

**UNIVERSIDAD NACIONAL DE INGENIERIA**

**Título:** Principios SOLID para el desarrollo de software

**Autores:** 1. Huamán Santamaría Fernando Ulises

2. Rivera Blas Frank

3. Rodas Quispe Marcela

**Profesor:** Coronel Castillo Eric Gustavo

Monografía presentada para el curso Java Programación

Lima, Enero del 2020

*Dedico este trabajo a mis enemigos*

*que tanto me han ayudado*

*en mi carrera*

ÍNDICE

Pág.

CAPÍTULO 1

Definición……………………………………………………………………………………4

CAPITULO 2

Single Responsibility Principle………………………………………………………………6

CAPÍTULO 3

Open / Closed Principle………………………………………………………………………………9

CAPÍTULO 4

Liskov Substitution Principle…………………………………………………………………………10

CAPÍTULO 5

Interface Segregation Principle Y Dependency Inversion Principle………………………………….11

CONCLUSIONES

BIBLIOGRAFIA

**CAPITULO 1**

**DEFINICIÓN**

SOLID es un acrónimo de 5 principios de diseño importantes al hacer OPP (Programación Orientada a Objetos). Si bien cada uno de estos cinco principios describe algo específico, también se superponen de tal manera que adoptar uno de ellos implica o lleva a adoptar otro.

Estos 5 principios fueron introducidos por Robert C. Martin en su documento “Design Principles and Design Patterns” pero el acrónimo SOLID fue identificado por Michael Feathers. El acrónimo SOLID proviene de las iniciales de Single Responsibility Principle, Open-Closed Principle, Liskov Substitution Principle, Interface Segregation Principle y Dependency Inversion Principle.

Los beneficios, para los desarrolladores de software, al utilizar los principios SOLID en el diseño de la programación orientada a objetos hace que el software sea más comprensible, más fácil de mantener y más fácil de extender. En consecuencia, a medida que nuestras aplicaciones crecen en tamaño, podemos reducir su complejidad y ahorrarnos muchos problemas más adelante.

En cambio, si desistimos en utilizar los principios SOLID pueden crear problemas duraderos para futuros desarrolladores que trabajan en el proyecto y limitar el éxito potencial de la aplicación que están desarrollando. Estos problemas se conocen comúnmente como "podredumbre de código" (code rot)

La “podredumbre de código” sucede cuando una aplicación se convierte en una gran cantidad de código que los desarrolladores encuentran cada vez más difícil de mantener. La podredumbre de código presenta ciertas características.

• **Rigidez:** pequeños cambios hacen que todo el sistema se reconstruya.

• **Fragilidad:** los cambios en un módulo hacen que otros módulos no relacionados se comporten mal. Imagine un sistema de automóvil en el que cambiar la estación de radio afecta a las ventanas.

• **Inmovilidad:** los componentes internos de un módulo no se pueden extraer y reutilizar en nuevos entornos. Por ejemplo, si el módulo de inicio de sesión de una aplicación no se puede utilizar en un sistema completamente diferente, entonces este módulo es inmóvil, debido a acoplamientos y dependencias entre diferentes módulos. La estrategia es desacoplar las abstracciones centrales de los detalles de bajo nivel, como un esquema de base de datos particular o implementación de UI (web, escritorio) o marcos específicos.

• **Viscosidad:** cuando la construcción y las pruebas son difíciles de realizar y su ejecución lleva mucho tiempo. Cuando incluso un simple cambio es costoso de hacer, y requiere que hagas cambios en múltiples lugares / niveles.

Por último, el objetivo de los desarrolladores de software es que el software cumpla con dos valores esenciales. El primero es que el software tiene que ser capaz de tolerar y facilitar el cambio continuo; y el segundo es el comportamiento que se logra cuando el software satisface las necesidades actuales del usuario sin fallas ni demoras.

A continuación, explicaremos en que consiste cada uno de estos principios del acrónimo SOLID.

