulo persona.py
class Persona:
    # (...) Código omitido

    # Método para agregar un libro a la lista de libros leídos
    def leer_libro(self, libro: Libro) -> None:
        libro.agregar_lector(self)  # Asociar la persona con este libro
        self.get_libros_leidos().append(libro)
```

En base a este código, y una vez hemos creado un objeto de la clase Persona, se tendrá que invocar al método *leer_libro* cada vez que la persona haya completado un libro. Así, este método se encargará primero de marcar el libro como leído por dicha persona (pasando el self como parámetro) y, finalmente, añadir este libro a la lista de libros leídos de la persona.

Llegados a este punto, si tratamos de ejecutar el *main* de la sesión 2, incluso sin haber realizado modificaciones para invocar los nuevos métodos en la clase *Persona*, obtendremos el siguiente error:

```
ImportError: cannot import name 'Libro' from partially initialized module
'app.tdas.libro' (most likely due to a circular import)
```

Este error se debe a un problema de importación circular: el intérprete de Python, en base a nuestro código, detecta que las clases *Persona* y *Libro* intentan importarse mutuamente, generando así una dependencia circular. Para solucionarlo, tendremos que asegurarnos de dos cosas:

- Solo una clase importará a la otra. Por ejemplo, solo *Persona* añadirá al inicio *from app.tdas.libro import Libro*. Sin embargo, Libro no importará a Persona.
- Cuando se haga referencia al tipo de datos (como *Persona* o *Libro*) desde la clase contraria, por ejemplo, en anotaciones de tipo de atributos o parámetros, utilizaremos el siguiente formato: 'NombreClase'. Es decir, escribiremos el nombre de la clase entre comillas simples. Esto le indica al intérprete que ese tipo existe con ese nombre, pero sin establecer una referencia directa inmediata, lo que permite que el intérprete resuelva correctamente las dependencias en tiempo de ejecución.

En general, cuando indiquemos una clase como tipo de datos de atributos, variables o parámetros, usaremos siempre el nombre con comillas simples, incluso cuando se trate de una relación unidireccional o el tipo de datos sea esa misma clase (por ejemplo, en el caso de hojear_libro(), donde se recibe como parámetro un *Libro*). Esto nos ahorrará muchos problemas de dependencias circulares según el código evolucione.

4.1.3. Relación de agregación

La relación de agregación es un **tipo especial de asociación** en la que una clase contiene o **tiene un objeto de otra clase**, pero **ambas pueden existir de forma independiente**. Esto se denomina relación ***has-a***. En la agregación, un objeto no depende del ciclo de vida del otro objeto: si el objeto contenedor (A) se destruye, los objetos que contiene (B) pueden seguir existiendo. Un ejemplo clásico es el de una **factura** y un **cliente**. La factura puede estar asociada a un cliente, pero el cliente seguirá existiendo aunque la factura sea eliminada. También, un cliente puede estar relacionado con varias facturas, pero las facturas no necesariamente tienen que destruirse si el cliente deja de existir. Otro ejemplo de agregación es la relación entre una **persona** y su **coche**. La persona puede tener uno o varios coches, pero si la persona ya no está, los coches pueden seguir existiendo sin ella. En este caso, el coche es parte de la persona en el sentido de que está vinculado a ella, pero ambos objetos pueden existir de manera independiente.

Retomemos el ejemplo de `Persona` y `Libro`. En este contexto, una posible relación de agregación consiste en que **una persona posee un listado de libros en propiedad, pero cada libro solo tiene únicamente un propietario**. A pesar de la relación, ambos objetos pueden existir independientemente:

- Si una persona es eliminada (deja de existir), los libros que poseía siguen existiendo y podrían transferirse a otro propietario o quedar sin dueño.
- Un libro tiene un único propietario, pero el libro puede seguir existiendo si se cambia su propietario o si la persona deja de existir.

Veamos cómo queda el código tras añadir esta nueva **relación de agregación bidireccional** entre clases:

