TDA: Resumen

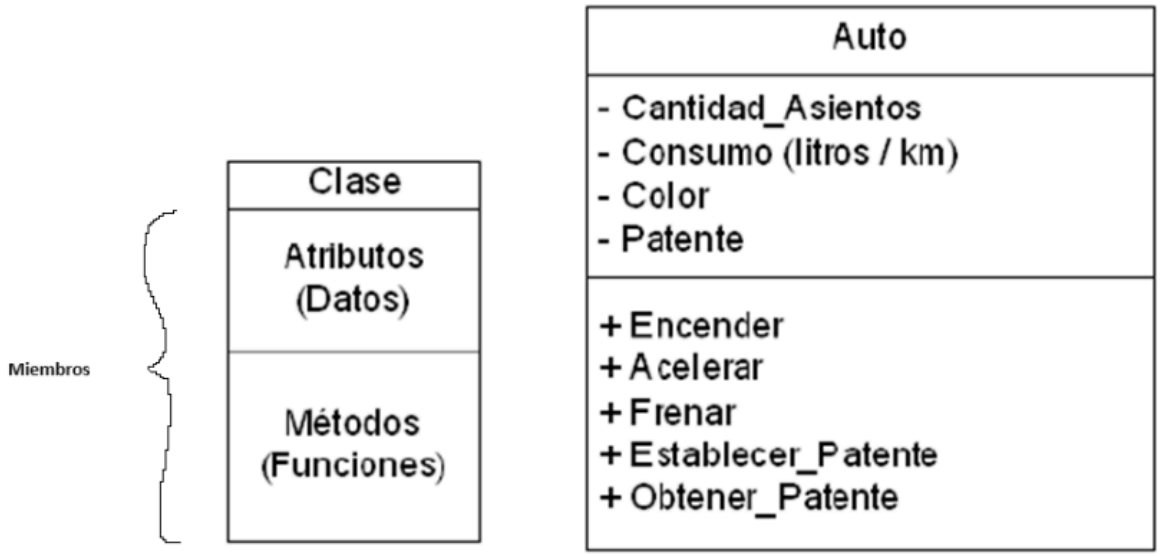
Un *Tipo de Dato Abstracto* (TDA) es un modelo que define valores y las operaciones que se pueden realizar sobre ellos. Es un mecanismo de descripción de alto nivel que modela un concepto y luego lo implementa con una clase.
La abstracción es la capacidad de encapsular y aislar la información del diseño, de la implementación y de la ejecución.

Un TDA se define indicando las siguientes cosas:

- Nombre: Se refiere al tipo de dato y debe ser único.
- Invariantes: indican la validez de los elementos que componen el TDA (qué valores son válidos para conformar un TDA).
- Operaciones y axiomas:
    - OPERACIONES: las que se necesitan que realice el TDA. Se indican con el nombre de la operación, los parámetros y el retorno.
    - AXIOMAS: son las PRE y POST condiciones.

Especificar un conjunto completo de operaciones y axiomas es esencial para garantizar la coherencia, la fiabilidad y la comprensión del TDA. Lo que a su vez facilita su uso, desarrollo y mantenimiento.

COMENZAR A IMPLEMENTAR EL TDA: Plantear el TDA en *UML.*



(a) Diseño genérico (b) Ejemplo de diseño en UML

CARACTERÍSTICAS DE UN TDA:

- Abstracción: Separar, por medio de una operación intelectual, las cualidades de un objeto para considerarlas para resolver el problema aisladamente.

- Encapsulamiento: los datos deberían ser (en general) inaccesibles desde el exterior, para que no puedan ser modificados sin autorización. Para lograr esto se implementan:
    - ➢ **Getters y setters:** Los atributos deben ser privados con el fin de proteger los datos. Los getters indican que podemos tomar algún valor de un atributo y los setters que podemos guardar algún valor sobre un atributo. NO SIEMPRE SON NECESARIOS AMBOS.
    - ➢ **Constructores:** método cuya misión es inicializar un objeto de una clase. En éste se asignan los valores iniciales del nuevo objeto. Posee el mismo nombre de la clase a la cual pertenece y NUNCA DEVUELVE NADA (ni siquiera se puede especificar la palabra reservada void).
    - ➢ **Puntero *this*:** *this* es una referencia a sí mismo que posee todo objeto y se genera de manera automática al invocar un método. Se puede utilizar siempre pero nos veremos obligados a usarlo cuando el compilador no pueda resolver una ambigüedad en los nombres.

