



Primer control práctico de MP. Resumen de aspectos a tener en cuenta en la implementación de un proyecto

General

1. Validación de parámetros en todos los métodos públicos y los constructores. Uso del proyecto útil.
 - Para validar aspectos específicos hay métodos `isNotNull(objeto, mensaje)`, `isEmpty(objeto, mensaje)`, `isNotBlank(objeto, mensaje)`.
 - Para validar en general se usa `isTrue(condición, mensaje)`.
 - Si la condición es compleja se crea un método auxiliar que la evalúe y devuelva el resultado booleano. Y se usa en lugar de la condición.
2. Máxima encapsulación. Sólo público lo que sea estrictamente necesario.
 - Atributos siempre privados
 - Get y set públicos, sólo cuando se necesite usarlos fuera del paquete
 - Todos los métodos auxiliares privados
 - El constructor los get y los set de superclase abstracta será paquete si las subclases están en el mismo paquete, y protegido en caso contrario.
 - Los métodos creados sólo para los test se declaran con acceso de paquete (sin modificador).
3. Creación de listas en la declaración (en lugar de en el constructor). Aumenta la robustez de código.
4. Uso de **Herencia para mejorar cohesión** agrupando atributos y métodos comunes en la superclase.
5. Herencia. El constructor de las subclases debe llamar al `super`, e inicializar sus propios atributos.
6. Uso de **herencia para generalizar** y tratar a todos los objetos del mismo modo **delegando la operación en las clases del modelo** (enlace dinámico) y usando polimorfismo (Ejemplo: para todos los ítems ... `item.getTotalPrice()`). `Item` es variable polimórfica (de tipo `Item` en tiempo de compilación, de algún tipo de las subclases en tiempo de ejecución).

Casos

- El método será abstracto en superclase si la operación es diferente para cada subclase.
- El método NO será abstracto si se puede implementar en superclase (`generateCode()`)
- El método se puede definir en la superclase y luego redefinir en alguna de las subclases (por ejemplo, un `print` básico en la superclase que solo imprima título y un `print` específico en alguna de las subclases (si no se implementa en la subclase usa el heredado básico).
- Método implementado con un código en superclase y otro en subclase. Desde el de la subclase se llama también al de la superclase con `super.métodoEnSuperclase()`.



7. Uso de **Interfaz para generalizar**, definiendo un tipo. Toda clase que implemente esa interfaz se puede considerar de este tipo (polimorfismo). Ejemplo `Drawable`. Todos los que se quieran pintar que implementen `Drawable`.
 - Lo que con herencia son métodos abstractos aquí son métodos de la interfaz (se implementan siempre en las subclases).
8. Uso de **Interfaz para reducir el acoplamiento**. En lugar de recibir como parámetro objeto de un tipo concreto (por ejemplo, `Rectangle`), recibimos objetos del tipo de una interfaz (por ejemplo, `Drawable`). No hace falta importar el tipo `Rectangle`, basta con que éste implemente la interfaz. `Rectangle` será de tipo `Rectangle` y de tipo `Drawable`. La interfaz es un contrato con las operaciones que debe implementar una clase que quiera comunicarse con otra.
9. Mejorar la cohesión revisando que los métodos sean cortos y realicen operaciones simples. Extraer bloques de código a métodos auxiliares si el método es demasiado largo (se puede hacer con `refactor / extract Method`).
10. Clases del modelo. Aquellas que almacenan los datos (`Cd`, `Dvd`, `Item`, `Message`, `Photo`, `Rectangle`, `Shape`, `Post`, `Door`, `TemperatureSensor` ...).
11. Clase servicio. Aquella que contiene las listas de objetos del modelo (`MediaLibrary`, `SocialNetwork`, `Canvas`, `TemperatureController` ...). Solo gestiona las listas. Las operaciones se delegan en las clases del modelo. (`calculatePrice`, `toHtmlFormat`, `draw`, `getTemperature` ... se implementan en las clases del modelo).
12. Uso de `instanceof` y `cast` (ver al final de este texto Uso en proyecto DOME).
13. Redefinición de métodos. Se debe incluir `@Override` para evitar errores.
 - Redefinición de `toString` para poder imprimir el objeto (sus datos).
 - Redefinición de `equals` para comparar dos objetos por contenido.
 - Redefinición de cualquier método de la superclase. Si interesa que una subclase se comporte de manera específica.
 - Redefinición de los métodos de una interfaz. Aunque sea una implementación por primera vez, como están definidos en la interfaz se considera `override`.
14. Uso de **`equals` para comparar Strings**.
15. Copia defensiva. Siempre que se devuelva una colección que constituye un atributo del objeto, se debe devolver una copia defensiva. De igual modo, al recibir una colección como parámetro de un constructor, se deberá crear una copia defensiva que será la que se asigna al atributo. La copia defensiva es una copia superficial (se copia la estructura, pero no los objetos).
 - Para hacer copia de un `ArrayList` llamado `list` se puede usar:
`copyList = new ArrayList<Tipo>(list)`
 - Para hacer una copia de un array llamado `array` se puede usar
`copyArray = Arrays.copyOf(array, array.length)`
 - Para hacer copia de un array de 2 dimensiones es necesario crear un array nuevo y asignar a cada fila la copia de cada uno de los arrays que constituyen las filas.

