Encapsulación en el Sistema de Biblioteca (Explicación Profundizada)

Retomando el ejemplo del sistema de biblioteca que vimos anteriormente, vamos a analizar cómo aplicar correctamente la encapsulación, marcando el **cómo**, **cuándo** y **porqué** en cada caso.

🔒 1. Atributos Privados en la Clase `Libro`

Cómo:

- Usamos `__` para marcar atributos que no deberían modificarse directamente desde fuera.
- Ejemplo: `__disponible` y `__id_libro` no deberían cambiarse arbitrariamente.

Cuándo:

- Cuando un atributo **debe ser controlado** (ej: `disponible` solo cambia al prestar/devolver).
- Cuando un atributo **es interno** (ej: `id_libro` es autoincremental y no debe modificarse).

Porqué:

- Evita que alguien establezca `disponible = True` sin pasar por `devolver()`.
- Previene que se modifique el `id_libro`, que debe ser único.

Código Modificado:

python

```
class Libro:
    contador_libros = 0

    def __init__(self, titulo, autor, paginas, isbn):
        self.titulo = titulo  # Público (puede consultarse/modificarse libremente)
        self.autor = autor    # Público
        self.paginas = paginas  # Público
        self.isbn = isbn      # Público (pero debería ser inmutable)
        self.__disponible = True  # Privado (solo se cambia con prestar/devolver)
        Libro.contador_libros += 1
```

```
        self.__id_libro = Libro.contador_libros  # Privado (no debe modificarse)

    # Getter para disponibilidad (solo lectura)
    @property
    def disponible(self):
        return self.__disponible

    # Getter para ID (solo lectura)
    @property
    def id_libro(self):
        return self.__id_libro
```

📝 2. Métodos Privados en la Clase `Biblioteca`

Cómo:

- Métodos que solo se usan internamente llevan `__` .
- Ejemplo: `__validar_isbn()` podría usarse antes de agregar un libro.

Cuándo:

- Cuando un método **es de uso interno** y no debe llamarse desde fuera.
- Cuando oculta lógica compleja que no es relevante para el usuario de la clase.

Porqué:

- Simplifica la interfaz pública de la clase.
- Evita que se usen métodos internos que podrían cambiar en el futuro.

Código Modificado:

python

```
class Biblioteca:
    def __init__(self, nombre):
        self.nombre = nombre
        self.__catalogo = []  # Privado (solo se modifica con métodos controlados)

    def agregar_libro(self, libro):
        if self.__validar_libro(libro):  # Método privado
            self.__catalogo.append(libro)
            return f"Libro '{libro.titulo}' agregado."
        return "Libro inválido."
```

```
    # Método privado (solo para uso interno)
    def __validar_libro(self, libro):
        return isinstance(libro, Libro) and libro.isbn
```

🔄 3. Propiedades para Atributos "Protegidos" en `LibroDigital`

Cómo:

- Usamos `@property` para exponer atributos de forma controlada.
- Ejemplo: `__descargas` no debería modificarse directamente, solo incrementarse con `descargar()`.

Cuándo:

- Cuando un atributo **debe ser de solo lectura** (ej: `descargas`).
- Cuando necesitas **calcular un valor al vuelo** (ej: `tamanio_en_kb`).

Porqué:

- Permite añadir lógica adicional (ej: validación, logging) sin cambiar la interfaz.

Código Modificado:

python

```
class LibroDigital(Libro):
    def __init__(self, titulo, autor, paginas, isbn, formato, tamanio_mb):
        super().__init__(titulo, autor, paginas, isbn)
        self.formato = formato  # Público
        self.__tamanio_mb = tamanio_mb  # Privado
        self.__descargas = 0  # Privado

    @property
    def descargas(self):  # Getter (solo lectura)
        return self.__descargas

    @property
    def tamanio_en_kb(self):  # Propiedad calculada
        return self.__tamanio_mb * 1024

    def descargar(self):
```

```
        self.__descargas += 1
        return f"Descargado ({self.__descargas} veces)"
```

⚡ 4. ¿Qué Ganamos con Esta Encapsulación?

Situación Sin Encapsulación	Con Encapsulación
`libro.disponible = True` (peligroso, no registra préstamo)	Solo se cambia con `prestar()` / `devolver()`.
`biblioteca.catalogo.append(libro_falso)` (cosa sin datos)	Solo se agrega con `agregar_libro()`, que valida.
`libro_digital.descargas = -100` (inválido)	Solo se incrementa con `descargar()`.

📌 Ejemplo Final Integrado

python

```
# Uso del sistema con encapsulación
libro1 = Libro("Cien años de soledad", "García Márquez", 432, "9780307474728")
biblioteca = Biblioteca("Central")

biblioteca.agregar_libro(libro1)

print(libro1.disponible)  # True (getter)
# libro1.disponible = False  # Error (no tiene setter)
print(libro1.prestar())  # Cambia __disponible internamente

libro_digital = LibroDigital("Python", "Autor", 300, "123", "PDF", 5)
print(libro_digital.descargar())  # Descargado (1 veces)
print(libro_digital.descargas)  # 1 (solo lectura)
# libro_digital.descargas = 0  # Error (atributo privado)
```

🔎 Conclusión: Encapsulación en el Contexto Real

- **En** `Libro` **:** Protegemos el estado interno (`__disponible` , `__id_libro`).

- **En `Biblioteca`:** Controlamos el acceso al `__catalogo` y validamos con métodos internos.
- **En `LibroDigital`:** Exponemos solo lo necesario (ej: `descargas` como solo lectura).

¿Por qué es importante aquí?

1. **Seguridad:** Evita que se corrompa el estado (ej: marcar un libro como disponible sin devolverlo).
2. **Consistencia:** Todos los cambios pasan por métodos que pueden llevar registro o validar.
3. **Flexibilidad:** Si luego queremos añadir logging o notificaciones, lo hacemos en un solo lugar.

La encapsulación **no es solo ocultar datos**, sino **definir contratos claros** sobre cómo interactuar con un objeto. 🏛️