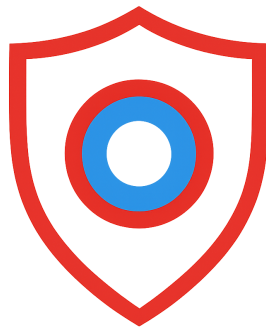


SentinelGuard

Sistema de Seguridad Inteligente



Trabajo Final de Desarrollo Orientado a Objetos

Autor/a:

María Muñoz Nadales

Índice

1. Introducción	3
2. Objetivos del proyecto	3
3. Visión general del sistema SentinelGuard	3
4. Arquitectura general	3
5. Diseño orientado a objetos	4
5.1. Encapsulación	4
5.2. Separación de responsabilidades	4
5.3. Uso de gestores	4
6. Modelado UML	4
6.1. Casos de uso: visión global del producto	5
6.2. Diagrama de clases: núcleo de dominio y servicios	6
6.3. Actividad: login y carga dinámica por perfil	7
6.4. Secuencia: disparo de sensor con sistema armado	8
6.5. Secuencia: creación de modo personalizado	9
6.6. Diagrama de estados: ciclo completo de una alarma	10
7. Modos de seguridad (catálogo completo y diseño de pantalla)	10
7.1. Modos por perfil (catálogo completo)	11
7.2. Diseño de la pantalla de Modos (UI/UX con lógica realista)	11
8. Sensores (catálogo completo, simulación y diseño de pantalla)	12
8.1. Sensores por perfil (catálogo completo)	12
8.2. Simulación realista: por qué no se puede “disparar” con el sistema desarmado	13
8.3. Diseño de la pantalla de Sensores (UI/UX)	13
8.4. Qué ocurre al simular un evento	13
9. Armado del sistema y temporizadores	14
10. Persistencia y trazabilidad (diseño empresarial)	14
10.1. Qué se persiste y por qué	14
10.2. Separación entre estado de sesión y persistencia	14
10.3. GestorBD como capa única de acceso	15
10.4. Historial como auditoría	15

11. Interfaz y experiencia de usuario (UX/UI con razonamiento real)	15
11.1. Lenguaje visual (colores semánticos)	16
11.2. Estructura de pantallas y consistencia	16
11.3. Pantalla Inicio: decisiones aparentemente pequeñas (pero críticas)	16
11.4. Cuenta atrás integrada (sin spam de notificaciones)	16
11.5. Pop-ups informativos persistentes (icono)	17
11.6. Logo y naming	17
12. Descripción de archivos (visión técnica y propósito)	17
12.1. Archivos de interfaz (UI)	17
12.2. Archivos de dominio (núcleo OO)	18
12.3. Persistencia (base de datos e historial)	18
12.4. Servicios	18
12.5. Recursos del proyecto	18
13. Errores comunes y dificultades	19
14. Errores comunes, dificultades y aprendizaje durante el desarrollo	19
14.1. Gestión del estado del sistema	19
14.2. Dependencia excesiva de la interfaz	19
14.3. Temporizadores y concurrencia	19
14.4. Refrescos de interfaz y experiencia de usuario	20
14.5. Definición del perfil mixto	20
14.6. Conclusión del aprendizaje	20

1 Introducción

SentinelGuard es un sistema de alarma inteligente diseñado como proyecto final de la asignatura *Diseño y Desarrollo Orientado a Objetos (DOO)*. El proyecto simula el funcionamiento realista de una alarma profesional, integrando lógica de negocio, persistencia, estados, temporizadores y una interfaz gráfica coherente con aplicaciones reales del sector de la seguridad.

A lo largo del desarrollo se ha priorizado que cada decisión técnica, estructural y visual tenga una justificación clara, alineada tanto con los contenidos teóricos de la asignatura como con criterios de experiencia de usuario y realismo funcional.

2 Objetivos del proyecto

- Aplicar principios de diseño orientado a objetos en un sistema completo.
- Modelar correctamente estados, sensores y modos de seguridad.
- Separar responsabilidades entre interfaz, lógica y persistencia.
- Simular comportamientos reales de sistemas de alarma profesionales.
- Diseñar una interfaz clara, consistente y justificada.

3 Visión general del sistema SentinelGuard

SentinelGuard es un sistema de seguridad modular que permite a distintos tipos de usuarios gestionar una alarma mediante perfiles, modos de seguridad y sensores simulados. El sistema está diseñado para comportarse como una alarma real, incorporando retardos de entrada y salida, disparos de sensores, estados de alarma y registro persistente de eventos.

4 Arquitectura general

El sistema se estructura conceptualmente en tres grandes capas:

- **Interfaz de usuario (UI):** pantallas de login, inicio, modos, sensores e historial.
- **Lógica de dominio:** gestión de alarmas, modos, sensores y estados.
- **Persistencia:** base de datos encargada de almacenar usuarios, modos y eventos.

Esta separación permite un desarrollo mantenible, facilita la depuración y refleja una arquitectura limpia alineada con los principios de DOO.

5 Diseño orientado a objetos

SentinelGuard ha sido diseñado siguiendo los principios fundamentales de la orientación a objetos:

5.1 Encapsulación

Cada entidad relevante del sistema (Usuario, Alarma, Sensor, ModoSeguridad) encapsula su propio estado y comportamiento. Esto evita dependencias innecesarias y facilita la evolución del sistema.

5.2 Separación de responsabilidades

La lógica de negocio no se implementa en la interfaz. Las pantallas de la UI únicamente reflejan el estado del sistema y delegan las acciones en clases de dominio o gestores.

5.3 Uso de gestores

Se introduce el concepto de *gestores* (GestorAlarma, GestorBD) para centralizar operaciones complejas y evitar que las entidades conozcan detalles de persistencia o coordinación global.

