

**CÓDIGO LIMPIO**

**Guía de Trabajo**

**Versión: 1.1**

**Febrero 2023**

CONTENIDO

[1. OBJETIVO 4](#_Toc128058583)

[2. ALCANCE 4](#_Toc128058584)

[3. NOMBRES CON SENTIDO 5](#_Toc128058585)

[3.1 Usar nombres que revelen intenciones 5](#_Toc128058586)

[3.2 Evitar la desinformación 5](#_Toc128058587)

[3.3 Realizar distinciones con sentido 5](#_Toc128058588)

[3.4 Usar nombres que se puedan pronunciar 5](#_Toc128058589)

[3.5 Usar nombres que se puedan buscar 5](#_Toc128058590)

[3.6 Evitar codificaciones 5](#_Toc128058591)

[3.7 Evitar notación húngara 5](#_Toc128058592)

[3.8 No usar prefijos, a menos que sea extremadamente necesario 5](#_Toc128058593)

[3.9 Interfaces e implementaciones 5](#_Toc128058594)

[3.10 Evitar asignaciones mentales 6](#_Toc128058595)

[3.11 Nombres de clases: no debe ser un verbo 6](#_Toc128058596)

[3.12 Nombres de métodos: deben tener nombres de verbo 6](#_Toc128058597)

[3.13 Una palabra por concepto 6](#_Toc128058598)

[3.14 Añadir contexto con sentido y no contextos innecesarios r 6](#_Toc128058599)

[4. FUNCIONES 6](#_Toc128058600)

[4.1 Tamaño reducido 6](#_Toc128058601)

[4.2 Bloques y sangrado 6](#_Toc128058602)

[4.3 Hacer una sola cosa 6](#_Toc128058603)

[4.4 Secciones en funciones 6](#_Toc128058604)

[4.5 Nivel de abstracción por función 6](#_Toc128058605)

[4.6 Leer funciones de arriba hacia abajo 7](#_Toc128058606)

[4.7 Instrucciones Switch 7](#_Toc128058607)

[4.8 Usar nombres descriptivos 7](#_Toc128058608)

[4.9 Argumentos de funciones 7](#_Toc128058609)

[4.9.1 Verbos y palabras claves 7](#_Toc128058610)

[4.9.2 Argumentos de salida 7](#_Toc128058611)

[4.9.3 Manejo de excepciones 7](#_Toc128058612)

[4.10 Separar funciones: una función por cada objetivo 7](#_Toc128058613)

[4.11 Programación Estructurada 7](#_Toc128058614)

[5. COMENTARIOS 8](#_Toc128058615)

[5.1 Introducción 8](#_Toc128058616)

[5.2 Los comentarios indican incapacidad de expresar ideas en el código. 8](#_Toc128058617)

[5.3 Comentarios de calidad – No Balbucear – Evitar la redundancia. 8](#_Toc128058618)

[5.4 Comentarios legales 8](#_Toc128058619)

[5.5 Comentarios informativos - ¿Obligatorios? ¿Periódicos? 8](#_Toc128058620)

[5.6 Advertir sobre consecuencias 8](#_Toc128058621)

[5.7 Amplificación 9](#_Toc128058622)

[5.8 JavaDoc en Apis públicas 9](#_Toc128058623)

[5.9 Autoría de cambios 9](#_Toc128058624)

[5.10 Código comentado 9](#_Toc128058625)

[6. PROCESAR ERRORES 9](#_Toc128058626)

[6.1 Utilizar excepciones en lugar de códigos de error 9](#_Toc128058627)

[6.2 Crear primero la instrucción Try-Catch-Finally 9](#_Toc128058628)

[6.3 Usar excepciones sin comprobar 9](#_Toc128058629)

[6.4 Ofrecer contexto junto a las excepciones 10](#_Toc128058630)

[6.5 Problemas con los “Null” 10](#_Toc128058631)

[7. CONCURRENCIAS: SEPARAR LO QUE SE HACE DE CÓMO SE HACE 10](#_Toc128058632)

[7.1 ¿Por qué concurrencia? 10](#_Toc128058633)

