Informe Práctica 4

Diseño Arquitectónicos y Patrones
Grado en Ingeniería Informática
Universidad de La Laguna

Alejandro Rodríguez Rojas
alu0101317038@ull.edu.es

1. Introducción

El presente proyecto consiste en el desarrollo de una aplicación Java que simula el funcionamiento de un **semáforo accesible para personas invidentes**, combinando señales visuales y auditivas. El objetivo es mostrar cómo el uso de **patrones de diseño orientados a objetos**, en especial el **Patrón Estado (State Pattern)**, permite estructurar de manera clara y extensible un sistema que debe reaccionar a diferentes fases temporales.

El semáforo cuenta con los tres estados clásicos: **rojo**, **ámbar** y **verde**, con las siguientes duraciones:

- Rojo: 10 segundos.
- Ámbar: 3 segundos.
- Verde: 10 segundos, parpadeando durante los últimos 3 segundos.

La aplicación ofrece una interfaz gráfica con control de velocidad, modo silencioso (Mute), y botones de *Iniciar*, *Pausar* y *Reset*. Además, la lógica se ha refinado para que:

- Al **pausar**, el semáforo se detenga exactamente donde estaba, y al reanudar, continúe desde ese punto.

- El **modo Mute** actúe de forma inmediata al marcar o desmarcar la casilla, sin esperar al siguiente cambio de estado.

Durante el desarrollo se empleó una **IA generativa (ChatGPT 5)** como herramienta de apoyo, especialmente en las fases de diseño y depuración. Su uso permitió acelerar la aplicación de principios SOLID, detectar errores de sincronización y sugerir mejoras de estructura, manteniendo siempre la supervisión y comprensión.

2. Diseño y arquitectura

El sistema está organizado en cuatro paquetes principales: model, controller, view y sound, siguiendo una arquitectura modular orientada a la separación de responsabilidades.

Dado el tamaño del diagrama UML general, se ha dividido en tres capturas más legibles, que muestran la estructura y las relaciones entre las clases en cada parte del sistema.

En la **Figura 1** se muestra el diagrama correspondiente a los paquetes **model y controller**. El primero contiene la interfaz TrafficLightState y las clases RedState, AmberState y GreenState, que implementan los distintos comportamientos del semáforo. El segundo incluye la clase TrafficLightContext, encargada de gestionar la transición de estados, la pausa, la reanudación y la coordinación con la interfaz y el sonido.