Test unitarios

16. Se crean en carpeta `test`
17. Nombre correcto de paquete de `test`



- Igual al paquete donde está la clase a probar.
- 18. Nombre correcto de clase de test
 - Igual a ClaseAProbarMétodoAProbarTest.
- 19. Casos de test a realizar
 - Todos los casos extremos
 - recibe algo no permitido, la lista está vacía, el valor está justo en el límite, el valor supera el límite.
 - Caso o casos generales
 - Hay varios elementos de todos los tipos, se reciben valores correctos, se reciben diferentes valores que dan lugar a diferentes resultados, etc.
- 20. Usar `@BeforeEach` para inicializar datos generales.
- 21. Añadir lista con todos los casos numerados en un comentario al comienzo de la clase de prueba.
- 22. Comentar siempre los casos con GIVEN... WHEN... THEN.

Herramienta eclipse

- 23. Generación rápida de código
 - Generación automática cuando haga falta a través de menú Source opciones generate. Se debe usar para generar:
 - getters o setters
 - constructores
 - toString
 - equals y hashCode
 - Control + barra espacio completa una palabra con el método, atributo, etc.
 - Uso de quik fix cuando hay error, a través de ventana sobre la palabra que contiene el error (usar solo si se entiende el error y las soluciones propuestas).
- 24. Refactorización
 - Cambio de nombre de clase, de método, de variable, a través de refactor / rename. Lo cambiará en el código en todas las apariciones.
- 25. Configuración.
 - Enlazar proyectos a través de Build path/Configure build path / projects
 - Enlazar Java JRE a través de Build path/Configure build path / libraries / Modulepath / add Libraries / JRE System Library / Execution environment/ JSE 16 (o 17).
- 26. Acesos rápidos
 - Control + click de ratón para navegar por el código hacia adelante
 - Alt + flecha izquierda para navegar hacia atrás.
 - Control + signo + para aumentar el tamaño de letra del código.
 - Control + / para poner comentarios de línea (o seleccionar un párrafo primero).
- 27. Importación / exportación



- Importar File / import / Existing projects into workspace / next / select archive file (si tenemos fichero comprimido) / Browse.
- Select root directory si queremos importar un proyecto que no está comprimido. En este caso seleccionar SIEMPRE opción copy Project into workspace.
- Exportar: File/ export / General / Archive File / (seleccionar ficheros a exportar) / Browse / (seleccionar carpeta donde guardar) / (indicar nombre de fichero comprimido con lo exportado) / guardar

