# Resumen de SOLID

Todos los principios SOLID hablan de cómo preparar el código para futuros cambios. La cuestión, es que cada principio se centra en un aspecto del cambio, pero en el fondo, todos los principios recomiendan: “Si algo va a cambiar, sepáralo”.

## 1. Single Responsibilite Principle

Una clase debería tener una única razón para cambiar.

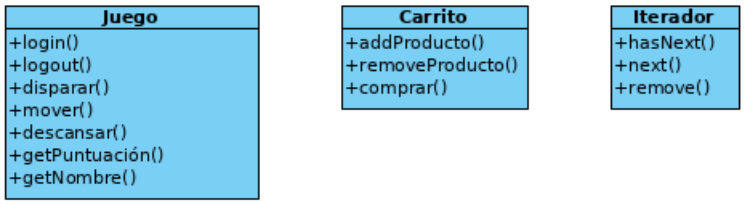
SRP enfoca la razón del cambio en las cosas que hace una clase y cómo es usada por otras clases cliente.

- Llevar a cabo actividades muy dispares. Por ejemplo, mezclar diferentes capas arquitectónicas, com una parte del modelo y de la interfaz.
- Ser utilizado por clientes de diferente naturaleza, para cuestiones diferentes.

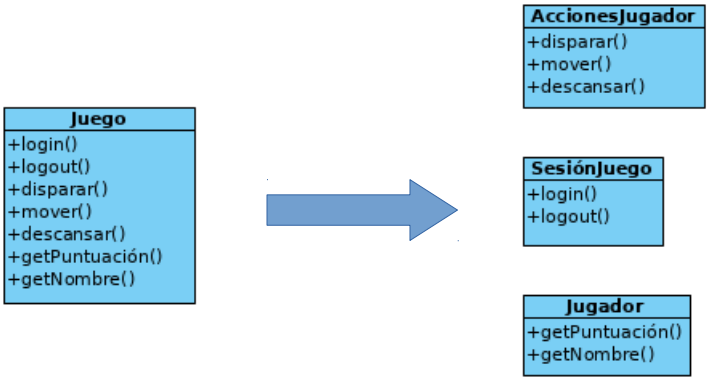
**`Estrategias`**

- `Dividir en clases más especializadas.`
- `Si no tiene sentido dividir, se pueden añadir clases intermedias, que hagan de intermediarias.`

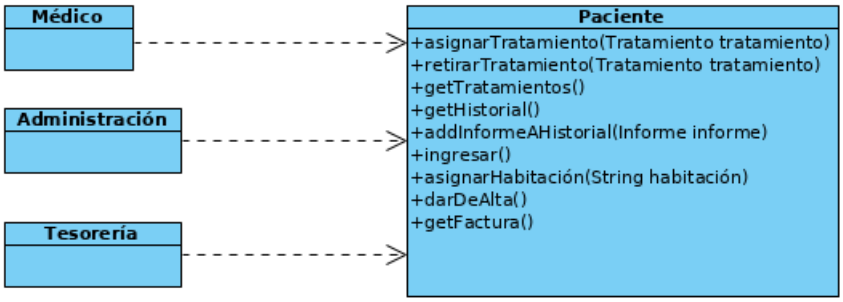
**Ejemplo 1. ¿Cuál de estas clases inclumple SRP?**




Solución:



**Ejemplo 2. ¿Por qué se incumple SRP en el siguiente diseño? Supón que la factura se calcula a partir de la lista de tratamientos, y que los médicos necesitan un cambio en la obtención de dicha lista, para que sólo se muestren ciertos tratamientos. ¿Cómo se puede solucionar si no es posible modificar la clase *Paciente*?**

