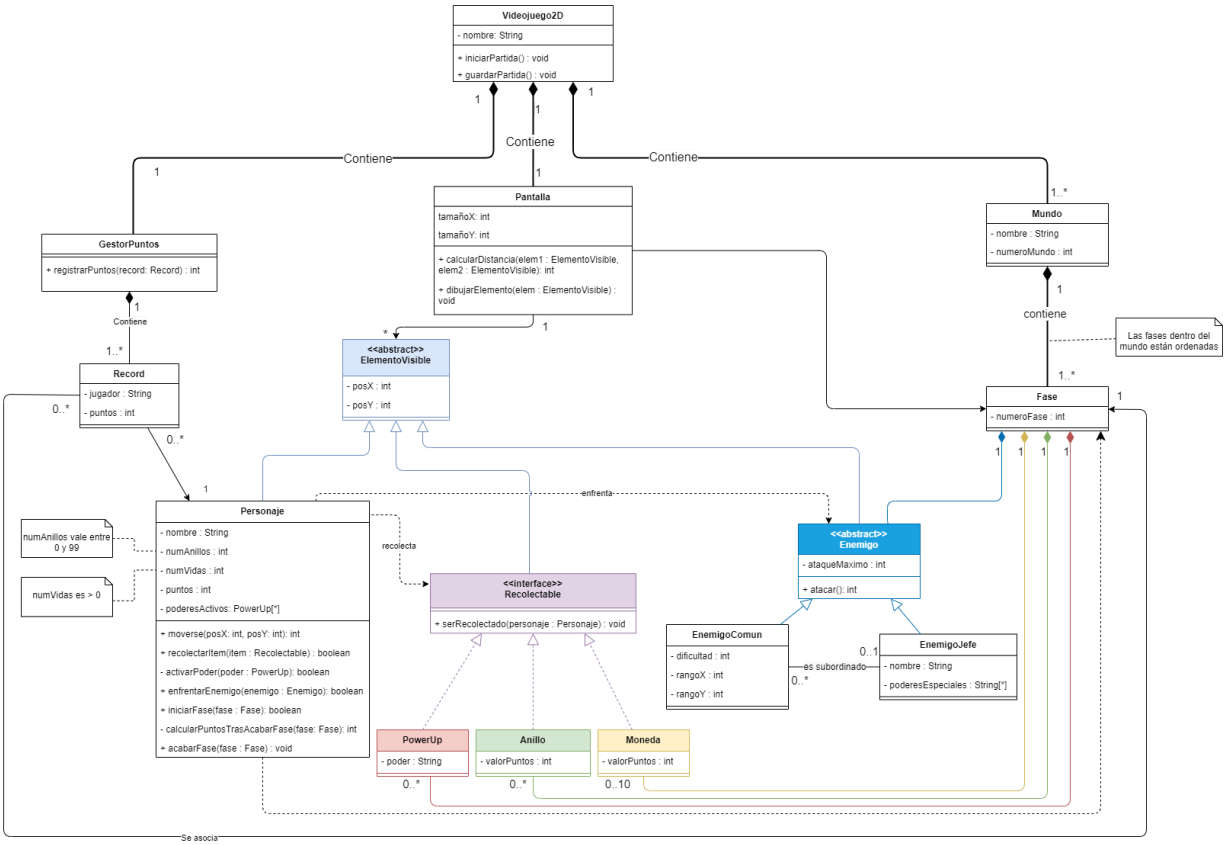
ED04_1 Sistema de Gestión para un Concesionario de Vehículos

Solución propuesta

A continuación se muestra el diagrama de clases propuesto para resolver el problema planteado.



Respuesta a las preguntas de reflexión

¿Por qué es útil definir la interfaz `DescuentoAplicable` para los clientes?

La interfaz `DescuentoAplicable` permite definir un contrato común para todos los clientes que pueden aplicar descuentos. De esta forma, se garantiza que cualquier cliente que implemente esta interfaz podrá recibir descuentos en sus compras, independientemente de su tipo concreto.

