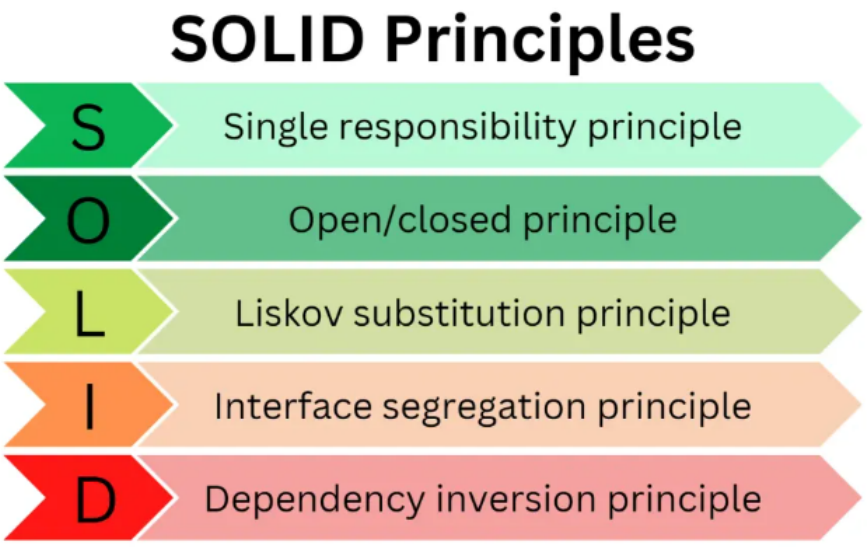
Principios SOLID



SOLID es el acrónimo que acuñó **Michael Feathers**, basándose en los principios de la programación orientada a objetos que **Robert C. Martin** había recopilado en el año 2000 en su paper "*Design Principles and Design Patterns*".

Ocho años más tarde, el *tío Bob* siguió compendiando consejos y buenas prácticas de desarrollo y se convirtió en el padre del código limpio con su célebre libro *'Clean Code'*.

Entre los objetivos a alcanzar si tenemos en cuenta estos 5 principios a la hora de escribir código encontramos:

- Crear un **software eficaz** que cumpla con su cometido y que sea **robusto y estable**.
- Escribir un **código limpio y flexible** ante los cambios, lo que significa que se pueda modificar fácilmente según necesidad, que sea **reutilizable y mantenible**.
- Permitir **escalabilidad**, o lo que es lo mismo, que acepte ser ampliado con nuevas funcionalidades de manera ágil.

En este sentido la aplicación de los principios SOLID está muy relacionada con la comprensión y el uso de **patrones de diseño**, que nos permitirán mantener una **alta cohesión** y, por tanto, un **bajo acoplamiento** de software. En definitiva, desarrollar un software de calidad.

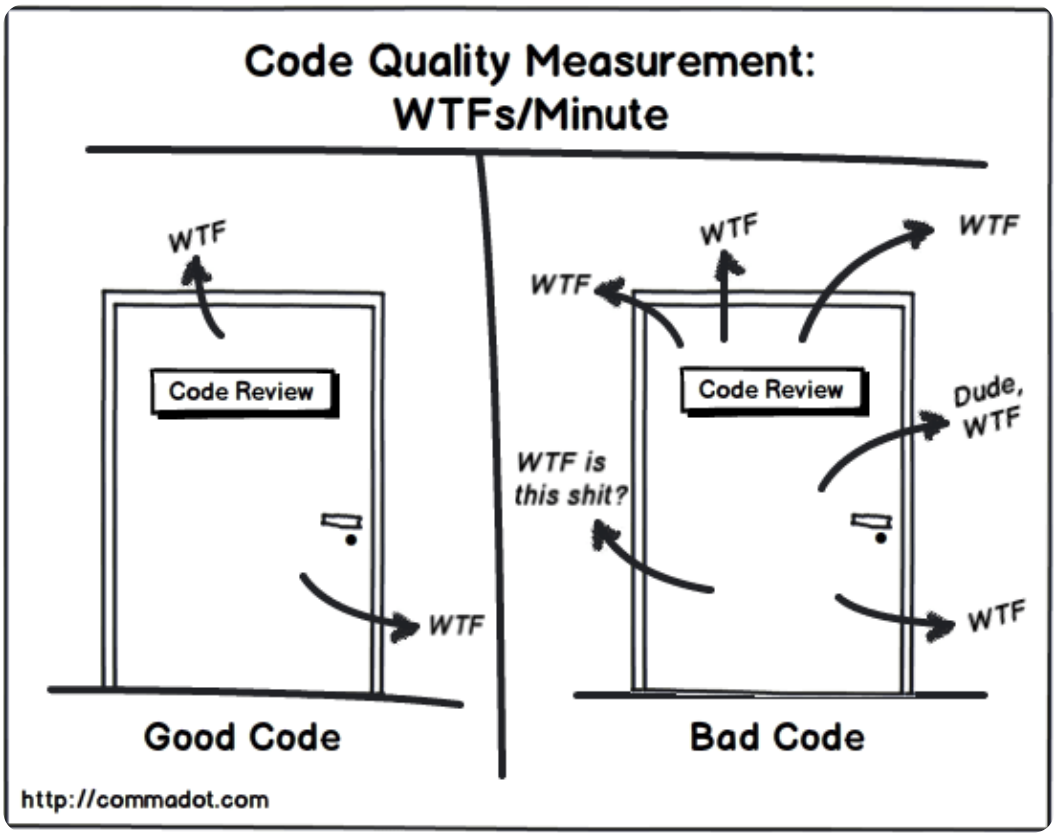
El acoplamiento se refiere al **grado de interdependencia que tienen dos unidades de software entre sí**, entendiendo por unidades de software: clases, subtipos, métodos, módulos, funciones, bibliotecas, etcétera… Si dos unidades de software son completamente independientes la una de la otra, decimos que están **desacopladas**.