[7.2 Mitos e imprecisiones 10](#_Toc128058634)

[7.3 Desafíos 10](#_Toc128058635)

[7.4 Principios de la concurrencia 11](#_Toc128058636)

[8. PRUEBAS DE UNIDAD 11](#_Toc128058637)

[8.1 Introducción 11](#_Toc128058638)

[8.2 Desarrollo Guiado por Pruebas 11](#_Toc128058639)

[8.3 Pruebas limpias 11](#_Toc128058640)

[8.4 Las pruebas propician posibilidades 11](#_Toc128058641)

[8.5 Lenguaje de pruebas específico del dominio 11](#_Toc128058642)

[8.6 F.I.R.S.T. 12](#_Toc128058643)

[9. CLASES 12](#_Toc128058644)

[9.1 Organización de clases 12](#_Toc128058645)

[9.2 Las clases deben ser de tamaño reducido 12](#_Toc128058646)

[9.3 Principio de responsabilidad única 13](#_Toc128058647)

[9.4 Cohesión 13](#_Toc128058648)

[9.5 Mantener resultados consistentes en muchas clases de tamaño reducido. 13](#_Toc128058649)

[9.6 Organizar los cambios 13](#_Toc128058650)

[9.7 Aislar el código de los cambios. 13](#_Toc128058651)

[10. PRINCIPIOS DE PROGRAMACIÓN 13](#_Toc128058652)

[10.1 El principio KISS 14](#_Toc128058653)

[10.2 El principio YAGNI 14](#_Toc128058654)

[10.3 El principio DRY 14](#_Toc128058655)

[10.4 El principio de separación de conceptos 15](#_Toc128058656)

[10.5 El principio de Least Astonishment 15](#_Toc128058657)

[10.6 Principio de la Ley de Demeter 15](#_Toc128058658)

[10.7 Principios SOLID 15](#_Toc128058659)

[11. VERSIÓN DE LA GUÍA 17](#_Toc128058660)

[12. HISTORIAL DE VERSIONES DE LA GUÍA 17](#_Toc128058661)

[13. VERSIÓN DE TEMPLATE BASE 17](#_Toc128058662)

# OBJETIVO

Definir pautas claras y concisas para facilitar la construcción de software e incrementar la legibilidad del código existente cuando se trate de un proceso de mantenimiento de aplicaciones.

# ALCANCE

Aplica a todos los proyectos de desarrollo de software de la empresa, utilizable en cualquier plataforma de desarrollo, y con cualquier tipo de arquitectura de sistemas.

# NOMBRES CON SENTIDO

Dar nombres con sentido implica que las entidades y relaciones que se crean / mantienen durante un proyecto de desarrollo de software deben cumplimentar pautas que faciliten la comprensión de lo que desea alcanzar con dicho elemento. No existe el nombre “perfecto” porque cada construcción tiene el toque de la personalidad del constructor, pero seguir ciertas pautas produce que se puede facilitar su mantenimiento.

## Usar nombres que revelen intenciones

En lugar de nombres indescifrables, utilizar uno que de pauta de su objetivo.

## Evitar la desinformación

Relacionado con el punto anterior, la intención del elemento se devela con el nombre.

## Realizar distinciones con sentido

Utilizar la misma variable para todas las funciones semejantes.

## Usar nombres que se puedan pronunciar

Evitar nombres ininteligibles, por ejemplo, cuando se eliminan las vocales. Tener un Código Limpio implica poder leer un desarrollo.

## Usar nombres que se puedan buscar

Evitar nombres que puedan asemejarse a constantes, ya que eso dificulta la búsqueda.

## Evitar codificaciones

Nombrar elementos de forma codificada produce que haya que realizar interpretaciones innecesarias.

## Evitar notación húngara

En los ámbitos modernos de desarrollo, no existe la limitación en cuanto a los nombres de elementos. Por lo tanto, se debe evitar este tipo de notación.

## No usar prefijos, a menos que sea extremadamente necesario

El creador del código hace un esfuerzo mental para poner prefijos en sus nombres, y quien debe mantener ese desarrollo, otro esfuerzo en descifrar a qué corresponde el prefijo. Evitar ambas situaciones; se facilita la comprensión del desarrollo.

