**Documento de Arquitectura y Patrones de Diseño del Sistema de Seguridad**

**Proyecto:** Sistema de Seguridad Avanzado para Urbanización **Autor:** [Andres Felipe Chicaiza, Santiago lopez león , Eduar Ruiz]

**1. Introducción: Fundamentos del Diseño Arquitectónico**

Este documento profundiza en la arquitectura de software del sistema de seguridad, detallando la aplicación de patrones de diseño y principios que garantizan su **modularidad, escalabilidad y mantenibilidad**. La elección de un diseño robusto y flexible es fundamental para la longevidad del proyecto, permitiendo que crezca y se adapte a nuevos requisitos sin requerir una reestructuración completa. El sistema está concebido no solo para funcionar, sino para evolucionar de manera eficiente.

**2. Principios de Diseño de Software: La Guía SOLID**

Los **principios SOLID**, un acrónimo para los cinco principios de diseño de software orientado a objetos, son una base crucial para tu proyecto. Aseguran una estructura limpia y un código de alta calidad.

* **Principio de Responsabilidad Única (SRP):**
  + **Concepto:** Cada clase, módulo o función debe tener una y solo una razón para cambiar. Esto significa que cada componente se especializa en una tarea específica, minimizando el impacto de los cambios.
  + **Aplicación en tu sistema:** Tu proyecto ejemplifica el SRP a través de una división de responsabilidades muy clara. El archivo **app.py** es un servidor web monolítico, pero con una responsabilidad única y bien definida: actúa como la **capa de API REST**. Su trabajo es recibir solicitudes HTTP, comunicarse con la base de datos y devolver respuestas. El archivo **python.py**, por otro lado, está especializado en la **lógica de negocio interna**, sin tener conocimiento del protocolo HTTP o la base de datos. Aloja el EventBus y los algoritmos. Esta separación de preocupaciones permite que un cambio en la base de datos (por ejemplo, de PostgreSQL a otra tecnología) solo afecte a app.py, mientras que la lógica de python.py permanece inalterada.
* **Principio Abierto/Cerrado (OCP):**
  + **Concepto:** Las entidades de software deben estar **abiertas para la extensión**, lo que permite agregar nueva funcionalidad, pero **cerradas para la modificación**, para que los cambios no alteren el código existente que ya funciona.
  + **Aplicación en tu sistema:** El patrón **Observer**, implementado como el **EventBus**, es una manifestación perfecta del OCP. Si necesitas agregar un nuevo comportamiento (por ejemplo, enviar un mensaje a un *smartphone* al registrar una ronda), no es necesario modificar la función registrarRonda() en Celador.html ni la clase EventBus en python.py. Simplemente creas un nuevo "observador" o servicio y lo suscribes al tema correspondiente. Esto hace que el sistema sea altamente extensible y resistente a cambios.

**3. Patrones de Diseño: Soluciones Probadas**

Tu sistema utiliza patrones de diseño para resolver problemas de arquitectura de forma eficiente y probada, lo que mejora la reusabilidad y el mantenimiento del código.

* **Patrón de Comportamiento: Observer**
  + **Concepto:** Define una dependencia **uno a muchos**, donde un objeto (sujeto) notifica a múltiples objetos dependientes (observadores) cuando su estado cambia. El objetivo es que los observadores puedan reaccionar a los eventos sin que el sujeto tenga que conocer la identidad o la estructura de los observadores.
  + **Aplicación en tu sistema:** El **EventBus** centralizado en python.py actúa como el **sujeto**. Las clases Logger y Notificador son los **observadores**. Cuando el usuario presiona el botón "Registrar Ronda", se publica un evento "ronda" al bus. El bus, sin saber quién está escuchando, distribuye el evento a todos los suscriptores. Esto crea un **acoplamiento débil**, lo que significa que los componentes son casi independientes entre sí, haciendo el sistema más robusto y flexible.
* **Patrón Estructural: Adapter (Uso Potencial y Estratégico)**
  + **Concepto:** El patrón Adapter permite que clases con **interfaces incompatibles** colaboren entre sí. Actúa como un "traductor" o "envoltorio" que adapta la interfaz de una clase existente a la interfaz que se espera.
  + **Uso potencial en el sistema:** El sistema actual utiliza psycopg2 para interactuar con la base de datos PostgreSQL. Si en el futuro se decide migrar a una base de datos NoSQL como MongoDB, un **DatabaseAdapter** podría crearse. Este *adapter* implementaría los métodos de consulta de tu API, como get\_accesos(), pero internamente los traduciría a las operaciones de MongoDB. El archivo app.py seguiría llamando a los mismos métodos, sin notar el cambio en la tecnología subyacente. Esto protegería al resto del código de una dependencia rígida en PostgreSQL, facilitando las futuras migraciones.
* **Patrón Creacional: Factory (Uso Potencial y Estratégico)**
  + **Concepto:** El patrón Factory proporciona una interfaz para la creación de objetos en una superclase, permitiendo a las subclases decidir qué tipo de objeto se creará. Su propósito es encapsular la lógica de creación.
  + **Uso potencial en el sistema:** Para la gestión de usuarios y accesos, se podría implementar una **AccessFactory**. Este *factory* recibiría un tipo de usuario (residente, visitante, celador) y devolvería el objeto de acceso correspondiente, cada uno con su propia lógica. Por ejemplo, AccessFactory.create\_access('residente') crearía un objeto AccesoResidente, mientras que AccessFactory.create\_access('visitante') crearía un AccesoVisitante. Esto centraliza la creación de objetos y evita que el código cliente esté lleno de condicionales (if/else) para determinar qué clase instanciar.

**4. Algoritmos y Lógica de Negocio: Más Allá de los Patrones**

Más allá de los patrones, el corazón de la arquitectura de tu sistema incluye algoritmos diseñados para la eficiencia.

* **Búsqueda Bidireccional:**
  + **Concepto:** Este algoritmo de búsqueda de ruta es más eficiente que una búsqueda tradicional en una sola dirección. Lo logra al buscar simultáneamente desde el punto de inicio y el de destino hasta que se encuentran en el punto medio. Esto reduce drásticamente el número de nodos que deben ser explorados, especialmente en grafos grandes.
  + **Aplicación en el sistema:** La función bidirectional\_search() en python.py es un componente clave. En el contexto del sistema de seguridad, este algoritmo es ideal para **optimizar las rondas del celador**. Al tener un mapa de la urbanización como un grafo, el algoritmo puede calcular rápidamente la ruta más corta desde la posición actual del celador hasta un punto de interés o entre dos puntos de control, mejorando los tiempos de respuesta ante incidentes.

**5. Conclusión General del Diseño**

El diseño arquitectónico del sistema es una base sólida y bien pensada. La adhesión a los principios SOLID y la implementación estratégica de un patrón **Observer** para el EventBus aseguran un código limpio, un bajo acoplamiento y una gran capacidad de expansión. La inclusión de un algoritmo tan eficiente como la **búsqueda bidireccional** demuestra un enfoque en el rendimiento. Finalmente, la consideración de patrones como **Factory** y **Adapter** para el futuro asegura que el sistema no solo funcione hoy, sino que pueda crecer y adaptarse a los desafíos tecnológicos de mañana.