La cohesión de software es el **grado en que elementos diferentes de un sistema permanecen unidos para alcanzar un mejor resultado** que si trabajaran por separado. Se refiere a la forma en que podemos agrupar diversas unidades de software para crear una unidad mayor.

Los principios SOLID son eso: principios, es decir, **buenas prácticas** que pueden ayudar a escribir un mejor código más limpio, mantenible y escalable.

Como indica el propio Robert C. Martin en su artículo "*Getting a SOLID start*" no se trata de reglas, ni leyes, ni verdades absolutas, sino más bien soluciones de sentido común a problemas comunes. Son heurísticos, basados en la experiencia: "se ha observado que funcionan en muchos casos; pero no hay pruebas de que siempre funcionen, ni de que siempre se deban seguir."

Dice el *tío Bob*, que SOLID nos ayuda a categorizar lo que es un buen o mal código y es innegable que un código limpio tenderá más a salir airoso del “control de calidad de código WTFs/Minute":



Los 5 principios SOLID son:

1. ***Single Responsability Principle (SRP)*** **- Principio de Responsabilidad Única**

   Este principio establece que cada módulo o clase debe tener **responsabilidad sobre una sola parte de la funcionalidad** proporcionada por el software y esta responsabilidad debe estar encapsulada en su totalidad por la clase. Todos sus servicios deben estar estrechamente alineados con esa responsabilidad.

2. ***Open/Closed Principle (OCP)*** **- Principio de Abierto/Cerrado**

   Este principio establece que **«una entidad de software (clase, módulo, función, etc.) debe quedar abierta para su extensión, pero cerrada para su modificación»**. Es decir, se debe poder extender el comportamiento de la entidad pero sin modificar su código fuente.

3. ***Liskov Substitution Principle (LSP)*** **- Principio de Substitución de Liskov**

   Este principio puede definirse como: **«cada clase que hereda de otra puede usarse como su padre sin necesidad de conocer las diferencias entre ellas»**.

4. ***Interface Segregation Principle (ISP)*** **- Principio de Segregación de la Interfaz**

   Este principio establece que los clientes de un programa dado sólo deberían conocer **aquellos métodos del programa que realmente usan, y no aquellos que no necesitan usar**.

5. ***Dependency Inversion Principle (DIP)*** **- Principio de Inversión de Dependencias**

   Este principio consta de dos partes:

   - **Módulos de alto nivel no deben depender de módulos de bajo nivel**. Ambos deben depender de abstracciones.

- **Abstracciones no deberían depender de detalles**. Los detalles debieran depender de abstracciones.

"Single Responsibility Principle"

> *"A class should have one, and only one, reason to change"*
>
> -- Robert C. Martin

Los requerimientos del código pueden cambiar con el tiempo. Cada uno de estos cambios en los requerimientos va a modificar al menos la responsabilidad de una clase. Si una clase tiene muchas responsabilidades deberá cambiar más a menudo que si sólo tuviera una responsabilidad.

Estos cambios tan reiterados pueden introducir errores o efectos secundarios en otras partes del código. Por tanto, **una clase sólo debería cambiar por una única razón** o lo que es lo mismo, que cambie la responsabilidad de la que se ocupa. Es esto, precisamente, "*razón para cambiar*", lo que Robert C. Martin identifica como **"responsabilidad"**.

Las clases con una única responsabilidad son más fáciles de mantener y menos propensas a errores.

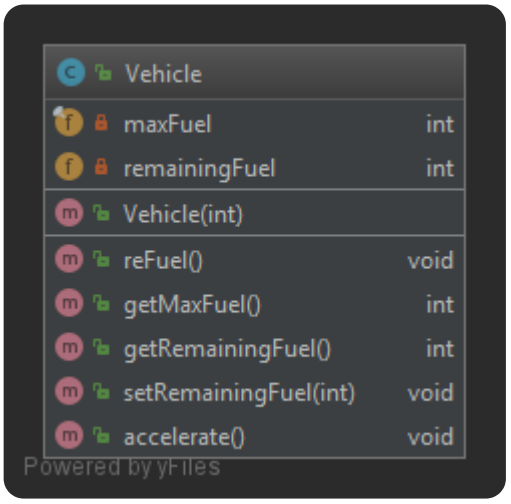
Este principio ayuda a crear código de calidad, mantenible, reusable, testeable, fácil de implementar y previene de efectos secundarios en los cambios.

