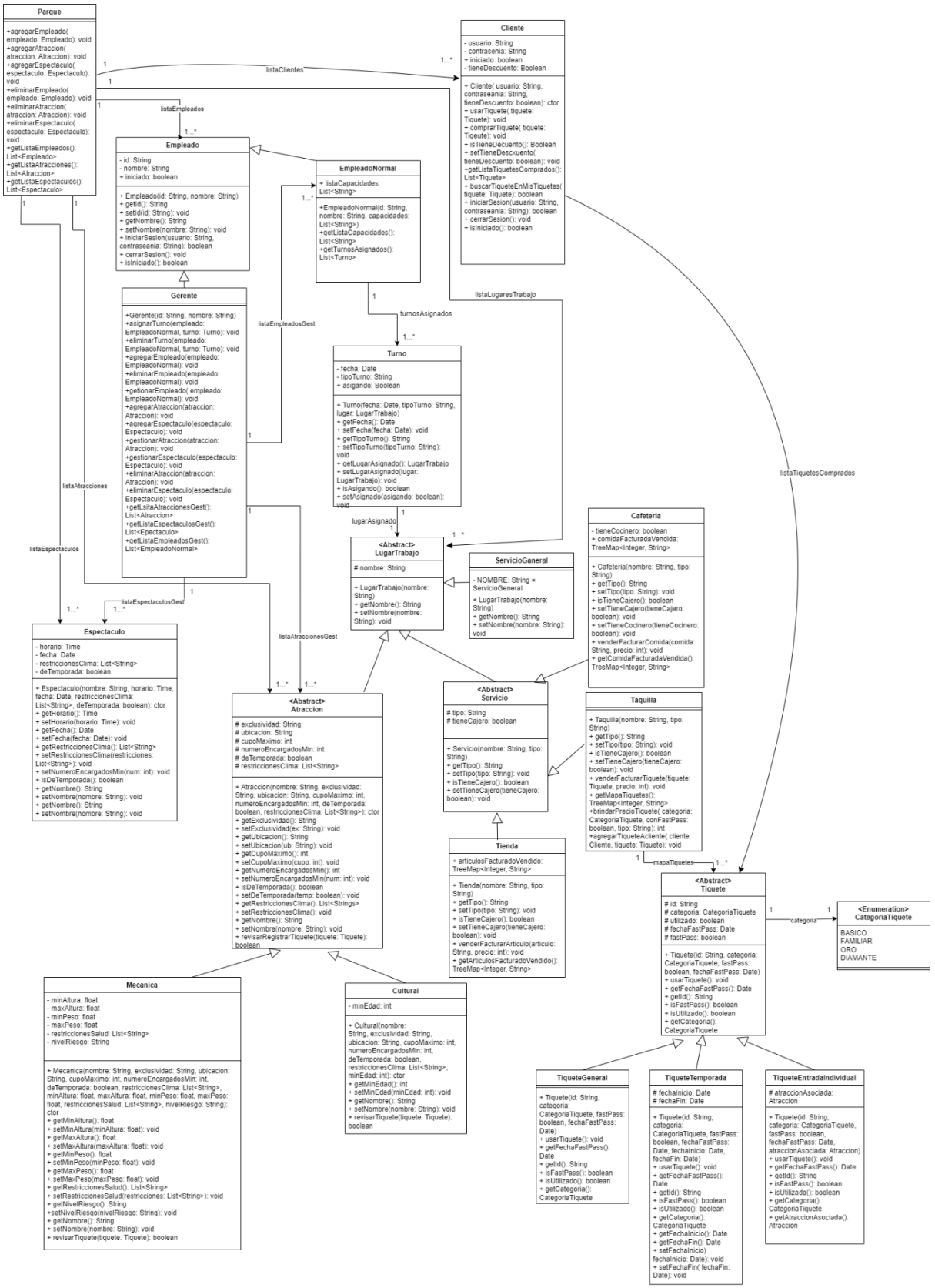
UML parque de diversión

Integrantes:

Juan Pablo Romero 202412107
Sergio Santana 202410732
Juan Camilo Panadero 202411474

ÍNDICE

Análisis UML



1.

Clase Atracción

Atributos	Métodos
1. **exclusividad: String** Almacena el nivel de exclusividad de la atracción (por ejemplo, "Familiar", "Oro", "Diamante"). Este atributo es clave para gestionar el acceso de los visitantes según el tipo de tiquete adquirido. 2. **ubicacion: String** Define la ubicación de la atracción dentro del parque, ayudando a organizar y localizar las atracciones en el mapa del parque. 3. **cupoMaximo: int** Establece el número máximo de personas que pueden disfrutar de la atracción al mismo tiempo, controlando la capacidad de la atracción. 4. **numeroEncargadosMin: int** Representa el número mínimo de empleados encargados para operar la atracción. Es importante para asegurar que la atracción funcione de manera segura. 5. **deTemporada: boolean** Indica si la atracción está disponible solo en ciertos períodos del año, como atracciones de verano o eventos especiales. 6. **restriccionesClima: List<String>** Define las restricciones relacionadas con las condiciones climáticas que pueden	1. **Atraccion(...)**: Constructor Inicializa los atributos de la clase Atraccion cuando se crea una nueva instancia. Recibe los valores de los atributos: nombre, exclusividad, ubicación, cupo máximo, número mínimo de encargados, si es de temporada y las restricciones climáticas. 2. **getExclusividad()**: Método getter que devuelve el valor del atributo exclusividad de la atracción. 3. **setExclusividad(ex: String)**: Método setter que permite modificar el valor de exclusividad. 4. **getUbicacion()**: Método getter que devuelve la ubicación de la atracción. 5. **setUbicacion(ub: String)**: Método setter que permite modificar la ubicación de la atracción. 6. **getCupoMaximo()**: Método getter que devuelve el valor del cupoMaximo de la atracción. 7. **setCupoMaximo(cupo: int)**: Método setter que permite modificar el

afectar la operación de la atracción (por ejemplo, tormentas o calor extremo)	número máximo de personas para la atracción. 8. **getNumeroEncargadosMin()**: Método getter que devuelve el número mínimo de encargados para operar la atracción. 9. **setNumeroEncargadosMin(num: int)**:Método setter que permite modificar el número mínimo de empleados necesarios para la atracción. 10. **isDeTemporada()**: Método que devuelve un valor booleano que indica si la atracción es de temporada (si está disponible solo en ciertas épocas del año). 11. **setDeTemporada(temp: boolean)**: Método setter que permite modificar el valor de si la atracción es de temporada. 12. **getRestriccionesClima()**: Método getter que devuelve las restricciones relacionadas con el clima de la atracción. 13. **setRestriccionesClima()**: Método setter que permite modificar las restricciones climáticas de la atracción.

	14. **getNombre()**: Método getter que devuelve el nombre de la atracción. **15. revisarRegistrarTiquete(tiquete: Tiquete): boolean** Este método parece ser responsable de verificar si un tiquete es válido para registrar en la atracción. Devuelve un valor booleano que indica si el tiquete puede ser utilizado o no.

2.

