Actividad – Estilos de Código

Elige tres aspectos de la guía de estilo con los que estés de acuerdo y, para cada uno, explica por qué. Si no encuentras suficientes aspectos con los que estés de acuerdo, da tu mejor interpretación sobre la razón detrás de la selección (con tus propias palabras) y los beneficios que proporciona. Nota: "No hay razón" y "No hay beneficios" no son respuestas aceptables aquí.

Utiliza Java Style Guide (Google), Python Style Guide (Google), PEP 8 (Python Enhancement Proposal).

Guía de Estilo: Java Style Guide(Google)

1. Consistencia de la indentación

Razón: La consistencia en la indentación mejora la legibilidad del código, facilita la colaboración entre desarrolladores y ayuda a evitar errores comunes en la estructura del código.

Beneficio: Un código con una indentación consistente es más fácil de mantener y entender, lo que reduce el tiempo de desarrollo y depuración.

Ejemplo:

```java
//Correcto
public void FundionDeEjemplo() {
    for (int i = 0; i < 10; i++) {
        System.out.println(i);
    }
}

//Incorrecto
public void FuncionDeEjemplo() {
for (int i = 0; i < 10; i++) {
System.out.println(i);
}
}
```

2. Uso de nombres descriptivos para variables y métodos

Razón: Usar nombres descriptivos hace que el código sea más intuitivo y autoexplicativo, lo que facilita la comprensión del propósito de las variables y métodos.

Beneficio: Mejora la mantenibilidad del código y reduce la necesidad de comentarios adicionales para explicar qué hace cada variable o método.

Ejemplo:

```java
//Correcto
public double CalculaAreaRectangulo(double ancho, double largo) {
    return ancho * largo;
}

//Incorrecto
public double calcArea(double a, double b) {
    return a * b;
}
```

3. Espacios alrededor de operadores

Razón: Incluir espacios alrededor de los operadores aritméticos y de asignación mejora la legibilidad del código al hacerlo visualmente más claro y menos denso.

Beneficio: Un código con espacios bien definidos es más fácil de leer y entender rápidamente, lo que puede reducir errores y manejar la productividad.

Ejemplo:

```java
//Correcto
int a = b + c;

//Incorrecto
int a=b+c;
```

Muestra código donde aplicas lo que estás de acuerdo de la guía que elegiste.

Elige tres aspectos de la guía de estilo con los que no estés de acuerdo y, para cada uno, explica por qué. Si no encuentras suficientes aspectos con los que no estés de acuerdo, proporciona tu mejor interpretación de por qué alguien podría no estar de acuerdo y qué posibles desventajas podría tener su uso.

1. Líneas máximas de 100 caracteres

Razón de desacuerdo: Limitar las líneas de código a 100 caracteres puede llevar a que el código se divida de maneras que no son naturales, haciendo que sea más difícil de leer y seguir.

Desventaja: Esto puede resultar en un código con muchas líneas cortadas de manera innecesaria, lo cual puede complicar la lectura y comprensión del flujo lógico del programa.

Ejemplo:

```
//Según la guía debe tener un límite de 100 caracteres
public void FuncionDeEjemploConParametrosLargos(String Parametro1, String
    Parametro2,
    String Parametro3) {
    // ...
}

//Preferencia personal más de 100 caracteres en una línea, pero más claro
public void FuncionDeEjemploConParametrosLargos(String Parametro1, String
    Parametro2, String Parametro3) {
    // ...
}
```

2. Uso obligatorio de espacios después de palabras clave

Razón de desacuerdo: La obligación de incluir un espacio después de cada palabra clave (como if, for, while) puede ser vista como una regla innecesaria que no contribuye significativamente a la legibilidad del código.

Desventaja: Algunos desarrolladores pueden encontrar que esta regla ralentiza su flujo de trabajo sin mejorar de manera apreciable la claridad del código.

Ejemplo:

```
//Según la guía debe de tener espacio después de palabra clave
if (condition) {
    // ...
}

// Preferencia personal sin espacio, sin pérdida de claridad
if(condition) {
    // ...
}
```

3. Comentarios Javadoc en métodos Privados

Razón de desacuerdo: Requerir comentarios Javadoc en métodos privados puede ser excesivo, ya que estos métodos no están destinados a ser utilizados fuera de su clase y a menudo su propósito es evidente en el contexto del código.

Desventaja: Esto puede llevar a una sobrecarga de documentación, haciendo que el código sea más difícil de mantener sin un beneficio claro en términos de comprensión.

Ejemplo:

```
// Según la guía debe de haber comentarios Javadoc en métodos privados
/**
 * este metodo procesa los datos.
 */
private void ProcesarDatos() {
    // ...
}

// Preferencia personal sin comentarios Javadoc en métodos privados
private void ProcesarDatos() {
    // ...
}
```

Análisis de código estático

Selecciona una herramienta de análisis estático reconocida con documentación en línea. Elige una de sus funciones, describe brevemente su propósito, cómo utilizarla y su importancia. Luego, selecciona otra característica de la misma herramienta, explica su propósito, uso e importancia.

Selección de la Herramienta de Análisis Estático: SonarQube

SonarQube es una herramienta de análisis estático bien establecida, conocida por su capacidad para inspeccionar continuamente la calidad del código. Ofrece documentación extensa en línea y es ampliamente utilizada en la industria.

Función Seleccionada: Análisis de Duplicación de Código
Propósito de la Función: El análisis de duplicación de código en SonarQube identifica bloques de código que se repiten en el proyecto. Esto ayuda a los desarrolladores a detectar y eliminar redundancias que pueden hacer el código difícil de mantener y propenso a errores.