## Interfaces e implementaciones

No es necesario develar si el desarrollo tiene interfaces. El desarrollador sabe a qué corresponde la llamada a un proceso específico dentro de la aplicación.

## Evitar asignaciones mentales

Relacionado con el punto 3.8. Si un desarrollador tiene que interpretar (por ejemplo un prefijo) puede que no sea la interpretación deseada. Evitar es el mejor camino.

## Nombres de clases: no debe ser un verbo

Una clase se refiere a una colección de elementos relacionados entre sí, no a una acción específica. Evitar los verbos es una buena práctica.

## Nombres de métodos: deben tener nombres de verbo

Un método es una acción que lleva a un resultado, por tal motivo, como el nombre debe revelar la intención, será un verbo parte del nombre del método.

## Una palabra por concepto

Elegir una palabra única por cada concepto abstracto que maneje la aplicación, evita confusiones e interpretaciones innecesarias.

## Añadir contexto con sentido y no contextos innecesarios r

Muchas veces, tener nombres con sentido no alcanza para develar el objetivo de cada elemento nombrado. Por eso, el contexto bien nombrado también es revelador del sentido que tiene cada elemento del desarrollo.

# FUNCIONES

Atributos que se deben tener presentes para que el lector de un desarrollo pueda intuir el tipo de programa a que pertenece.

## Tamaño reducido

Funciones extensas resultan ilegibles para el lector del desarrollo.

## Bloques y sangrado

Los bloques IF/THEN/ELSE por ejemplo, al estar dentro del mismo sangrado, reducen significativamente el tiempo empleado en entender una función.

## Hacer una sola cosa

Las funciones deben hacer una sola cosa. Exclusivamente. Y bien hechas.

## Secciones en funciones

Al ser cada función de uso exclusivo, se deben evitar las secciones en las funciones: agrandan el código innecesariamente.

## Nivel de abstracción por función

Para que las funciones realicen «una cosa», asegurar que las instrucciones de la

función se encuentren en el mismo nivel de abstracción.

## Leer funciones de arriba hacia abajo

Al conseguir que el código se lea de arriba a abajo, se mantiene la coherencia de los niveles de abstracción.

## Instrucciones Switch

Las instrucciones switch siempre hacen “N” cosas, por lo que se deben evitar, o reducir al máximo su uso, por tratar de reducir el código al mínimo.

## Usar nombres descriptivos

Nombre que describa el uso de la función.

## Argumentos de funciones

El número ideal de argumentos (o parámetros) de funciones es cero. Aunque existen excepciones. Después uno (monádico) y dos (diádico). Siempre que sea posible, evitar la presencia de tres argumentos (triádico). Más de tres argumentos (poliádico) requiere una justificación especial y no es muy habitual.

## Verbos y palabras claves

La selección de nombres correctos para una función mejora la explicación de su cometido, así como el orden y el cometido de los argumentos.

## Argumentos de salida

Por lo general, habrá que evitarlos. Si la función tiene que cambiar el estado de un elemento, es mejor hacer que cambie el estado de su objeto contenedor

## Manejo de excepciones

Es preferible el manejo de excepciones que el procesamiento de errores.

## Separar funciones: una función por cada objetivo

Las funciones deben hacer algo o responder algo, pero no ambas cosas.

## Programación Estructurada

Esta regla implica utilizar una sola entrada y una sola salida por cada función (sin breaks, ni goto’s ni sentencias similares que impliquen una salida errática del código). Pero al utilizar funciones sencillas y abreviadas, esta regla resulta anacrónica. Se debe evaluar su uso en una función reducida, y evitar definitivamente su uso en una función extensa.

# COMENTARIOS

## Introducción

No hay nada más útil que un comentario bien colocado. No hay nada que colapse más un módulo que comentarios dogmáticos innecesarios. No hay nada más dañino que un comentario antiguo que propague mentiras y desinformación.

## Los comentarios indican incapacidad de expresar ideas en el código.

Si los lenguajes de programación fueran más expresivos o si se pudieran dominar para expresar las intenciones del desarrollador, no se necesitarían demasiados comentarios, puede que incluso ninguno.