6 Modelado UML

En un sistema de seguridad real, el UML no es “decoración académica”: es un instrumento de diseño que permite:

- Validar que la lógica del sistema refleja comportamientos reales (alarmas, retardos, disparos).
- Anticipar inconsistencias de estado (por ejemplo, armar dos veces, cambiar de modo con la alarma activa).
- Definir responsabilidades: qué capa decide, cuál ejecuta y cuál persiste.
- Facilitar mantenimiento y extensibilidad: nuevos sensores, nuevos modos, nuevos perfiles.

6.1 Casos de uso: visión global del producto

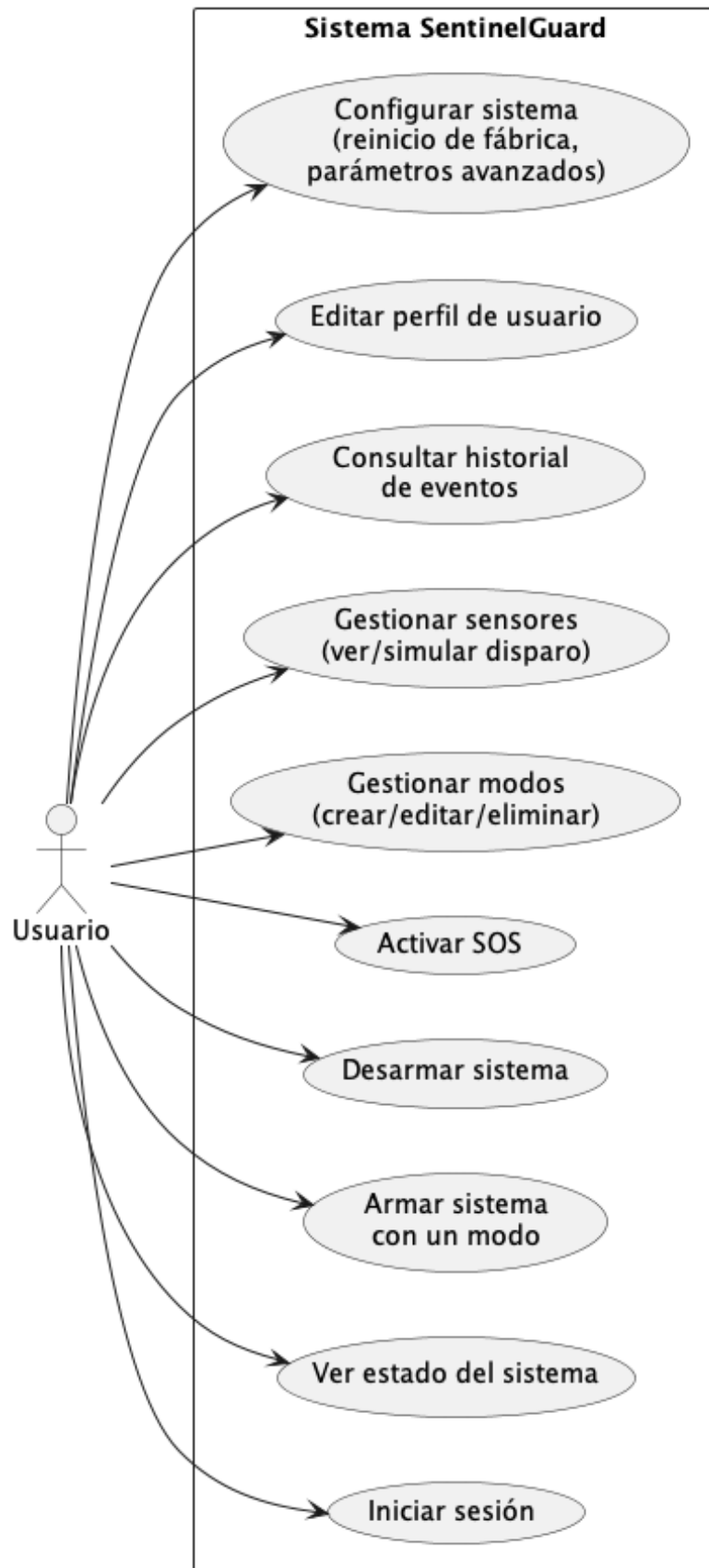


Figura 1: Casos de uso: visión global del sistema SentinelGuard.

Interpretación (como sistema real): este diagrama define el “contrato” de valor con el usuario. Un producto de seguridad se juzga por acciones concretas: armar, desarmar, consultar eventos, gestionar sensores y actuar ante emergencias (SOS). Cada caso de uso se traduce en un flujo de UI y en eventos persistentes (auditoría).

Decisión clave: el sistema se diseña alrededor de *gestión de estado* (armado/desarmado/alarma activa) y *trazabilidad* (historial). Esto es exactamente lo que diferencia una app “bonita” de un producto operativo.