Clase <u>Mecanica</u>	
Atributos	**Métodos**
1. **minAltura: float** Define la altura mínima requerida para poder acceder a la atracción. Es crucial para las atracciones mecánicas como las montañas rusas, que tienen límites de altura para garantizar la seguridad de los usuarios. 2. **maxAltura: float** Define la altura máxima permitida para usar la atracción. Esto asegura que los usuarios que excedan esta altura no puedan usar la atracción por razones de seguridad.	1. **Mecanica(...)**: Constructor Inicializa los atributos de la clase Mecanica. Recibe valores para el nombre, exclusividad, ubicación, cupo máximo, número mínimo de encargados, si es de temporada, las restricciones climáticas, altura mínima y máxima, peso mínimo y máximo, restricciones de salud y nivel de riesgo. 2. **getMinAltura() / setMinAltura(minAltura: float)**:

3. **minPeso: float**
   Define el peso mínimo permitido para la atracción. Esto es importante para garantizar que los usuarios estén dentro de los límites de peso adecuados para el funcionamiento seguro de la atracción.

4. **maxPeso: float**
   Define el peso máximo permitido para la atracción. Al igual que el mínimo, es esencial para la seguridad de los usuarios.

5. **restriccionesSalud: List<String>**
   Una lista de restricciones de salud que pueden afectar a los usuarios de la atracción, como problemas de vértigo, enfermedades cardiacas, o cualquier otra condición que contraindique el uso de la atracción.

6. **nivelRiesgo: String**
   Define el nivel de riesgo de la atracción mecánica (por ejemplo, "Medio" o "Alto"). Este atributo se utilizará para asignar empleados con la capacitación adecuada para operar la atracción dependiendo de su nivel de riesgo.

Métodos getter y setter para obtener y modificar la altura mínima permitida para la atracción.

3. **getMaxAltura() / setMaxAltura(maxAltura: float)**:
   Métodos getter y setter para obtener y modificar la altura máxima permitida para la atracción.

4. **getMinPeso() / setMinPeso(minPeso: float)**:
   Métodos getter y setter para obtener y modificar el peso mínimo permitido para la atracción.

5. **getMaxPeso() / setMaxPeso(maxPeso: float)**:
   Métodos getter y setter para obtener y modificar el peso máximo permitido para la atracción.

6. **getRestriccionesSalud() / setRestriccionesSalud(restricciones: List<String>)**:
   Métodos getter y setter para obtener y modificar las restricciones de salud asociadas con la atracción.
7. **getNivelRiesgo() / setNivelRiesgo(nivelRiesgo: String)**:
   Métodos getter y setter para obtener y modificar el nivel de riesgo de la atracción mecánica (por ejemplo,

	"Medio" o "Alto"). 8. **getNombre() / setNombre(nombre: String)**: Métodos getter y setter para obtener y modificar el nombre de la atracción. **9. revisarTiquete(tiquete: Tiquete): boolean** Método que probablemente verifica si un tiquete es válido para esta atracción específica, teniendo en cuenta las restricciones de altura, peso, y salud asociadas con la atracción mecánica.

3.

Clase Cultural	
Atributos	**Métodos**
7. **minEdad: int** Este atributo define la edad mínima requerida para participar en la atracción cultural. Es útil para controlar restricciones como las de edad, por ejemplo, en espectáculos que no son apropiados para niños pequeños.	1. **Cultural(...)**: Constructor Este constructor inicializa los atributos de la clase **Cultural** al crear una nueva instancia. Recibe los parámetros: nombre, exclusividad, ubicación, cupo máximo, número de encargados, si es de temporada, las restricciones climáticas y la edad mínima para participar. 2. **getMinEdad() / setMinEdad(minEdad: int)**: Métodos getter y setter para obtener y

	modificar la edad mínima permitida para la atracción cultural. 3. **getNombre() / setNombre(nombre: String)**: Métodos getter y setter para obtener y modificar el nombre de la atracción cultural. **4. revisarTiquete(tiquete: Tiquete): boolean** Método que probablemente verifica si un tiquete es válido para la atracción cultural, asegurándose de que el tiquete cubra las condiciones necesarias (como la edad del visitante).

4.

Clase Espectáculo	
Atributos	**Métodos**
1. **horario: Time** Define la hora en la que el espectáculo se lleva a cabo. Este atributo es importante para gestionar los horarios de las diferentes presentaciones y asegurar que los visitantes puedan planificar su día de acuerdo con las actividades disponibles. 2. **fecha: Date** Establece la fecha en la que el espectáculo tiene lugar. Los espectáculos pueden estar programados para fechas específicas y, por lo tanto, este atributo ayuda a gestionar la	1. **Espectaculo(...)**: Constructor Este constructor inicializa los atributos de la clase Espectáculo. Recibe parámetros como el nombre, horario, fecha, restricciones climáticas y si es de temporada. 2. **getHorario() / setHorario(horario: Time)**: Métodos getter y setter para obtener y modificar el horario en el que se realiza el

programación de eventos.

3. **restriccionesClima: List<String>**
   Similar a otras clases de atracciones, este atributo especifica las restricciones climáticas que pueden afectar la realización del espectáculo, como tormentas o condiciones extremas de temperatura.

4. **deTemporada: boolean**
   Indica si el espectáculo está disponible solo en una temporada específica o durante todo el año.

espectáculo.

3. **getFecha() / setFecha(fecha: Date)**:
   Métodos getter y setter para obtener y modificar la fecha del espectáculo.

4. **getRestriccionesClima() / setRestriccionesClima(restricciones: List<String>)**:
   Métodos getter y setter para obtener y modificar las restricciones climáticas del espectáculo.

5. **setNumeroEncargadosMin(num: int)**:
   Método para establecer el número mínimo de encargados necesarios para la realización del espectáculo. Aunque los espectáculos no requieren operadores como las atracciones, es posible que se necesiten encargados para la gestión del evento.

6. **isDeTemporada()**:
   Método que verifica si el espectáculo es de temporada, lo cual es importante para la programación y la disponibilidad en ciertos períodos del año.

7. **getNombre() / setNombre(nombre: String)**:
   Métodos getter y setter para obtener y modificar el nombre del espectáculo.

5.

Clase Empleado	
Atributos	**Métodos**
1. **id: String** Representa un identificador único para cada empleado. Este atributo es clave para gestionar y diferenciar a los empleados en el sistema. 2. **nombre: String** Almacena el nombre del empleado. Es un atributo básico que se utiliza para referirse al empleado dentro del sistema. 3. **usuario: String** Define el nombre de usuario que utiliza el empleado para acceder al sistema. Esto es necesario para la autenticación dentro de la plataforma del parque. 4. **contrasenia: String** Almacena la contraseña asociada al usuario del empleado, utilizada para verificar la identidad del empleado al momento de ingresar al sistema.	1. **Empleado(id: String, nombre: String)**: Constructor Este constructor inicializa los atributos básicos de la clase **Empleado**, como el **id** y el **nombre**. 2. **getId() / setId(id: String)**: Métodos getter y setter para obtener y modificar el **id** del empleado. El **id** es único y se usa para identificar al empleado dentro del sistema. 3. **getNombre() / setNombre(nombre: String)**: Métodos getter y setter para obtener y modificar el nombre del empleado. **4. iniciarSesion(String usuario, String contraseña): boolean** Permite al usuario iniciar sesion **5. getUsuario(): String** Retorna el Usuario **6. getContrasenia():String** Retorna la contraseña del usuario **7. setUsuario(usuario: String): Void** Establece el usuario **8. setContrasenia (contraseña: String): Void** Establece la contraseña del usuario

	9. IsIniciado ():Boolean Retorna un boolean si el usuario esta iniciado o no

6.