Es aplicable a clases, componentes de software o microservicios.

Implementación

En el siguiente ejemplo la clase `Vehicle` modela un vehículo y sus propiedades. Además tiene la responsabilidad de repostar el vehículo.

Por tanto, si cambia el modelo o si cambia la forma de repostar combustible esta clase tendrá **dos** motivos para cambiar, es decir, ser modificada por lo que esta clase no estaría cumpliendo este principio:



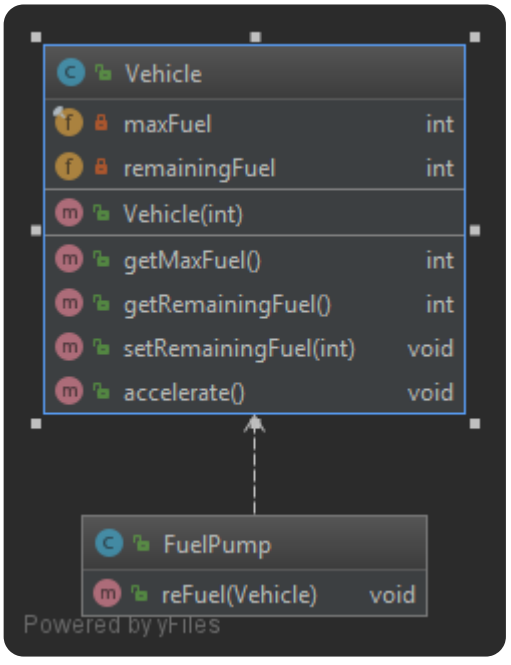

```
class Vehicle {
    private final int maxFuel;
    private int remainingFuel;

    public Vehicle(final int maxFuel) {
        this.maxFuel = maxFuel;
        remainingFuel = maxFuel;
    }

    // Esto no es responsabilidad de la clase 'Vehicle'
    public void reFuel() {
```

```
        remainingFuel = maxFuel;
    }

    public int getMaxFuel() {
        return maxFuel;
    }

    public int getRemainingFuel() {
        return remainingFuel;
    }

    // ....
}
```

Para aplicar este principio deberemos refactorizar la clase `Vehicle` y crear una clase responsable del respostaje del vehículo como por ejemplo `FuelPump`:

```
class FuelPump {
    void reFuel(final Vehicle vehicle) {
        final int remainingFuel = vehicle.getRemainingFuel();
        final int additionalFuel = vehicle.getMaxFuel() - remainingFuel;
        vehicle.setRemainingFuel(remainingFuel + additionalFuel);
    }
}
```

Eliminando este método de la clase `Vehicle` eliminamos la doble responsabilidad que tenía.



"Open/Closed Principle"

> *"Software entities (classes, modules, functions, etc...) should be open for extension, but closed for modification"*
>
> -- Robert C. Martin

La idea es escribir código de forma que sea posible **añadir nuevas funcionalidades pero sin modificar el código existente**. Esto previene situaciones en que al modificar clases base nos veamos obligados también a adaptar todas las clases dependientes.

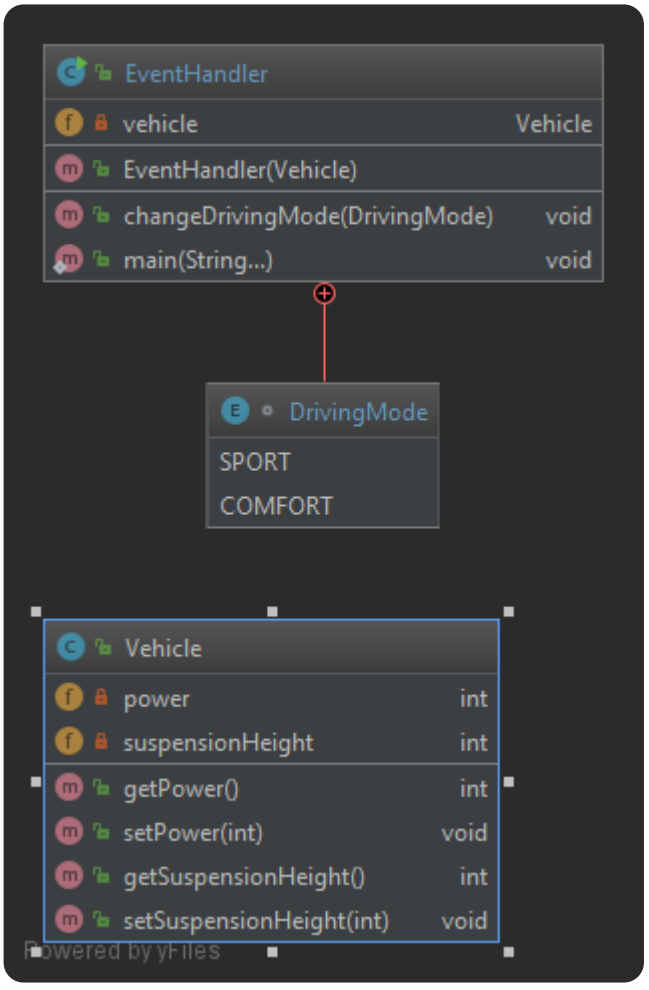
Inicialmente este principio se basaba en el uso de la herencia pero **Robert C. Martin** y otros autores, basándose en su experiencia llegaron a la conclusión de que la herencia crea una fuerte dependencia entre las clases. Por tanto la recomendación es el **uso de interfaces en lugar de la herencia**.