6.2 Diagrama de clases: núcleo de dominio y servicios

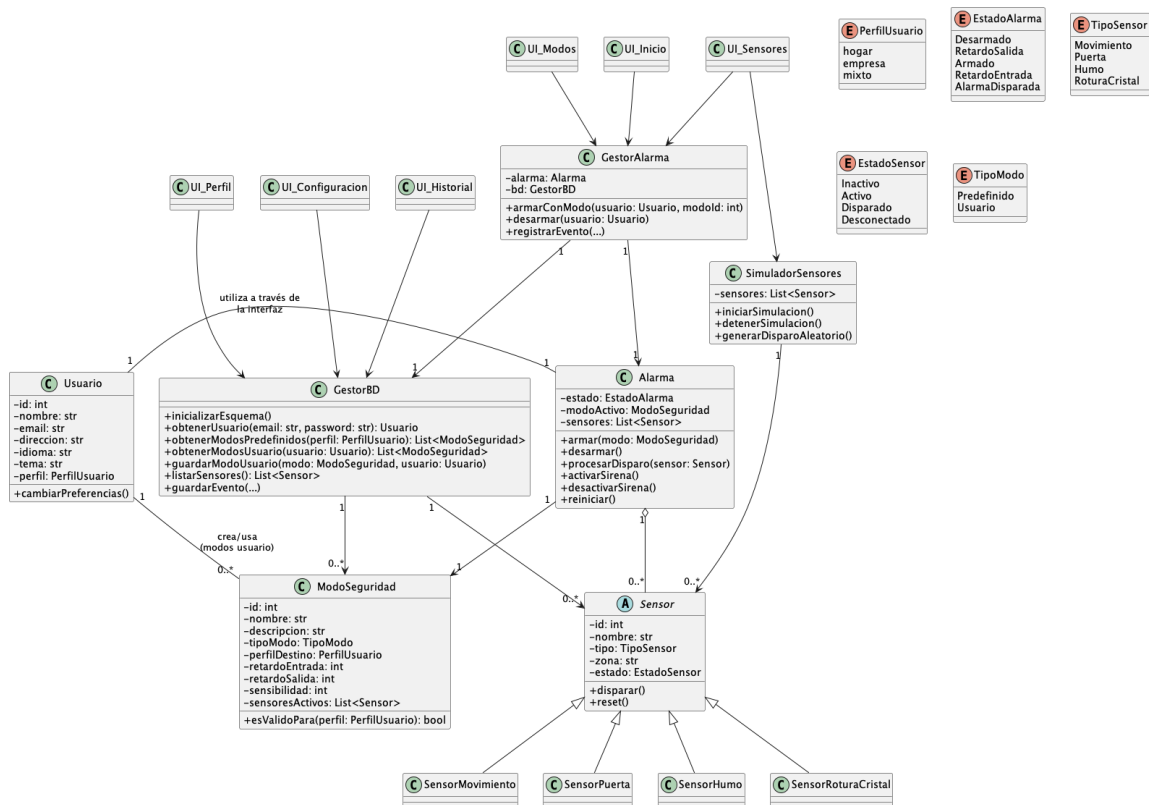


Figura 2: Diagrama de clases principal: dominio, gestores y UI.

Interpretación: el diagrama muestra que la clase **Alarma** actúa como núcleo del dominio (estado, modo activo, sensores), mientras que **GestorBD** encapsula persistencia y **GestorAlarma** coordina flujos de alto nivel entre UI–dominio–BD.

Por qué esta separación (empresa real):

- **Evita acoplamiento UI–BD:** si mañana se cambia SQLite por un servicio remoto, la UI no debería cambiar.
- **Protege la coherencia del estado:** la lógica crítica no puede depender de una pantalla concreta.