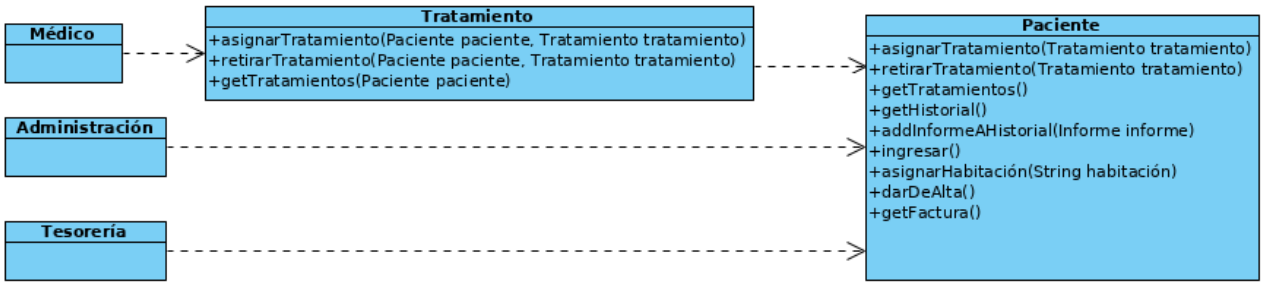



Solución.

Se incumple SRP porque:

- La clase *Paciente* tiene muchas responsabilidades diferentes.
- Las clases *Médico, Administración* y *Tesorería* utilizan la clase *Paciente* de manera diferente, y son potenciales motivos de cambio.

Para poder modificar la lista de tratamientos que obtienen los médicos y que no afecte a la factura, se puede hacer lo siguiente:




*Nota: Se sigue incumpliendo SRP, por las razones comentadas anteriormente, pero hemos conseguido resolver el cambio sin modificar Paciente*

# 2. Open-Close Principle

Todo artefacto debería ser escrito para su extensión, pero también para evitar cambios.

OCP se centra en la amplicación de las funcionalidades de una parte del código.

- Por ejemplo, un módulo que ya es capaz de guardar datos en una base de datos, ahora debe hacerlo también en archivos XML.
- Otro ejemplo puede ser un cambio en la forma en que se representa cierta información. Por ejemplo, una clase que calcula perímetros de polígonos mediante listas de longitudes, ahora también debe calcular perímetros mediante listas de coordenadas.

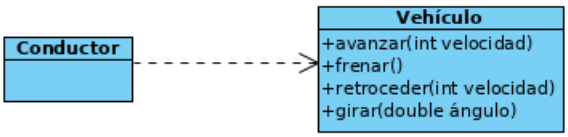
**`Estrategias`**

- `Los componentes de mayor nivel de abstracción deben estar protegidos de los cambios/ampliaciones en los componentes de menor nivel de abstracción.`
- `Separar las partes del código que puedan ser susceptibles de cambio.`
- `Abstraer mediante clases abstractas o interfaces las partes del código que puedan sufrir ampliaciones.`

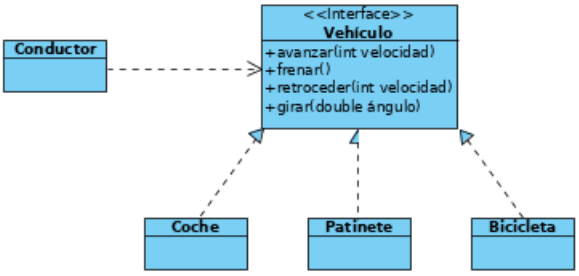
**Ejemplo 1. En un sistema, la clase *Conductor* depende de la clase *Vehículo*. Supongamos que las especificaciones requieren que comportamientos diferentes de la clase *Vehículo* dependiendo del tipo de vehículo. Por ejemplo:**

- **Un patinete eléctrico no puede ir a más de 60 Km/h y no puede dar marcha atrás.**
- **Un coche no puede ir a más de 120 Km/h y puede dar marcha atrás a 15 Km/h como máximo.**

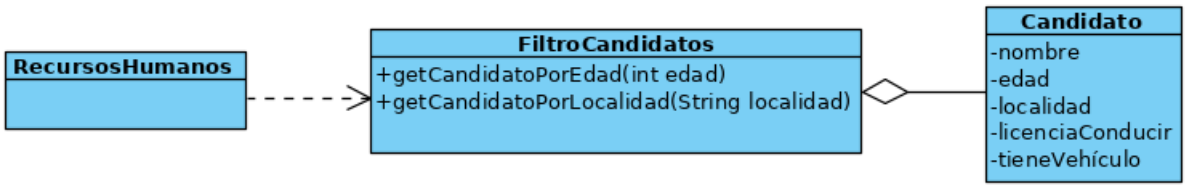
**Supongamos que se pide incluir ahora un nuevo vehículo, la bicicleta, entre los vehículos ya existentes. ¿Por qué se incumple OCP en este diseño?**