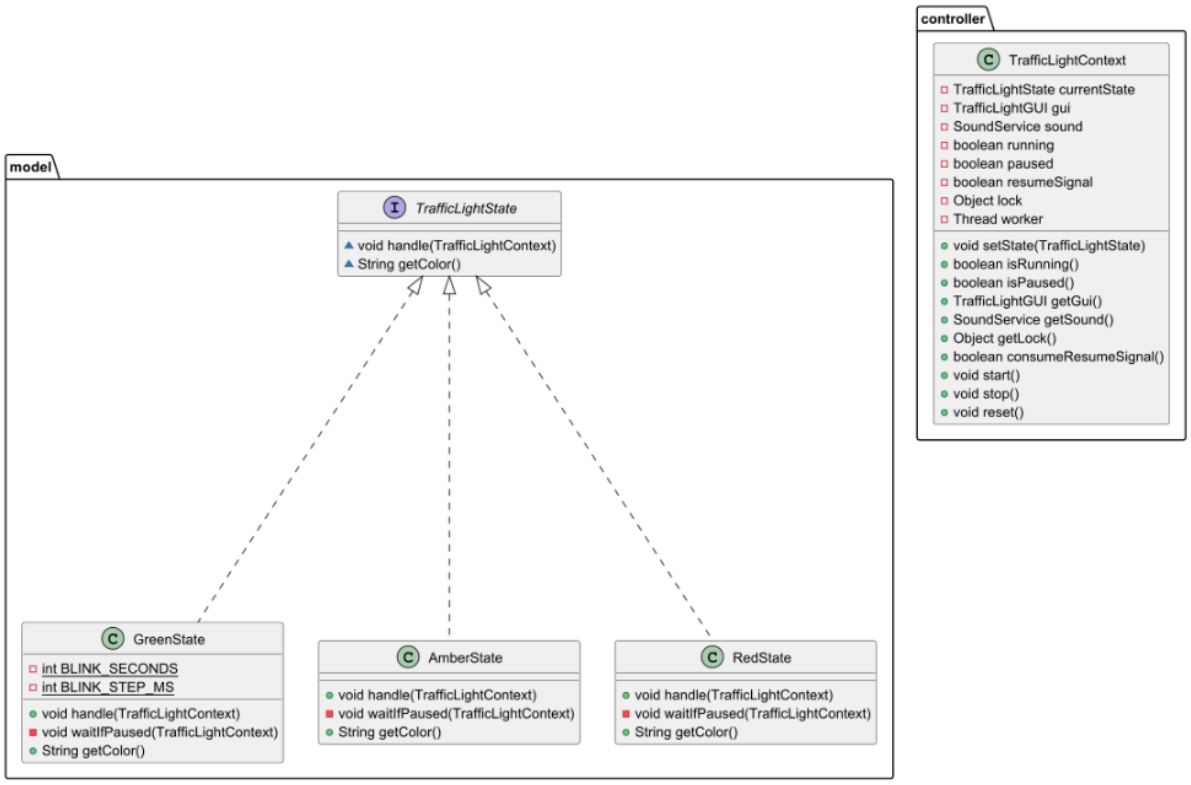

Figura 1. Diagrama UML de los paquetes model y controller, que muestran la jerarquía de estados y el contexto principal del patrón Estado.

En la **Figura 2** se representa el diagrama del **paquete sound** junto con la clase Main. Aquí se observa la abstracción SoundService y su implementación concreta ToneSoundService, encargada de generar sonidos mediante la API javax.sound.sampled. También se incluye Main, punto de entrada del programa, donde se crean los componentes y se conectan los eventos de la interfaz.

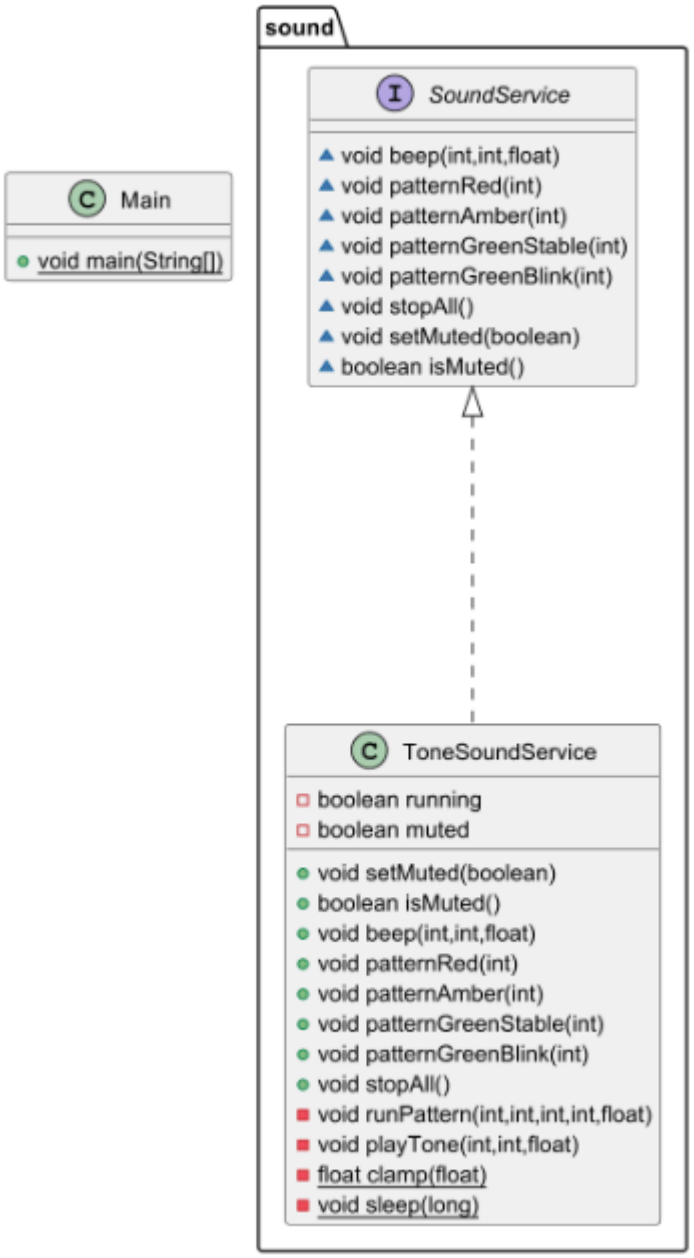

Figura 2. Diagrama UML del paquete sound y la clase Main, donde se define la arquitectura del sistema de sonido y la inicialización de la aplicación.