## Comentarios de calidad – No Balbucear – Evitar la redundancia.

El código debe ser auto-descriptivo. Si no lo es, el comentario debe ser breve y conciso como para aportar lo que el nombre no dice. Por ende, no debe tener la debilidad de no poder interpretarse a primera vista, así como tampoco tendría que encontrarse en el código redundancia de explicación: si la función se entiende, por su sencillez y breve extensión, el comentario pasa a ser redundante.

## Comentarios legales

Comentarios como declaración de derechos de autor (Copyright) se deben incluir en el código, aun si se está trabajando con desarrollos de código abierto. Esta premisa debe estar acotada a lo indicado en el punto anterior.

## Comentarios informativos - ¿Obligatorios? ¿Periódicos?

Como se define previamente, el comentario debe ser breve y conciso cuando la función no se auto-describe. Cuando no es posible desarrollar una función breve, se deberán informar en los bloques del código qué se pretende en dicho bloque. Si se desarrollan funciones breves y auto descriptivas, el comentario es innecesario, por lo tanto, no hay obligatoriedad de colocarlo. En algunos desarrollos (cuentas / clientes), se acostumbraba agregar un comentario con el autor del cambio en el código. Con las herramientas de control de edición y versionado que se cuentan en la actualidad, esos comentarios periódicos (y casi obligatorios en muchos casos) pasa a ser obsoleto. Si se encuentran, deben ser eliminados para tener código limpio.

## Advertir sobre consecuencias

En ciertas ocasiones, se agrega al código un pequeño módulo utilizado en el testeo. Por ejemplo, un bloque de grabación de movimientos en una tabla para hacer un chequeo de estrés de la función. El buen desarrollador indica que dicho módulo dentro de la función tiene un uso determinado, y advertirá que, si se lo invoca, la ejecución de dicho código implicará la tardanza de x tiempo.

## Amplificación

Cuando una parte del código tiene importancia, pero por su brevedad no resalta del resto del bloque, el comentario debe hacer la función de amplificación de dicho bloque.

## JavaDoc en Apis públicas

Si se publican Apis para su uso público, se debe comentar escuetamente contenido de dicha función.

## Autoría de cambios

Antiguamente, se exigía declarar en un comentario el desarrollador que había realizado un cambio en el código. Las herramientas actuales de control de versiones almacenan también el usuario que realiza el cambio, y registran todas las secuencias que se modifican de un bloque de código, por lo que se considera redundante registrar en un comentario el autor del cambio.

## Código comentado

Bajo ninguna circunstancia el código obsoleto debe quedar escrito en el código productivo. Si una parte del código está comentada, es porque esa parte no se utiliza más, y dejarla implica un esfuerzo innecesario en tratar de interpretar por qué se dejó dicha parte en el código fuente. Cuando se la encuentra, hay que eliminarla.

# PROCESAR ERRORES

El control de errores es importante, pero si oscurece la lógica es incorrecto.

## Utilizar excepciones en lugar de códigos de error

Los lenguajes de programación modernos manejan excepciones, pero antiguamente no lo hacían, por eso existe el uso y costumbre de manejar códigos de error. El “código limpio” tendrá como premisa la utilización de excepciones.

## Crear primero la instrucción Try-Catch-Finally

Las excepciones definen un ámbito en el programa. Al ejecutar la parte Try de esta instrucción, la misma se puede cancelar en cualquier momento y retomar en la parte Catch. Intentar crear pruebas que fuercen las excepciones, para después añadir al controlador un comportamiento que satisfaga dichas pruebas. De este modo crear el ámbito de transacción del bloque try y poder mantener la naturaleza de transacción del ámbito. La parte Catch nunca se debe dejar sin realizar una función.

## Usar excepciones sin comprobar

Las excepciones comprobadas pueden ser útiles si tiene que crear una biblioteca crítica: hay que capturarlas. Pero en el desarrollo de aplicaciones generales, los costes de dependencia superan las ventajas.

## Ofrecer contexto junto a las excepciones

Las excepciones que se generan deben proporcionar el contexto adecuado para determinar el origen y la ubicación del error.