Solución: OCP recomienda utilizar abstracciones en lugar de clases concretas. En este caso, incluir nuevos tipos de vehículos requiere modificar los métodos de *Vehículo*, por lo que se viola OCP. Una posible solución sería:

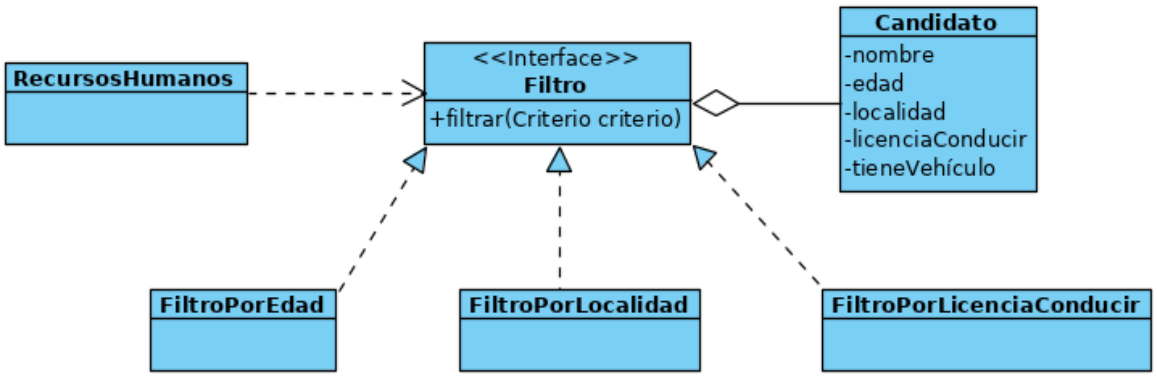



`Ejemplo 2. La clase RecursosHumanos utiliza una clase llamada FiltroCandidatos, para seleccionar candidatos por un criterior. Inicialmente se filtra por edad y por localidad. Pero se necesita añadir como criterio de filtro el tipo de licencia conducir. ¿Por qué se viola OCP en el siguiente diseño?`




Solución: Para añadir el nuevo filtro, habría que modificar la clase *FiltroCandidatos* y añadir un nuevo método *getCandidatosPorLicenciaConducir.* Por lo tanto se está violando OCP.

Una posible solución sería abstraer los filtros mediante una interfaz *Filtro.* El criterio se podría abstraer mediante una clase Criterio, que incluya información sobre la consulta.



# 3. Liskov Substitution Principle

Una clase padre debería ser intercambiable con cualquiera de sus clases hijas sin que el software falle.

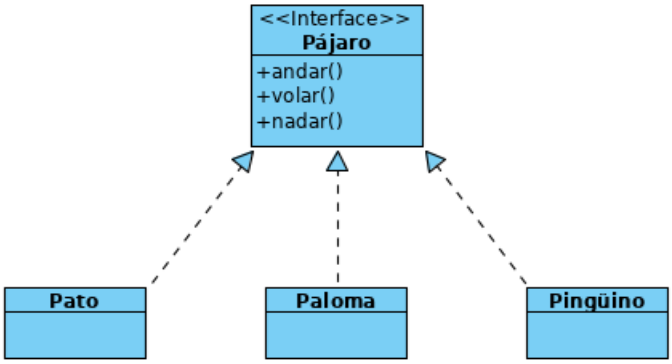
LSP se centra en cómo aplicar correctamente la herencia y el polimorfismo.

- Los métodos de una clase padre y sus subclases deberían ser coherentes.
- Una subclase debería poder aceptar, al menos, el mismo rango de valores que su clase padre.
- Una subclase debería saber resolver los mismos problemas que su clase padre.
- Los resultados de una subclase no deberían salirse de lo que se espera de su clase padre.

