1. R. Sí, el proyecto Healthy+ implementa una arquitectura modular organizada en capas, donde la estructura del proyecto separa la configuración de la base de datos en la carpeta config, los modelos de datos como Medicamento, EventoToma y Sintoma en la carpeta models, los controladores que manejan la lógica de negocio en controllers, los servicios especializados como el Administrador de Almacenamiento y los alertadores en servicios, y al final las vistas en views por lo que esta separación permite que si se necesita cambiar la base de datos, únicamente se modificarían los archivos Gestor Almacenamiento.js y Database.js sin afectar a los controladores ni a las vistas.

2. R. En el proyecto se observa alta cohesión ya que cada componente tiene una responsabilidad única y bien definida, por ejemplo el ControladorMedicamentos se encarga exclusivamente de la gestión de medicamentos, el Controlador Alertas maneja únicamente las alertas y notificaciones, el GestorAlmacenamiento se dedica solo a la persistencia de datos, el Notificador Sonoro genera exclusivamente avisos sonoros, mientras que el Notificador Visual se encarga únicamente de las notificaciones visuales en la pantalla.

Por otro lado en cuanto al bajo acoplamiento, el proyecto implementa el patrón Observer mediante las clases Sujeto y Observador, aquí la clase Sujeto define un contrato abstracto con métodos para suscribir, desuscribir y notificar observadores, también la clase Observador define una interfaz con el método actualizar que debe ser implementado por las clases concretas, también, el RelojSistema, que actúa como sujeto, no conoce a los observadores como NotificadorSonoro o NotificadorVisual, sino que trabaja únicamente con la abstracción Observador, esto permite agregar nuevos tipos de notificadores, como por ejemplo uno de mensajes de texto o correo electrónico, sin necesidad de modificar el código del reloj.

3. R. El proyecto aplica esto correctamente, pues en el caso del Medicamento, toda la revisión de los datos se hace dentro de su función de validar, por otro lado, los modelos tienen reglas de conversión usando toJSON y fromJSON, donde estos métodos funcionan como traductores, pues se encargan de transformar la información de la aplicación a un formato listo para guardarse, y también hacen el proceso inverso, de esta manera podemos guardar y recuperar datos sin necesidad de revelar cómo está organizada la base de datos por dentro, además, la clase ConexionBD implementa el patrón Singleton para asegurar una sola instancia de conexión a la base de datos, encapsulando todo el proceso de inicialización y configuración de IndexedDB.

4. R. El proyecto cumple con este principio mediante una separación en tres capas, donde la capa de interfaz de usuario está representada por las clases VistaListaMedicamentos, VistaSintomas y VistaHistorial, las cuales se encargan solo del renderizado de elementos en el DOM y la captura de eventos del usuario, sin contener lógica de negocio.

La capa de lógica de negocio está implementada en los controladores como Controlador Medicamentos, Controlador Alertas, Controlador Formulario, Controlador Historial y Controlador Sintomas, donde estos componentes contienen las reglas de negocio, las validaciones y la coordinación entre las otras capas.

La capa de datos está compuesta por el GestorAlmacenamiento y Database, los cuales gestionan todo el acceso a IndexedDB de forma independiente, por otro lado el archivo app.js conecta las vistas con los controladores mediante callbacks, pero sin mezclar las responsabilidades de cada capa.

5. R. El proyecto aplica patrones de diseño como el patrón Singleton que se implementa en la clase ConexionBD para garantizar una única conexión a la base de datos durante toda la ejecución de la aplicación, evitando conflictos y consumo innecesario de recursos, también el patrón Observer está implementado a través de las clases Sujeto, Observador y Reloj Sistema, donde este se extiende de Sujeto y monitorea continuamente si hay medicamentos pendientes de tomar y cuando detecta una alerta, notifica a todos los observadores suscritos, que en este caso son Notificador Visual y Notificador Sonoro, donde ambos se extienden de La clase Observador y aplican el método actualizar de manera particular, y como se nota este patrón facilita la ampliación del sistema ya que para agregar un nuevo tipo de notificación solo se necesita crear una nueva clase que extienda Observador y suscribirla al Reloj Sistema.

6. R. Entre los aspectos positivos incluyen que los modelos como Medicamento y Eventos Toma tienen métodos de validación que pueden probarse de forma unitaria sin dependencias externas, se puede crear una instancia de Medicamento con datos erróneos y verificar que el método validar retorne los errores esperados, todo esto sin necesidad de base de datos ni interfaz gráfica.

Sin embargo el Gestor Almacenamiento depende directamente de IndexedDB sin una capa de abstracción que permita simular la base de datos.

7.R. Nuestro proyecto tiene partes de seguridad, como la validación de entradas, y también tanto la clase Medicamento como la clase Evento Toma tienen funciones de comprobación que aseguran que los campos esenciales están presentes, los formatos son adecuados y los valores se encuentran dentro de límites permitidos, por ejemplo, se valida que el nombre del medicamento no esté vacío, que la dosis sea mayor a cero, que la frecuencia sea válida, que esté dentro del rango, y que la hora tenga el formato correcto HH:MM.

Con respecto a la auditoría, el modelo EventoToma tiene información completa sobre cada acción relacionada con la toma de medicamentos, teniendo también el id del medicamento, el tipo de evento que puede ser como omitido o pospuesto, la fecha y hora programada para su toma, la fecha y hora real de la acción, la dosis administrada y el motivo en caso de omisión o postergación.