6.3 Actividad: login y carga dinámica por perfil

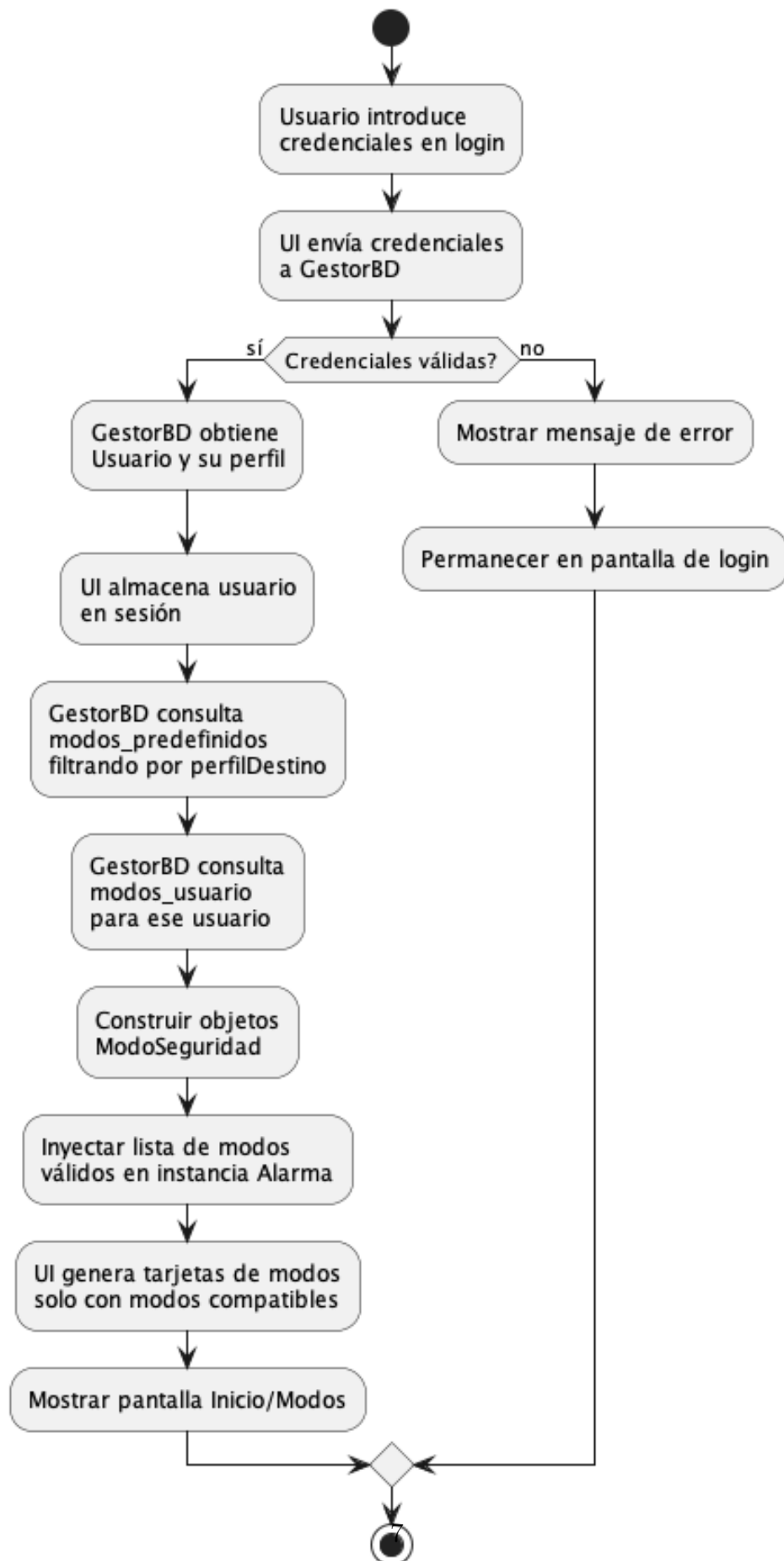


Figura 3: Actividad: autenticación y carga dinámica de modos según perfil.

Interpretación: un producto real no muestra opciones que no aplican. El usuario entra y el sistema construye su configuración (perfil, modos compatibles y personalización). Esto reduce errores y mejora la percepción de robustez.

Decisión UX importante: el usuario ve *solo* los modos válidos para su perfil, evitando confusión y acciones incoherentes (ej.: un modo “Mascotas” no tiene sentido en empresa).

6.4 Secuencia: disparo de sensor con sistema armado

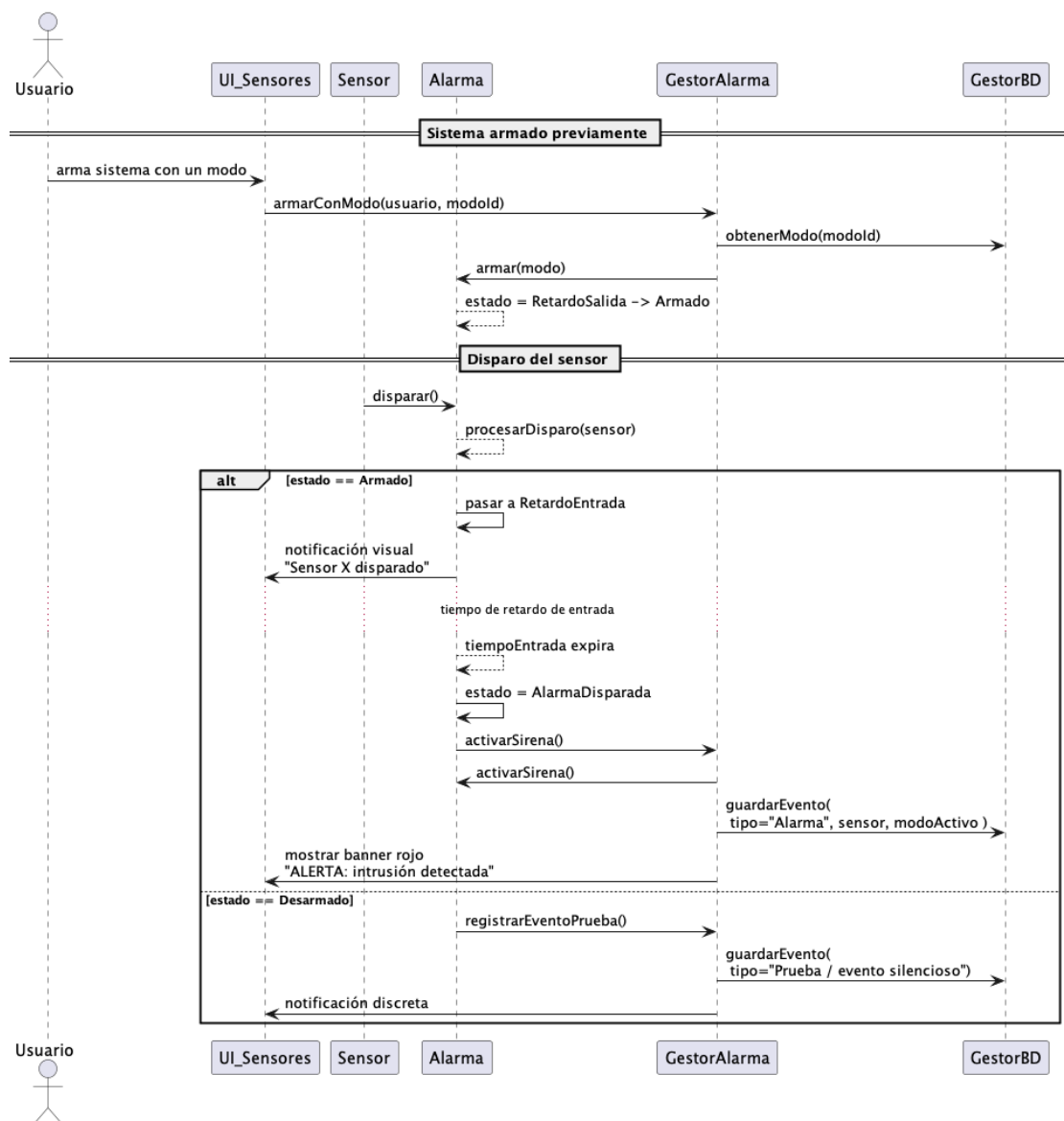


Figura 4: Secuencia: disparo de sensor con sistema armado.

Interpretación (seguridad real): un sensor no “dispara” si el sistema está desarmado. Si está armado, el disparo inicia un flujo controlado: notificación, posible retardo de entrada y, si no se desarma, activación de alarma y registro.

Decisión clave de coherencia: se impide simular eventos con el sistema desarmado y se registra todo en historial. En sistemas de seguridad, el historial funciona como *auditoría* y como *explicación* al usuario.