Clase <u>LugarTrabajo</u>	
Atributos	**Métodos**
1. nombre: String Este atributo define el nombre del lugar de trabajo (por ejemplo, "Cafetería", "Montaña rusa", "Tienda de recuerdos"). Es un atributo básico que permite identificar el lugar de trabajo dentro del parque.	1. **LugarTrabajo(nombre: String)**: Constructor Este constructor inicializa la instancia de la clase LugarTrabajo, tomando el **nombre** del lugar de trabajo como parámetro. 2. **getNombre()**: Método getter que devuelve el nombre del lugar de trabajo. 3. **setNombre(nombre: String)**: Método setter que permite modificar el nombre del lugar de trabajo.

7.

Clase <u>Cafeteria</u>	
Atributos	**Métodos**

1. **tieneCocinero: boolean** Este atributo indica si la cafetería tiene un cocinero asignado para preparar la comida. Es esencial para las operaciones dentro de una cafetería, ya que la preparación de comida generalmente requiere personal especializado. 2. **comidaFacturadaVendida: TreeMap<Integer, String>** Este atributo lleva un registro de las ventas de comida. Utiliza un TreeMap, donde la clave Integer podría ser la cantidad de artículos vendidos o el precio, y el valor String representa el nombre de la comida o artículo vendido.	1. **Cafeteria(nombre: String, tipo: String): Constructor** Inicializa los atributos de la clase Cafeteria, tomando el nombre y tipo de servicio como parámetros. El tipo podría ser "Comida rápida", "Bebidas", etc. 2. **getTipo() / setTipo(tipo: String):** Métodos getter y setter para obtener y modificar el tipo de servicio de la cafetería. 3. **isTieneCajero() /** setTieneCajero(tieneCajero: boolean): Métodos para verificar si la cafetería tiene un cajero y asignar uno según sea necesario. 4. **getTieneCocinero() / setTieneCocinero(tieneCocinero: boolean):** Métodos para verificar si la cafetería tiene un cocinero y para asignar un cocinero a la cafetería. 5. **venderFacturarComida(comida: String, precio: int):** Método que permite registrar la venta de un artículo de comida. Recibe el nombre de la comida y su precio, actualizando el registro de ventas. 6. **getComidaFacturadaVendida():** Devuelve la lista de comidas facturadas vendidas, representada por un TreeMap donde se almacena la cantidad o el precio de los productos vendidos junto con su nombre.

8.

Clase <u>Gerente</u> **(hereda de Empleado)**	
Atributos	**Métodos**
Hereda todos los atributos de Empleado, es decir, id, nombre, usuario y contraseña, ya que Gerente es una subclase de Empleado.	1. **Gerente(id: String, nombre: String)**:Constructor Inicializa los atributos de la clase Gerente, que incluye el id y nombre del gerente. **2. asignarTurno(empleado: EmpleadoNormal, turno: Turno): void** Asigna un turno a un empleado EmpleadoNormal. El método toma el objeto de EmpleadoNormal y el objeto de Turno como parámetros. 3. **eliminarTurno(empleado:EmpleadoNormal): void** Elimina el turno asignado a un EmpleadoNormal. 4. **agregarEmpleado(empleado: EmpleadoNormal): void** Agrega un nuevo empleado de tipo EmpleadoNormal al sistema. 5. **eliminarEmpleado(empleado: EmpleadoNormal): void** Elimina un empleado de tipo EmpleadoNormal del sistema. 6. **getEmpleado(empleado: EmpleadoNormal): EmpleadoNormal** Devuelve el objeto EmpleadoNormal

	correspondiente al empleado especificado. 7. **agregarAtraccion(atraccion: Atraccion): void** Permite agregar una nueva atracción al parque, pasando como parámetro un objeto Atracción. 8. **agregarEspectaculo(espectaculo: Espectaculo): void** Agrega un espectáculo al sistema. Toma como parámetro el objeto Espectaculo. 9. **gestionarAtraccion(atraccion: Atraccion): void** Permite gestionar una atracción existente, lo que implica controlar su operación y detalles asociados. 10. **gestionarEspectaculo(espectaculo: Espectaculo): void** Permite gestionar un espectáculo, similar al caso de las atracciones, controlando su operación y programación. 11. **eliminarAtraccion(atraccion: Atraccion): void** Elimina una atracción del sistema. 12. **eliminarEspectaculo(espectaculo: Espectaculo): void** Elimina un espectáculo del sistema. 13. **getListaAtraccionesGest(): List<Atraccion>** Devuelve una lista de todas las atracciones

	gestionadas por el gerente. 14. **getListaEspectaculosGest(): List<Espectaculo>** Devuelve una lista de todos los espectáculos gestionados por el gerente. 15. **getListaEmpleadosGest(): List<EmpleadoNormal>** Devuelve una lista de todos los empleados normales gestionados por el gerente. 16. **comprarTiquete(tiquete: Tiquete): void** Permite que el gerente compre un tiquete para sí mismo o para otros empleados, tomando un objeto Tiquete como parámetro.

9.

Clase EmpleadoNormal **(hereda de Empleado)**	
Atributos	**Métodos**
listaCapacidades: List<String> Representa las capacidades del empleado, es decir, las habilidades o funciones que el empleado puede desempeñar. Por ejemplo, podría ser capaz de operar una atracción, atender en una tienda, o realizar tareas de mantenimiento.	10. **EmpleadoNormal(d: String, nombre: String, capacidades: List<String>)**: Constructor Este constructor inicializa los atributos del EmpleadoNormal. Recibe el id del empleado, su nombre, y una lista de capacidades que define las habilidades del empleado. 11. **getListaCapacidades()**: Método que devuelve la lista de capacidades del EmpleadoNormal, indicando las tareas que puede realizar en el parque.

	12. **getTurnosAsignados()**: Método que devuelve la lista de turnos asignados al EmpleadoNormal. Este atributo es importante para gestionar el horario de trabajo del empleado dentro del parque. 13.

10.

Clase Parque	
Atributos	**Métodos**
1. **listaEmpleados:** Una lista de empleados del parque. 2. **listaAtracciones**: Una lista de atracciones disponibles en el parque. 3. **listaEspectaculos:** Una lista de espectáculos programados en el parque.	1. **agregarEmpleado(empleado: Empleado):** Este método agrega un nuevo Empleado al parque. Recibe un objeto Empleado y lo inserta en el sistema de empleados del parque. 2. **agregarAtraccion(atraccion: Atraccion):** Permite agregar una nueva Atraccion al parque. Este método toma un objeto Atraccion y lo agrega al parque. 3. **agregarEspectaculo(espectaculo: Espectaculo):** Permite agregar un nuevo Espectaculo al parque. Este método toma un objeto Espectaculo y lo agrega al parque. 4. **eliminarEmpleado(empleado: Empleado):** Este método elimina un Empleado del sistema del parque. El objeto Empleado que se pasa como parámetro será eliminado de la lista de empleados.