El mayor beneficio es que las interfaces introducen una capa extra de abstracción que otorga un bajo nivel de acoplamiento. La implementación que hace cada clase de esa interfaz son independientes unas de otras y no necesitan compartir el código.

Sin embargo, en el caso de que los beneficios de compartir código fueran notables sería mejor optar por la herencia o la composición.

Implementación

En el siguiente ejemplo la clase `Vehicle` modela un vehículo y sus propiedades. Por otro lado, la clase `EventHandler` con el método `changeDrivingMode(...)` permite cambiar ciertos parámetros según el modo de conducción:



Este modo de conducción se codifica en una enumeración. Una posible implemetación sería:

```
class EventHandler {

    enum DrivingMode {
        SPORT, COMFORT
    }

    private Vehicle vehicle;

    public EventHandler(final Vehicle vehicle) {
        this.vehicle = vehicle;
    }
```

```
    void changeDrivingMode(final DrivingMode drivingMode) {
        switch (drivingMode) {
            case SPORT:
                vehicle.setPower(500);
                vehicle.setSuspensionHeight(10);
                break;
            case COMFORT:
                vehicle.setPower(400);
                vehicle.setSuspensionHeight(20);
                break;
            default:
                vehicle.setPower(200);
                vehicle.setSuspensionHeight(30);
                break;
            // Cuando necesitemos añadir otro modo (e.g. ECONOMY)
            // deberemos cambiar la clase 'EventHandler'
            // y la enumeración 'DrivingMode'.
        }
    }
}
```

```
class Vehicle {
    private int power;
    private int suspensionHeight;

    int getPower() {
        return power;
    }

    void setPower(final int power) {
        this.power = power;
    }

    int getSuspensionHeight() {
        return suspensionHeight;
    }

    void setSuspensionHeight(final int suspensionHeight) {
        this.suspensionHeight = suspensionHeight;
    }
}
```

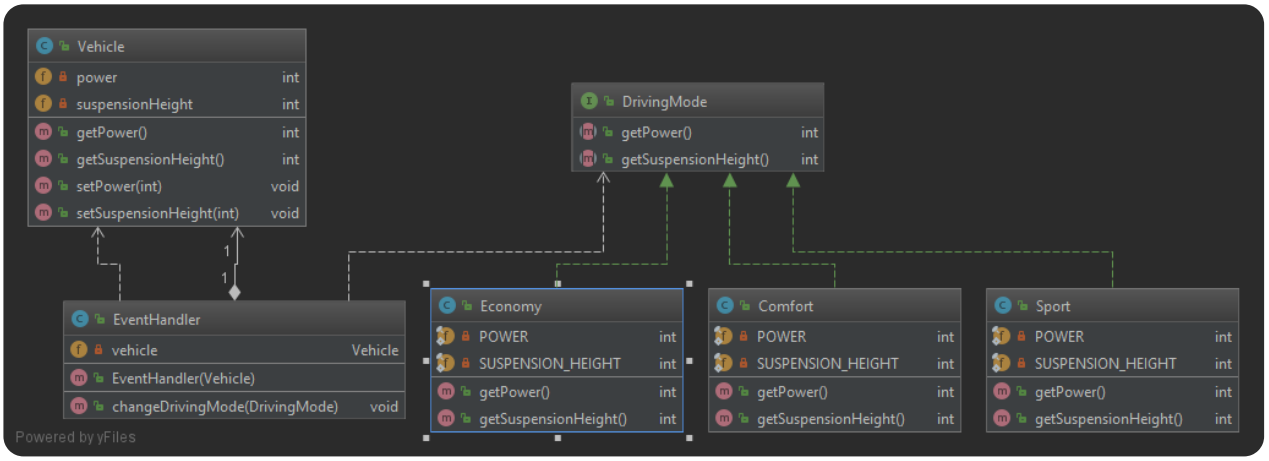
El principio se incumple ya que si tenemos que añadir un nuevo modo de conducción, deberemos añadir el nuevo modo en la enumeración y además deberemos modificar el método `changeDrivingMode(...)` para tener en cuenta este nuevo modo. Por tanto, al añadir nueva funcionalidad estamos obligados a modificar el código existente.

Además, si esta modificación se tuviera que hacer en varias partes del código, en caso de "despiste" por parte del programador, se producirían errores que son complicados de detectar.

Para cumplir este principio deberemos refactorizar el código de forma que el método `changeDrivingMode(...)` no necesite ser modificado si se añade nueva funcionalidad o nuevos modos de conducción. **Este método debe permanecer cerrado a la modificación**.