Finalmente, la **Figura 3** muestra el diagrama del **paquete view**, que contiene la clase TrafficLightGUI. En ella se implementa la interfaz gráfica mediante Swing, con los paneles de luces, los botones de control, el deslizador de velocidad y la opción de silencio, así como la comunicación con el TrafficLightContext.

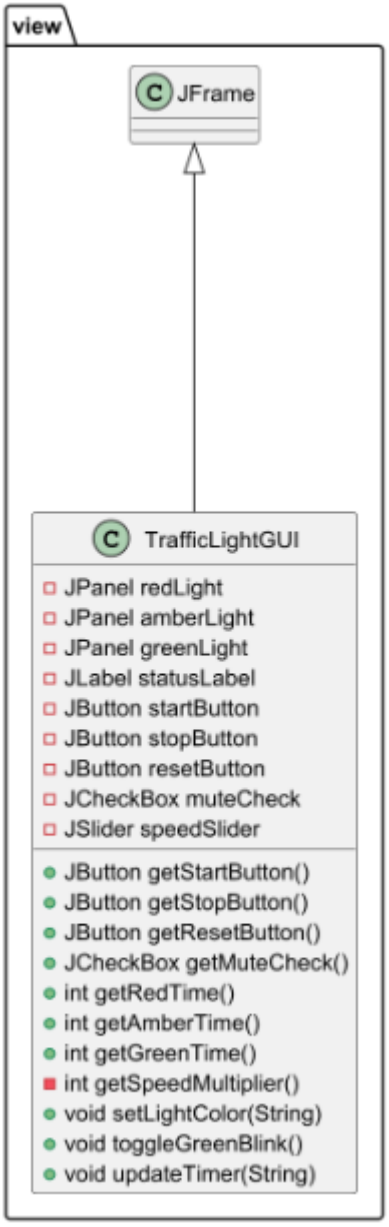

Figura 3. Diagrama UML del paquete view, correspondiente a la interfaz gráfica y su interacción con el controlador.

Durante la fase de diseño, la IA colaboró en la revisión de esta arquitectura, sugiriendo la inyección de dependencias para el sonido y la separación de responsabilidades entre estados y controlador. Estas aportaciones permitieron reforzar la adherencia a los principios **SOLID**, especialmente el **SRP** y el **DIP**.

3. Patrón Estado y principios SOLID aplicados

El **Patrón Estado** se aplicó para encapsular el comportamiento de cada fase del semáforo en clases independientes. TrafficLightContext mantiene una referencia al estado actual (TrafficLightState) y delega en él las acciones a realizar.

Cada estado gestiona su propia lógica:

- **RedState:** mantiene la luz roja y un tono grave intermitente.
- **AmberState:** activa la luz ámbar y un tono medio durante tres segundos.
- **GreenState:** muestra luz verde con un tono agudo estable y, durante los últimos tres segundos, un parpadeo rápido con beeps cortos.

La IA ayudó a refinar esta implementación proponiendo un sistema de **bucle cooperativo con espera activa** que permite pausar y reanudar sin reiniciar el temporizador. Gracias a ello, el programa ahora conserva los segundos restantes de cada estado al detenerse y reanuda exactamente donde estaba.

El diseño cumple con los principios **SOLID**:

- **SRP:** cada clase cumple una única función (estado, vista, controlador o sonido).
- **OCP:** es posible añadir nuevos estados (por ejemplo, "intermitente de emergencia") sin modificar las clases existentes.
- **LSP:** todos los estados son intercambiables a través de la interfaz TrafficLightState.
- **ISP:** las interfaces (SoundService) exponen solo los métodos necesarios.
- **DIP:** el controlador depende de la abstracción de sonido y no de una implementación concreta.

4. Interfaz gráfica (Swing)

La interfaz fue diseñada con **Swing** para mantener compatibilidad con cualquier entorno de escritorio. Se puede observar en la **Figura 4**. Consta de tres paneles verticales para las luces roja, ámbar y verde, un área inferior que muestra la cuenta atrás y un conjunto de controles en la parte superior: *Iniciar*, *Pausar*, *Reset*, un deslizador de velocidad y una casilla de silencio (*Mute*).

