Práctica Clean Code II

(https://github.com/FranciscoHernandezPuertas/CleanCode2Ev)

Índice

BLOQUE IV: OBJETOS Y ESTRUCTURAS DE DATOS

17. Diferencias entre objetos y estructuras de datos

- La clase `Libro` actúa más como un objeto, ya que encapsula datos (título, autor, ISBN, disponibilidad) y proporciona métodos para operar con esos datos (`marcarComoPrestado`, `marcarComoDevuelto`).

```
13    public class Libro {
14        private final String tituloDefault = "";
15        private final String autorDefault = "" ;
16        private final String ISBNDefault = "";
17        private final boolean disponibilidadDefault = false;
18        private String titulo;
19        private String autor;
20        private String ISBN;
21        private boolean disponible;
```

```
// Método para marcar el libro como prestado
public void marcarComoPrestado() {
    disponible = false;
}

// Método para marcar el libro como devuelto
public void marcarComoDevuelto() {
    disponible = true;
}
```

- La clase `Estanteria` es más como una estructura de datos, ya que expone directamente una lista de libros (`libros`) y proporciona métodos simples para agregar, quitar y buscar libros, sin mucha lógica adicional.

Si estuviera mal:

- Sería incorrecto si la clase `Estanteria` también tuviera lógica compleja asociada, como reglas de negocios específicas de la aplicación. En ese caso, estaría violando el principio de separación entre objetos y estructuras de datos.

```
public class Estanteria {
    private int numeroEstanteria;
    private List<Libro> libros;

    public Estanteria(int numeroEstanteria, List<Libro> libros) {
        this.numeroEstanteria = numeroEstanteria;
        this.libros = libros;
    }

    public int getNumeroEstanteria() {
        return numeroEstanteria;
    }

    public void setNumeroEstanteria(int numeroEstanteria) {
        this.numeroEstanteria = numeroEstanteria;
    }

    // Método para añadir un libro a la estantería
    public void agregarLibro(Libro libro) {
        libros.add(libro);
    }

    // Método para quitar un libro de la estantería
    public void quitarLibro(Libro libro) {
        libros.remove(libro);
    }

    // Método para buscar un libro en la estantería por título
    public Libro buscarLibroPorTitulo(String titulo) {
        for (Libro libro : libros) {
            if (libro.getTitulo().equals(titulo)) {
                return libro;
            }
        }
        return new Libro(); // Si no se encuentra el libro, devolver un libro por defecto si no se encuentra ninguno.
    }
```

18. La Ley de Demeter

- La clase `DAM1HernandezPuertas_Francisco_ActividadCleanCodeII (el método main)` interactúa principalmente con objetos de alto nivel como `Libro`, `Estanteria`, `Seccion` y `Usuario`.
- No accede directamente a los detalles internos de estos objetos, sino que utiliza métodos proporcionados por estas clases, como `tomarLibroPrestado`, `devolverLibro`, `buscarLibroPorTitulo`, etc.

Si estuviera mal:

- Sería incorrecto si el código principal (por ejemplo, el método `main` en este caso) estuviera profundamente involucrado en los detalles internos de las clases, accediendo directamente a los campos internos o métodos de bajo nivel de los objetos.

```
try {
    // Intentar tomar prestado un libro disponible
    if (usuario1.tomarLibroPrestado(libro1)) {
        System.out.println("Libro prestado con éxito a " + usuario1.getNombre());
    } else {
        System.out.println("El libro no está disponible para préstamo");
    }
} catch (RuntimeException e) {
    System.out.println("Error al intentar tomar prestado el libro: " + e.getMessage());
}

try {
    // Intentar devolver un libro que el usuario no tiene prestado
    if (usuario1.devolverLibro(libro2)) {
        System.out.println("Libro devuelto con éxito por " + usuario1.getNombre());
    } else {
        System.out.println("El usuario no tiene este libro prestado");
    }
} catch (RuntimeException e) {
    System.out.println("Error al intentar devolver el libro: " + e.getMessage());
}
```

BLOQUE V: MANEJO DE ERRORES

19. Usa excepciones en lugar de código de retorno

En el método `tomarLibroPrestado` de la clase `Usuario`, se utiliza una excepción (`RuntimeException`) para manejar el caso en que el libro no esté disponible para préstamo.