6.5 Secuencia: creación de modo personalizado

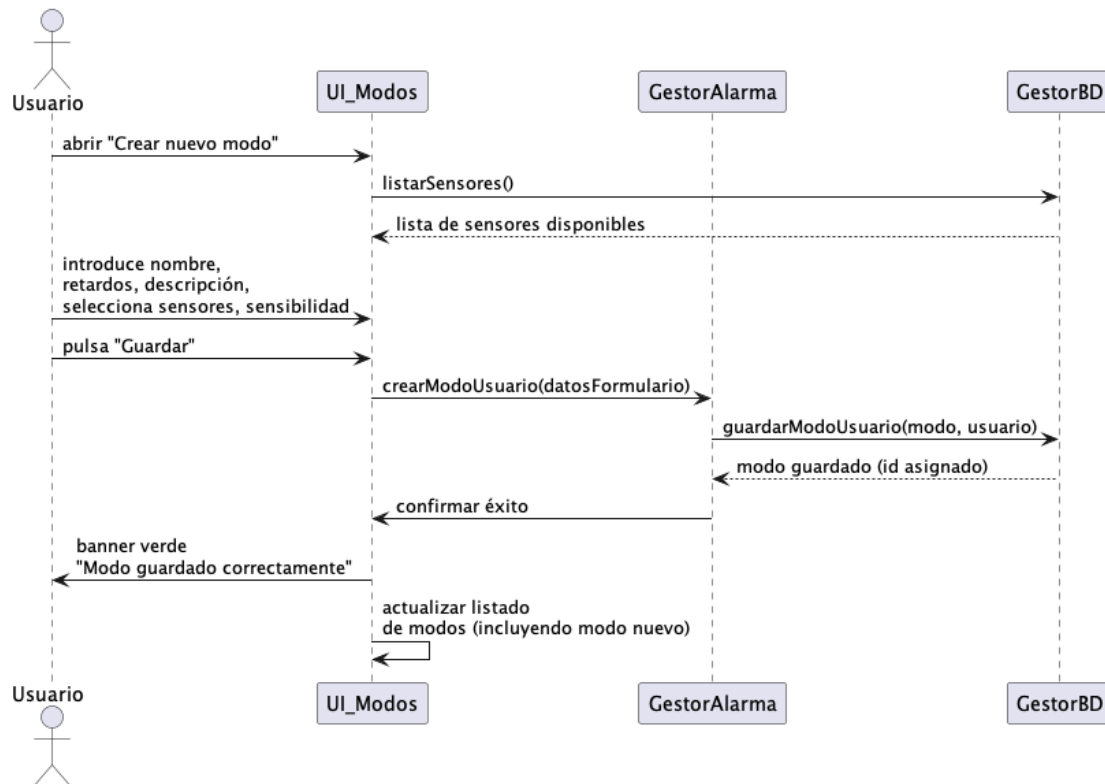


Figura 5: Secuencia: creación y persistencia de un modo personalizado.

Interpretación (producto real): la creación de modos no puede ser una acción “local de UI”. Debe:

- Validarse con sensores disponibles,
- Persistirse en base de datos,
- Refrescar la UI para que el usuario vea el cambio.

Motivo: una app real debe garantizar consistencia. Si el modo se crea pero no se persiste, el usuario perdería configuración y se rompe confianza.