**CAPITULO 2**

**SINGLE RESPONSIBILITY PRINCIPLE**

Es posible que haya escuchado la cita: "Haz una cosa y hazlo bien". Esto se refiere al principio de responsabilidad única. En programación, el Principio de responsabilidad única establece que cada módulo o clase debe tener responsabilidad sobre una sola parte de la funcionalidad proporcionada por el software.

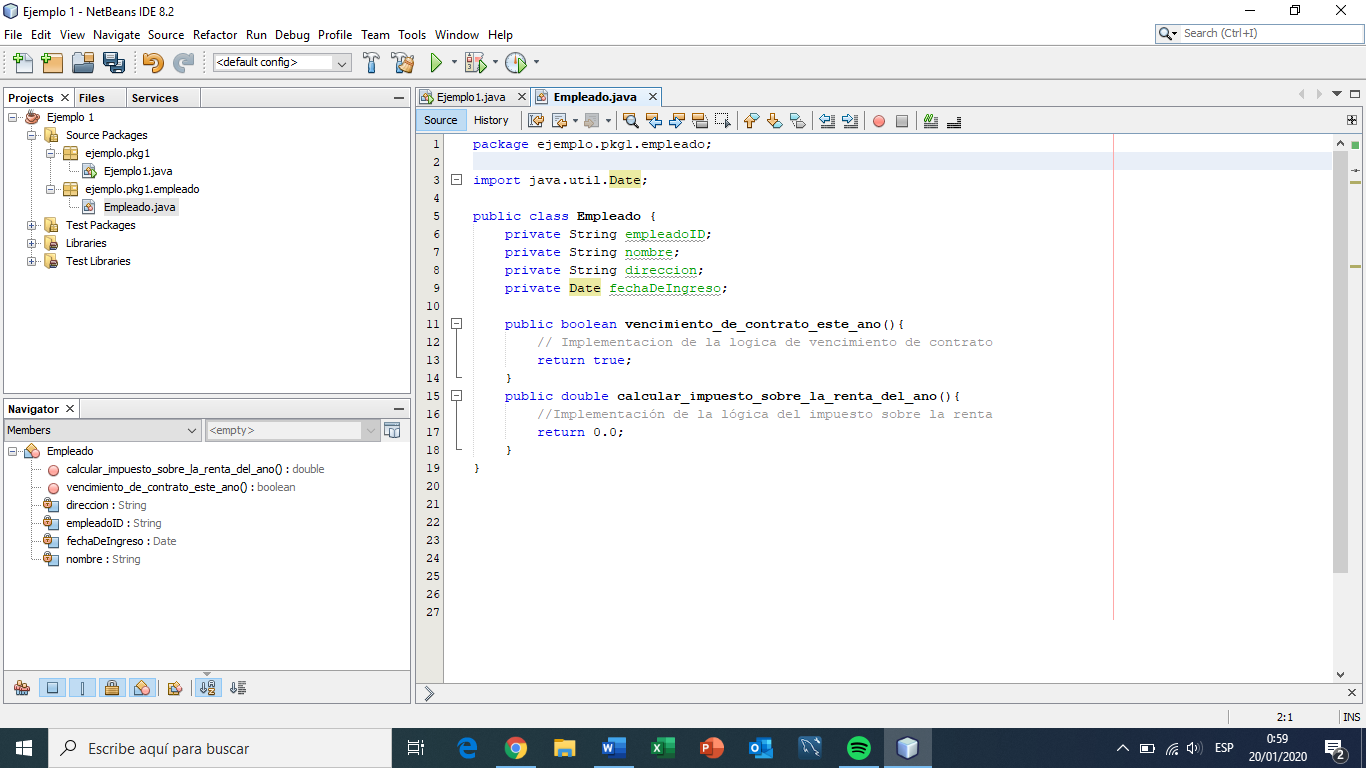
En el artículo “Principles of Object Oriented Design”, Robert C. Martin define una responsabilidad como una 'razón para cambiar', y concluye que una clase o módulo debe tener una, y solo una, razón para cambiar, lo que significa que cada clase debe tener una responsabilidad única, un trabajo único o un propósito único.

