DEFINIR ESTÁNDARES DE CODIFICACIÓN DE ACUERDO CON LA PLATAFORMA ELEGIDA

EVIDENCIA GA7-220501096-AA1-EV02

Nicolas Tamayo Jimenez
Fabian Rodríguez Alzate

Instructor: José Carmelo Álzate

SENA

Ficha #2977373 – ADSO

Agosto de 2025

Introducción

Este informe define los estándares de codificación que se aplicarán al desarrollo de software bajo el paradigma de programación orientada a objetos (POO), empleando los lenguajes Java y Python los cuales utilizaremos para el desarrollo de nuestro proyecto, es importante reconocer el uso consistente de estándares mejora la legibilidad, mantenibilidad y calidad del código, facilitando el trabajo en equipo y reduce los posibles defectos. Las directrices aquí descritas cubren nombramiento, estructura de clases y métodos, documentación, estilo, manejo de excepciones, pruebas y herramientas de automatización.

Objetivo

Establecer un conjunto claro y verificable de estándares de codificación para el proyecto, alineado con buenas prácticas reconocidas en Java (Oracle Java Conventions) y Python (PEP 8/PEP 257), garantizando uniformidad, claridad semántica y calidad técnica a lo largo del ciclo de vida del software.

Nombramiento

Acá encontramos la forma de nombramiento de las diferentes, clases, métodos, variables, constantes y métodos en los diferentes lenguajes que vamos a utilizar con su respectivo ejemplo.

Java

- Paquetes: minúsculas con puntos. Ej.: com.empresa.proyecto.modulo.
- Clases/Interfaces: PascalCase, nombre sustantivo. Ej.: CustomerService, PaymentStrategy.
- Métodos: camelCase, verbo al inicio. Ej.: calculateTotal(), findById().
- Variables y atributos: camelCase, nombres descriptivos. Ej.: maxRetries, customerName.
- Constantes: MAYÚSCULAS_CON_GUIONES_BAJOS. Ej.: DEFAULT_TIMEOUT_MS.
- Genéricos: T, E, K, V cuando el significado sea estándar; usar nombres expresivos si no. Ej.: Result

Python

- Paquetes/Módulos: snake_case en minúsculas. Ej.: data_access, user_service.
- Clases: PascalCase. Ej.: CustomerService.
- Funciones/Métodos: snake_case, verbo al inicio. Ej.: calculate_total(), find_by_id().
- Variables y atributos: snake_case descriptivo. Ej.: max_retries, customer_name.
- Constantes: MAYÚSCULAS_CON_GUIONES_BAJOS. Ej.: DEFAULT_TIMEOUT_MS.
- Tipado: anotaciones de tipo (PEP 484) para firmas públicas.

Declaración y Estructura de Clases

Cada clase debe declararse en el codigo, por ende, las clases deben seguir una estructura para tener buenas practicas logrando un codigo limpio

Java

- Una clase pública por archivo, nombre del archivo igual a la clase.
- Orden recomendado:
    - Constantes.
    - Atributos.
    - Constructores
    - Métodos Públicos.
    - Protegidos.
    - Privados.
    - ToString/Equals/hashCode.
- Aplicar encapsulación (atributos privados, getters/setters cuando corresponda).

Python

- Una clase por archivo cuando sea razonable; agrupar sólo si están estrechamente relacionadas.
- Orden recomendado:
    - Constantes.
    - Atributos de Clase.
    - Init.
    - Métodos Públicos.
    - Privados (prefijo).
    - Repr/Str.
- Encapsulación por convención: prefijos _interno; usar propiedades (@property) para exponer atributos calculados o validados.
- Favorecer dataclasses para modelos inmutables o de datos simples (@dataclass(frozen=True)).

Ejemplo (Java):

```
public final class Money {
    private final long amountInCents;
    private final String currency;

    public Money(long amountInCents, String currency) {
        if (amountInCents < 0) throw new IllegalArgumentException("amount must be >= 0");
        this.amountInCents = amountInCents;
        this.currency = currency;
    }

    public long amountInCents() { return amountInCents; }
    public String currency() { return currency; }

    @Override public String toString() { return amountInCents + " " + currency; }
}
```

Ejemplo (Python):

```
from dataclasses import dataclass

@dataclass(frozen=True)
class Money:
    amount_in_cents: int
    currency: str

    def __post_init__(self):
        if self.amount_in_cents < 0:
            raise ValueError("amount must be >= 0")

    def __str__(self) -> str:
        return f"{self.amount_in_cents} {self.currency}"
```

Declaración de Métodos y Firmas

Los métodos juegan papel fundamental, a través de los métodos se tiene la función definida dentro de una clase que describe el comportamiento de un objeto. A través de los métodos, los atributos de la clase pueden ser utilizados, modificados o consultados, permitiendo que el objeto realice acciones y se comunique con otros objetos.

