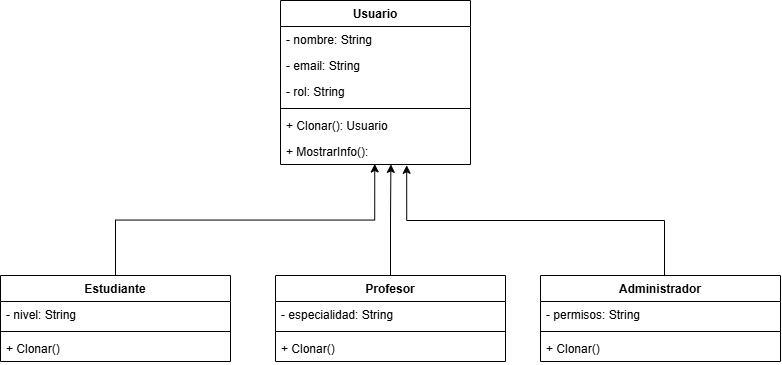
**1. Patrón del caso: Prototype.**

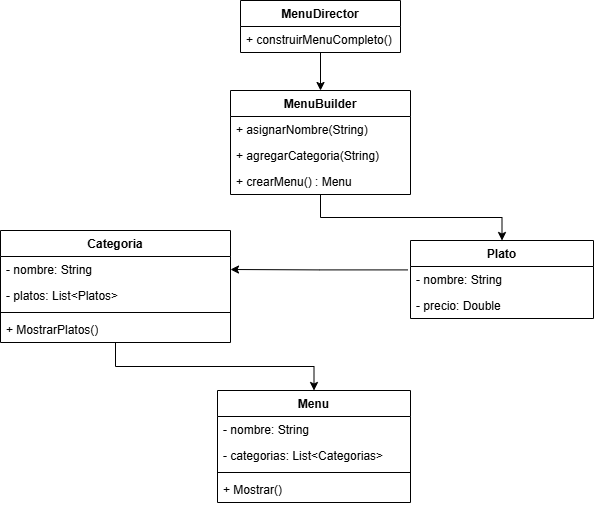
En una plataforma educativa los usuarios pueden tener diferentes roles (estudiantes, profesores, administradores) con atributos y funcionalidades específicos, si se crea un usuario por cada rol esto nos llevaría a duplicar código dado que los usuarios independientemente del rol comparten atributos y funcionalidades. El patron Prototype nos permite implementar una clase UsuarioBase que sirve como plantilla y permite extender la clase con los atributos y funcionalidades específicas del rol. Esto optimiza la creación de usuarios sin repetir código.

****

**2. Generación de menús de restaurante dinámicos (Patrón Builder)**

**Patrón del caso: Builder.**

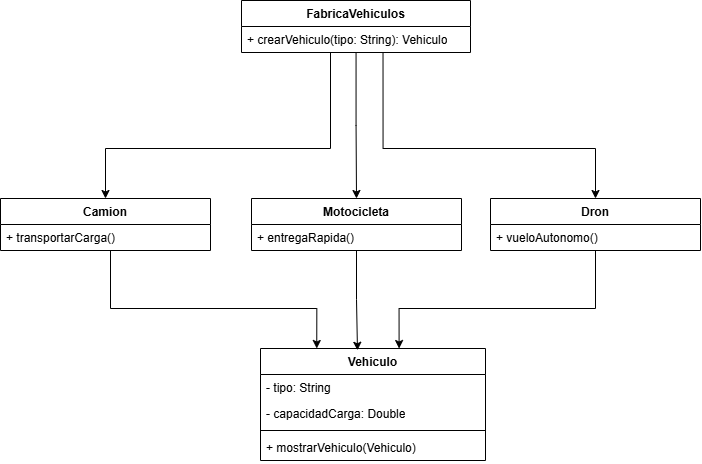
Un sistema de pedidos en línea necesita generar menús personalizados según el restaurante, incluyendo distintos tipos de platos, tipos de bebidas y adicionales. La estructura del menú debe ser flexible para adaptarse a las necesidades de cada negocio. El patrón Builder nos permite definir una clase GeneradorMenu que se encargara de construir un menú paso a paso, agregando las categorías y platos según las preferencias de cada restaurante facilitando la personalización sin afectar la lógica interna de la aplicación.



**3. Creación de vehículos en una aplicación de logística (Patrón Factory Method)**

**Patrón del caso: Factory Method.**

Una empresa de transporte necesita gestionar distintos tipos de vehículos (camiones, motocicletas, drones) según el tipo de carga y destino. La creación manual de cada vehículo dificulta el mantenimiento del código, el patron Factory Method permite Implementar una clase FabricaVehiculos que nos permita instanciar vehículos según el tipo requerido sin modificar el código base facilitando la escalabilidad y adaptación a nuevos tipos de vehiculos.

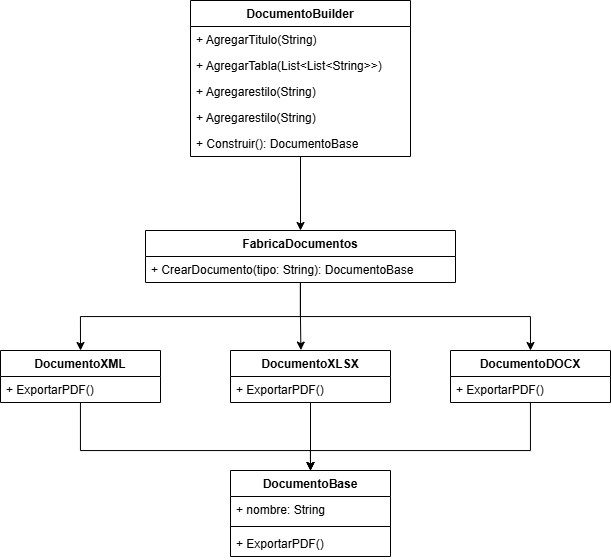


**1. Diseño del sistema y elección de patrones**

Como solución a este problema se plantea el uso del Patrón Factory Method para la selección flexible del tipo de documento, sin que el cliente tenga que conocer los detalles de cada formato.

De igual manera el patron builderfacilitará la construcción de documentos complejos, asegurando que se puedan agregar columnas, tablas, títulos y estilos de manera flexible.

**2. Diagrama UML de Clases**



**4. Explicación de la solución**

Para la solución propuesta se propone el uso de los patrones Factory Mehod y Builder trabajando de manera conjunta de la siguiente forma:

1. Selección del tipo de documento: El patron Factory Method nos permitirá encapsular la creación de documentos según el formato (XLSX, DOCX, XML). La clase FabricaDocumentos creara objetos sin que el cliente conozca los detalles de cada formato.
2. Construcción del documento flexible:

Se emplea el patron Builder para agregar dinámicamente elementos como títulos, tablas y estilos. La clase DocumentoBuilder nos permitirá estructurar los documentos de forma flexible sin modificar el código base.

1. Extensibilidad del sistema:

Si en el futuro se requieren nuevos formatos de documentos, simplemente se añade una nueva clase sin modificar la fábrica ni el cliente, De igual manera la clase se puede extender la clase DocumentoBuilder para admitir nuevos tipos de contenido sin afectar el código existente.