Si estuviera mal:

- Sería incorrecto si el método devolviera un código de retorno, como `true` o `false`, para indicar el resultado de la operación en lugar de lanzar una excepción. Esto complicaría la lógica de llamada ya que el código cliente tendría que verificar el resultado después de cada invocación.

```
// Método para tomar un libro prestado
public void tomarLibroPrestado(Libro libro) {
    if (libro != null && libro.isDisponible()) {
        librosPrestados.add(libro);
        libro.marcarComoPrestado();
    } else {
        throw new RuntimeException("El libro no está disponible para préstamo");
    }
}

// Método para devolver un libro prestado
public void devolverLibro(Libro libro) {
    if (libro != null && librosPrestados.contains(libro)) {
        librosPrestados.remove(libro);
        libro.marcarComoDevuelto();
    } else {
        throw new RuntimeException("El usuario no tiene este libro prestado");
    }
}
```

20. Escribe primero el try-catch-finally

En el método `main` del archivo `DAM1HernandezPuertas_Francisco_ActividadCleanCodeII.java`, alrededor de los bloques de código que intentan tomar y devolver libros prestados, se utiliza try-catch-finally para manejar excepciones y realizar acciones de limpieza.

```
try {
    // Intentar tomar prestado un libro disponible
    usuario1.tomarLibroPrestado(libro1);
    System.out.println("Libro prestado con éxito a " + usuario1.getNombre());
} catch (RuntimeException e) {
    System.out.println("Error al intentar tomar prestado el libro: " + e.getMessage());
}

try {
    // Intentar devolver un libro que el usuario no tiene prestado
    usuario1.devolverLibro(libro2);
    System.out.println("Libro devuelto con éxito por " + usuario1.getNombre());
} catch (RuntimeException e) {
    System.out.println("Error al intentar devolver el libro: " + e.getMessage());
}
```

21. Usa excepciones unchecked

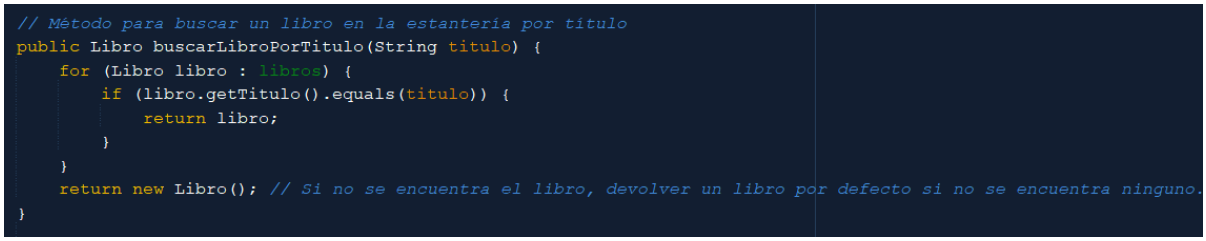
En el método `tomarLibroPrestado` de la clase `Usuario`, se utiliza una excepción no verificada (`RuntimeException`) para indicar problemas durante el proceso de préstamo.

Si estuviera mal:

- Sería incorrecto si se usaran excepciones verificadas (`checked exceptions`) en lugar de excepciones no verificadas (`unchecked exceptions`) en situaciones donde la captura obligatoria podría ser innecesaria y dificultar la legibilidad del código.

22. No devuelvas Null

En el método `buscarLibroPorTitulo` de la clase `Estanteria`, se utiliza un `assertEquals` tanto para un libro existente, como para uno vacío, ya que, si el libro no existe, para evitar devolver null e incumplir el punto 22 (No devuelvas null), se crea automáticamente un libro vacío utilizando el constructor por defecto, y utilizando el método `equals` de la clase `Libro`, podemos igualar el título de dos libros vacíos.



```
// Método para buscar un libro en la estantería por título
public Libro buscarLibroPorTitulo(String titulo) {
    for (Libro libro : libros) {
        if (libro.getTitulo().equals(titulo)) {
            return libro;
        }
    }
    return new Libro(); // Si no se encuentra el libro, devolver un libro por defecto si no se encuentra ninguno.
}
```

BLOQUE VI: TESTS UNITARIOS

24. Mantén limpios los tests & 25. Clean tests

- Los tests están organizados de manera clara y ordenada, con un método de test para cada funcionalidad y utilizando anotaciones como `@DisplayName` y `@Tag` para una mejor identificación.
- El código de los tests sigue buenas prácticas de legibilidad. Los nombres de los métodos y variables son descriptivos y siguen la convención camelCase.
- Los comentarios y anotaciones se utilizan de manera eficiente para explicar el propósito de los tests.

```
@Test
@DisplayName("Prueba para el método tomarLibroPrestado")
@Tag("PruebaBiblioteca")
@Test
@DisplayName("Prueba para el método devolverLibro")
@Tag("PruebaBiblioteca")
```

26. Un Assert por test

Hay tests que cuentan con múltiples assert, esto infringe la regla 26, que no permite más de un assert por test, a continuación, voy a tomar como ejemplo el método `testTomarLibroPrestado`:

```
void testDevolverLibro() {
    // Caso 1: Devolver un libro prestado
    assertDoesNotThrow(() -> {
        usuario1.tomarLibroPrestado(libro1);
        usuario1.devolverLibro(libro1);
        assertTrue(libro1.isDisponible());
        assertFalse(usuario1.getLibrosPrestados().contains(libro1));
});
    // Caso 2: Intentar devolver un libro no prestado
    assertThrows(RuntimeException.class, () -> {
        usuario1.devolverLibro(libro2); // Debe lanzar una excepción
porque el libro2 no está prestado }); }
```