```python
# Módulo libro.py
class Libro:
    def __init__(self, titulo: str, autor: str, paginas: int, propietario: 'Persona' = None):
        self._titulo: str = titulo
        self._autor: str = autor
        self._paginas: int = paginas
        self._listado_lectores: list['Persona'] = []  # Lista de personas que han leído el libro
        self._propietario: 'Persona' = propietario  # Relación de agregación: un libro tiene un propietario

    # (...) Métodos get/set no mostrados por simplicidad
```

```
    # Métodos getter y setter para el propietario del libro
    def get_propietario(self) -> 'Persona':
        return self._propietario

    def _set_propietario(self, propietario: 'Persona'):
        self._propietario = propietario

    def comprar_libro(self, propietario: 'Persona'):
        self._set_propietario(propietario)
```

Para realizar esta relación de agregación es necesario incluir un nuevo atributo, `propietario`, añadiéndose en el constructor. Además, hemos añadido un método get para obtener el propietario de un libro, y la funcionalidad de establecer el propietario. Sin embargo, podemos ver cómo el método set se ha definido como protegido. Así, en su lugar se expone el método *comprar_libro()* por el que protegemos el método setter. Por su parte, veamos los cambios a realizar en la clase Persona:

```
# Módulo persona.py
class Persona:
    especie: str = "Humano"

    def __init__(self, nombre: str, edad: int, altura: float = None, activo: bool = None,
                 hobbies: list = None:
        self._nombre: str = nombre
        self._edad: int = edad
        self._altura: float = altura
        self._activo: bool = activo

        if hobbies is None:
            self._hobbies: list = []
        else:
            self._hobbies: list = hobbies

        self._libros_leidos: list[Libro] = [] # Relación de asociación

        # Relación de agregación: la persona tiene libros en propiedad
        self._libros_propiedad: list[Libro] = []

    # (...) Métodos get/set no mostrados por simplicidad

    # Método para comprar un libro (lo agrego a mi listado de libros en propiedad)
    def comprar_libro(self, libro: Libro) -> None:
        # Asociar el libro con esta persona como propietario (delegación de métodos)
        libro.comprar_libro(self)
        self._libros_propiedad.append(libro)

    # Método para listar los libros en propiedad de la persona
    def listar_libros_propiedad(self) -> None:
        print(f"Libros en propiedad de {self.get_nombre()}:")
        for libro in self.get_libros_propiedad():
            print("> ", libro)
```

En la clase Persona es necesario añadir el atributo `_libros_propiedad`, definido como un listado de libros que la persona posee. Además, definimos dos métodos: *comprar_libro()* y *listar_libros_propiedad()*. Es importante destacar dos aspectos relevantes del método encargado de comprar. En primer lugar, este método se encarga de establecer la relación bidireccional, pues invoca a la clase Libro para actualizar su propietario. Por otro lado, el método *comprar_libro()* de la clase Persona realiza una invocación al método *comprar_libro()* de la clase Libro, usando así **delegación de métodos**.

Es importante remarcar que una persona puede tener libros en propiedad que todavía no ha leído, libros que ha leído pero no tiene en propiedad, o libros que posee y que además ha leído.

4.1.4. Relación de composición

La **relación de composición** es un tipo más fuerte de asociación, donde un objeto es una **parte esencial** de otro. A diferencia de la agregación, en la composición **los objetos dependen completamente del ciclo de vida del objeto contenedor**. Si el objeto que **contiene** se destruye, entonces todas sus **partes** también se destruyen. Esta relación se denomina ***part-of***. Otra característica clave de la composición es que el **objeto contenedor** es responsable de la creación y destrucción de los objetos **contenidos**. En la mayoría de los casos, el objeto contenido se inicializa y se destruye junto con el contenedor. Un ejemplo típico de composición es la relación entre una **casa** y sus **habitaciones**. Las habitaciones no tienen sentido fuera del contexto de la casa; si la casa deja de existir, las habitaciones también dejan de hacerlo.

Vamos a implementar composición en el contexto de `Persona` y `Libro`. Para ello, podemos considerar un **diario** como un libro que solo puede ser utilizado por una persona concreta. Así, si la persona se elimina, el diario debería también eliminarse.