## Problemas con los “Null”

No devolver Null. Esto puede provocar perder el control de la aplicación si el “null” no está controlado en todos los bloques donde se utilice una función llamada. No pasar como parámetro un valor “null”: hay que evitarlos siempre que sea posible y verificar en el código que se chequee el valor del parámetro para evitar problemas y que los errores sean menores.

# CONCURRENCIAS: SEPARAR LO QUE SE HACE DE CÓMO SE HACE

## ¿Por qué concurrencia?

La concurrencia es una estrategia de desvinculación, que permite diferenciar lo que se hace de dónde se hace. Esto provoca la mejora sustancial del rendimiento y la estructura de una aplicación. Imaginar una fila de personas esperando que los atiendan en una sola caja. Ahora, la misma cantidad de personas, pero con 4 cajas habilitadas. ¿Y con 8 cajas? ¿Y con “n” cajas? Claramente hay limitaciones (no puede haber infinitas cajas, ni espacio suficiente en el local como para atender a infinitas personas) entonces, el código limpio deberá tener en cuenta el equilibrio entre lo posible y lo necesario.

## Mitos e imprecisiones

La concurrencia siempre mejora el rendimiento: en ocasiones lo hace, pero hay que evaluar todas las situaciones posibles.

El diseño no cambia al crear programas concurrentes: Si, de hecho, puede ser muy distinto.

Se deben entender los problemas de concurrencia al trabajar con contenedores WEB o EJB.

La concurrencia genera cierta sobrecarga: es una afirmación correcta.

Se debe tener en cuenta que administrar la concurrencia es algo complejo, que pueden ocurrir errores (que generalmente se los ignora, en lugar de considerarse verdaderos problemas).

## Desafíos

Cuando la concurrencia de procesos pone en riesgo la integridad de un sistema, habrá que tener en cuenta todos y cada uno de los riesgos, para evaluar cómo pueden evitarse los riesgos y manejar los errores como se describe en el capítulo anterior. Tener presente el caso de la numeración de comprobantes, por ejemplo.

## Principios de la concurrencia

Principio de responsabilidad única: un método, clase o componente solo debe tener un motivo para cambiar.

El código relacionado con la concurrencia tiene su propio ciclo de desarrollo.

El código relacionado con la concurrencia tiene sus propios desafíos, diferentes y más complicados que los códigos no relacionados con la concurrencia.

Por esto, se debe separar el código de concurrencia del resto del código.

# PRUEBAS DE UNIDAD

## Introducción

Las pruebas de unidad, o pruebas unitarias, deben efectuarse antes de la implementación del código productivo. Suena como algo que todo el mundo sabe, pero no todos lo ponen en práctica. Es imperativo para complementar la creación de código limpio, que se realicen todos los pasos estipulados en esta guía de trabajo.

## Desarrollo Guiado por Pruebas

Primera ley: No debe crear código de producción hasta que haya creado una prueba

de unidad que falle.

Segunda ley: No debe crear más de una prueba de unidad que baste como fallida y

no compilar se considera un fallo.

Tercera ley: No debe crear más código de producción que el necesario para superar

la prueba de fallo actual.

## Pruebas limpias

Los entornos de test (y peor aún, los de desarrollo) siempre contienen datos inconsistentes. Tener lotes de prueba (también se los denomina Casos de Uso) lo más parecido (sino los mismos) que los datos de producción ayudan a que las pruebas sean consistentes con la realidad del proyecto. Lo que define que una prueba sea limpia también es la legibilidad.

## Las pruebas propician posibilidades

Si las pruebas no son limpias, se pierden. Las pruebas de unidad son las que permiten que el código sea flexible, que se pueda mantener y reutilizar. Sin pruebas, cada cambio es potencialmente un error.

## Lenguaje de pruebas específico del dominio

Son funciones y utilidades que se convierten en una API especializada usada por las pruebas. Son un lenguaje de pruebas que los desarrolladores utilizan para crear sus pruebas, y ayudar a quien lee las mismas con posterioridad. Esta API evoluciona con la refactorización continuada del código de prueba.