**`Estrategias`**

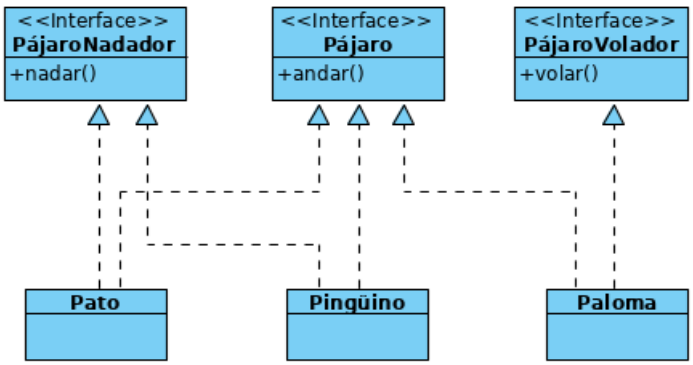
- `Separar en interfaces diferentes aquellas características no implementables por todas las subclases.`

**Ejemplo 1. En un juego se utiliza la interfaz *Pájaro,* implementada por *Pato, Paloma* y *Pinguïno.* ¿Por qué se incumple LSP?**

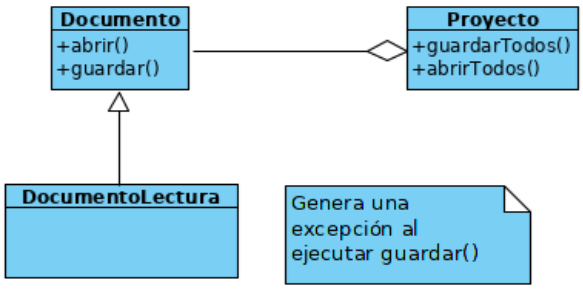



Solución: se incumple LSP porque algunas de las clases no pueden implementar la interfaz debido a que son más restrictivas que la interfaz. Por ejemplo, una paloma no puede implementar *nadar,* ni *Pingüino* puede implementar *volar.*

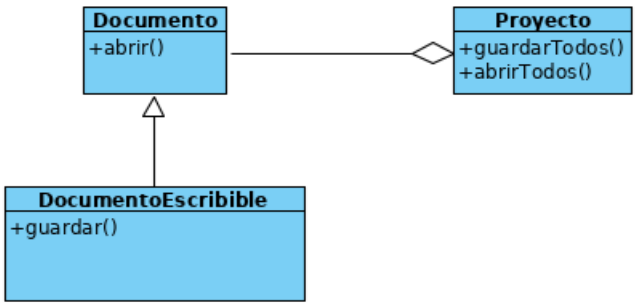
Una posible solución podría ser separar las partes de la interfaz que no pueden implementar todas las subclases.



**Ejemplo 2. Un sistema de gestión de documentos incluye dos tipos de documentos: documentos de lectura y documentos que se pueden modificar. ¿Por qué el siguiente diseño no cumple LSP?**




Solución: no cumple LSP porque la clase *DocumentoLectura* no soporta el método heredado *guardar.* El comportamiento de *DocumentoLectura* es más restrictivo que el de su clase padre, y por eso no se puede intercambiar la clase padre por la clase hija.
Una posible solución podría ser cambiar la jerarquía de herencia:



# 4. Interface Segregation Principle

Una clase no debería depender de partes de otra que no utiliza.

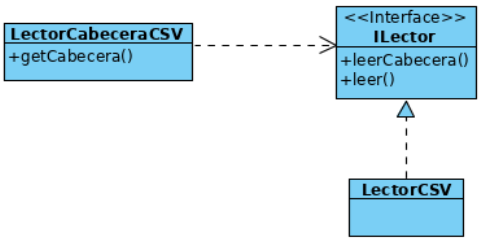
ISP se puede enfocar desde el punto de vista de la herencia o del uso como cliente.

- Desde el punto de vista de la herencia, una clase hija no debería heredar código que no va a ser utilizado o no tiene sentido. Por ejemplo, una clase *Pájaro* que implementa el método *volar* no debería tener una subclase *Pingüino,* que lo heredaría. Según esto, un pingüino vuela.
- Desde el punto de vista de las dependencias, una clase cliente debe tener acceso sólo a los métodos que que necesita. Por ejemplo, supongamos una clase *RecursosHumanos* que tiene varios métodos entre los que están *getNóminaEmpleado* y *altaEmpleado.* Una clase llamada *Empleado* debería tener acceso sólo al método *getNóminaEmpleado*, pero no a *altaEmpleado.*