¿Qué ventajas aporta el uso de clases abstractas (`Vehiculo` y `Cliente`) para modelar jerarquías?

Las clases abstractas permiten definir un conjunto de propiedades y métodos comunes a todas las clases que heredan de ellas. En este caso, la clase `Vehiculo` define propiedades y métodos comunes a todos los vehículos, mientras que la clase `Cliente` define propiedades y métodos comunes a todos los clientes. Esto facilita la reutilización de código y la definición de comportamientos comunes. Estas clases se definen como abstractas ya que no hay intención de instanciar objetos de ellas directamente, sino que se utilizan como base para otras clases concretas.

Justifica la relación empleada entre `Venta` y `Vehiculo`/`Cliente` (asociación, composición, agregación...).

En ambos casos, la relación empleada es la de asociación, una relación estructural pero flexible, que no implica propiedad, relación todo-parte (composición o agregación) ni de jerarquía (herencia).

¿Cómo el uso de interfaces y herencia mejora la extensibilidad del sistema?

El uso de interfaces y herencia permite definir contratos comunes y comportamientos genéricos que pueden ser implementados por diferentes clases. Esto facilita la extensibilidad del sistema, ya que se pueden añadir nuevas clases que implementen las interfaces o hereden de las clases base sin modificar el código existente. Por ejemplo, si se desea añadir un nuevo tipo de vehículo o cliente, basta con crear una nueva clase que implemente la interfaz `DescuentoAplicable` o herede de la clase `Vehiculo` o `Cliente`, respectivamente.

¿Qué ventajas ofrece dividir la clase `Concesionario` en varios componentes especializados?

Dividir la clase `Concesionario` en varios componentes especializados permite separar las responsabilidades y mejorar la cohesión del sistema. Cada componente se encarga de una funcionalidad específica, como la gestión de vehículos, clientes o ventas, lo que facilita la comprensión y mantenimiento del código. Además, esta división permite reutilizar los componentes en otros contextos y facilita la extensibilidad del sistema. Esta división en componentes especializados sigue el principio de responsabilidad única (SRP) de SOLID.

¿Cómo garantizarías que el sistema sea extensible para incluir nuevos tipos de vehículos o clientes?

Las jerarquías creadas sobre clases abstractas aseguran que el sistema sea extensible para incluir nuevos tipos de vehículos o clientes. Si se desea añadir un nuevo tipo de vehículo, basta con crear una nueva clase que herede de la clase `Vehiculo` y defina sus propiedades y métodos específicos. De la misma forma, si se desea añadir un nuevo tipo de cliente, basta con crear una nueva clase que herede de la clase `Cliente` y defina sus propiedades y métodos específicos. Además, si se desea que un cliente pueda aplicar descuentos, basta con implementar la interfaz `DescuentoAplicable` en la nueva clase de cliente.

¿Qué principios SOLID están mejor representados en este diseño y cómo?

Fundamentalmente:

- **SRP (Single Responsibility Principle):** Cada clase y componente del sistema tiene una única responsabilidad, lo que facilita la comprensión y mantenimiento del código. La división del `Concesionario` en varios gestores especializados sigue este principio.
- **OCP (Open/Closed Principle):** El sistema es extensible para incluir nuevos tipos de vehículos o clientes sin modificar el código existente. La jerarquía de clases y el uso de interfaces permiten añadir nuevas clases sin alterar las existentes.

¿Es recomendable o deseable representar los constructores y getters/setters de las clases en el diagrama de clases? ¿Hay casos donde esa representación tenga más sentido? Justifica si, en tu caso, has decidido incluirlos en el diagrama o no.

En general, no es necesario representar los constructores y getters/setters en el diagrama de clases, ya que son detalles de implementación que no afectan a la estructura del sistema. Sin embargo, en algunos casos puede ser útil incluirlos para mostrar cómo se inicializan las clases o cómo se accede a sus atributos. En este caso, se ha decidido no incluirlos en el diagrama de clases para mantener el foco en la estructura y relaciones entre las clases.