Imagine diseñar clases con más de una responsabilidad o implementar más de una funcionalidad. No hay nadie que te detenga para hacer esto. Pero imagine la cantidad de dependencia que su clase puede crear dentro de sí misma en el transcurso del tiempo de desarrollo. Entonces, cuando se le pide que cambie una determinada funcionalidad, no está realmente seguro de cómo afectaría a las otras funcionalidades implementadas en la clase. El cambio puede o no afectar otras características, pero realmente no puede correr riesgos, especialmente en aplicaciones de producción. Entonces terminas probando todas las características dependientes.

Se podría decir que tenemos pruebas automatizadas y que la cantidad de pruebas que se deben verificar es baja, pero imagine el impacto con el tiempo. Este tipo de cambios se acumulan debido a la viscosidad del código que lo hace realmente frágil y rígido.

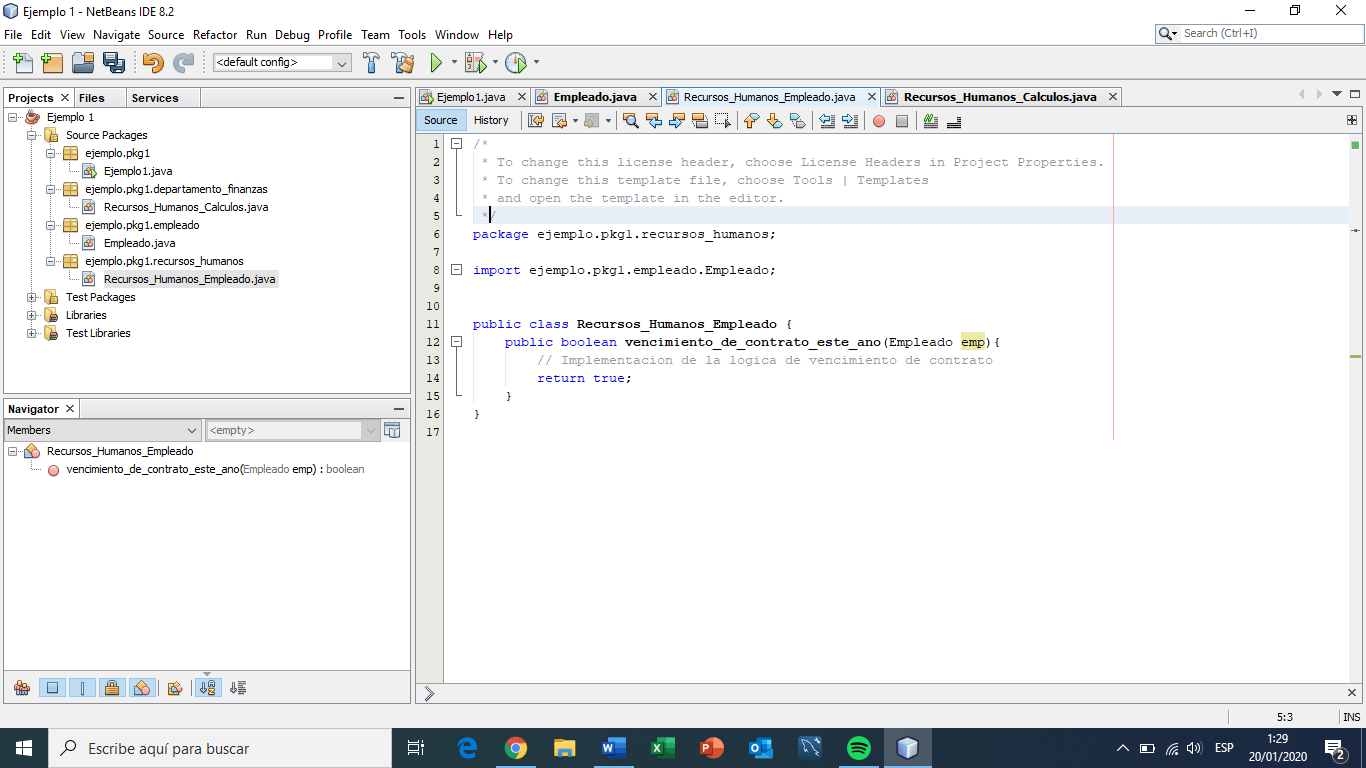
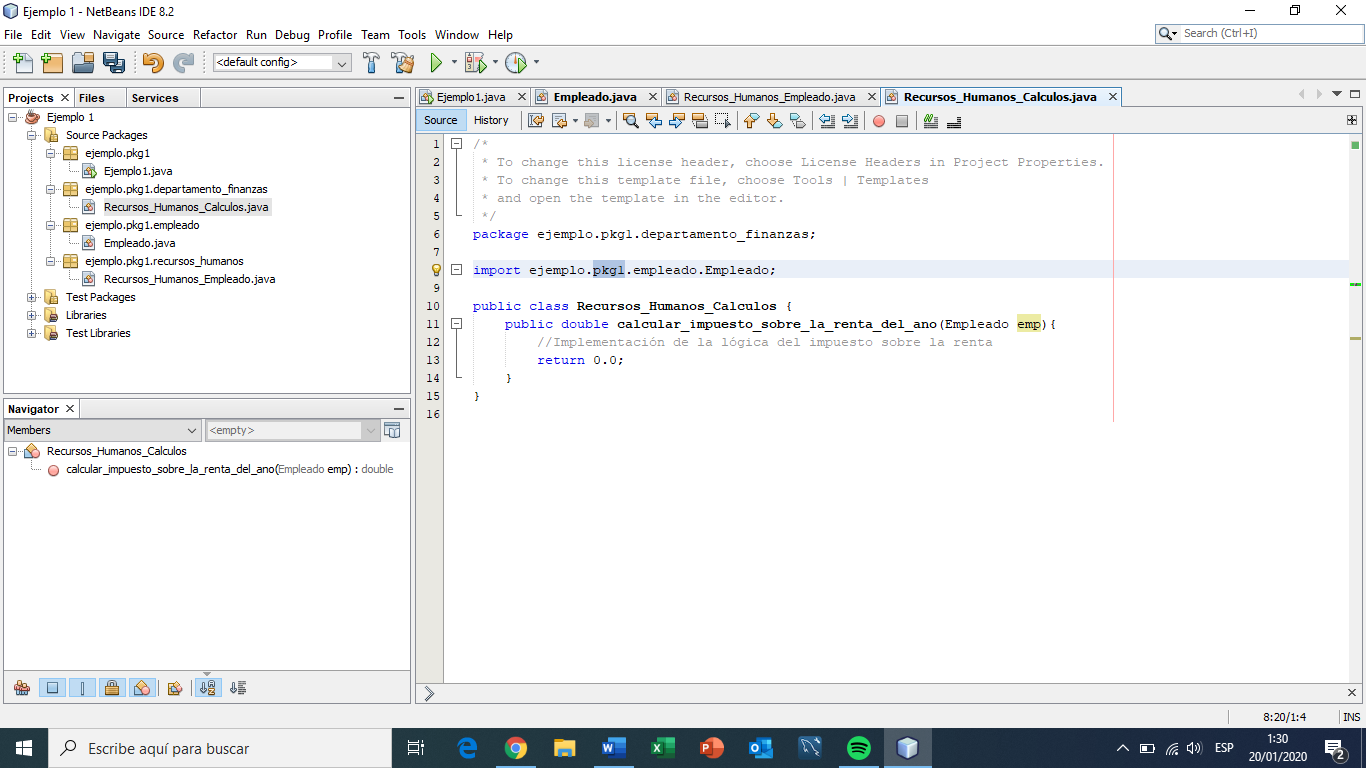
Una forma de corregir la violación de SRP, como también se menciona al principio, es descomponer las funcionalidades de la clase en diferentes clases, cada una de las cuales confirma a SRP.

Ejemplo.

* Tenemos la clase pública “**Empleado**” que contiene ciertas variables privadas (**private**) y, además, contiene dos métodos (una booleana y otra double) donde el primer método (“**vencimiento\_de\_contrato\_este\_ano ( )**”) nos permite saber si el vencimiento del contrato del empleado de la empresa es este año y el otro método (“**calcular\_impuesto\_sobre\_la\_renta\_del\_ano ( )**”) permite calcular el impuesto a la renta del empleado. Por lo tanto, la clase tendría dos responsabilidades o métodos.