```
# Módulo persona.py
class Persona:
```

```
    especie: str = "Humano"

    # Constructor con sobrecarga mediante parámetros opcionales
    def __init__(self, nombre: str, edad: int, altura: float = None, activo: bool = None,
                 hobbies: list = None:
        self._nombre: str = nombre
        self._edad: int = edad
        self._altura: float = altura
        self._activo: bool = activo

        if hobbies is None:
            self._hobbies: list = []
        else:
            self._hobbies: list = hobbies

        self._libros_leidos: list[Libro] = []

        # Relación de agregación: la persona tiene libros en propiedad
        self._libros_propiedad: list[Libro] = []

        # Relación de composición
        self._diario: 'Libro' = Libro(titulo="Diario", autor=self._nombre, paginas=0, propietario=self)

    # (...) Métodos get/set no mostrados por simplicidad
```

Como podemos ver, el atributo *diario* se crea en el constructor, definiendo un Libro en el que el título es "Diario", el nombre del autor es el nombre de la persona que lo crea, el número de páginas es cero (todavía está vacío), y el propietario es la propia persona que lo crea (se pasa como parámetro al constructor de Libro el self de la propia persona). Así, podremos usar el diario como cualquier otro libro, con la diferencia de que si se elimina la persona, el diario también se debería eliminar (cosa que no pasaría con el resto de libros restantes). No implementaremos más funcionalidad sobre el diario por simplicidad.

4.1.5. Relación de herencia

La **herencia** es un mecanismo fundamental en la programación orientada a objetos, donde una clase **hereda** atributos y métodos de otra, estableciendo una relación **es-un** (*is-a*). Esto permite que una clase hija utilice o modifique las propiedades y comportamientos de una clase padre o base. A través de la herencia, es posible **reutilizar código** de manera eficiente, evitando la duplicación y promoviendo la creación de jerarquías entre clases. En la siguiente sesión exploraremos en detalle cómo implementar herencia en Python.

4.2 Delegación

La **delegación** es un principio clave en el diseño de software, donde una clase **cederá** la ejecución de ciertos métodos o acciones en instancias de otras clases. Este enfoque es útil cuando se desea **distribuir la responsabilidad de ciertas funcionalidades entre diferentes clases**, lo que ayuda a mantener el **código más modular, claro y fácil de mantener**. La delegación permite que una clase principal no se sobrecargue con demasiada lógica, sino que reparta tareas específicas a otras clases especializadas. Un ejemplo común es cuando una clase *Rectángulo* tiene un método *trasladar()*, que delega la tarea de ajustar las coordenadas en una instancia de la clase *Punto*, encargada de gestionar las posiciones x e y. Esta forma de dividir las responsabilidades mejora la reutilización de código y su legibilidad.

Ya hemos visto un ejemplo anteriormente, donde Persona y Libro definen un método con el mismo nombre: *comprar_libro()*, donde el método de *Persona* delega funcionalidad en el de *Libro*. Además, podríamos aplicar delegación en otros aspectos del código:

1. **Delegación en la gestión de libros**: La clase Persona puede delegar las acciones relacionadas con los libros leídos o en propiedad a la clase `Libro`. Por ejemplo, si un libro es leído por una persona, la gestión de la lista de lectores o la asignación de un propietario puede delegarse en la instancia de *Libro*. De esta manera, la clase *Persona* no se encargará de los detalles internos de la gestión del libro, promoviendo una separación de responsabilidades.
2. **Delegación en la visualización de datos**: Cuando la clase `Persona` quiera mostrar sus libros leídos, puede delegar esta tarea a la clase `Libro`. Esto permite que cada clase sea responsable de gestionar y presentar su propia información, manteniendo el código más organizado y fácil de mantener.

Por simplicidad no implementaremos estas ideas de delegación. Sirven únicamente como ejemplo para comprender cómo podríamos aplicar estos principios en el código que implementemos.

4.3 Clonación de objetos

La clonación de objetos es un proceso importante en POO, ya que permite crear copias de instancias sin alterar el objeto original. Existen dos tipos principales de clonación, que elegiremos en función del contexto particular:

1. **Copia superficial**: Esta técnica crea una nueva instancia de la clase, copiando los valores de los atributos del objeto original. Sin embargo, en el caso de los **atributos mutables**, como listas, diccionarios u otros objetos, **solo se copia la referencia** al mismo objeto. Esto implica que si se modifica el contenido mutable en la copia, también se verá reflejado en el objeto original, lo que puede dar lugar a efectos no deseados si no se tiene cuidado. En Python, la copia superficial puede realizar con el método `copy.copy()` del módulo `copy`.
2. **Copia profunda**: En contraste con la copia superficial, una copia profunda no solo crea una nueva instancia del objeto, sino que **también realiza una copia de todos los objetos referenciados dentro de los atributos mutables**. Esto significa que los objetos anidados también se clonan, lo que **previene la compartición de referencias entre el objeto original y la copia**. Este tipo de clonación es **más costosa** en términos de rendimiento, pero garantiza que la copia sea completamente independiente del objeto original. Se puede realizar con el método `copy.deepcopy()` del módulo `copy`.

Además de los métodos ofrecidos por el módulo `copy`, la clonación en Python también se puede personalizar en cada clase mediante los métodos mágicos `__copy__()` y `__deepcopy__()`, permitiendo un mayor control sobre el proceso de clonación para las clases definidas por el usuario.

Trabajaremos sobre la clase `Persona` para realizar, en primer lugar, una **copia superficial** de un objeto. Sin tener que realizar cambios en la clase `Persona`, podemos hacer uso del siguiente código en el *main* para realizar una copia superficial:

```
# Módulo main.py
if __name__ == "__main__":
    # (...) Código omitido

    # Crear una copia superficial de persona1
    persona_copia = copy.copy(persona1)

    # Modificar el objeto original (lista de libros en propiedad)
    print("\nAntes de modificar persona1:")
    persona1.listar_libros_propiedad()
    persona_copia.listar_libros_propiedad()

    print("\nModificando la lista de libros en propiedad de persona1...")
    libro_nuevo = Libro("El hobbit", "J.R.R. Tolkien", 310)
    persona1.comprar_libro(libro_nuevo)

    # Mostrar los datos del original y de la copia después de la modificación
    print("\nDatos de persona1 tras la modificación:")
    persona1.listar_libros_propiedad()

    print("\nDatos de persona_copia tras la modificación en persona1 (copia superficial):")
    persona_copia.listar_libros_propiedad()
```

Un ejemplo de salida esperada podría sería la siguiente:

```
(...)

Modificando la lista de libros en propiedad de persona1...

Datos de persona1 tras la modificación:
Libros en propiedad de Sergio:
>  'El señor de los anillos' de J.R.R. Tolkien, 1178 páginas.
>  '1984' de George Orwell, 328 páginas.
>  'El hobbit' de J.R.R. Tolkien, 310 páginas.

Datos de persona_copia tras la modificación en persona1 (copia superficial):
Libros en propiedad de Sergio:
>  'El señor de los anillos' de J.R.R. Tolkien, 1178 páginas.
>  '1984' de George Orwell, 328 páginas.
>  'El hobbit' de J.R.R. Tolkien, 310 páginas.
```

Como podemos observar, el listado de libros en propiedad cambia en ambos objetos tras modificar la copia, debido a que las listas en Python son mutables y, por tanto, el objeto copiado almacena para esta lista una referencia (**aliasing**) a la lista del objeto original. Si queremos redefinir el comportamiento de la copia superficial de forma manual tendremos que definir el método mágico __copy__ en la clase Persona, de la siguiente forma:

```
# Módulo persona.py
class Persona:
    # (...) Código omitido

    # Método mágico __copy__ para la copia superficial
    def __copy__(self):
        # Crear una nueva instancia de Persona con la referencia a las mismas listas
        nueva_persona = Persona(
            self.get_nombre(),
            self.get_edad(),
            self.get_altura(),
            self.get_activo(),
```