Se puede alcanzar haciendo uso de las **interfaces** (en vez del uso de la herencia) de forma que en el método `changeDrivingMode(...)` utilice la nueva interfaz `DrivingMode`. Las clases que modelan los modos de conducción implementarán dicha interfaz.

Si en el futuro se necesita añadir un nuevo modo de conducción únicamente será necesario añadir la nueva clase que representa el modo de conducción y que implementa la interfaz `DrivingMode` para que el sistema tenga en cuenta el nuevo modo. El método `changeDrivingMode(...)` permanecerá inalterado y plenamente funcional ya que este método hace uso de la interfaz y ésta no se ha modificado.




```
interface DrivingMode {
    int getPower();
    int getSuspensionHeight();
}
```

```
class Comfort implements DrivingMode {
    private static final int POWER = 400;
    private static final int SUSPENSION_HEIGHT = 20;

    @Override
    public int getPower() {
        return POWER;
    }

    @Override
    public int getSuspensionHeight() {
        return SUSPENSION_HEIGHT;
    }
}
```

```
class Sport implements DrivingMode {
    private static final int POWER = 500;
    private static final int SUSPENSION_HEIGHT = 10;

    @Override
    public int getPower() {
        return POWER;
    }

    @Override
    public int getSuspensionHeight() {
        return SUSPENSION_HEIGHT;
    }
}
```

```
class EventHandler {
    private Vehicle vehicle;

    public EventHandler(final Vehicle vehicle) {
        this.vehicle = vehicle;
    }

    public void changeDrivingMode(final DrivingMode drivingMode) {
        vehicle.setPower(drivingMode.getPower());
        vehicle.setSuspensionHeight(drivingMode.getSuspensionHeight());
        // Ahora, cuando necesitemos añadir otro modo (e.g. ECONOMY)
```

```
        // sólo hay que crear la clase 'Economy'.
    }
}
```

"Liskov Substitution Principle"

Este principio extiende el ***"Open/Closed Principle"*** pero focalizado en el comportamiento de una superclase y sus subtipos.

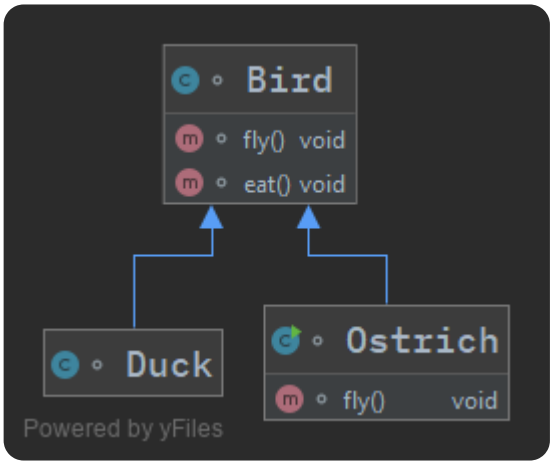
Este principio define que los **objetos de una superclase deben ser reemplazables por objetos de sus subclases** sin "romper" la aplicación o sistema y sin efectos secundarios. Eso requiere que los objetos de las subclases se comporten de la misma manera que los objetos de la superclase de forma que se puedan usar de forma **indistinta**.

Para conseguir esto las subclases deberían seguir estas reglas:

- No implementar reglas de validación más estrictas en los parámetros de entrada que las implementadas por la clase base.
- Aplicar al menos las mismas reglas a todos los parámetros de salida aplicados por la clase base.

Implementación

En el ejemplo las clases `Duck` y `Ostrich` heredan de la clase base `Bird` y por tanto también sus métodos. Sin embargo, la clase `Ostrich` sobreescribe el método `fly()` porque no aplica:




```
// Bird.java
class Bird {
    void fly() {}
    void eat() {}
}
```

```
// Duck.java
class Duck extends Bird { }
```

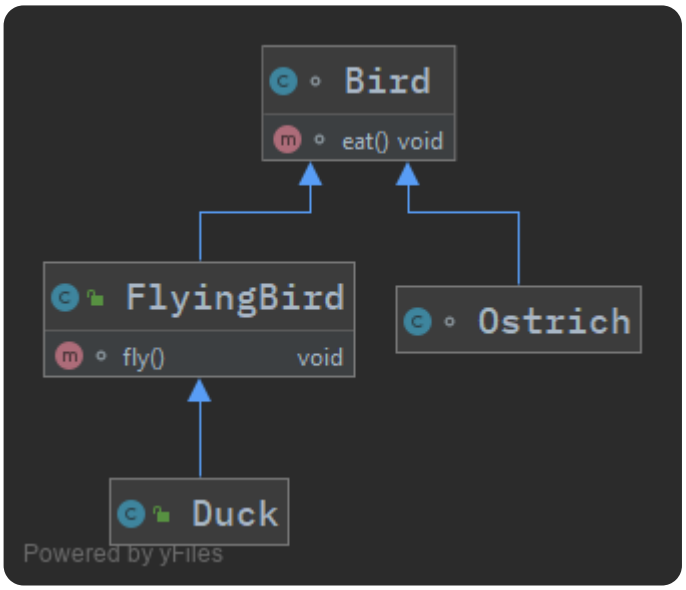
```
// Ostrich.java
class Ostrich extends Bird {
    void fly(){
        throw new UnsupportedOperationException();
    }
}
```

Según este principio se debería poder utilizar las subclases `Duck` y/o `Ostrich` en lugar de la superclase `Bird`. Debido a que no se cumple este principio no se puede usar de forma indistinta la superclase o las subclases sin generar errores en la aplicación.