**Figura 2.1:** La clase “Empleado” tiene dos responsabilidades.

* Como la clase “Empleado” contiene dos métodos, estaríamos violando el principio. Además, los dos métodos que hemos declarado ninguno puede ser realizado por la clase “Empleado” porque quien se encarga de verificar si el contrato de un empleado ya va a vencer es Recursos Humanos y quien se encargar de calcular el impuesto sobre la renta de un empleado es el departamento de Finanzas. Por lo cual crearemos dichas clases que contendrán dichos métodos.



En conclusión, Si su código se adhiere a este principio, es más fácil de seguir, comprender, depurar, eliminar y refactorizar. Puedes hacer cambios más valientes. Después de algunos años, es posible que desee (necesitará, se verá obligado a) cambiar las cosas, y respetar este principio será una clave vital para cambios más fáciles. Y si rompe algo, rompe una cosa (o menos), no un sistema completo.

**CAPÍTULO 3**

**OPEN / CLOSED PRINCIPLE**

Atribuido a Bertrand Meyer, quien utilizó dicho principio en su obra “Oriented Software Construction”, este principio se resume en que un código y sus entidades, esto incluye sus clases, módulos y funciones; debe estar abierto para la extensión, pero a la vez cerrada para la modificación; mitigando cambios, haciendo que estos sean mínimos, dicho esto, podemos recalcar un poco más cada punto:

- Por abierto, se puede referir por ejemplo al momento de añadir campos a la estructura de datos, en otras palabras, agregar nuevas funcionalidades que requiera el usuario, cualidad muy común en cualquier programa que se realice.

- Por cerrado, hablamos del módulo y su cualidad y necesidad de gozar una descripción estable, a la vez de estar definida claramente, implicando a la vez el concepto de interfaz pública.

Aplicando este principio, lo que se obtendrá como resultado será un acople mucho más relajado, además de mejorar la lectura para poder corregir algún u otro error, que de vez en cuando se puede hacer presente, y finalmente, y quizá una de las ventajas más importantes del Open / closed principle es que se reducirá notoriamente el riesgo de poder romper alguna funcionalidad ya existente en nuestro código fuente, por ello, desemboca así en un usualmente denominado código limpio.

**CAPÍTULO 4**

**LISKOV SUBSTITUTION PRINCIPLE**

Principio atribuido a Barbara Liskov, la noción aquí es que los objetos deberían ser reemplazados por ejemplos de su subtipo, y todo ello sin que la funcionalidad del sistema desde el punto de vista de los usuarios se vea dañada. En otras palabras, podemos describirlo como en vez de utilizar la implementación actual, deberías ser capaz de utilizar una clase base y obtener el resultado esperado. Concluyendo lo expuesto, se puede colegir lo siguiente, que dicho principio nos hace muestra de la gran importancia de poder crear todas las clases derivadas, esto para que también puedan ser tratadas como la propia clase base.

Cumpliendo este principio fundamentalmente se confirmará y nos ayudará a demostrar que nuestro programa tiene una jerarquía de clases fácil de entender y un código reutilizable.

Algo que adicionar a este principio es que muchos especialistas del tema afirman que el principio de Liskov tiene una fuerte relación con el principio anteriormente mencionado, el denominado Open / close principle; incluso Robert c Martin dijo en alguna oportunidad que “una infracción del Liskov substitution principle es una infracción latente del Open / close principle”.

**CAPÍTULO 5**

**INTERFACE SEGREGATION PRINCIPLE Y**

**DEPENDENCY INVERSION PRINCIPLE**

* **INTERFACE SEGREGATION PRINCIPLE**

Este principio fue concebido con la finalidad de mantener un sistema desacoplado de los demás sistemas de los cuales depende, es decir, los clientes de un programa deben conocer y usar a los métodos que necesiten independientemente de aquellos que no son necesarios para ellos.