## F.I.R.S.T.

Las pruebas limpias siguen otras cinco reglas, cuyas iniciales forman las siglas FIRST en inglés:

Rapidez (Fast): Las reglas deben ser rápidas y ejecutarse de forma rápida. Si lo hacen lentamente, no las ejecutará con frecuencia. Al no hacerlo, no detectará los problemas con la suficiente antelación como para solucionarlos. No se sentirá con libertad para limpiar el código, que acabará corrompiéndose.

* **Independencia** (Independent): Las pruebas no deben depender entre ellas. Una prueba no debe establecer condiciones para la siguiente. Debe poder ejecutar cada prueba de forma independiente y en el orden que desee. Si las pruebas dependen unas de otras, la primera que falle provocará una sucesión de fallos, dificultará el diagnóstico y ocultará efectos posteriores.
* **Repetición** (Repeatable): Las pruebas deben pode repetirse en cualquier entorno. Debe poder ejecutarlas en el entorno de producción, en el de calidad y en su portátil de camino a casa en un tren sin red. Si no puede repetir las pruebas en cualquier entorno, siempre tendrá una excusa de su fallo. También verá que no puede ejecutar las pruebas si el entorno no está disponible.
* **Validación** automática (Self-Validating): Las pruebas deben tener un resultado booleano: o aciertan o fallan. No debe tener que leer un extenso archivo de registro para saber si una prueba ha acertado, ni comparar manualmente dos archivos de texto distintos para ello. Si las pruebas no se validan automáticamente, el fallo puede ser subjetivo y la ejecución de las pruebas puede requerir una extensa evaluación manual.
* **Puntualidad** (Timely): Las pruebas deben crearse en el momento preciso: antes del código de producción que hace que acierten. Si crea las pruebas después del código de producción, puede que resulte difícil probarlo. Puede decidir qué parte del código de producción sea demasiado difícil de probar. No diseñe código de producción que no se pueda probar

# CLASES

## Organización de clases

De acuerdo a la convención estándar de Java, una clase debe comenzar con una lista de variables, con el siguiente orden:

* Constantes estáticas públicas (si existen)
* Variables estáticas privadas
* Variables de instancia privada

## Las clases deben ser de tamaño reducido

Como se explicó en capítulos anteriores: reducción de tamaño implica facilidad de lectura y rapidez en la comprensión del objetivo.

## Principio de responsabilidad única

Una clase o módulo debe tener uno y solo un objeto a cambiar. No tener una clase donde se actualice un comprobante y un cliente, por ejemplo. Lo deseable es una clase para la actualización del comprobante, y otra clase para la actualización del cliente. Más fácil de probar, más fácil de entender.

## Cohesión

Las clases deben tener un número reducido de variables de instancia. Los métodos de una clase deben manipular una o varias de dichas variables. Por eso, cuantas más variables manipule un método, más cohesión tendrá con su clase.

## Mantener resultados consistentes en muchas clases de tamaño reducido.

Dividir una gran función en otras más reducidas también permite dividir varias clases más reducidas.

## Organizar los cambios

En muchos sistemas, el cambio es continuo. Cada cambio supone un riesgo que el resto del sistema no funciones de la forma esperada. En un sistema limpio se organizarán las clases para reducir los riesgos de los cambios. Se deben estructurar los sistemas para “ensuciarlos” lo menos posible cuando se actualicen con nuevas funciones o cambios.

## Aislar el código de los cambios.

Las necesidades cambian, y también el código. En la programación orientada a objetos, hay clases concretas que contienen detalles de implementación (el código) y clases abstractas que solo representan conceptos. Las dependencias de detalles de concretos crean retos para los sistemas.

Si se disecciona un sistema para poder probarlo de esta forma, resultará más flexible y se podrá reutilizar. La ausencia de conexiones significa que los elementos del sistema se aíslan entre ellos y de otros cambios. Este aislamiento hace que se comprendan mejor los elementos del sistema. Al minimizar las conexiones de esta forma, las clases cumplen otro principio de diseño: Principio de inversión de dependencias. Básicamente afirma que nuestras clases deben depender de abstracciones, no de detalles concretos.