6.6 Diagrama de estados: ciclo completo de una alarma

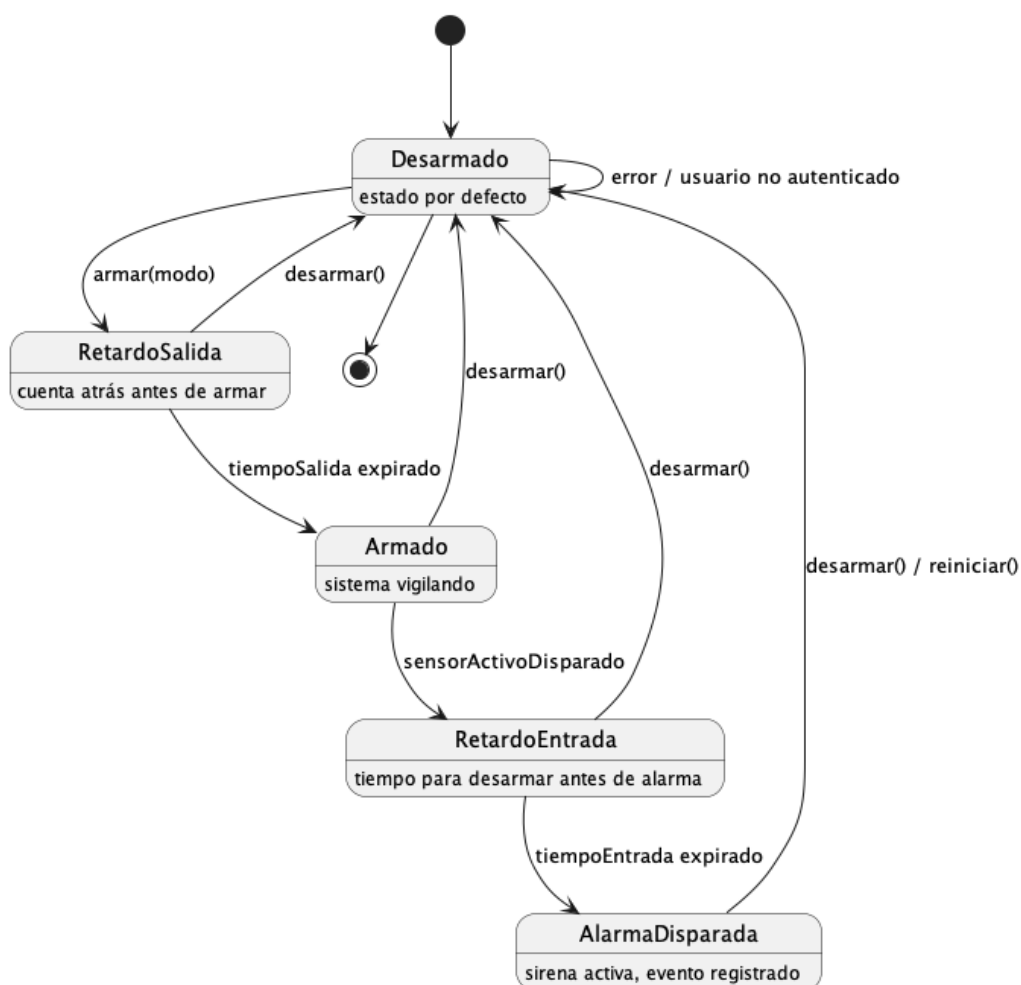


Figura 6: Estados del sistema de alarma y transiciones.

Interpretación: el estado por defecto es **Desarmado**. El armado pasa por **RetardoSalida** (cuenta atrás realista). Un disparo con el sistema armado entra en **RetardoEntrada** y finalmente en **AlarmaDisparada** si no se desarma.

Decisión de negocio: esto replica exactamente la lógica real: evitar falsas alarmas (retardo), permitir salir/entrar, y solo disparar sirena si procede.

7 Modos de seguridad (catálogo completo y diseño de pantalla)

En una empresa de seguridad, los “modos” son el corazón del producto: transforman un conjunto fijo de sensores en comportamientos adaptados a situaciones reales (noche, jornada laboral, ausencia total, presencia parcial). SentinelGuard implementa modos por perfil para reducir errores y aumentar la claridad.

7.1 Modos por perfil (catálogo completo)

Perfil Hogar

- **Modo Casa:** protege el perímetro manteniendo libertad de movimiento en el interior.
- **Modo Noche:** activa sensores estratégicos mientras el usuario duerme.
- **Modo Total:** activa todos los sensores del sistema sin excepciones.
- **Modo Mascotas:** reduce falsas alarmas provocadas por mascotas en casa.
- **Modo Silencioso:** registra actividad sin activar sirena audible.

Perfil Empresa

- **Modo Perímetro Nocturno:** protege accesos exteriores fuera de horario laboral.
- **Modo Total:** todos los sensores activos.
- **Modo Empleados:** permite circulación interior controlada durante la jornada.
- **Modo Almacén:** protege zonas sensibles sin interferir en zonas públicas.
- **Modo Silencioso Profesional:** monitorización discreta sin sirena.

Perfil Mixto

- **Modo Total:** combina protección de hogar y empresa (ausencia total).
- **Modo Limpieza:** permite movimiento interior limitado por personal autorizado.

Modos personalizados

Además de los modos predefinidos, SentinelGuard permite la creación de modos personalizados (desde administración), que quedan ligados al usuario y persisten en base de datos. Esta capacidad replica un requisito real: cada instalación es distinta y los usuarios avanzados requieren configuraciones a medida.

7.2 Diseño de la pantalla de Modos (UI/UX con lógica realista)

La pantalla de modos se diseña como un **catálogo visual** de tarjetas, por razones de negocio y UX:

- **Lectura rápida:** el usuario identifica su modo actual de un vistazo.

- **Acción directa:** cada tarjeta contiene la acción principal (activar).
- **Aprendizaje asistido:** un icono de información explica cada modo.

Decisiones clave de comportamiento (realismo)

- **No se permite activar un modo si ya hay uno activo:** en sistemas reales se exige desarmar antes de cambiar configuración operativa.
- **Cuenta atrás de armado (RetardoSalida):** evita que el usuario dispare su propia alarma al salir.
- **Durante el armado no se permiten acciones conflictivas:** no se puede rearmar, ni activar otro modo, ni generar estados incoherentes.

Persistencia y trazabilidad

Cada activación inicia un registro en historial. En un producto real, esto sirve para:

- Auditoría (qué modo estaba activo),
- Diagnóstico de incidentes,
- Confianza del usuario (explicación de por qué ocurrió algo).

8 Sensores (catálogo completo, simulación y diseño de pantalla)

En un sistema de alarma real, los sensores son fuentes de eventos. SentinelGuard implementa sensores por perfil para reflejar instalaciones típicas y reforzar coherencia: un usuario de empresa ve sensores de oficina/almacén, mientras que un hogar ve acceso, movimiento y humo doméstico.

8.1 Sensores por perfil (catálogo completo)

Perfil Hogar

- **Movimiento Salón** (Movimiento)
- **Puerta Principal** (Apertura)
- **Ventana Dormitorio** (Apertura)
- **Humo Cocina** (Humo)

Perfil Empresa

- **Acceso Principal** (Apertura)
- **Movimiento Oficina** (Movimiento)
- **Movimiento Almacén** (Movimiento)
- **Humo Zona Técnica** (Humo)

Perfil Mixto

- **Acceso Vivienda** (Apertura)
- **Movimiento Zona Mixta** (Movimiento)

8.2 Simulación realista: por qué no se puede “disparar” con el sistema desarmado

Una decisión esencial para realismo fue impedir que el usuario simule eventos cuando el sistema está desarmado. En una alarma real, un sensor puede detectar movimiento, pero el **evento de intrusión** solo tiene sentido si el sistema está armado. Esto evita confusión y mantiene la narrativa de seguridad: *solo hay alerta cuando existe vigilancia activa*.

8.3 Diseño de la pantalla de Sensores (UI/UX)

La pantalla se basa en tarjetas (similar a Modos) por consistencia visual. La decisión más relevante fue mostrar un mensaje global (“arma el sistema para simular eventos”) cuando está desarmado, en vez de repetirlo en cada tarjeta, por:

- **Reducir ruido visual** (menos texto duplicado),
- **Aumentar percepción de producto profesional**,
- **Mantener el foco en la acción principal**.