	5. **eliminarAtraccion(atraccion: Atraccion):** Permite eliminar una Atraccion del parque. El objeto Atraccion que se pasa como parámetro será eliminado del sistem**a.** 6. **eliminarEspectaculo(espectaculo: Espectaculo):** Elimina un Espectaculo del parque. El objeto Espectaculo que se pasa como parámetro será eliminado de la lista de espectáculos programados. 7. **getListaEmpleados():** Devuelve la lista de todos los Empleados del parque. Este método probablemente devuelve una lista de objetos Empleado. 8. **getListaAtracciones():** Devuelve la lista de todas las Atracciones del parque. Este método devuelve una lista de objetos Atraccion, que pueden ser tanto mecánicas como culturales. 9. **getListaEspectaculos():** Devuelve la lista de todos los Espectaculos programados en el parque. Este método devuelve una lista de objetos Espectaculo, que representa los eventos en vivo o presentaciones dentro del parque.

11.

Clase <u>Turno</u>

Atributos	Métodos
1. **fecha: Date -** La fecha en que se asigna el turno. 2. **horaInicio: Tim**e - La hora de inicio del turno. 3. **horaFin: Time** - La hora de finalización del turno. 4. **tipoTurno: String** - El tipo de turno, por ejemplo, matutino, vespertino, nocturno, etc. 5. **lugarTrabajo:** String - La ubicación o el servicio donde el empleado trabajará durante su turno	1. **getTurno():** Devuelve el turno asignado con la fecha, hora de inicio y hora de fin. 2. **setTurno(fecha: Date, horaInicio: Time, horaFin: Time, tipoTurno: String, lugarTrabajo: String):** Asigna un turno a un empleado con la fecha, hora de inicio, hora de fin, tipo de turno y lugar de trabajo. 3. **getFecha():** Devuelve la fecha del turno. 4. **getHoraInicio():** Devuelve la hora de inicio del turno. 5. **getHoraFin():** Devuelve la hora de fin del turno. 6. **getTipoTurno():** Devuelve el tipo de turno (por ejemplo, matutino, vespertino). 7. **getLugarTrabajo():** Devuelve el lugar de trabajo asignado en el turno.

12.

Clase Tienda	
Atributos	**Métodos**

1. **nombreTienda**: String El nombre de la tienda (por ejemplo, "Tienda de souvenirs", "Tienda de ropa", etc.). 2. **productosDisponibles**: List<String> Una lista de productos disponibles en la tienda (por ejemplo, camisetas, juguetes, artículos de recuerdo). 3. **horarioApertura**: String El horario de apertura de la tienda. 4. **ubicación**: String La ubicación dentro del parque donde se encuentra la tienda (por ejemplo, "Zona A", "Cerca de la entrada"). 5. **stock**: int El número de productos o el stock disponible en la tienda. Esto puede variar según la tienda y sus productos.	1. **getNombreTienda():** Devuelve el nombre de la tienda. 2. **getProductosDisponibles():** Devuelve la lista de productos disponibles en la tienda. 3. **getHorarioApertura():** Devuelve el horario de apertura de la tienda. 4. **getUbicación():** Devuelve la ubicación de la tienda dentro del parque. 5. **getStock():** Devuelve la cantidad de productos disponibles en la tienda. 6. **setNombreTienda(nombre: String):** Establece el nombre de la tienda. 7. **setProductosDisponibles(productos: List<String>):** Establece la lista de productos disponibles en la tienda. 8. **setHorarioApertura(hora: String):** Establece el horario de apertura de la tienda. 9. **setUbicación(ubicación: String):** Establece la ubicación de la tienda dentro del parque.

	10. **setStock(stock: int):** Establece la cantidad de productos disponibles en la tienda.

13.

Clase Servicio	
Atributos	**Métodos**
1. **nombre: String** El nombre del servicio (por ejemplo, "Cafetería", "Taquilla", etc.). 2. **tipo: String** El tipo de servicio (por ejemplo, "Restaurante", "Venta de boletos", etc.). 3. **descripcion: String** Descripción breve del servicio, detallando lo que ofrece (por ejemplo, "Servicio de comida rápida", "Venta de entradas al parque"). 4. **ubicacion: String** La ubicación del servicio dentro del parque (por ejemplo, "Zona A", "Cerca de la entrada", etc.).	1. **getNombre()**: Devuelve el nombre del servicio. 2. **getTipo()**: Devuelve el tipo de servicio (por ejemplo, "Restaurante", "Venta de boletos", etc.). 3. **getDescripcion()**: Devuelve la descripción del servicio. 4. **getUbicacion()**: Devuelve la ubicación del servicio dentro del parque. 5. **setNombre(nombre: String)**: Establece el nombre del servicio. 6. **setTipo(tipo: String)**: Establece el tipo de servicio. 7. **setDescripcion(descripcion: String)**: Establece la descripción del servicio.

	8. **setUbicacion(ubicacion: String)**: Establece la ubicación del servicio dentro del parque.

14.

Clase <u>Taquilla</u> **(hereda de Servicio)**	
Atributos	**Métodos**
1. **mapaTiquetes: TreeMap<Integer, String>** Este atributo almacena los tiquetes vendidos. Utiliza un TreeMap donde la clave Integer podría representar el número de tiquetes vendidos o el precio, y el valor String representa el tipo de tiquete (por ejemplo, "Básico", "Oro", "Diamante").	1. **Taquilla(nombre: String, tipo: String): Constructor** Este constructor inicializa la instancia de la clase Taquilla. Toma el nombre de la taquilla y el tipo de servicio (por ejemplo, si la taquilla es para tiquetes generales o exclusivos) como parámetros. 2. **getTipo() / setTipo(tipo: String):** Métodos getter y setter para obtener y modificar el tipo de taquilla. El tipo puede indicar si la taquilla vende tiquetes generales o si tiene algún tipo de restricción de acceso. 3. **isTieneCajero() / setTieneCajero(tieneCajero: boolean):** Métodos para verificar si la taquilla tiene un cajero asignado para realizar las transacciones y asignar un cajero a la taquilla según sea necesario. 4. **venderFacturarTiquete(tiquete: Tiquete, precio: int):**

	Método que permite registrar la venta de un tiquete. Toma un objeto Tiquete y su precio, actualizando el sistema de ventas de tiquetes**.** 5. **getMapaTiquetes():** Devuelve el mapa de tiquetes vendidos, representado por un TreeMap, lo que permite mantener un registro ordenado de los tiquetes vendidos en términos de cantidad o precio.

15.

Enumeración: CategoriaTiquete
Valores de la enumeración:
BÁSICO Este valor representa el tiquete más básico, que probablemente permite el acceso general al parque pero no incluye el acceso a atracciones exclusivas o especiales. **FAMILIAR** Este valor representa un tiquete que probablemente permite el acceso a las atracciones destinadas a la familia, que pueden incluir una mayor variedad de actividades dentro del parque. **ORO** Este valor representa un tiquete que otorga acceso a una categoría superior de atracciones o servicios dentro del parque. Los usuarios con un tiquete Oro tienen acceso a las atracciones de exclusividad Familiar y Oro. **DIAMANTE** Este valor representa el tiquete de mayor exclusividad, probablemente dando acceso completo a todas las atracciones disponibles en el parque, incluyendo las más exclusivas o de alto nivel.

16.