Cómo Usarla:
Configuración Inicial: Instala y configura SonarQube siguiendo la documentación oficial.
Proyecto de Análisis: Ejecuta un análisis en tu proyecto utilizando el SonarQube Scanner, que puede ser integrado en tu pipeline de CI/CD.
Revisión de Resultados: Después del análisis, revisa los resultados en el tablero de SonarQube. Los bloques duplicados se marcarán claramente en los informes.
Refactorización: Basado en los resultados, refactoriza el código para eliminar duplicaciones y mejorar la mantenibilidad.
Importancia de la Capacidad: Detectar duplicaciones de código es crucial para mantener un código limpio y manejable. Las duplicaciones aumentan la complejidad del código, dificultan la corrección de errores y la

implementación de nuevas funcionalidades. La capacidad de SonarQube para identificar estas duplicaciones ayuda a los equipos a mantener un alto estándar de calidad del código.

Segunda Característica: Análisis de Complejidad Ciclomática
Propósito de la Característica: La complejidad ciclomática mide el número de caminos independientes a través del código. Esta métrica ayuda a los desarrolladores a entender la complejidad de sus funciones y clases, lo que puede influir en la facilidad de prueba y mantenimiento.

Cómo Usarla:
Configuración Inicial: Utiliza SonarQube como se describió anteriormente.
Ejecución del Análisis: Realiza el análisis del proyecto.
Revisión de Resultados: Examina los resultados en el tablero de SonarQube, donde se presentará la complejidad ciclomática para cada función y clase.
Refactorización: Identifica funciones con alta complejidad ciclomática y refactorízalas para reducir la complejidad. Esto puede implicar dividir funciones grandes en funciones más pequeñas y manejables.
Importancia de la Capacidad: La alta complejidad ciclomática puede hacer que el código sea difícil de entender, probar y mantener. Reducir la complejidad ciclomática mejora la legibilidad del código y facilita la detección y corrección de errores, así como la adición de nuevas características. Tener esta capacidad de análisis en SonarQube permite a los desarrolladores identificar rápidamente áreas problemáticas y abordarlas proactivamente.

Comentarios y documentación

Selecciona un proyecto de código abierto que te interese. Encuentra y revisa un archivo del proyecto mediante URL.

Describe el propósito del archivo y proporciona el enlace. Evalúa si sigue un estándar de estilo de código, indicando ejemplos de por qué no cumple o detallando el estándar formal si lo sigue. Proporciona sugerencias de mejora para el archivo o menciona tres aspectos del estilo con los que estás de acuerdo. Evalúa si el archivo está bien documentado. Explica la autodocumentación, proporcionando ejemplos de comentarios y ubicaciones comunes en el código. Identifica lugares donde el código se autodocumenta bien y mal, explicando las razones.

Selección del Proyecto
Nombre del Proyecto: React

Propósito del Archivo: El archivo App.js en un proyecto típico de React sirve como el componente principal de la aplicación. Es el punto de entrada donde se estructuran y organizan los componentes hijos.

Enlace al Archivo:
https://github.com/facebook/react/blob/main/packages/react-dom/src/client/ReactDOM.js

Evaluación del Estándar de Estilo de Código
Sigue un estándar de estilo de código? : Sí

Estándar Formal: Utiliza ESLint con reglas de Airbnb, un estándar formal ampliamente adoptado.

Ejemplos de Cumplimiento del Estándar:
1. Indentación y espacios: El archivo utiliza una indentación consistente de 2 espacios, lo cual es recomendado por el estándar de Airbnb.

```
function App() {
  return (
    <div className="App">
      <header className="App-header">
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
      </header>
    </div>
  );
}
```

2. Uso de comillas simples: Las comillas simples se utilizan para las cadenas de texto, siguiendo la convención de estilo.

```
const greeting = 'Hello, World!';
```

3. Funciones flecha: Se utilizan funciones flecha para definir componentes funcionales, siguiendo las recomendaciones de ES6.

```
const App = () => {
  return (
    <div className="App">
      {/* contenido */}
    </div>
  );
}
```

Propuestas de mejora

El archivo “App.js” ya sigue buenas prácticas y estándares de estilo. Sin embargo, se pueden considerar las siguientes mejoras:

1. Comentarios: Incluir más comentarios explicativos para mejorar la comprensión del código.

```jsx
// Componente principal de la aplicación
const App = () => {
  return (
    <div className="App">
      {/* Contenido del encabezado */}
      <header className="App-header">
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
      </header>
    </div>
  );
}
```

2. Pro Types: Definir las PropTypes para los componentes, lo que ayuda a la validación de las propiedades.

```jsx
import PropTypes from 'prop-types';

const App = ({ title }) => {
  return (
    <div className="App">
      <header className="App-header">
        <p>{title}</p>
      </header>
    </div>
  );
}

App.propTypes = {
  title: PropTypes.string.isRequired,
};
```

Documentación del archivo

¿Está bien documentado?: No complemente. Si bien el código es claro y sigue un estándar de estilo, le falta documentación explícita como comentarios y ProTypes.

Autodocumentación del código

1. Lugar bien documentado:

```
// Componente principal de la aplicación
const App = () => { ... }
```

Por qué es bueno: Proporciona contexto inmediato al lector sobre la función del componente.

2. Lugar bien documentado:

```
<header className="App-header">
  {/* Contenido del encabezado */}
  <p>...</p>
</header>
```

Por qué es bueno: Clarifica la estructura y propósito de la sección dentro del JSX.

Código no autodocumentado:

1. Lugar mal documentado:

```
const App = () => { ... }
```

Por qué es malo: La función podría beneficiarse de comentarios sobre el propósito y uso del componente.

2. Lugar mal documentado:

```
<div className="App">
  {/* contenido */}
</div>
```

Por qué es malo: La falta de descripciones detalladas puede dificultar la comprensión del código para nuevos desarrolladores.