Diseño del Sistema de Parking

1. Descripción del Tipo de Parking

Parking Privado Automatizado con Plazas Especiales para Minusválidos

Se trata de un parking privado de múltiples plantas con sistema de control automatizado. El parking cuenta con:

- **Sistema de acceso controlado**: Detección automática de matrículas mediante cabina de entrada/salida
- **Plazas exclusivas para minusválidos**: 15% de las plazas están reservadas exclusivamente para vehículos con ocupantes minusválidos
- **Tarificación por tiempo**: 30 segundos gratuitos, después 1.5€ cada 20 segundos
- **Asignación aleatoria inteligente**: Los vehículos buscan plaza de forma aleatoria (máximo 5 intentos)
- **Persistencia de datos**: Todo el estado se guarda en formato JSON
- **Operación automática**: El sistema puede funcionar de forma autónoma, con entradas y salidas automáticas
- **Registro completo**: Sistema de logging que registra todas las operaciones

---

2. Listado de Clases

Clase: Coche

Qué representa en el mundo real:
Representa un vehículo que accede al parking. Es la unidad básica que utiliza el servicio del parking.

Atributos:

- `matricula` (str): Identificador único del vehículo (formato español: 4 dígitos + 3 letras)
- `es_minusvalido` (bool): Indica si el vehículo transporta ocupantes con movilidad reducida

Métodos:

- `to_dict()`: Serializa el objeto a diccionario para guardarlo en JSON
- `from_dict(data)`: Método estático que crea un objeto Coche desde un diccionario JSON

Por qué es una clase independiente:

- Encapsula toda la información relevante de un vehículo
- Permite agregar fácilmente nuevos atributos (tipo de vehículo, matrícula extranjera, etc.)
- Facilita la serialización/deserialización sin afectar otras clases
- Cumple el principio de responsabilidad única: solo gestiona datos del vehículo
- No sería apropiado como función suelta porque necesitamos mantener estado y comportamiento juntos

---

Clase: Aparcamiento

Qué representa en el mundo real:
Representa una plaza física de estacionamiento dentro del parking. Es el espacio donde se estaciona un vehículo.

Atributos:

- `id` (str): Identificador único (ej: "A1", "B5")
- `fila` (str): Letra que identifica la fila/planta
- `columna` (int): Número de columna dentro de la fila
- `solo_minusvalidos` (bool): Indica si es una plaza exclusiva para minusválidos
- `ocupado` (bool): Estado actual de ocupación
- `coche` (Coche): Referencia al vehículo estacionado (None si está libre)
- `timestamp_entrada` (datetime): Momento exacto en que el vehículo ocupó la plaza

Métodos:

- `puede_ocupar(coche)`: Verifica si un coche específico puede ocupar esta plaza (considera si está libre y si respeta las restricciones de minusválidos)
- `ocupar(coche)`: Asigna un coche a la plaza y registra el timestamp de entrada
- `liberar()`: Libera la plaza, retorna el coche y el tiempo que estuvo estacionado
- `to_dict()`: Serializa el estado del aparcamiento
- `from_dict(data)`: Reconstruye un aparcamiento desde JSON

Por qué es una clase independiente:

- Gestiona su propio estado de ocupación de forma autónoma
- Encapsula la lógica de restricciones (plazas para minusválidos)
- Mantiene la trazabilidad temporal (timestamp)
- Permite fácil escalabilidad (agregar sensores, cámaras, etc.)
- Una función suelta no podría mantener el estado persistente de cada plaza individual

---

Clase: Cabina

Qué representa en el mundo real:
Representa el sistema de control de acceso del parking (barrera de entrada/salida). Es el punto de interacción entre los vehículos y el sistema del parking.

Atributos (constantes de clase):

- `TIEMPO_GRATIS_SEGUNDOS` (int): 30 segundos de estacionamiento gratuito
- `TARIFA_POR_SEGUNDO` (float): 0.075€ por segundo (1.5€ cada 20 segundos)
- `MAX_INTENTOS_BUSQUEDA` (int): 5 intentos máximos para encontrar plaza

Métodos:

- `generar_matricula()`: Genera una matrícula española aleatoria
- `detectar_minusvalido()`: Simula la detección de tarjeta/distintivo de minusválido (15% probabilidad)
- `calcular_tarifa(tiempo_estacionado)`: Calcula el importe a pagar según el tiempo
- `procesar_entrada(parking)`: Gestiona todo el proceso de entrada de un vehículo (generación matrícula, detección minusválido, búsqueda de plaza)

- `procesar_salida(parking, id_aparcamiento)`: Gestiona la salida, libera plaza y calcula tarifa

Por qué es una clase independiente:

- Centraliza toda la lógica de negocio relacionada con tarifas y control de acceso
- Separa las responsabilidades: Cabina gestiona entrada/salida, Parking gestiona plazas
- Facilita cambios en el sistema de tarifas sin afectar otras clases
- Permite simular hardware real (lectores de matrícula, sensores de minusválidos)
- Como función suelta perdería cohesión y sería difícil mantener consistencia en las reglas de negocio

---

Clase: Parking

Qué representa en el mundo real:
Representa el parking completo como sistema. Es el contenedor principal que coordina todas las plazas y el sistema de control.