Clase <u>Tiquete</u>

Atributos	Métodos
1. **tipoTiquete: String** El tipo de tiquete. Puede ser algo como "General", "VIP", "Familiar", entre otros. Define el tipo de entrada que se ofrece. 2. **fecha: Date** La fecha en que el tiquete fue emitido. 3. **hora**: Time La hora en la que se emitió el tiquete. 4. **lugarAsignado**: String El lugar o área del parque al que está asociado el tiquete, como una atracción específica, un espectáculo o un área general del parque.	1. **getTiquete()**: Devuelve la información completa del tiquete (tipo, fecha, hora y lugar asignado). 2. **setTiquete(tipoTiquete: String, fecha: Date, hora: Time, lugarAsignado: String)**: Establece los detalles del tiquete, como el tipo, la fecha, la hora de emisión y el lugar al que da acceso. 3. **getTipoTiquete()**: Devuelve el tipo del tiquete (por ejemplo, "General", "VIP", "Familiar"). 4. **getFecha()**: Devuelve la fecha en la que se emitió el tiquete. 5. **getHora()**: Devuelve la hora de emisión del tiquete. 6. **getLugarAsignado()**: Devuelve el lugar o atracción al que el tiquete otorga acceso. 7. **setTipoTiquete(tipoTiquete: String)**: Establece el tipo de tiquete. 8. **setFecha(fecha: Date)**: Establece la fecha de emisión del tiquete. 9. **setHora(hora: Time)**: Establece la hora de emisión del tiquete.

	10. **setLugarAsignado(lugarAsignado: String)**: Establece el lugar o atracción asociado al tiquete. 11.

17.

Clase Tiquete general ext Tiquete	
Atributos	**Métodos**
1. **Hereda los atributos de Tiquete.** 2. **categoria: String** La categoría del tiquete, por ejemplo, "Básico", "Familiar", "Oro", "Diamante", etc. Define el tipo de acceso que se tiene al parque (puede implicar distintos niveles de privilegios o precios). 3. **lugarAsignado: String** El lugar o área dentro del parque al que el tiquete da acceso (por ejemplo, "Zona A", "Atracción X", "Espectáculo Y").	1. **TiqueteGeneral(id: String, categoria: CategoriaTiquete): Constructor** específico para los tiquetes generales. 2. **usarTiquete(): void:** Marca el tiquete general como utilizado. 3. **esValido(): boolean:** Verifica si el tiquete general es válido.

18.

Clase Tiquete temporada ext Tiquete	
Atributos	**Métodos**
1. **echaInicio: Date**: Fecha de inicio de validez del tiquete.	1. **TiqueteTemporada(id: String, categoria: CategoriaTiquete, fechaInicio: Date,**

2. **fechaFin: Date**: Fecha de finalización de validez del tiquete.	**fechaFin: Date): Constructor** específico para tiquetes de temporada. 2. **usarTiquete(): void: Marca el tiquete de temporada como utilizado.** 3. **esValido(): boolean:** Verifica si el tiquete de temporada es válido, considerando las fechas de inicio y fin.

19.

Clase TiqueteentradaIndivual ext Tiquete	
Atributos	**Métodos**
1. **atraccionAsociada: Atraccion**: Atracción asociada al tiquete de entrada individual.	1. **TiqueteEntradaIndividual(id: String, categoria: CategoriaTiquete, atraccionAsociada: Atraccion): Constructor** específico para tiquetes de entrada individual. 2. **usarTiquete(): void:** Marca el tiquete de entrada individual como utilizado. 3. **esValido(): boolean:** Verifica si el tiquete de entrada individual es válido.

20.

Clase ServicioGeneral	
Atributos	**Métodos**
1. **nombre: String** El nombre del servicio (por ejemplo,	1. **getNombre():** Devuelve el nombre del servicio**.**

"Cafetería", "Taquilla", etc.). 2. **tipo: String** El tipo de servicio (por ejemplo, "Restaurante", "Venta de boletos", etc.). 3. **descripcion: String** Una descripción del servicio, detallando lo que ofrece. 4. **ubicacion: String** La ubicación del servicio dentro del parque (por ejemplo, "Zona A", "Cerca de la entrada", etc.).	2. **getTipo():** Devuelve el tipo de servicio (por ejemplo, "Restaurante", "Venta de boletos", etc.). 3. **getDescripcion():** Devuelve la descripción del servicio. 4. **getUbicacion():** Devuelve la ubicación del servicio dentro del parque. 5. **setNombre():** Establece el nombre del servicio. 6. **setTipo():** Establece el tipo del servicio. 7. **setDescripcion():** Establece la descripción del servicio. 8. **setUbicacion():** Establece la ubicación del servicio dentro del parque.

21.

Clase <u>Cliente</u>	
Atributos	**Métodos**
1. **usuario:** El usuario del cliente en la plataforma	1. **iniciarSesion(String usuario, String contraseña): boolean** Permite al usuario iniciar sesion

2. **contraseña:** Tipo de dato String. También es un atributo privado. 3. **Iniciado:** ya inició sesión dentro del sistema 4. **ListaTiquetes**: lista de tiquetes que el cliente compra 5. tieneDescuento: si tiene descuento por ser empleado	**2. getUsuario(): String** Retorna el Usuario **3. getContrasenia():String** Retorna la contraseña del usuario **4. setUsuario(usuario: String): Void** Establece el usuario **5. setContrasenia (contraseña: String): Void** Establece la contraseña del usuario **6. IsIniciado ():Boolean** Retorna un boolean si el usuario esta iniciado o no 7. **comprarTiquete(tiquete: Tiquete): void** Se compra el tiquete de la preferencia del usuario. **8. usarTiquete(tiquete: Tiquete): void** Marca el tiquete como utilizado si pertenece al cliente y aún no ha sido usado. **9. isTieneDescuento(): boolean** Retorna true si el cliente cuenta con un descuento activo, false si no. **10.setTieneDescuento(tieneDescuento:** boolean): void Establece si el cliente tiene o no descuento en sus compras. **11.getListaTiquetesComprados():** List<Tiquete> Retorna la lista de tiquetes que el cliente ha comprado.

	12buscarTiqueteEnMisTiquetes(tiquete: Tiquete): boolean Verifica si el tiquete dado se encuentra en la lista de tiquetes del cliente. **13 cerrarSesion(): void** Cierra la sesión activa del cliente, estableciendo el estado como no iniciado

Requerimientos

Requerimientos NO Funcionales	
1. **Usabilidad**:	

	La interfaz del sistema debe ser intuitiva y fácil de usar tanto para los administradores como para los usuarios. La aplicación debe ser accesible desde múltiples dispositivos, como computadoras y teléfonos móviles.
2. **Rendimiento**:	El sistema debe ser capaz de manejar un número elevado de usuarios concurrentes sin deteriorar su rendimiento. Las consultas y operaciones deben ser rápidas, especialmente en la gestión de tiquetes y turnos.
3. **Seguridad**:	El sistema debe asegurar que la información personal y financiera de los empleados y clientes esté protegida. Debe contar con autenticación de usuarios para acceder a las funciones administrativas.
4. **Escalabilidad**:	El sistema debe ser escalable para soportar el aumento de la cantidad de usuarios, atracciones y espectáculos con el tiempo.
5. **Fiabilidad**:	El sistema debe ser confiable, con un bajo índice de fallos, y debe ofrecer una recuperación rápida ante cualquier error o caída.