# PRINCIPIOS DE PROGRAMACIÓN

Los principios aquí descriptos no se deben cumplir al 100% o ciegamente cómo fanáticos. Siempre se debe valorar y buscar el equilibrio al hacer la implementación. Se debe buscar un balance entre los atributos de calidad del software: mantenibilidad, funcionalidad, rendimiento y fiabilidad.

## El principio KISS

Este principio sugiere que para que un sistema funcione mejor se debe mantener simple, no se deben aplicar patrones de diseño que complejicen la solución si no es necesario. Esto significa que no debemos agregar clases e interfaces solo por agregarlas, por si a futuro se debe hacer algo. En cambio, se debe crear solo lo necesario para que la solución funcione de forma óptima. Una solución sencilla, es cuando es fácil de explicar y de entender.

Para lograr la simplicidad se debe:

* Preferir la composición en lugar de la herencia, siempre que sea posible. Usar polimorfismo en lugar de muchos if-else o grandes switch-case.
* Evitar la sobre optimización. Muchas veces no es perceptible por el usuario. Optimice cuando vea que hay un problema.
* Clases y métodos pequeños son mejores.
* Tenga métodos que solo resuelvan un problema específico.
* Siempre refactorice buscando reducir su código.

En resumen, se busca hacer la implementación más sencilla evitando la complejidad innecesaria. No agregar abstracciones que no son fundamentales, que no surgen del negocio o que no aportan a la solución del requerimiento.

## El principio YAGNI

Este principio sugiere que no se deben implementar funcionalidades que no son necesarias o que no se requirieron. No se debe anticipar a las funcionalidades que Ud. cree que se van a requerir. Esto provoca sistemas más complejos y difíciles de entender, además de todo el esfuerzo extra en hs de diseño, programación y testing. Siempre se debe implementar solo aquello que se pide, nada más.

Cuando vaya a implementar una función, hágase esta pregunta:

* ¿Esta función o funcionalidad es necesaria ahora?
* Si la respuesta es si, entonces impleméntela. Si la respuesta es no, hágase esta pregunta:
* ¿Cuánto esfuerzo me llevará implementar esta funcionalidad, si es requerida en el futuro?

Si la respuesta es: será sencillo de implementar, entonces deje todo cómo está y no agregue nada adicional. Si la respuesta es: Me va a llevar mucho esfuerzo, muchas hs, para implementar y reescribir código. Entonces en ese caso hago unos refactors que eviten a futuro tanto esfuerzo si es que se llega a necesitar agregar esa funcionalidad.

En resumen, lo que se busca es no tener código innecesario por una funcionalidad que no esté orientada a la problemática actual.

## El principio DRY

No debe tener código duplicado, esto solo genera bugs y mucho retrabajo. Cada entidad de la realidad, debe tener una única representación en el sistema. Evite duplicar lógica en distintos lugares. Evite repetir múltiples if o siwtch-cases, use abstracciones para sustituirlos. Evite hardcodear valores en el código, use enums o declare variables constantes que expresen que significa ese valor, para que al leer el código se entienda que significa.

## El principio de separación de conceptos

Este principio busca que se divida en capas, donde cada capa represente un concepto y cada uno de estar aislado del otro, sin que uno de ellos conozca los detalles del otro. Cada concepto debe tener su módulo. Por ejemplo

* UI
* Capa de Negocio
* Capa de presentación
* Capa de datos

En resumen, promueve la creación de sistemas modulares o en capas.

## El principio de Least Astonishment

Este principio hace énfasis en que cada componente de un sistema debe comportarse de una manera consistente y esperada para los clientes de ese componente. Por ejemplo, la inmutabilidad está basada en este principio. Lo que significa que si una funcionalidad tiene un comportamiento en alto grado tendiente a ser diferente cada vez que se llama, entonces es muy probable que a futuro se deba modificar.

## Principio de la Ley de Demeter

Esta ley indica como un método f de una clase C debe comunicarse con otros métodos. De tal forma, dicho método f debe invocar métodos de:

* C
* Un objeto creado por f
* Un objeto pasado como argumento a f.
* Un objeto en una variable de instancia de C

Este principio expresa que una instancia de una clase no debería conocer los detalles internos de otra instancia de clase con la que interactúa. Permite detectar el acoplamiento entre clases. Evitar usar cadenas de llamadas de métodos.