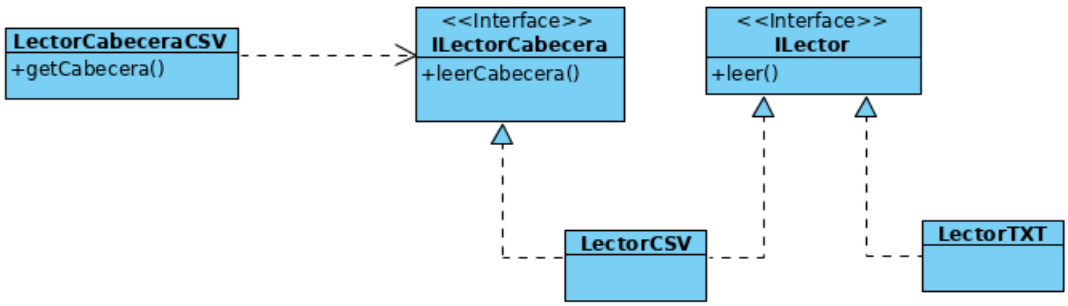
**`Estrategias`**

- `Al igual que en LSP, cuando se enfoca en la herencia, separar las superclases en interfaces.`
- `Cuando se enfoca en las dependencias, utilizar interfaces intermedias con los métodos necesarios exclusivamente.`

**Ejemplo 1. El siguiente diseño incluye una interfaz *Ilector*, que ofrece la interfaz general para leer documentos y sus cabeceras. La clase *LectorCabeceraCSV* hace uso de la clase *LectorCSV* a través de la interfaz *Ilector. Este diseño* incumple ISP. ¿Por qué?**

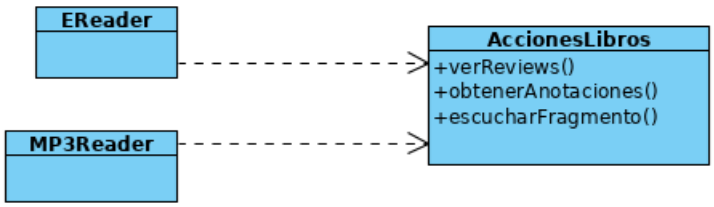



Solución: La clase *LectorCabeceraCSV* tiene acceso al método *leer* innecesariamente. Según ISP, debería segregarse el método *leerCabecera* a una interfaz específica. De esta manera, además, podrían añadirse nuevas clases *Lector* sin tener que implementar *leerCabecera.* Una posible solución sería:

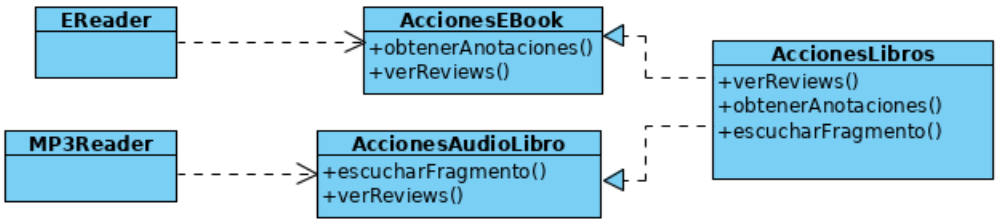


NOTA: La clase *LectorTXT* sólo se ha hañadido a modo de ejemplo.

**Ejemplo 2. En una librería online se pueden hacer ciertas acciones sobre los libros. El diseño original incluye una clase llamada *AccionesLibros* que contiene todas las operaciones posibles sobre un libro. También existen las clases *Ereader* y *MP3Reader* que son clientes de la primera. Este diseño incumple *ISP* ¿Por qué?**




Solución: La clase *AccionesLibros* está sobrecargada. Esto hace que las clases *EReader* y *MP3Reader* dependan de métodos que no necesitan. Una posible solución que evita modificar *AccionesLibros,* sería utilizar interfaces intermedias con los métodos precisos para cada tipo de libro.



# 5. Dependency Inversion Principle

Una clase no debería crear directamente clases concretas, sino que debería usar abstracciones.