Requerimientos Funcionales

5. Gestión general de parque ParqueTest	El sistema permite administrar la lista de clientes, empleados, atracciones y espectáculos.
6. **Gestión de Empleados**: GerenteTest	El sistema debe permitir la asignación de turnos a los empleados y gestionar la disponibilidad de los lugares de trabajo. El sistema debe permitir asignar turnos a los empleados y asociarlos a un lugar de trabajo específico. El sistema debe gestionar la disponibilidad de los empleados para cada turno.
7. **Gestión de Tiquetes**: TiquetesTest TiqueteValidezTest	El sistema debe permitir la creación de tiquetes de diferentes tipos, como generales, de temporada y de entrada individual. Cada tiquete debe tener un identificador único, una categoría y un estado de utilización. El sistema debe verificar la validez de los tiquetes dependiendo de su tipo y fecha de uso.
8. **Control de Servicios**: ServicioTest	El sistema debe gestionar los servicios ofrecidos en el parque, como alimentos, limpieza, entre otros. Y la respectiva facturación y registro.
9. **Gestión de Espectáculos y Atracciones**: AtraccionesEspectaculoTest	El sistema debe permitir la creación, modificación y programación de espectáculos y atracciones.

10. Gestion y compra de tiquetes para los Clientes: ClientesTest	El sistema permite a los clientes iniciar sesión, gestionar su usuario y contraseña y comprar tiquetes.

Público Objetivo de la Aplicación

La aplicación está dirigida a:

- **Administradores del Parque**: Para gestionar las atracciones, empleados, servicios, espectáculos y tiquetes.
- **Empleados**: Para consultar sus turnos, capacidades y lugares de trabajo asignados.
- **Visitantes del Parque**: Para adquirir tiquetes, consultar el horario de espectáculos y atracciones disponibles, y acceder a información sobre los servicios del parque.

Restricciones

1. **Las capacidades de un empleado serán medidas por un índice de experticia:** El índice de experiencia será un número entero que traduce las capacidades de un empleado.
2.
3. **Un tiquete solo será válido para entrar a una atracción si y sólo si su categoría es la misma que la exclusividad de la atracción:** Si una atracción es Oro, para permitir el ingreso, la categoría del tiquete también debe serlo.
4.
5. **El tiquete general es marcado como utilizado únicamente después de salir del parque:** dado que el tiquete general puede servir para entrar a varias atracciones, solo será marcado como utilizado una vez el cliente salga del parque.
6.
7. **El tiquete individual solo se podrá usar una vez**: El tiquete individual está destinado a una solo atracción y solo se podrá acceder a esta atracción una vez.
8.
9. **El tiquete de temporada será marcado como utilizado una vez que la temporada haya finalizado:** El tiquete de temporada admite la entrada durante cierto periodo, será marcado una vez el período haya finalizado.
10.
11. **Un Cliente no puede modificar la estructura del parque:** Los clientes solo pueden comprar tiquetes y consultar información, pero no pueden agregar o eliminar atracciones, espectáculos o servicios.

12. .**Un EmpleadoNormal no puede despedirse a sí mismo:** Solo un Gerente tiene la capacidad de eliminar empleados del sistema.

13. **Solo el Gerente puede asignar turnos**: Un EmpleadoNormal no puede asignarse turnos a sí mismo o a otros empleados, esta función está reservada para el Gerente.

14. **La exclusividad de una atracción debe corresponder con la categoría del tiquete:** Una atracción de nivel "Diamante" solo puede ser accedida por tiquetes de categoría DIAMANTE.
15. **Un tiquete utilizado no puede volver a ser usado**: Una vez marcado como utilizado mediante usarTiquete() , el tiquete no puede reutilizarse.

16. **El número de empleados asignados a una atracción no puede ser menor al mínimo requerido**: Cada atracción especifica un numeroEncargadosMin que debe respetarse por razones de seguridad.

17. **Un TiqueteTemporada no puede tener fechaInicio posterior a fechaFin -** La validación temporal exige que la fecha de inicio sea anterior a la de finalización.

18. . **Un espectáculo no puede ser programado en dos lugares diferentes al mismo tiempo:** No se pueden solapar horarios para un mismo espectáculo.

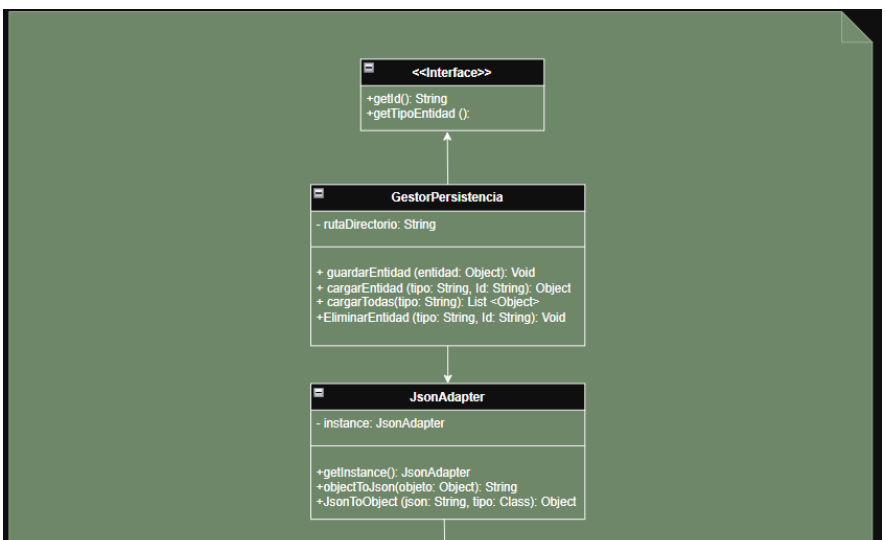
19. **Un cliente no puede acceder a una atracción si no cumple las restricciones de salud o físicas**: El método revisarRegistrarTiquete() debe validar estas condiciones.

20. **Las atracciones con restricciones climáticas no pueden operar bajo condiciones prohibidas:** El sistema debe verificar las condiciones climáticas actuales contra las restricciones de cada atracción.

21. **No se puede exceder el cupo máximo de una atracción**: El atributo cupoMaximo establece un límite estricto para la capacidad de cada atracción.
22. **El gerente sólo podrá realizar acciones si se enceutra iniciado.**

Persistencia

Justificación del Diseño de Persistencia



¿Por qué elegimos este enfoque de persistencia?

Nuestro diseño de persistencia para el sistema del parque de diversiones responde a necesidades específicas del proyecto y sigue principios de diseño sólidos. Veamos las razones detrás de esta arquitectura:

1. Enfoque Centralizado

Decidimos implementar un sistema centralizado de persistencia por varias razones clave:

- **Mantenimiento simplificado**: Al centralizar toda la lógica de persistencia en una única clase GestorPersistencia, cualquier modificación en la forma de almacenar datos solo requiere cambios en un lugar, sin afectar al resto del sistema.

- **Consistencia**: Un punto único de acceso a los datos garantiza que todas las operaciones de guardado y carga sigan los mismos patrones y convenciones, evitando inconsistencias.

- **Control de cambios**: Facilita la implementación de funcionalidades como registro de cambios (logging), validación de datos, o políticas de respaldo centralizadas.

2. Formato JSON

La elección de JSON como formato de almacenamiento está respaldada por:

- **Legibilidad**: Los archivos JSON son legibles por humanos, lo que facilita la depuración y revisión manual si fuera necesario.

- **Flexibilidad estructural**: JSON maneja naturalmente estructuras anidadas y listas, adaptándose bien a las complejas relaciones entre objetos del parque.

