Estándares de Codificación para Canadian Visa Advise

Informe Técnico

Grupo 3: Emiliano Mendoza, Cesar Quiroga, Manuel Galindo, Deywid Mora

Tabla de Contenidos

1. Introducción

Este informe define los estándares de codificación que se seguirán en el desarrollo del proyecto Canadian Visa Advise (CVA). Si bien el proyecto no utilizará Java, el material en el que se basa nuestro desarrollo hace referencia a este lenguaje, por lo que los estándares reflejarán las mejores prácticas de codificación en Java. Estos estándares buscan asegurar consistencia, legibilidad, mantenibilidad y adherencia a principios fundamentales del desarrollo orientado a objetos.

2. Objetivo

El objetivo de este informe es establecer un conjunto de reglas y convenciones claras para asegurar que el código desarrollado sea fácil de mantener, entender y extender a lo largo del tiempo. De esta manera, se facilita la colaboración entre desarrolladores y se asegura que el proyecto CVA cumpla con altos estándares de calidad.

3. Lenguaje de Programación

Aunque el proyecto CVA no usará Java, este documento describe las convenciones en base a este lenguaje debido a que se utilizará como referencia académica en las sesiones de clase. Java es un lenguaje orientado a objetos ampliamente utilizado para desarrollar aplicaciones robustas y escalables.

4. Convenciones de Nomenclatura

4.1 Nombres de Clases

- **Formato**: Las clases deben ser nombradas utilizando el formato **UpperCamelCase**, donde cada palabra en el nombre comienza con mayúscula.
- **Sustantivos**: El nombre debe ser un sustantivo que represente una entidad o concepto.

  Ejemplo: `VisaApplication`, `UserProfile`.

Explicación: UpperCamelCase es una convención que mejora la legibilidad y consistencia en los nombres de las clases, permitiendo identificar rápidamente los objetos y sus relaciones en el código.

4.2 Nombres de Métodos

- **Formato**: Los métodos deben usar **lowerCamelCase**, donde la primera palabra comienza en minúscula y las siguientes en mayúscula.
- **Acción**: El nombre debe ser un verbo que describa claramente la acción o función del método.

  Ejemplo: `submitApplication()`, `validateUserInput()`.

Explicación: Los métodos son funciones que realizan acciones específicas, y el uso de verbos facilita entender el propósito del método. LowerCamelCase permite diferenciar fácilmente los métodos de las clases.

4.3 Variables y Atributos

- **Formato**: Las variables y atributos deben nombrarse en **lowerCamelCase**, similar a los métodos.
- **Descripción**: Deben ser sustantivos descriptivos que indiquen claramente su propósito o contenido.

  Ejemplo: `firstName`, `applicationStatus`.

Explicación: Los atributos representan datos o propiedades del objeto. Un nombre descriptivo ayuda a entender qué almacena cada variable y su uso en el código.

4.4 Constantes

- **Formato**: Las constantes se escriben en **UPPER_SNAKE_CASE**, con palabras separadas por guiones bajos.

  Ejemplo: `MAX_UPLOAD_SIZE`, `DEFAULT_TIMEOUT`.

Explicación: Las constantes son valores fijos que no cambian durante la ejecución del programa, y el uso de UPPER_SNAKE_CASE facilita su identificación rápida en el código.

5. Estructura de Clases

5.1 Orden de Elementos

- **Variables de clase (static)**: Se definen primero, ya que afectan a todas las instancias de la clase.
- **Variables de instancia**: Atributos propios de cada objeto, se definen después de las variables de clase.
- **Constructores**: Definen cómo se crean los objetos de la clase.
- **Métodos**: Agrupados por funcionalidad, primero los públicos y luego los privados.

Explicación: Este orden asegura que el código sea consistente y fácil de seguir, facilitando su comprensión y mantenibilidad.

5.2 Encapsulación

- Utilizar modificadores de acceso como `private`, `protected` y `public` para controlar la visibilidad de los atributos y métodos.
- Implementar **getters** y **setters** para acceder a los atributos privados de manera controlada.

Explicación: La encapsulación es clave en la programación orientada a objetos, ya que protege los datos internos de una clase y controla cómo se accede a ellos desde el exterior.

6. Comentarios y Documentación

- Usar **Javadoc** para documentar las clases y métodos públicos, incluyendo descripciones claras de los parámetros y valores de retorno.
- Incluir **comentarios inline** para explicar secciones complejas del código.
- Mantener los comentarios actualizados con el código para evitar confusiones.

Explicación: La documentación clara y precisa ayuda a otros desarrolladores a entender el código sin necesidad de revisar toda la implementación. Los comentarios también facilitan la depuración y el mantenimiento.

7. Manejo de Excepciones

- Utilizar **bloques try-catch** para manejar errores de manera segura.
- Crear excepciones personalizadas cuando sea necesario para situaciones específicas del negocio.
- Registrar (loggear) las excepciones con suficiente detalle para facilitar el diagnóstico de problemas.

Explicación: El manejo adecuado de excepciones previene que el programa falle de manera inesperada y asegura que los errores sean detectados y resueltos de manera oportuna.

8. Principios SOLID

Estos principios son fundamentales para asegurar que el diseño del software sea robusto, escalable y mantenible:

- **S**ingle Responsibility Principle (SRP): Cada clase debe tener una única responsabilidad o motivo para cambiar.
- **O**pen/Closed Principle (OCP): Las clases deben estar abiertas a extensión, pero cerradas a modificación.
- **L**iskov Substitution Principle (LSP): Los objetos de una clase derivada deben poder sustituir a los de su clase base sin alterar el comportamiento del programa.
- **I**nterface Segregation Principle (ISP): Las interfaces deben ser específicas y no forzar a las clases a implementar métodos que no necesitan.
- **D**ependency Inversion Principle (DIP): Las clases deben depender de abstracciones y no de implementaciones concretas.

Explicación: Estos principios guían el diseño de software para que sea flexible y fácil de mantener. Aplicar SOLID asegura que el código sea modular y que los cambios o nuevas características no afecten al resto del sistema.

9. Patrones de Diseño

El uso de patrones de diseño asegura que las soluciones a problemas recurrentes sean claras y eficientes. Algunos de los patrones que podrían aplicarse en CVA son:

- **MVC (Model-View-Controller)**: Para separar la lógica de negocio (Modelo), la interfaz de usuario (Vista) y la lógica de control (Controlador).
- **Factory**: Para crear objetos sin necesidad de especificar la clase exacta a instanciar.
- **Singleton**: Para asegurar que solo exista una instancia de una clase global en todo el sistema.

Explicación: Los patrones de diseño aportan soluciones probadas a problemas comunes en el desarrollo de software, mejorando la arquitectura y reduciendo la complejidad.

10. Conclusiones

Seguir estos estándares de codificación garantiza que el código del proyecto CVA sea claro, consistente y fácil de mantener. La adherencia a principios SOLID y el uso de patrones de diseño proporcionarán una base sólida para el desarrollo de un sistema escalable y robusto.

11. Recomendaciones

- Realizar revisiones de código periódicas para asegurar el cumplimiento de estos estándares.
- Implementar herramientas de análisis estático de código como SonarQube para detectar problemas en el código.
- Revisar y actualizar este documento según evolucione el proyecto.