La interfaz que creemos no debe tener clases con métodos que obliguen agregar funcionalidades extras, sino que las clases que implementen esas interfaces vayan a necesitar y ser capaces de agregar comportamientos a todos los métodos. Para cumplir con ello debemos crear otra interfaz más pequeña y así mantener un sistema desacoplado e independiente.

* **DEPENDENCY INVERSION PRINCIPLE**

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.

El principio de inversión de dependencias nos permite desacoplar los distintos módulos de un programa. Lo que nos dice este principio es que no deben existir dependencias entre los módulos, en especial entre módulos de bajo nivel y de alto nivel.  
Dicho de otro modo nuestro programa no debería depender, por ejemplo, de cómo están implementado los frameworks para acceso a base de datos o conexiones con el servidor.

Para este fin debemos depender de interfaces sin conocer qué sucede exactamente en la implementación de dichas interfaces.

Lo que se desea es que las clases de nuestro código no dependan directamente de las clases de bajo nivel. Debemos usar abstracciones a través de interfaces y solo depender de estas abstracciones. Lo que hace este principio es dividir la dependencia entre los módulos de alto y bajo nivel introduciendo una abstracción entre ellos.

Este principio está relacionado con el concepto de [*"Inyección de Dependencias"*](https://es.wikipedia.org/wiki/Inyecci%C3%B3n_de_dependencias) ya que será otro sistema el que *'inyecte'* en tiempo de ejecución la implementación que requiera la clase en el constructor.

**CONCLUSIONES**

* Cuando desarrollas cualquier software, hay dos conceptos muy importantes: cohesión (cuando dos partes distintas de un sistema trabajan juntas para tener mejor resultado que ambas partes trabajando de manera individual) y acoplamiento (puede verse como un grado de dependencia en una clase, método u otra entidad de software)
* El acople se presenta normalmente en muchos códigos, por lo cual, la situación más óptima sería tener un bajo acoplamiento y alta cohesión. Con esta pequeña introducción a los 5 principios SOLID, has podido ver que nos ayudan a lograrlo.
* Hay muchos principios en la ingeniería de software y yo recomendaría que antes de escribir un código, hay que indagar un poco sobre ello, leer e intentar entender sus principios. Aunque pueda parecer mucho, SOLID se convierte en una parte de ti y tu código utilizándolo con continuidad y adaptándose a sus guías.
* Recuerda que los principios no son reglas y no siempre, en todas las situaciones deben ser aplicados. Te darás cuenta cuando y donde aplicarlas cuando vayas tomando experiencia. Por lo tanto, sería factible que ya empieces a implementarlos en tus desarrollos, te harán un mejor programador y en un futuro te sentirás mejor de todo lo que haz hecho.

**BIBLIOGRAFÍA**

* [*https://www.javacodegeeks.com/2011/11/solid-single-responsibility-principle.html*](https://www.javacodegeeks.com/2011/11/solid-single-responsibility-principle.html)
* [*https://www.javabrahman.com/programming-principles/single-responsibility-principle-with-example-in-java/*](https://www.javabrahman.com/programming-principles/single-responsibility-principle-with-example-in-java/)
* [*https://java2blog.com/solid-principles-java/*](https://java2blog.com/solid-principles-java/)
* [*https://enmilocalfunciona.io/principios-solid/*](https://enmilocalfunciona.io/principios-solid/)
* *https://apiumhub.com/es/tech-blog-barcelona/principios-solid/*
* [*https://uniwebsidad.com/libros/tdd/capitulo-7/principios-solid*](https://uniwebsidad.com/libros/tdd/capitulo-7/principios-solid)
* *https://enmilocalfunciona.io/principios-solid/*
* *https://devexperto.com/principio-de-segregacion-de-interfaces/*
* *https://devexperto.com/principio-de-inversion-de-dependencias/*
* *https://www.mundojava.net/interfaces.html?Pg=java\_inicial\_4\_5.html*
* *https://github.com/alxgcrz/apuntes-principios-solid*
* *https://styde.net/abstraccion-programacion-orientada-a-objetos/*
* *https://medium.com/eduesqui/solid-e79c3f7eb459*