Java

- Nombres en camelCase, verbales y precisos.
- Parámetros: evitar más de 4; usar objetos de parámetro cuando sea necesario.
- Retornos: preferir tipos específicos a *Object*; usar Optional<*T*> para ausencias.
- Documentar precondiciones/postcondiciones en Javadoc. No atrapar excepciones genéricas salvo justificación.

Python

- Nombres en snake_case, precisos.
- Anotar tipos (PEP 484) y documentar comportamiento con docstrings (PEP 257).
- Evitar argumentos mutables por defecto; usar *None* y documentar.
- Retornar *Optional[T]* o lanzar excepciones; no mezclar ambos sin necesidad.

Ejemplo (Java):

```
/**
 * Calcula el total aplicando impuestos.
 * @param base monto base en centavos
 * @param taxRate porcentaje de impuesto [0..1]
 * @return total en centavos
 * @throws IllegalArgumentException si taxRate es inválido
 */
public long calculateTotal(long base, double taxRate) {
    if (taxRate < 0 || taxRate > 1) throw new IllegalArgumentException("invalid taxRate");
    return Math.round(base * (1 + taxRate));
}
```

Ejemplo (Python):

```
def calculate_total(base: int, tax_rate: float) -> int:
    """Calcula el total aplicando impuestos.
    :param base: monto base en centavos
    :param tax_rate: porcentaje de impuesto [0..1]
    :return: total en centavos
    :raises ValueError: si tax_rate es inválido
    """
    if tax_rate < 0 or tax_rate > 1:
        raise ValueError("invalid tax_rate")
    return round(base * (1 + tax_rate))
```

Comentarios y Documentación

Mediante los comentarios y documentación podemos llevar un control del porque de las cosas, porque se hizo de esta forma, que camino se toma y la explicación de todas las decisiones.

Java

- Javadoc para clases y métodos públicos. Etiquetas comunes: *@param*, *@return*, *@throws*, *@since*.
- Evitar comentarios redundantes. Comentar el porqué, no el qué. Mantener ejemplos de uso cuando agregue valor.

Python

- Docstrings estilo PEP 257 (triple comillas). Estilo Google para parámetros y retornos.
- Usar comentarios en línea sólo cuando aclaren decisiones no obvias. Preferir nombres claros en lugar de comentar obviedades.

Formato de Código (Indentación, Llaves/Espacios)

El formato del codigo es importante para saber el estilo con el que se va a codificar en base a los lenguajes utilizados para el desarrollo del proyecto.

Java

- Identación: 4 espacios. Llaves estilo K&R; en la misma línea.
- Autoformato con Spotless/Checkstyle.

Python

- Identación: 4 espacios. Longitud 88–100. Formateo con Black; verificación con Ruff/Flake8.

Manejo de Excepciones y Errores

El manejo de errores y excepciones es importante manejarlo para tener planes de contingencia por si se presentan errores en el desarrollo del codigo, el como se va a actuar y que excepciones manejar cuando se presenten estos errores.

Java

- Excepciones específicas, no silenciar. try-with-resources para I/O.

Python

- Capturas específicas, no usar except Exception sin necesidad. Context managers (with) para recursos.

Organización de Paquetes y Módulos

Una buena organización de los paquetes y módulos permite mejorar nuestro codigo, teniendo buenas prácticas.

Java

- Estructura por capas o dominio:

com.empresa.proyecto.{domain|app}.{api,service,repository,model}. Una responsabilidad por clase.

Python

- Paquetes con __init__.py. Separar dominio, servicios, adaptadores y presentación. Evitar módulos monolíticos.

Conclusiones

La definición y aplicación de estándares de codificación es un aspecto fundamental dentro del desarrollo de software, ya que permiten garantizar la calidad, mantenibilidad y comprensión del código a lo largo de su ciclo de vida. Tanto en Java como en Python, el uso de convenciones bien establecidas facilita que los equipos de trabajo puedan colaborar de manera más eficiente, reduciendo errores y asegurando que el producto final cumpla con los principios de la programación orientada a objetos (POO), los estándares presentados, que abarcan desde la forma de nombrar variables, clases y métodos, hasta la correcta documentación mediante comentarios y el manejo de excepciones, representan buenas prácticas que promueven el orden y la coherencia. Estas reglas no buscan limitar la creatividad del programador, sino más bien servir como una guía que ayude a construir sistemas robustos, escalables y fáciles de mantener en el tiempo.