```
            self.get_hobbies()  # Aquí no se crea una copia nueva de la lista, se usa la misma referencia
        )

        nueva_persona._diario = copy.copy(self.get_diario())  # Invocación a copy por ser un objeto
        nueva_persona._libros_leidos = self.get_libros_leidos()  # Se copia la referencia
        nueva_persona._libros_propiedad = self.get_libros_propiedad()

        return nueva_persona
```

En este código vemos que creamos un objeto *nueva_persona* a partir de los atributos de la propia persona (self). Además, debemos copiar aquellos atributos que no se usan en el constructor, como son el diario y las dos listas. En el caso de las listas de libros leídos y en propiedad, bastará con asignar el atributo de la clase actual. Sin embargo, para el diario, como es un objeto, debemos delegar en el método mágico *copy* de la clase Libro. En caso de que no esté definido (como es el caso actual), aplicará copia superficial por defecto. Finalmente, es esencial destacar que este código tendrá el mismo efecto que no incluirlo, pues implícitamente Python ya implementa la copia superficial cuando hacemos uso del método `copy()`.

En cuanto a la **copia profunda**, podemos hacer uso del método `deepcopy()` de forma equivalente a como lo hicimos con `copy()`. Si queremos implementar nuestra propia versión de copia profunda, lo podremos hacer de la siguiente forma:

```
# Módulo persona.py
class Persona:
    # (...) Código omitido

    # Método mágico __deepcopy__ para la copia profunda
    def __deepcopy__(self, memo):
    nueva_persona = Persona(
        self.get_nombre(),  # El nombre es inmutable (string), por lo que puede copiarse superficialmente
        self.get_edad(),  # La edad es un entero (inmutable), también puede copiarse superficialmente
        self.get_altura(),  # Altura es un float (inmutable)
        self.get_activo(),  # Activo es un booleano (inmutable)
        copy.deepcopy(self.get_hobbies(), memo),  # Se hace una copia profunda de la lista de hobbies
    )

    nueva_persona._diario = copy.deepcopy(self.get_diario(), memo)
    nueva_persona._libros_leidos = copy.deepcopy(self.get_libros_leidos(), memo)
    nueva_persona._libros_propiedad = copy.deepcopy(self.get_libros_propiedad(), memo)

    return nueva_persona
```

En primer lugar, para aquellos atributos que se pasan en el constructor y que son inmutables, simplemente pasaremos el atributo al constructor. Sin embargo, como *hobbies* es una lista y, por tanto, mutable, se debe invocar a su vez a *deepcopy* sobre este atributo. Lo mismo sucede para aquellos atributos de la clase Persona que no se usan en el constructor, requiriendo hacer uso de *deepcopy* para todos ellos al ser mutables (dos listas y un objeto de la clase Libro)

Es importante destacar que el método mágico `__deepcopy__` requiere de un parámetro denominado memo en su funcionamiento interno. Sin embargo, nosotros cuando invoquemos al método lo haremos pasándole como parámetro únicamente el objeto que queremos copiar, de forma similar a *copy()*, abstrayéndonos de este aspecto. Esta copia profunda hará que los atributos mutables generen una copia idéntica y, por tanto, no se produzca aliasing entre objetos.

4.4 Relación de ejercicios

Esta sesión de prácticas parte del código implementado en el último ejercicio de la Sesión 3 donde tenemos implementadas completamente las clases de Polígono, Punto y Línea. A continuación se listan los atributos y métodos que deberían estar definidos en cada clase.

```
Clase Punto

Atributos:
    _coordenada_x: Coordenada X del punto.
    _coordenada_y: Coordenada Y del punto.

Métodos:
    get_coordenada_x()
    set_coordenada_x()
    get_coordenada_y()
    set_coordenada_y()
    obtener_cuadrante()
```

```
Clase Linea

Atributos:
    _punto_inicio: Punto inicial de la línea.
    _punto_fin: Punto final de la línea.

Métodos:
    get_punto_inicio()
```

```
    set_punto_inicio()
    get_punto_fin()
    set_punto_fin()
    calcular_longitud()
    mostrar_puntos()
```

```
Clase Poligono

Atributos:
    numero_poligonos: Atributo de clase que cuenta el número de polígonos creados.
    _numero_lados: Número de lados del polígono.
    _color: Color del polígono.
    _forma: Forma del polígono (convexo, cóncavo).
    _relacion_lados: Relación entre los lados (regular, irregular).
    _lados: Lista de objetos Linea que representan los lados del polígono.
    __contraseña_base_datos: Contraseña interna del polígono.