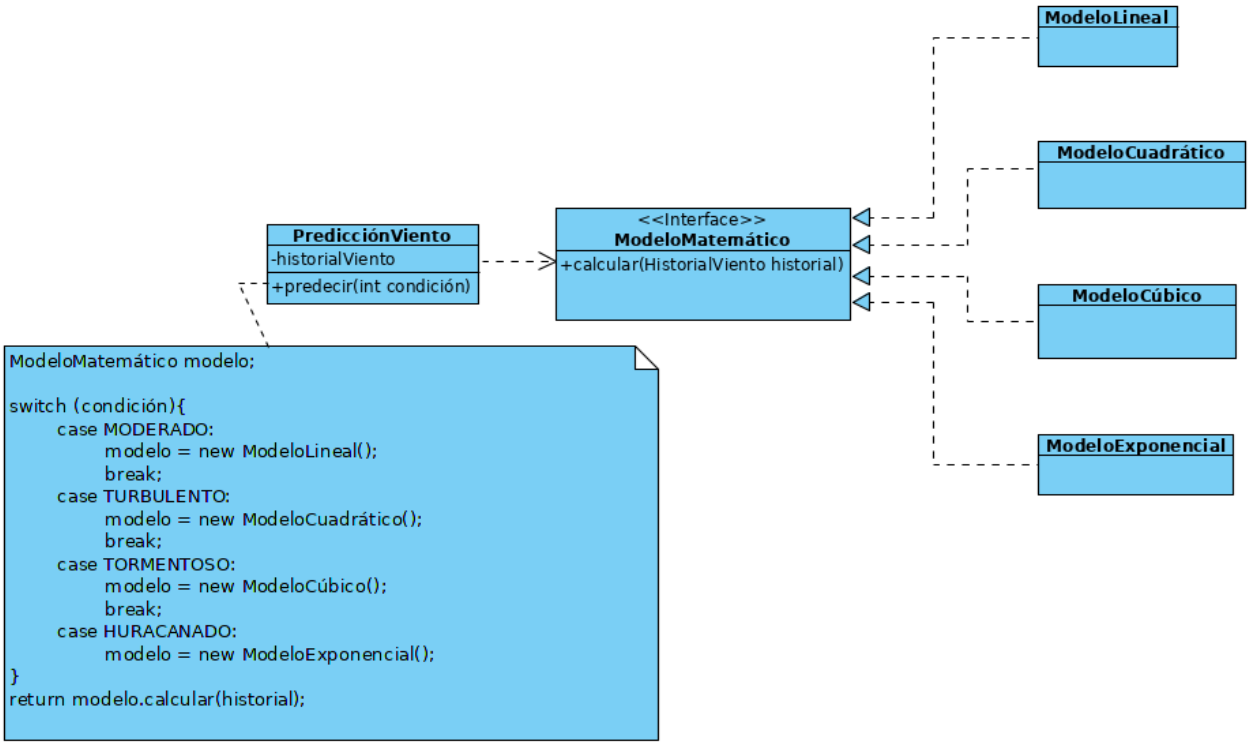
DIP se centra en los cambios en tiempo de ejecución de una subclase por otra.

- Supongamos que existe una clase abstracta llamada *ProveedorDatos*. Supongamos también que tiene dos subclases llamadas *ProveedorDatosXML* y *ProveedorDatosSQL*. Una clase que utilice *ProveedorDatos*, no debería tener que saber si en tiempo de ejecución se está usando *ProveedorDatosXML* o *ProveedorDatosSQL*.