## Principios SOLID

Estos principios ayudan a tomar buenas decisiones al desarrollar. Pero cómo no debemos ser puritanos y hacerlos cumplir estrictamente, sino que siempre se debe buscar un balance. Muchos de estos principios están relacionados unos con otros. Dichos principios son:

1. **(S) - Principio de Responsabilidad única.**

Plantea que una clase o método o paquete o namespace debe tener un solo objetivo, es decir una sola razón para cambiar o ser modificado.

1. **(O) - Principio Abierto/Cerrado**

Dice que los objetos o entidades o módulos o métodos deben estar abiertos por extensión, pero cerrados para modificación. Es decir, se busca tener un sistema donde se pueda de forma fácil agregar nuevas funcionalidades sin tener que modificar, recompilar y redeployar los componentes core o importantes. En resumen debe permitirse que se amplíen las funcionalidades sin tener que modificar lo existente.

1. **(L) - Principio de sustitución de Liskov**

Manifiesta que, teniendo una clase abstracta o interfaz, todas las clases derivadas o que implementen la interfaz, debe poder ser sustituidas por sus pares en el mismo contexto. Es decir, podría usar los subtipos de forma indistinta reemplazando unos por otros y esto no afectaría el código existente.

1. **(I) - Principio de segregación de interfaz**

Expresa que ningún cliente debe ser forzado a implementar métodos que no usa. Es decir, cada clase debe tener un propósito bien definido y solo debe exponer comportamiento alineado a ese propósito. El cliente o consumidor de ese comportamiento no debe ser forzado a depender de métodos que no requiere. Por lo que cuando se define la interfaz esta debe ser lo más atómica posible según el contexto lo requiera.

1. **(D) - Principio de inversión de dependencias**

Plantea que un módulo de alto nivel no debe depender de un módulo de bajo nivel. Sino que ambos deben depender de una abstracción. La abstracción no debe depender de detalles. Los detalles deben depender de las abstracciones. En otras palabras, se debe buscar reducir el acoplamiento entre componentes al usar abstracciones(interfaces) entre ellos. Las clases no deben depender directamente unas de otras, sino que debemos pasarlas cómo parámetro en el constructor (inyectarlas). Por lo tanto será más fácil mantenerlas en caso de que necesitemos modificar su funcionamiento. pero mejor aún, si en lugar de que nuestro método reciba la clase B recibe una abstracción superior, como una clase abstracta o una interfaz, pues estamos desacoplando aún más nuestro código y serás capaz de intercambiar objetos externos a esta clase sin que por ello se afecte el funcionamiento de la misma, en este momento tendrás bajo acoplamiento y una mejor estructura en tu proyecto.

# VERSIÓN DE LA GUÍA

|  |  |  |  |
| --- | --- | --- | --- |
| Fecha | Versión | Nombre de la guía | Ubicación |
| 23-02-2023 | 1.1 | DES-CNS-GT-Código Limpio | SGC |

# HISTORIAL DE VERSIONES DE LA GUÍA

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Fecha | Versión | Descripción | Realizó | Controló | Aprobó |
| 06-12-2022 | 1.0 | Primera versión | Leonardo Mierez | Sebastián Filippelli | Pablo Behrend |
| 23-02-2023 | 1.1 | Se agregar la sección principios de programación | Jorge Martínez | Pablo Behrend | Pablo Behrend |

# VERSIÓN DE TEMPLATE BASE

|  |  |  |  |
| --- | --- | --- | --- |
| Fecha | Versión | Nombre del template | Ubicación |
| 10-11-2022 | 1.1 | GQA-CID-TP-Template Base Word-Principales | SGC |

**[](https://www.linkedin.com/company/it-resources---soluciones-informaticas)[](https://www.facebook.com/itresources/)[](https://www.instagram.com/itr.ar/)**

**Seguinos**

**Contacto**

+54 9 11 5218-4444

calidad@itrsa.com.ar