8.4 Qué ocurre al simular un evento

Cuando el sistema está armado y el usuario simula un evento:

- Se genera una notificación de alerta,
- Se marca el estado interno de alarma activa,
- Se registra el evento en el historial con sensor y modo activo.

Esta secuencia replica el comportamiento de una central: alerta + registro + contexto (modo/sensor).

9 Armado del sistema y temporizadores

El sistema incorpora retardos de salida y entrada mediante temporizadores, simulando el tiempo que un usuario tiene para salir o desarmar la alarma antes de que se dispare.

Estos temporizadores se gestionan de forma centralizada para evitar bloqueos de la interfaz y mantener la coherencia del estado global.

10 Persistencia y trazabilidad (diseño empresarial)

En sistemas de seguridad, la persistencia no es un añadido: es parte del producto. Una alarma sin historial ni configuración persistente se percibe como frágil e inservible, porque:

- El usuario necesita explicación tras un incidente (qué pasó y cuándo).
- Un técnico necesita diagnosticar (sensores que disparan con frecuencia, modos mal configurados).
- En entornos empresa, la auditoría es requisito operativo (responsabilidad y control).

10.1 Qué se persiste y por qué

SentinelGuard persiste al menos tres tipos de información crítica:

1. **Usuarios y perfil:** permite personalizar la experiencia y filtrar modos/sensores.
2. **Modos (predefinidos y personalizados):** garantiza que configuraciones del usuario no se pierdan.
3. **Historial de eventos:** proporciona trazabilidad de lo ocurrido (armados, desarmados, disparos, etc.).

10.2 Separación entre estado de sesión y persistencia

Una decisión clave fue diferenciar:

- **Estado de sesión (memoria):** valores como `modo_activo`, `armando`, `segundos_armado` viven en sesión para respuesta inmediata.
- **Persistencia (base de datos):** eventos y configuraciones que deben sobrevivir a reinicios se guardan en BD.

En una empresa real esto evita sobrecargar la base de datos con estados temporales, mientras garantiza trazabilidad de lo importante. El resultado es un sistema ágil (UI responde rápido) y fiable (historial y modos no se pierden).

10.3 GestorBD como capa única de acceso

Se usa un **GestorBD** para centralizar consultas y escrituras. Esta decisión tiene ventajas claras:

- **Seguridad y consistencia:** todas las escrituras pasan por la misma lógica.
- **Mantenibilidad:** si cambia el esquema, se modifica un punto único.
- **Evolución tecnológica:** permite migrar a una BD remota sin reescribir UI.

10.4 Historial como auditoría

El historial no se limita a “mostrar mensajes”. Está diseñado como una auditoría mínima:

- Tipo de evento (armado, desarmado, sensor, etc.).
- Descripción contextual (qué y por qué).
- Modo activo y sensor implicado si aplica.
- Fecha/hora.

En un producto real, esta auditoría reduce incidencias y aumenta confianza: el usuario entiende el sistema.

11 Interfaz y experiencia de usuario (UX/UI con razonamiento real)

SentinelGuard no se diseñó como una interfaz académica, sino como un producto real. En aplicaciones de seguridad, la UI es crítica por dos razones:

- **Reducir errores:** el usuario debe entender si está armado o no sin interpretaciones.
- **Generar confianza:** una interfaz inconsistente reduce la percepción de seguridad.

11.1 Lenguaje visual (colores semánticos)

Se seleccionaron colores por significado operativo:

- **Azul (3154A6):** acciones generales. Evita agresividad y transmite estabilidad (común en apps corporativas).
- **Verde (2F7D32):** armado/seguro. Asociado universalmente a “correcto” y “protegido”.
- **Rojo (B3261E):** emergencia (SOS) y alerta. Debe destacar y comunicar urgencia.
- **Azul oscuro (223663):** administración avanzada (modos personalizados). Diferencia acciones de experto del uso básico.

11.2 Estructura de pantallas y consistencia

Banner superior azul: navegación constante. En un producto real, el usuario debe encontrar Modos/Sensores/Historial sin fricción.

Tarjetas grises sin sombras blancas: se eligió una estética limpia y profesional. Las sombras fuertes suelen parecer “demo”. El gris uniforme reduce ruido visual y centra la atención en la información operativa.

11.3 Pantalla Inicio: decisiones aparentemente pequeñas (pero críticas)

Nombre del usuario en saludo: refuerza personalización y reduce la sensación de sistema genérico. En seguridad, el usuario quiere sentir que el sistema es “su instalación”.

Estado textual humano: “Tu sistema está desarmado/protegido” se prioriza frente a tecnicismos. En un entorno real, la claridad reduce errores operativos.

11.4 Cuenta atrás integrada (sin spam de notificaciones)

Se evitó notificar cada segundo porque:

- En un producto real sería molesto y poco profesional.
- Las notificaciones son para eventos relevantes (armado completado, sensor detectado).
- La cuenta atrás debe vivirse como estado del sistema, no como alerta continua.

Por eso la cuenta atrás se muestra como estado en pantalla (inline), manteniendo realismo.

11.5 Pop-ups informativos persistentes (icono)

En modos, el icono de información abre un pop-up anclado (`ui.menu`) que:

- Permanece abierto hasta que el usuario lo cierra o toca fuera,
- Usa animación suave (scale) y color uniforme,
- Evita parpadeos (problema típico al refrescar UI).

Esto replica un patrón real: ayuda contextual sin abandonar la pantalla.

11.6 Logo y naming

Nombre SentinelGuard: sugiere vigilancia (Sentinel) y protección (Guard). Suena a producto real, fácil de recordar, y coherente con una app de seguridad.