Atributos:

- `aparcamientos` (list[Aparcamiento]): Lista con todas las plazas del parking
- `cabina` (Cabina): Referencia al sistema de control de acceso
- `filas` (int): Número de filas/plantas del parking
- `columnas` (int): Número de columnas por fila

Métodos:

- `_crear_aparcamientos(filas, columnas, porcentaje_minusvalidos)`: Crea la estructura completa de plazas, asignando aleatoriamente cuáles son exclusivas para minusválidos
- `buscar_aparcamiento_por_id(id)`: Busca una plaza específica por su identificador
- `obtener_ocupacion()`: Calcula el porcentaje de ocupación actual
- `guardar_estado(archivo)`: Persiste el estado completo del parking en JSON
- `cargar_estado(archivo)`: Método estático que reconstruye un parking desde JSON

Por qué es una clase independiente:

- Es el punto de entrada principal del sistema (fachada)
- Coordina la interacción entre todas las demás clases
- Gestiona la colección de plazas como un todo
- Proporciona operaciones de alto nivel (estadísticas, persistencia)
- Encapsula la complejidad del sistema completo
- Como función suelta sería imposible mantener el estado global del parking y coordinar todas las operaciones

---

Clase: InterfazParking

Qué representa en el mundo real:
Representa la interfaz gráfica de usuario y el panel de control del parking. Es lo que vería un operador del sistema.

Atributos:

- `parking` (Parking): Referencia al parking que está visualizando/controlando
- `automatico` (bool): Estado del modo automático
- `ventana` (tk.Tk): Ventana principal de la aplicación
- `canvas` (tk.Canvas): Lienzo donde se dibuja el parking
- `boton_automatico` (tk.Button): Botón para activar/desactivar modo automático
- `label_info` (tk.Label): Etiqueta con información de ocupación
- `hilo_automatico` (Thread): Hilo que ejecuta las operaciones automáticas

Métodos:

- `dibujar_parking()`: Renderiza el estado actual del parking en el canvas (plazas de colores según estado)
- `entrada_vehiculo()`: Procesa manualmente la entrada de un vehículo
- `salida_vehiculo()`: Procesa manualmente la salida de un vehículo
- `toggle_automatico()`: Activa/desactiva el modo de operación automática
- `proceso_automatico()`: Bucle que genera entradas y salidas aleatorias (corre en hilo separado)
- `guardar_estado()`: Invoca el guardado del estado del parking
- `iniciar()`: Inicia el bucle principal de la interfaz gráfica

Por qué es una clase independiente:

- Separa completamente la presentación de la lógica de negocio (patrón MVC)
- Permite cambiar la interfaz (web, móvil, consola) sin tocar las clases de negocio
- Gestiona su propio estado (modo automático, elementos gráficos)
- Coordina múltiples hilos de ejecución (interfaz + automatización)
- Encapsula toda la complejidad de Tkinter
- Una función suelta no podría mantener el estado de la UI ni gestionar eventos de forma eficiente

---

3. Relación entre Clases

Descripción de las Relaciones:

Parking ──► Cabina

- **Relación**: Parking "tiene una" Cabina
- **Por qué**: El parking necesita un sistema de control de acceso. La cabina no tiene sentido sin un parking
- **Conocimiento**: Parking conoce a Cabina y la utiliza para procesar entradas/salidas

Parking ──► Aparcamiento

- **Relación**: Parking "tiene muchos" Aparcamientos
- **Por qué**: Un parking está compuesto por múltiples plazas de estacionamiento
- **Conocimiento**: Parking conoce todos sus aparcamientos y los gestiona como colección

Aparcamiento ──► Coche

- **Relación**: Aparcamiento "puede tener" un Coche
- **Por qué**: Una plaza puede estar ocupada o vacía. Cuando está ocupada, mantiene referencia al coche
- **Conocimiento**: Aparcamiento conoce al Coche que lo ocupa (si hay alguno)

Cabina ──► Coche

- **Relación**: Cabina "crea y procesa" Coches
- **Por qué**: La cabina genera las matrículas y detecta atributos de los coches entrantes
- **Conocimiento**: Cabina crea objetos Coche pero no los mantiene, los pasa al Parking/Aparcamiento

Cabina ──► Parking

- **Relación**: Cabina "usa" Parking
- **Por qué**: La cabina necesita acceder a la lista de aparcamientos para buscar plazas
- **Conocimiento**: Cabina recibe el Parking como parámetro en sus métodos para acceder a los aparcamientos

InterfazParking ──► Parking

- **Relación**: InterfazParking "visualiza y controla" Parking
- **Por qué**: La interfaz necesita acceso al parking para mostrar su estado y ejecutar operaciones
- **Conocimiento**: InterfazParking mantiene una referencia al Parking y delega todas las operaciones de negocio en él

---

Flujo de Comunicación Típico:

Entrada de Vehículo:

1. `InterfazParking.entrada_vehiculo()`
2. `Cabina.procesar_entrada()`
3. `Cabina.generar_matricula()`
4. `Cabina.detectar_minusvalido()`
5. Crear `Coche(matricula, es_minusvalido)`
6. Cabina busca plaza aleatoriamente en `parking.aparcamientos`
7. `Aparcamiento.puede_ocupar(coche)`
8. `Aparcamiento.ocupar(coche)`
9. `InterfazParking.dibujar_parking()`

Salida de Vehículo:

1. `InterfazParking.salida_vehiculo()`
2. `Cabina.procesar_salida(parking, id)`
3. `Parking.buscar_aparcamiento_por_id(id)`
4. `Aparcamiento.liberar()`
5. `Cabina.calcular_tarifa(tiempo_estacionado)`
6. `InterfazParking.dibujar_parking()`

Persistencia:

1. `InterfazParking.guardar_estado()`
2. `Parking.guardar_estado()` → Archivo JSON