La subclase `Ostrich` tiene unas restricciones superiores a la superclase en el método `fly()` debido a que lanza una excepción de tipo `'UnsupportedOperationException'` que no se lanza ni en la otra subclase ni en la superclase. No se pueden usar de forma indistinta ya que si se usa esta subclase se deberá capturar o relanzar dicha excepción.

Para cumplir con este principio se refactoriza la superclase `Bird` y se crea la clase `FlyingBird` que heredará de esta superclase. El método `fly()` se mueve a la subclase correspondiente y la clase `Duck` pasa a heredar de la clase `FlyingBrid`. De esta forma se puede utilizar las subclases y la superclase de forma indistinta.

El método `fly()` podrá ser invocado independientemente de que tengamos un objeto de tipo `Duck` o `FlyingBird` y a su vez el método `eat()` podrá ser invocado independientemente de que tengamos un objeto de tipo `Bird`, `Ostrich`, `Duck` o `FlyingBird`.




```
// Bird.java
class Bird {
    void eat() {}
}

// Ostrich.java
class Ostrich extends Bird { }

// FlyingBird.java
public class FlyingBird extends Bird {
    void fly() {}
}

// Duck.java
public class Duck extends FlyingBird { }
```

"Interface Segregation Principle"

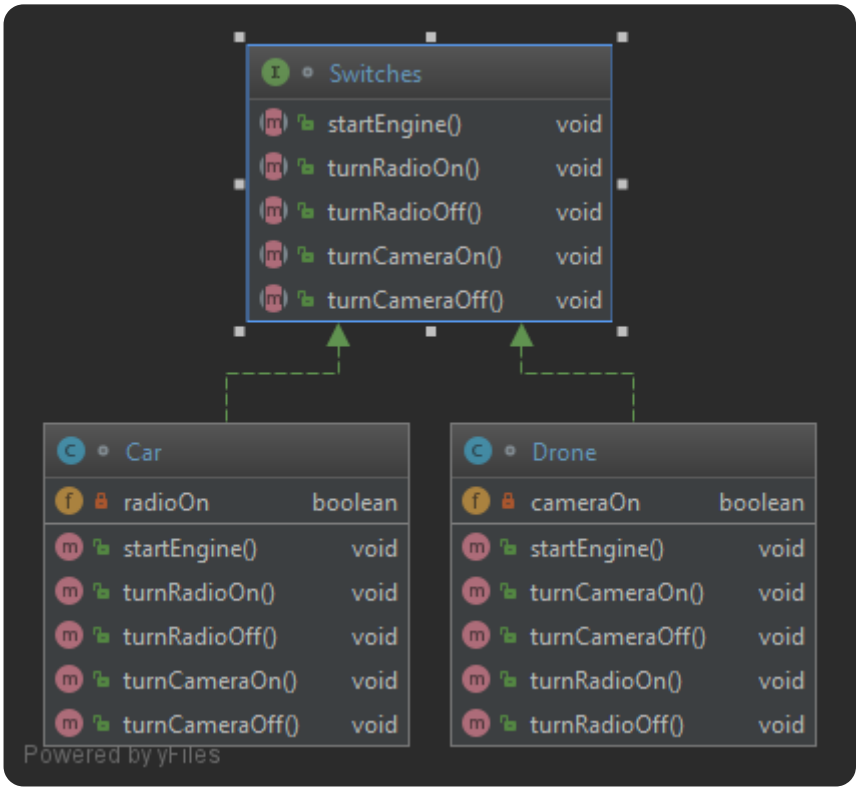
> ***"Clients should not be forced to depend upon interfaces that they do not use"***
>
> -- Robert C. Martin

El objetivo de este principio, al igual que el ***"Single Responsibility Principle"*** es reducir los efectos secundarios y la frecuencia de los cambios si dividimos el código en múltiples partes independientes.

Al seguir este principio se evitan interfaces infladas que definen métodos para múltiples responsabilidades.