Métodos:
    get_numero_poligonos()
    set_numero_poligonos()
    get_numero_lados()
    _set_numero_lados()
    get_color()
    set_color()
    get_forma()
    set_forma()
    get_relacion_lados()
    set_relacion_lados()
    get_lados()
    set_lados()
    __get_contraseña()
    __set_contraseña()
    cambiar_contraseña()
    calcular_perimetro()
    escalar_poligono()
    __str__()
    __eq__()
    __len__()
    __add__()
    __lt__()
```

1. **Relación de uso**.
   a) Implementa el método **calcular_distancia_puntos** que calcule la distancia de Manhattan entre dos puntos, *¿en qué clase debes hacerlo?*
   b) ¿Podríamos aprovechar este nuevo método para reutilizarlo en funcionalidad ya existente? ¿Podríamos hacer uso de delegación de métodos entre clases en este ejercicio? Si la respuesta a alguna de estas preguntas es sí, implementa los cambios necesarios.
2. **Relación de asociación**. En este ejercicio, se trabajará con la idea de polígonos vecinos. Un polígono se considera vecino de otro si existe al menos un vértice de uno de los polígonos que se encuentra a una distancia menor o igual a un valor dado (distancia máxima) de cualquier vértice del otro polígono.
   a) Define el atributo **distancia_maxima_vecinos** que representa el valor de la distancia a partir de la cual dos polígonos ya no se considerarían vecinos. El valor de la distancia máxima es un **valor común a todos los polígonos existentes**. Por simplicidad, le asignaremos un valor de 10.
   b) Define el atributo **poligonos_vecinos** que representa un listado de polígonos vecinos asociados al polígono actual (inicialmente vacío).
   c) Implementa el método **es_vecino()** capaz de comparar si otro polígono es vecino del polígono actual.
   d) Implementa el método **agregar_vecino()** para añadir un polígono a la lista de polígonos, siempre que se cumplan las condiciones para serlo.
3. **Relación de composición**. Amplía la clase Poligono para que contenga un **conjunto** de vértices.
   a) Añade el método **get_puntos()** a la clase Linea para que devuelva los dos puntos que componen la línea.
   b) Define el atributo **vertices** que representa la colección de vértices como un conjunto de puntos, creado a partir de las líneas que definen a un polígono.
   c) ¿Debemos modificar métodos ya existentes en la clase Poligono?

   Nota: Este ejercicio asume que existe coherencia en las líneas que forman un polígono. Sin embargo, como no se implementan mecanismos de coherencia, podría pasar que las líneas establecidas no generen realmente un polígono en el código actual.
4. **Clonación**. Este ejercicio hace uso de los métodos **copy** y **deepcopy** proporcionados por el módulo *copy* de Python (sin implementar métodos mágicos) sobre un objeto de la clase Poligono. En tu función *main* realiza:
   a) **Copia superficial**: Clona de manera superficial un polígono. Modifica los atributos **color** y **lados** del polígono clonado. ¿Se modifica el color en ambos objetos? ¿Y los lados? ¿Por qué?
   b) **Copia profunda**: Clona de manera profunda un polígono. Modifica los atributos **color** y **lados** del polígono

clonado. ¿Se modifica el color en ambos objetos? ¿Y los lados? ¿Por qué?

Nota: para hacer este ejercicio crea un método en la clase Polígono llamado **eliminar_lado_aleatorio()**. Este método seleccionará uno de los lados del polígono de forma aleatoria y lo eliminará. ¿Sería necesario actualizar el conjunto de vértices?

5. **Relación de asociación bidireccional**. Implementa los siguientes pasos:
   a) Haz las modificaciones necesarias para que un punto almacene la línea a la que pertenece.
   b) Haz las modificaciones necesarias para que una línea almacene el polígono al que pertenece.