UML

28. Deben estar todas las clases o interfaces del diseño.

29. Relaciones entre clase A y B

- De dependencia (uso) (línea discontinua) si la clase B no es un atributo de A y la clase A necesita a la clase B para compilarse y ejecutarse (por ejemplo, solo se crea B dentro de un método A, se usa y desaparece o B es un parámetro de algún método de A). Esta relación es la menos intensa.
- De **asociación** si B es atributo de A pero no se inicializa en el constructor (hay luego otra operación como un set público o un add...).
- De **agregación** si B es atributo de A y se recibe como parámetro un objeto de A en el constructor. Al eliminar A no se elimina B porque alguien de fuera tiene su referencia (el que la pasó como parámetro).
- De **composición** si B es atributo de A y se crea en el propio constructor. Al eliminar A se elimina B.
- De **herencia** (entre superclase y subclases) triángulo en superclase, línea continua con las subclases.
- De **implementación** (entre interfaz y clases que la implementan) triángulo en interfaz, línea discontinua con las clases que la implementan.

30. Atributos y métodos públicos

- Los atributos de tipo básico (primitivo o clases como String) se incluyen en el UML de la clase.
- Los atributos que son objetos de clases del diseño no se incluyen dentro de la clase, sino que habrá un enlace entre las dos clases. En la línea del enlace se puede poner el nombre del atributo. Por ejemplo, el nombre de la lista.

31. No se incluyen los métodos privados, si se incluyen los públicos.

Proyecto Dome. Dos posibles diseños. Uso de instanceof y cast

Caso 1: Lista de Items y lista de Drawables

Al añadir, hay que añadir en las dos listas. Se usa el instanceof para añadir un objeto a la otra lista.



```
public void add(Item theItem) {
    ArgumentChecks.isNull(theItem, "Invalid item");
    items.add(theItem);
    if (theItem instanceof Borrowable) {           // usamos instanceof
        borrowables.add((Borrowable)theItem);      // hacemos cast
    }
}
```

Para sacar todos los disponibles recorreremos directamente los Borrowables

```
public ArrayList<Borrowable> getAvailables(){
    ArrayList<Borrowable> disponibles = new ArrayList<>();
    for (Borrowable borrowable: borrowables) {
        if (borrowable.isAvailable()) {
            disponibles.add(borrowable);
        }
    }
    return disponibles;
}
```

Caso 2: Únicamente lista de Items

Al añadir, solo se añade en una lista.

```
public void add(Item theItem) {
    ArgumentChecks.isNull(theItem, "Invalid item");
    items.add(theItem);
}
```


Para sacar todos los disponibles hay que usar el instanceof y hacer un cast. Se recorre la lista de items pero se quieren solo los borrowables.

```
public ArrayList<Borrowable> getAvailables(){
    ArrayList<Borrowable> disponibles = new ArrayList<>();
    for (Item item: items) {
        if (item instanceof Borrowable) {           // usamos instanceof
            Borrowable borrowable = (Borrowable)item; // hacemos un cast
            if (borrowable.isAvailable()) {
                disponibles.add(borrowable);
            }
        }
    }
    return disponibles;
}
```

Dónde **NO** se usa instanceof



```
public double getTotalPrice() {  
    double total = 0;  
    for (Item itemInList: items) {  
        if (itemInList instanceof Cd) {  
            total += itemInList.getBasePrice() + TAX;  
        }  
        if (itemInList instanceof Dvd) {  
            total += itemInList.getBasePrice();  
        }  
        if (itemInList instanceof VideoGame) {  
            total += itemInList.getBasePrice() + itemInList.getBasePrice() * PERCENTAGE;  
        }  
    }  
    return total;  
}
```



Cuál sería lo correcto:

- Delegar la operación en las clases del modelo.
- calculateFinalPrice sería abstracto en Item y se implementa en las subclases.

```
public double getTotalPrice() {  
    double total = 0;  
    for (Item itemInList: items) {  
        total += itemInList.calculateFinalPrice();  
    }  
    return total;  
}
```