- **Estándar ampliamente adoptado**: Existen bibliotecas maduras y bien probadas para la serialización/deserialización en prácticamente todos los lenguajes, incluyendo Java.

- **Tamaño eficiente**: Comparado con otros formatos como XML, JSON tiende a ser más compacto, usando menos espacio en disco.

3. Separación de Responsabilidades

La división en componentes específicos permite una clara separación de responsabilidades:

- **GestorPersistencia**: Coordina el proceso completo y conoce las reglas de negocio.
- **JsonAdapter**: Se especializa exclusivamente en conversión de formatos.
- **ArchivoUtil**: Maneja las operaciones de bajo nivel con el sistema de archivos.

Esta separación sigue el principio de responsabilidad única (SRP) y facilita las pruebas unitarias al poder reemplazar componentes individuales con dobles de prueba.

4. Interfaz Persistable

Implementamos una interfaz común por varios motivos:

- **Polimorfismo**: Permite tratar de manera uniforme a todas las entidades persistentes.
- **Contrato claro**: Define qué operaciones debe soportar cualquier clase que quiera ser persistente.
- **Desacoplamiento**: Las clases de persistencia no necesitan conocer detalles específicos de cada entidad del dominio.

5. Referencias mediante IDs

La decisión de usar IDs en lugar de referencias directas entre objetos responde a:

- **Prevención de ciclos**: Evita problemas de serialización circular (cuando A referencia a B, que a su vez referencia a A).
- **Carga eficiente**: Permite cargar solo los objetos necesarios (lazy loading) en lugar de cargar grafos completos de objetos.
- **Integridad referencial**: Ayuda a mantener la consistencia entre referencias, incluso si algunos objetos no están cargados en memoria.

Beneficios para el Proyecto del Parque de Diversiones

Esta arquitectura de persistencia es particularmente adecuada para un sistema de parque de diversiones porque:

1. **Maneja entidades diversas**: Puede almacenar empleados, atracciones, espectáculos, tiquetes, etc., con un mecanismo único.

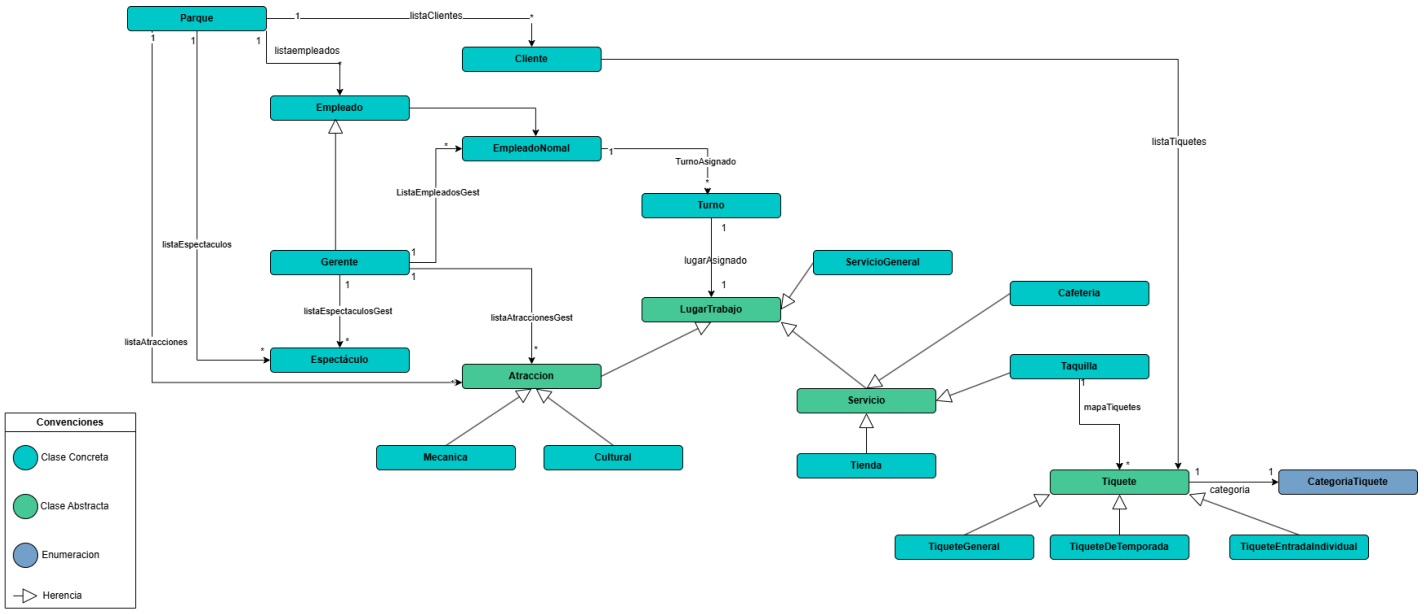
2. **Escalabilidad**: Se adapta fácilmente tanto a parques pequeños como grandes, simplemente cambiando el almacenamiento físico.

3. **Mantenimiento a largo plazo**: El diseño modular facilita la evolución del sistema a medida que cambian los requisitos del parque.

4. **Robustez**: La centralización permite implementar mecanismos de recuperación y verificación de integridad de datos.

5. **Independencia tecnológica**: Si en el futuro se decide cambiar el mecanismo de almacenamiento (por ejemplo, a una base de datos), el impacto en el resto del sistema sería mínimo.

Diagrama de clases de alto nivel



El diagrama de clases nos permite visualizar con claridad cómo es la conexión entre las clases del sistema Mediante líneas de conexión y representaciones gráficas, podemos comprender la estructura jerárquica y las relaciones entre las diferentes clases, mostrando de manera intuitiva cómo cada elemento se integra en el sistema.

En este formato de Diagrama nos permite ver cosas como:

Estructura Principal:

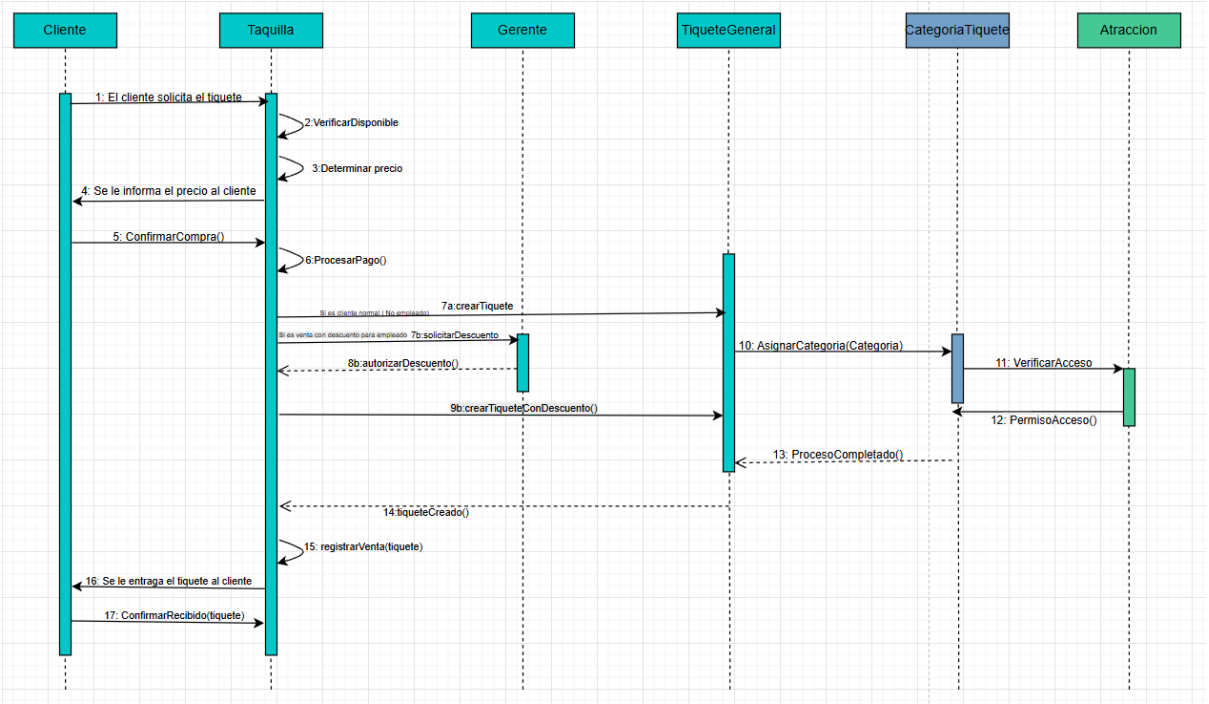
- El diagrama muestra las clases y sus relaciones en un sistema de gestión de parque de diversiones