Durante la implementación, la IA asistió en la disposición de los componentes mediante GridLayout y BorderLayout, y sugirió el uso de SwingUtilities.invokeLater() para actualizar la interfaz de forma segura desde el hilo de ejecución del semáforo.

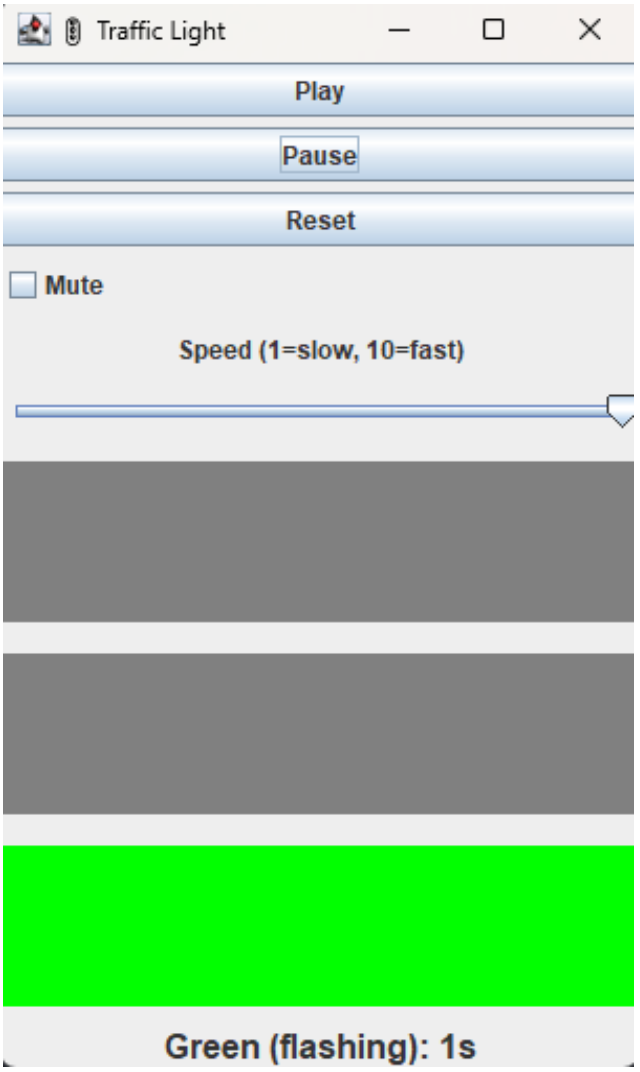

Figura 4. Interfaz gráfica del semáforo durante la ejecución en verde parpadeante con contador descendente.

La versión final mejora la experiencia del usuario al permitir pausar el ciclo en cualquier momento y continuar justo donde se dejó, algo poco común en implementaciones básicas de este tipo.

5. Sistema de sonido accesible

El sistema de sonido se implementó mediante la interfaz SoundService, con la clase ToneSoundService como su realización concreta. Esta clase genera tonos en tiempo real usando la API javax.sound.sampled, lo que evita depender de archivos de audio externos.

Los tonos se diferencian por su frecuencia y ritmo:

- Rojo: tono grave (440 Hz) intermitente cada dos segundos.
- Ámbar: tono medio (700 Hz) con ritmo moderado.
- Verde: tono agudo (880 Hz) estable, seguido de un parpadeo con beeps cortos (1200 Hz).

El modo **Mute** (**Figura 5**) ahora actúa instantáneamente gracias a un *listener* en la interfaz gráfica que actualiza el estado de SoundService tan pronto como el usuario marca o desmarca la casilla.

Figura 5. Modo silencioso activado durante la ejecución. El sonido se detiene inmediatamente sin afectar al ciclo de luces.

6. Ejecución y control de flujo

El flujo del semáforo se controla desde TrafficLightContext, que gestiona el estado actual y coordina el hilo de ejecución. Al pulsar *Iniciar*, el programa lanza un hilo que ejecuta los estados de forma secuencial.
Si se pulsa *Pausar*, el hilo no se interrumpe, sino que entra en modo de espera utilizando un monitor (wait()/notifyAll()). Al reanudar, el hilo continúa desde el punto exacto en el que se pausó, conservando los segundos restantes del estado activo. En la **Figura 6** se puede observar la ejecución del programa

El botón *Reset* interrumpe la ejecución, apaga los sonidos, limpia la interfaz y vuelve al estado inicial (rojo).
Estas mejoras se desarrollaron tras identificar, mediante pruebas prácticas, que la versión inicial reiniciaba los contadores al reanudar. Con ayuda de la IA, se propuso e implementó un sistema de pausa cooperativa más preciso.