Implementación

En el ejemplo las subclases `Drone` y `Car` modelan diferntes tipos de vehículos y además implementan la interfaz `Switches`:

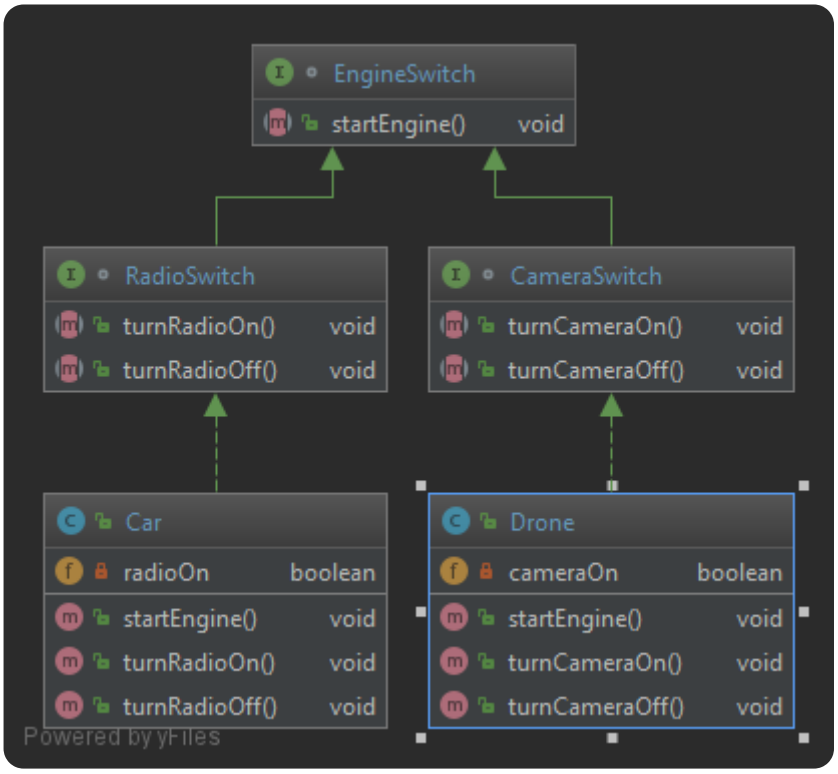



```
// Switches.java
interface Switches {
    void startEngine();
    void turnRadioOn();
    void turnRadioOff();
    void turnCameraOn();
    void turnCameraOff();
}

// Car.java
class Car implements Switches {
    private boolean radioOn;

    @Override
    public void startEngine() {
        // ...
    }

    @Override
    public void turnRadioOn() { radioOn = true; }

    @Override
    public void turnRadioOff() { radioOn = false;  }

    @Override
    public void turnCameraOn() {
        // nothing to do here
    }

    @Override
    public void turnCameraOff() {
        // nothing to do here
    }
}

// Drone.java
```

```java
class Drone implements Switches {
    private boolean cameraOn;

    @Override
    public void startEngine() {
        // ...
    }

    @Override
    public void turnCameraOn() { cameraOn = true; }

    @Override
    public void turnCameraOff() { cameraOn = false; }

    @Override
    public void turnRadioOn() {
        // nothing to do here
    }

    @Override
    public void turnRadioOff() {
        // nothing to do here
    }
}
```

Las subclases, debido a la herencia, se ven obligadas a implementar con un cuerpo vacío los métodos que no les son necesarios. La subclase `Car` se ve obligada a implementar los métodos `turnCameraOn()` y `turnCameraOff()` que son más propios de la subclase `Dron` y al revés.

Para cumplir con este principio se debe refactorizar el código de forma que en vez de tener una única interfaz con demasiada responsabilidad, haya tres interfaces con menor responsabilidad y que se adapten mejor al modelo y a la lógica de negocio.




```java
// EngineSwitch.java
interface EngineSwitch {
    void startEngine();
}

// CameraSwitch.java
```

```
interface CameraSwitch extends EngineSwitch {
    void turnCameraOn();
    void turnCameraOff();
}

// RadioSwitch
interface RadioSwitch extends EngineSwitch {
    void turnRadioOn();
    void turnRadioOff();
}
```

```
// Car.java
class Car implements RadioSwitch {
    private boolean radioOn;

    @Override
    public void startEngine() {
        // ....
    }

    @Override
    public void turnRadioOn() { radioOn = true; }

    @Override
    public void turnRadioOff() { radioOn = false; }
}

// Drone.java
class Drone implements CameraSwitch {
    private boolean cameraOn;

    @Override
    public void startEngine() {
        // ....
    }

    @Override
    public void turnCameraOn() { cameraOn = true; }

    @Override
    public void turnCameraOff() { cameraOn = false; }
}
```

"Dependency Inversion Principle"

La idea general de este principio es tan simple como importante: **los módulos de alto nivel**, que brindan una lógica compleja, **deben ser fácilmente reutilizables** y no verse afectados por los cambios en los módulos de bajo nivel, que brindan funciones de utilidad.

Para lograr eso, se deben introducir **una abstracción que desacople los módulos de alto y bajo nivel entre sí**.

La definición de este principio según **Robert C. Martin** consta de dos partes:

- Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deberían depender de abstracciones.
- Las abstracciones no deben depender de los detalles. Los detalles deben depender de las abstracciones.

Un importante detalle de esta definición es que tanto los módulos de alto nivel como los de bajo nivel dependen de una abstracción. Por tanto no se invierte la dirección de la dependencia como cabría esperar por el nombre del principio sino que se divide la dependencia entre los módulos de alto y bajo nivel introduciendo una abstracción entre ellos.