Ayuda de los colores

- Para simplificar el diseño del diagrama decidimos utilizar colores para representar los diferentes componentes del UML, esto nos permitió ahorrar más espacio y hacer mas agradable a la vista el diagrama. Específicamente Utilizamos 3 Colores: Turquesa para las clases concretas, verde para las clases abstractas y para finalizar utilizamos azul para las enumeraciones.

Diagrama de secuencia

Venta De Tiquete



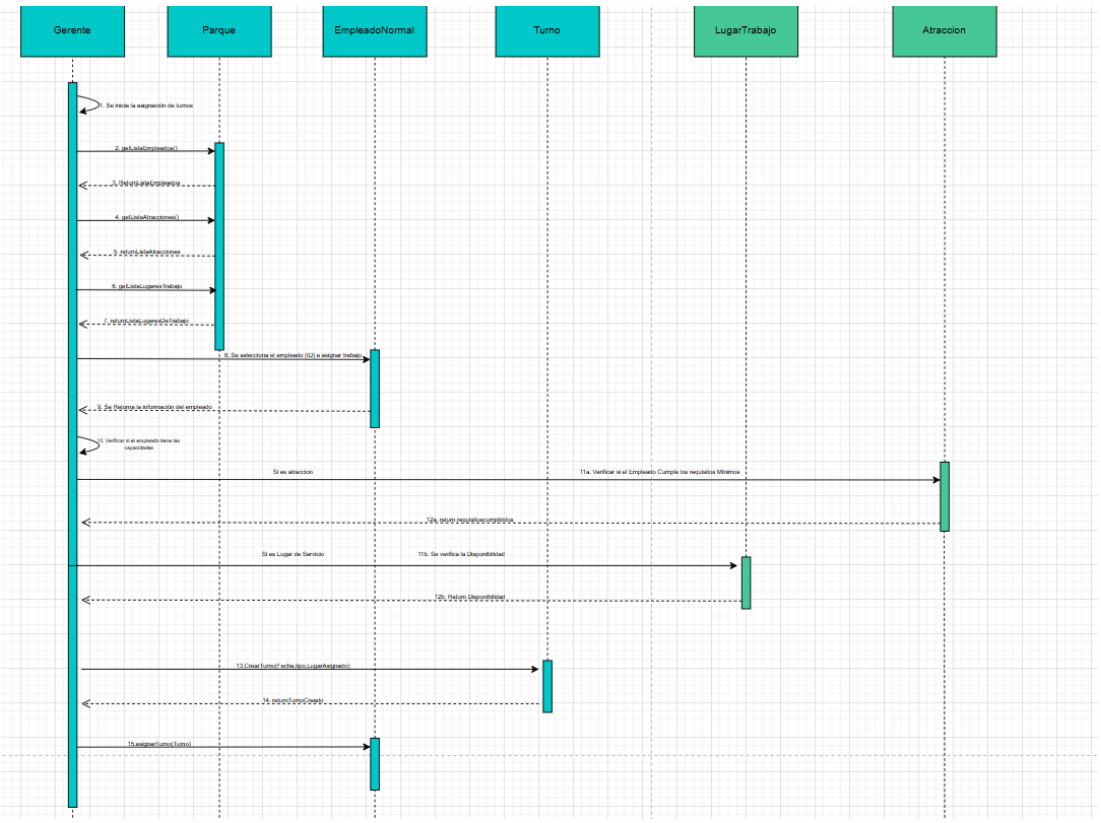
Justificación:

El diagrama de secuencia presenta un proceso de venta de tiquetes meticulosamente diseñado, donde cada actor desempeña un papel específico en un flujo de trabajo complejo y estructurado. Desde el momento en que el cliente solicita un tiquete, el sistema activa una serie de verificaciones y procesos interconectados que involucran múltiples componentes como gerencia, sistemas de ticketing y control de acceso.

La interacción comienza con la solicitud del cliente, seguida por verificaciones de disponibilidad y determinación de precios, donde el sistema demuestra flexibilidad para manejar diferentes escenarios, como ventas normales o con descuentos especiales para empleados. Cada paso está cuidadosamente orquestado, con mensajes síncronos y asíncronos que permiten una comunicación precisa entre los diferentes actores.

Un aspecto destacable es la gestión de categorías de tiquetes, donde el sistema realiza verificaciones de acceso y permisos, asegurando un control riguroso desde la solicitud hasta la confirmación final. Las líneas de comunicación revelan una arquitectura que prioriza la seguridad, la trazabilidad y la flexibilidad, permitiendo manejar diferentes tipos de transacciones con un alto grado de precisión y adaptabilidad.

Asignar Turno



Justificación:

El diagrama de secuencia para la asignación de turnos representa un proceso administrativo fundamental en la operación del parque de diversiones, donde el Gerente cumple un rol central como orquestador del flujo completo. Este proceso ha sido estructurado meticulosamente para asegurar una asignación eficiente que cumpla con todos los requisitos operativos y de seguridad del parque.

La secuencia comienza con una fase preliminar de recopilación de información, donde el Gerente consulta sistemáticamente las bases de datos del Parque para obtener listas actualizadas de empleados, atracciones y lugares de servicio. Esta etapa inicial es crucial pues proporciona el panorama completo de los recursos humanos disponibles y las necesidades operativas del parque, estableciendo así la base para tomar decisiones informadas.

Un aspecto particularmente destacable del diseño es la bifurcación condicional que ocurre cuando se verifica el tipo de lugar de trabajo. El sistema implementa verificaciones específicas según se trate de una atracción o un lugar de servicio, reconociendo que cada tipo de asignación tiene requisitos y restricciones diferentes. Para las atracciones, se verifica si el empleado cumple con los requisitos mínimos de capacitación, un aspecto crítico especialmente para atracciones de riesgo alto que requieren personal con entrenamiento especializado. En el caso de lugares de servicio, el enfoque cambia hacia la verificación de disponibilidad, asegurando una distribución adecuada del personal.