Figura 6. Estado rojo activo durante la cuenta atrás inicial.

7. Validación y pruebas

La validación se realizó mediante pruebas manuales repetitivas para comprobar la correcta sincronización entre luz, contador y sonido. Se verificaron los siguientes escenarios:

- Funcionamiento completo de la secuencia Rojo → Ámbar → Verde → Rojo.
- Precisión del parpadeo en verde (intervalos de ~200 ms).
- Reanudación exacta tras pulsar *Pausar*.
- Activación inmediata del modo *Mute*.
- Comportamiento estable tras múltiples ciclos y reinicios consecutivos.

Durante la depuración, la IA resultó útil para revisar el manejo de hilos y proponer una estructura de control sincronizada basada en wait()/notifyAll(), evitando bloqueos o reinicios innecesarios.

El resultado fue una aplicación robusta, capaz de ejecutar indefinidamente los ciclos sin pérdida de precisión ni errores de concurrencia.

8. Conclusiones

El proyecto demuestra cómo el **Patrón Estado** y los **principios SOLID** pueden aplicarse de forma efectiva en un entorno práctico. La separación entre lógica, presentación y sonido permitió integrar mejoras significativas sin comprometer la arquitectura original.

Las nuevas funcionalidades —pausa exacta y mute inmediato— aportan realismo y mejor experiencia de usuario, manteniendo una estructura limpia y extensible.

El uso de la **inteligencia artificial** como apoyo fue un factor clave para acelerar la toma de decisiones de diseño, encontrar soluciones seguras para la concurrencia y elaborar documentación técnica coherente. No sustituyó el razonamiento del desarrollador, sino que actuó como **asistente de desarrollo**, facilitando el aprendizaje de buenas prácticas y la exploración de alternativas de implementación.

9. Tabla de tiempos (estimado vs real)

En la **Figura 7** se muestra un listado de tareas así como el tiempo estimado que se consideraban que iban a tardar en realizarse, en comparación con el tiempo real que tardaron.

Tarea	Estimado	Real
Análisis de requisitos	0 h 30 m	0 h 40 m
Diseño UML y arquitectura (incluyendo división por paquetes)	0 h 50 m	0 h 55 m
Implementación base del patrón Estado	1 h 15 m	1 h 20 m
Creación GUI (luces, botones, contador)	1 h	1 h 10 m
Sistema de sonido (sintetizado)	0 h 40 m	0 h 45 m
Parpadeo verde + sincronización	0 h 30 m	0 h 35 m
Controles Start/Stop/Reset (versión 4.0 con pausa real)	0 h 40 m	0 h 50 m
Modo silencioso inmediato (Mute)	0 h 20 m	0 h 25 m
Validación y pruebas	0 h 40 m	0 h 45 m
Documentación e informe final	0 h 50 m	1 h
Total	6 h 55 m	8 h 15 m

Figura 7: Tabla de tiempos

10. Referencias

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). Design patterns: Elements of reusable object-oriented software. Addison-Wesley Professional. Recuperado de https://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612

(Resumen libre sobre el patrón State: https://refactoring.guru/design-patterns/state)

Oracle. (2025). Java Platform, Standard Edition & Java Development Kit Documentation. Oracle. Recuperado de https://docs.oracle.com/en/java/javase/

Documentación específica utilizada:

API de Swing → https://docs.oracle.com/javase/8/docs/api/javax/swing/package-summary.html

API de sonido (javax.sound.sampled) → https://docs.oracle.com/javase/8/docs/api/javax/sound/sampled/package-summary.html

Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., & Lea, D. (2006). Java concurrency in practice. Addison-Wesley Professional. Recuperado de https://jcip.net/

(Descripción oficial del libro: https://www.pearson.com/en-us/subject-catalog/p/java-concurrency-in-practice/P200000000207/9780321349606)

OpenAI. (2025). ChatGPT: Asistencia generativa aplicada al desarrollo de software. Recuperado de https://chat.openai.com/