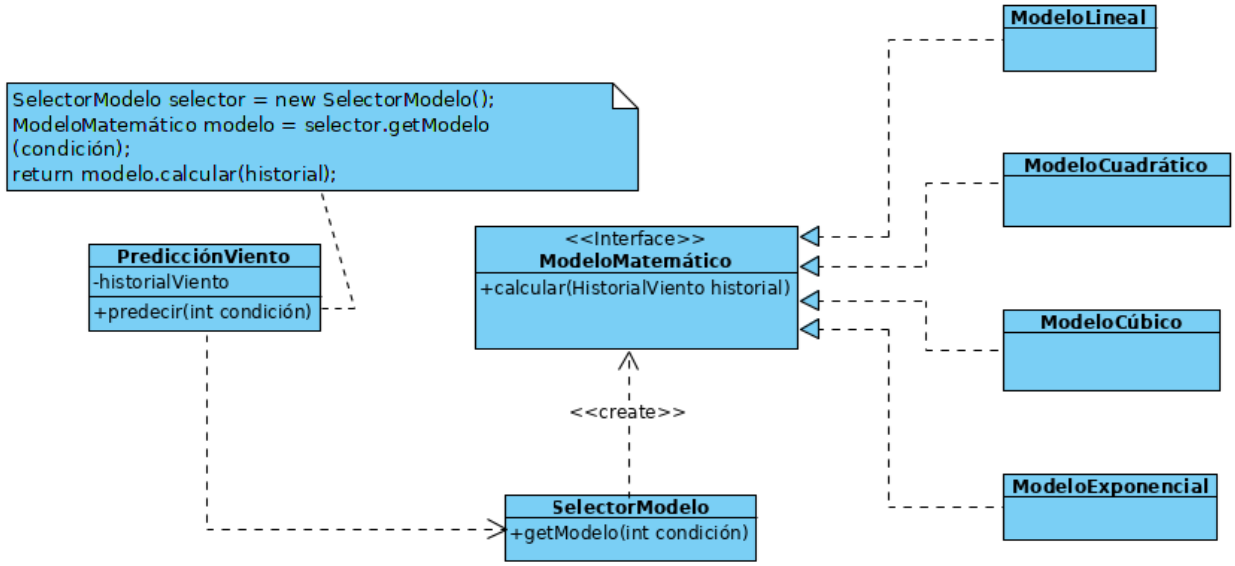
**`Estrategias`**

- `Pasar el objeto utilizado como un parámetro.`
- `Usar una clase intermedia que cree el objeto.`

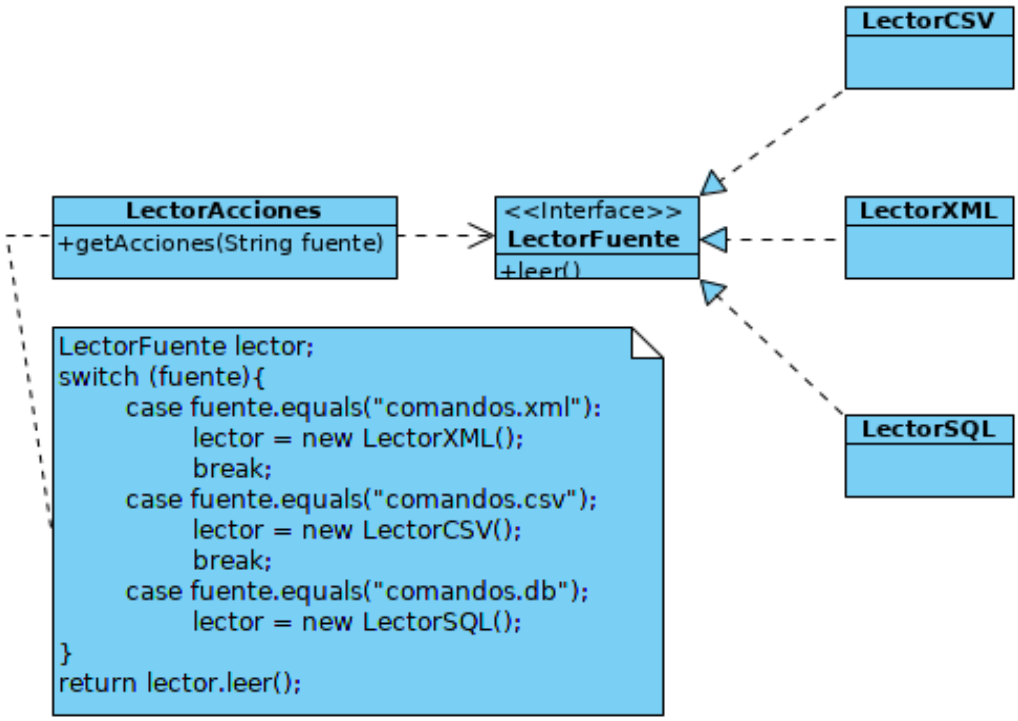
**Ejemplo 1. En un sistema de control de vuelo la clase *PrediccionViento* predice la fuerza y dirección del viento en los próximos segundos. Para ello, la clase *PrediccionViento* utiliza diferentes modelos matemáticos que se adaptan mejor dependiendo de las condiciones climáticas. El diseño inicial es el siguiente ¿Por qué incumple DIP?**