Dado que hay varios assert en este método, podríamos dividirlo en varios métodos de prueba más pequeños, cada uno centrado en un aspecto específico:

```
@Test
@DisplayName("Tomar libro prestado marca el libro como no disponible")
@Tag("PruebaBiblioteca")
void testTomarLibroPrestado_MarcaNoDisponible() {
    usuario1.tomarLibroPrestado(libro1);
    assertFalse(libro1.isDisponible()); }
@Test
@DisplayName("Tomar libro prestado agrega el libro a la lista de libros
prestados del usuario")
@Tag("PruebaBiblioteca")
void testTomarLibroPrestado_AgregaALaLista() {
    usuario1.tomarLibroPrestado(libro1);
    assertTrue(usuario1.getLibrosPrestados().contains(libro1)); }
@Tag("PruebaBiblioteca")
void testTomarLibroPrestado_AgregaALaLista() {
    usuario1.tomarLibroPrestado(libro1);
    assertTrue(usuario1.getLibrosPrestados().contains(libro1));}
```

Con esto, el test `testTomarLibroPrestado` no tendría más de un assert, cumpliendo así con el punto 26.

27. Un único concepto por test

Cada método de test está diseñado para probar una única funcionalidad o concepto. Por ejemplo, `testDevolverLibro` aborda dos casos diferentes (devolver un libro prestado e intentar devolver un libro no prestado), pero ambos están relacionados con la operación de devolución de libros.

```java
@Test
@DisplayName("Prueba para el método devolverLibro")
@Tag("PruebaBiblioteca")
void testDevolverLibro() {
    // Caso 1: Devolver un libro prestado
    assertDoesNotThrow(() -> {
        usuario1.tomarLibroPrestado(libro1);
        usuario1.devolverLibro(libro1);
        assertTrue(libro1.isDisponible());
        assertFalse(usuario1.getLibrosPrestados().contains(libro1));
    });

    // Caso 2: Intentar devolver un libro no prestado
    assertThrows(RuntimeException.class, () -> {
        usuario1.devolverLibro(libro2); // Debe lanzar una excepción porque el libro2 no está prestado
    });
}
```

BLOQUE VII: CLASES

29. Organización de clases

La clase `Libro` sigue el siguiente orden de organización (en caso de existir)

1. Constantes públicas
2. Constantes privadas
3. Variables públicas
4. Variables privadas
5. Funciones públicas
6. Las funciones privadas que son llamadas por las públicas justo debajo de la pública que las llama.

```java
public class Libro {
    private final String tituloDefault = "";
    private final String autorDefault = "" ;
    private final String ISBNDefault = "";
    private final boolean disponibilidadDefault = false;
    private String titulo;
    private String autor;
    private String ISBN;
    private boolean disponible;

    public Libro(String titulo, String autor, String ISBN, boolean disponible) {
        this.titulo = titulo;
        this.autor = autor;
        this.ISBN = ISBN;
        this.disponible = disponible;
    }

    public Libro() {
        this.titulo = tituloDefault;
        this.autor = autorDefault;
        this.ISBN = ISBNDefault;
        this.disponible = disponibilidadDefault;
    }

    public String getTitulo() {
        return titulo;
    }

    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }

    public String getAutor() {
        return autor;
    }

    public void setAutor(String autor) {
        this.autor = autor;
    }
```

30. Las clases deberían ser pequeñas

La clase `Libro` está enfocada en representar un libro y no tiene un crecimiento indiscriminado. El nombre es claro y conciso.

```
public Libro(String titulo, String autor, String ISBN, boolean disponible) {
    this.titulo = titulo;
    this.autor = autor;
    this.ISBN = ISBN;
    this.disponible = disponible;
}
```

31. Principio de Responsabilidad Única

La clase `Estanteria` tiene la responsabilidad de gestionar libros en una estantería, siguiendo el principio de responsabilidad única.

```
// Método para añadir un libro a la estantería
public void agregarLibro(Libro libro) {
    libros.add(libro);
}

// Método para quitar un libro de la estantería
public void quitarLibro(Libro libro) {
    libros.remove(libro);
}
```

32. Cohesión

La clase `Estanteria` tiene cohesión, ya que sus métodos manipulan las variables relacionadas con la estantería.

33. Organiza tu código para prepararlo para el cambio

La clase `Libro` tiene una razón para cambiar: el estado de disponibilidad, por lo que solo tiene una responsabilidad, facilitando su alteración en el futuro.

34. Separa la construcción de un sistema de su uso.

La clase `Seccion` solo tiene métodos para agregar estanterías, lo cual separa la construcción de la sección y su uso.

35. Utiliza copias de objetos para trabajar con concurrencia

En el código, *no* existe la manipulación de objetos utilizando copias de objetos. Si bien sería posible cambiar el código para trabajar con copias en lugar de con el objeto, el diseño de las clases permite la manipulación segura de objetos, ya que se accede a través de métodos y se utilizan listas para almacenar elementos.