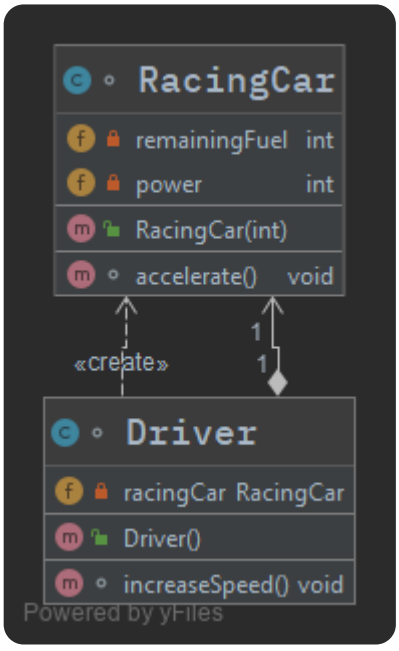
Si se han aplicado correctamente el ***"Open/Closed Principle"*** y el ***"Liskov Substitution Principle"*** también se ha aplicado este principio.

El ***"Open/Closed Principle"*** requiere que el componente esté abierto a extensión pero cerrado a modificación. Se puede lograr introduciendo interfaces para las que puede proporcionar diferentes implementaciones. La interfaz en sí misma está cerrada a modificaciones y puede ampliarse fácilmente proporcionando una nueva implementación de interfaz.

Sus implementaciones deben seguir el ***"Liskov Substitution Principle"*** para que pueda reemplazarlas con otras implementaciones de la misma interfaz sin "romper" la aplicación o sistema.

Implementación

En el ejemplo la clase `Driver` tiene una dependencia con la clase `RacingCar` ya que en su constructor se instancia un objeto de esta clase:

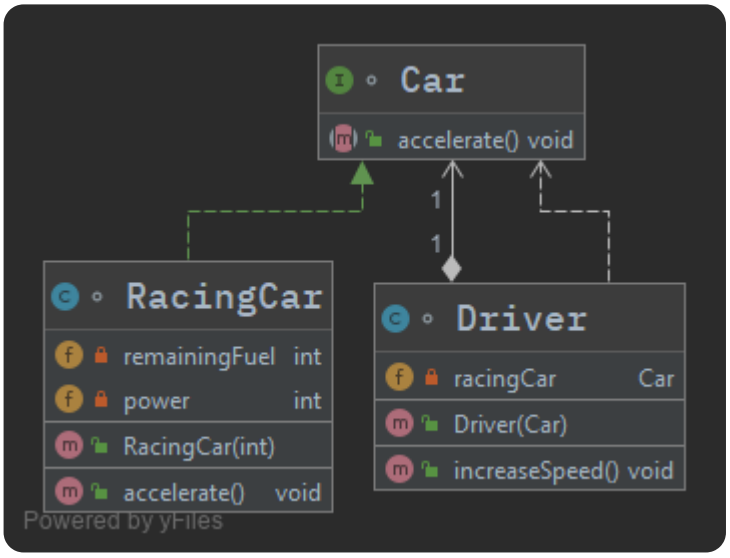



```java
// RacingCar.java
class RacingCar {
    private int remainingFuel;
    private int power;

    public RacingCar(final int fuel) {
        remainingFuel = fuel;
    }

    void accelerate() {
        power++;
        remainingFuel--;
    }
}

// Driver.java
class Driver {
    private RacingCar racingCar;

    public Driver() {
        this.racingCar = new RacingCar(100);
    }

    void increaseSpeed() {
        this.racingCar.accelerate();
```

```
    }
}
```

Para introducir una abstracción que desacople ambas clases se crea la interfaz `Car` de forma que la clase `Driver` en su constructor recibirá un objeto que implementa dicha interfaz.

En el ejemplo la clase `RacingCar` implementa dicha interfaz. Si se han aplicado correctamente los otros principios se podrá utilizar otras implementaciones y/o ampliar la funcionalidad del sistema sin que se produzcan errores.




```java
// Car.java
interface Car {
    void accelerate();
}

// RacingCar.java
class RacingCar implements Car {
    private int remainingFuel;
    private int power;

    public RacingCar(final int fuel) {
        remainingFuel = fuel;
    }

    @Override
    public void accelerate() {
        power++;
        remainingFuel--;
    }
}

// Driver.java
class Driver {
    private Car racingCar;

    public Driver(final Car racingCar) {
        this.racingCar = racingCar;
    }

    public void increaseSpeed() {
        this.racingCar.accelerate();
    }
}
```

Este principio está relacionado con el concepto de inyección de dependencias ya que será otro sistema el que *'inyecte'* en tiempo de ejecución la implementación que requiera la clase en el constructor.

Enlaces

- https://es.wikipedia.org/wiki/SOLID
- https://blogs.msdn.microsoft.com/cdndevs/2009/07/15/the-solid-principles-explained-with-motivational-posters/
- https://www.baeldung.com/solid-principles
- https://www.baeldung.com/java-single-responsibility-principle
- https://www.baeldung.com/java-open-closed-principle
- https://www.baeldung.com/java-liskov-substitution-principle
- https://www.baeldung.com/java-interface-segregation
- https://www.baeldung.com/java-dependency-inversion-principle

Licencia