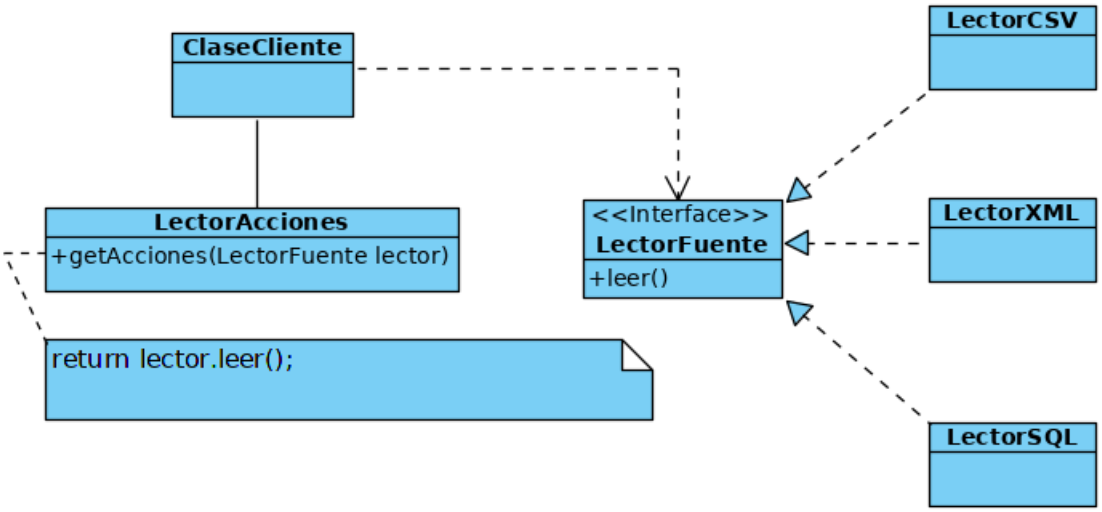
Solución: incumple ISP porque crea directamente los objetos a partir de sus clases concretas. Esto crea un gran acoplamiento entre *PredicciónViento* y las clases *ModeloX.* Además, obligará a modificar la clase *PredicciónViento* en caso de añadir nuevos modelos matemáticos.
Para inyectar la dependencia se pueden tomar dos vías: pasar como parámetro la dependencia, o bien dejar que sea otra clase la que cree el objeto necesario. Siguiendo esta última vía, una posible solución podría ser la siguiente:




**Ejemplo 2. La clase *LectorAcciones* lee un conjunto de comandos a realizar desde una cierta fuente, que puede ser una base de datos, un archivo CSV o un documento XML. Esta clase utiliza un lector determinado dependiendo la configuración almacenada en un archivo de configuración. Si el archivo de configuración incluye el valor "fuente=comandos.csv" utiliza la clase *LectorCSV.* Cuando el valor es "fuente=comandos.xml" utiliza la clase *LectorXML.* Si el valor es "fuente=comandos.db" utiliza la clase *LectorSQL.* El diseño original es el siguiente. ¿Porqué incumple DIP?**




Solución: El diseño incumple DIP porque la clase *LectorAcciones* crea directamente las clases concretas de clase *LectorFuente*. Por plantear una solución diferente al ejemplo anterior, se podría elegir la inyección de dependencias por parámetro. Es decir, la clase que utilice *LectorAcciones* (en este caso llamada *ClaseCliente*)le pasará el tipo de lector conveniente.




NOTA: en esta solución, *LectorAcciones* ya está cumpliendo DIP, ya que no depende de clases concretas sino de la abstracción *LectorFuente*, pero no es seguro que *ClaseCliente* lo haga. En alguna parte habrá un fragmento de código que cree clases concretas *LectorCSV, LectorXML* o *LectorSQL*. Lo ideal es que sea una clase cuya función sea únicamente esa (como un patrón *Factory Method* que ya veremos).