Logo (escudo + rojo/azul): el escudo simboliza protección; el contraste rojo/azul refleja alerta/control. El objetivo es comunicar “seguridad moderna”, no estética infantil ni genérica.

12 Descripción de archivos (visión técnica y propósito)

A continuación se describen los archivos principales del proyecto y su responsabilidad. El objetivo es que un lector pueda entender el sistema sin leer todo el código.

12.1 Archivos de interfaz (UI)

- **app.py:** punto de entrada; configura NiceGUI, rutas y arranque de la aplicación.
- **layout.py:** define la estructura visual global (banner superior, navegación y estilo consistente).
- **login.py:** pantalla de autenticación; valida credenciales y arranca sesión.
- **inicio.py:** dashboard del sistema; muestra estado (desarmado/armando/armado) y acciones rápidas.
- **modos.py:** catálogo de modos por perfil; activa modos con cuenta atrás y UI informativa.
- **admin_modos.py:** administración de modos personalizados; creación/edición con persistencia.
- **sensores.py:** lista sensores por perfil; permite simular eventos solo si el sistema está armado.

- **admin_sensores.py**: administración/gestión avanzada de sensores (configuración y control).
- **historial.py**: pantalla de auditoría; lista eventos con contexto (modo, sensor, fecha).
- **perfil.py**: lógica de selección/validación del perfil y su impacto en modos/sensores.
- **perfil_usuario.py**: utilidades para obtener perfil actual y mantener coherencia por sesión.
- **armado.py**: motor del armado con temporizador global; evita duplicidades y gestiona cuenta atrás.

12.2 Archivos de dominio (núcleo OO)

- **alarma.py**: clase central; gestiona estado, modo activo, activación/desactivación y disparos.
- **modo.py**: modelo de modo de seguridad; describe configuración, perfil destino y parámetros.
- **sensor.py**: modelo de sensor; tipo, estado y comportamiento ante disparo/reseteo.
- **usuario.py**: modelo de usuario; identidad y perfil asociado.

12.3 Persistencia (base de datos e historial)

- **basedatos.py**: acceso estructurado a la base de datos; consultas y escrituras centralizadas.
- **historial_bd.py**: API específica de eventos; inserción y lectura del historial.

12.4 Servicios

- **auth.py**: servicio de autenticación; valida credenciales y controla acceso.

12.5 Recursos del proyecto

- **logo.png**: identidad visual; refuerza marca y realismo del producto.
- **sentinelguard.db**: base de datos local; conserva modos y eventos entre sesiones.

13 Errores comunes y dificultades

Durante el desarrollo surgieron dificultades relacionadas con temporizadores, refrescos de interfaz y sincronización de estados. Estas complicaciones permitieron reforzar la arquitectura y mejorar la separación entre lógica y presentación.

14 Errores comunes, dificultades y aprendizaje durante el desarrollo

El desarrollo de SentinelGuard ha seguido un proceso iterativo, similar al de proyectos reales de software, en el que la aparición de errores y dificultades ha sido clave para la maduración del sistema. Esta sección resume los principales problemas encontrados, su causa y las soluciones adoptadas.

14.1 Gestión del estado del sistema

Uno de los principales retos fue garantizar que el sistema solo permitiera transiciones de estado válidas. En versiones iniciales era posible intentar armar la alarma cuando ya estaba armada, cambiar de modo sin desarmar previamente o interactuar con sensores durante el proceso de armado.

Este tipo de errores son especialmente críticos en sistemas de seguridad, ya que generan comportamientos impredecibles. La solución consistió en reforzar el modelo de estados de la alarma, asegurando que todas las acciones pasaran por validaciones lógicas coherentes con el diagrama de estados UML, y no dependieran únicamente de la interfaz.

14.2 Dependencia excesiva de la interfaz

Inicialmente, ciertas restricciones se aplicaban solo a nivel visual (botones deshabilitados), lo que suponía un riesgo si la lógica se ejecutaba desde otro punto. Se corrigió trasladando las validaciones al dominio, de modo que acciones como disparar un sensor o armar el sistema verificaran siempre el estado real antes de ejecutarse.

Este enfoque sigue principios de diseño orientado a objetos y refleja prácticas habituales en software crítico.

14.3 Temporizadores y concurrencia

La implementación de la cuenta atrás de armado generó problemas como temporizadores duplicados o estados bloqueados. Esto evidenció la complejidad de gestionar tiempo y concurrencia en sistemas reactivos.

La solución fue centralizar la lógica de armado en un único motor de temporización por sesión, evitando condiciones de carrera y garantizando una transición limpia entre estados.

14.4 Refrescos de interfaz y experiencia de usuario

El uso intensivo de refrescos automáticos provocaba parpadeos y la desaparición inmediata de elementos informativos. En una aplicación de seguridad, este comportamiento reduce la sensación de fiabilidad.

Se ajustó la frecuencia de refresco y se diferenciaron claramente los elementos que debían actualizarse de forma periódica (cuenta atrás) de aquellos que debían permanecer estables (pop-ups informativos).

14.5 Definición del perfil mixto

El perfil mixto supuso una dificultad conceptual, al no encajar completamente en los modelos de hogar o empresa. Finalmente se diseñó como un perfil flexible, orientado a usuarios con necesidades híbridas (autónomos, pequeños negocios con vivienda), priorizando la adaptabilidad frente a la especialización.

14.6 Conclusión del aprendizaje

Las dificultades encontradas permitieron mejorar la arquitectura del sistema, reforzar la separación entre dominio e interfaz, y aplicar de forma práctica conceptos clave de la asignatura como gestión de estados, concurrencia y diseño orientado a objetos. Estos ajustes han convertido SentinelGuard en un sistema más robusto, coherente y cercano a un producto real.

El resultado final no es solo una aplicación funcional, sino un sistema coherente, extensible y alineado con los objetivos de la asignatura DOO.