- Documentación: al principio de cada método, deben estar las pre y post condiciones.
    - **PRE:** en qué estado debe estar el objeto para llamar a determinado método y cuáles son los valores válidos de sus parámetros.
    - **POST:** cómo va a quedar el estado de la clase luego de ejecutar el método y qué devuelve. Ejemplo:

```
/**
 * pre:
 * @param fecha
 * @param monto en $ y mayor a 0
 * @param tipoDeTransaccion
 *
 * pos: crea una transaccion con los valores dados
 * @throws Exception
 */
public Transaccion(LocalDateTime fecha, double monto, TipoDeTransaccion tipoDeTransaccion) throws Exception {
```

- Robustez: capacidad de un sistema para resistir y recuperarse de errores, fallos o entradas no válidas sin que esto cause un colapso total del sistema. Consiste en: manejar excepción, validar entradas y/o parámetros, la recuperación elegante y pruebas exhaustivas.

- Testeo: hay que hacer una TDA de testeo o un main de prueba usando *JUnit*.
    - ➢ **Método *setUp()*:** utiliza la anotación *@BeforeEach* para inicializar una nueva instancia de la clase antes de cada prueba. Esto asegura que cada prueba comienza con un estado limpio.
    - ➢ **Método *testGetYSet()*:** verifica que los elementos se setean correctamente, y que la pila esté vacía después de desapilar todos los elementos.

- ➢ **Ejecutar las pruebas:** utilizas una herramienta de integración como JUnit 5 en tu IDE (desde propiedades del proyecto). Si las pruebas pasan, significa que tu TDA funciona como se espera. Si alguna prueba falla, deberías revisar la implementación para corregir cualquier error.
- ➢ **OTRAS IDEAS PRINCIPALES:** Asegúrate de que cada prueba sea independiente y se enfoque en una sola funcionalidad; Intenta cubrir todos los caminos posibles en tu código con las pruebas.

Ejemplo:

```java
public class TesteoDeComplejo {

    private Complejo complejo = null;

    @BeforeEach
    void setUp() {
        complejo = new Complejo();
    }

    @Test
    void testApilarYDesapilar() {
        complejo.setParteReal(2);
        complejo.setParteImaginaria(5);
        assertEquals(2, complejo.getParteReal());
        assertEquals(5, complejo.getParteImaginaria());
    }
}
```

- Modularización: dividir un sistema en TDAs más pequeños y cohesivos. Cada TDA tiene una responsabilidad única y realiza una función específica dentro del sistema. A su vez, deben tener una interfaz clara y bien definida que especifica cómo interactuar con él. De esta forma pueden ser reutilizados. TODOS los TDAs deben ser independientes entre sí y tener pocas dependencias externas.

- Campos estáticos, ***static***: es uno que no depende de un objeto en particular sino de la clase en sí. existe una sola copia por clase de un campo que es static.

- Sobrecarga de métodos: aunque el nombre suene negativo, está bien aplicarlo cuando corresponde. Dos o más métodos pueden tener el mismo nombre siempre y cuando difieran en: la cantidad de parámetros y los tipos de los parámetros (o ambos).Esto vale también para los constructores.

- Herencia: permite que una clase "*subclase*" herede atributos y métodos de otra clase "*superclase*". Esto facilita la reutilización de código y la creación de relaciones jerárquicas entre clases.
    - **Superclase:** es de la que se heredan atributos y métodos. Se suele considerar como una clase más general.
    - **Subclase:** es la clase que hereda de la *superclase*. Puede tener atributos y métodos adicionales o sobreescribir los métodos heredados.
    - ***extends*:** se utiliza para indicar que una clase hereda de otra:
        public class Subclase extends Superclase {}.

- Polimorfismo: permite que un objeto de una clase se pueda tratar como un objeto de una superclase o de una interfaz, así un solo método tiene diferentes comportamientos dependiendo del tipo de objeto que lo invoca. TIPOS:
    - **Polimorfismo en tiempo de compilación (Sobrecarga de métodos):** permite tener múltiples métodos en la misma clase con el mismo nombre pero con diferentes parámetros El compilador decide qué método invocar basándose en la firma del método. Ejemplo:

```
public void imprimir() {
    System.out.println("El animal tiene " + edad + " años de edad");
}

public void imprimir(int edadMaxima) {
    System.out.println("El animal tiene " + this.edad + " años de edad (" + this.edad / edadMaxima + ")");
}
```

    - **Polimorfismo en tiempo de ejecución (Sobreescritura de métodos):** cuando una *subclase* proporciona una implementación específica de un método que ya está definido en su *superclase*. El método que se invoca es decidido en tiempo de ejecución, basado en el tipo real del objeto. Ejemplo:

```
3  public class Perro extends Animal {
4
5      public Perro() {}
6
7
8      @Override
9      public void imprimir() {
10         System.out.println("El perro tiene " + getEdad() + " años de edad");
11     }
```