Directrices de implementación enfocadas en la calidad.

Dirección General de Sistemas de Información y Comunicaciones

Áreas de Gobierno Tecnológico de SSII y del Dato

Servicio Andaluz de Salud
Consejería de Sanidad, Presidencia y Emergencias

Versión

Guía de implementación Vigente

Versión: **1.0.1**

Estado: ACTIVO

Entrada en vigor desde: 11/07/2024

Guía de implementación PRE-RELEASE

Versión: -

Estado: -

Entrada en vigor desde: N/A

Tabla de Contenido

Cumplimiento Normativo

Las normas expuestas son de obligado cumplimiento. La STIC podrá estudiar los casos excepcionales los cuales serán gestionados a través de los responsables del proyecto correspondiente y autorizados por el Área de Gobernanza de la STIC. Asimismo cualquier aspecto no recogido en estas normas deberá regirse en primera instancia por las guías técnicas correspondientes al esquema nacional de seguridad y esquema nacional de interoperabilidad según correspondencia y en su defecto a los marcos normativos y de desarrollo software establecidos por la Junta de Andalucía, debiendo ser puesto de manifiesto ante la STIC.

La STIC se reserva el derecho a la modificación de la norma sin previo aviso, tras lo cual, notificará del cambio a los actores implicados para su adopción inmediata según la planificación de cada proyecto.

En el caso de que algún actor considere conveniente y/o necesario el incumplimiento de alguna de las normas y /o recomendaciones, deberá aportar previamente la correspondiente justificación fehaciente documentada de la solución alternativa propuesta, así como toda aquella documentación que le sea requerida por la STIC para proceder a su validación técnica.

Contacto Arquitectura: l-arquitectura.stic@juntadeandalucia.es

Histórico de cambios

Los cambios en la normativa vendrán acompañados de un registro de las modificaciones. De este modo se podrá realizar un seguimiento y consultar su evolución. Ordenándose de mas recientes a menos recientes, prestando especial cuidado a las cabeceras de la tablas dónde se indican las fechas de entrada en vigor y versión.

Versión	Pre-release	Adopción	Activa	Retiro	Alcance

v01r00		11 jul 2024	11 jul 2024		• Versión inicial.
					  ◦ SOLID
					  ◦ Pautas Clean Code
					  ◦ Dry
					  ◦ Kiss
					  ◦ Yagni

1.2. Ciclo de vida de la arquitectura de referencia

Los cambios normativos dentro de la arquitectura de referencia seguirán el siguiente ciclo de vida:

1. Introducción

Esta guía busca facilitar el desarrollo de productos dentro del

Se proporciona una serie de pautas para el diseño y el desarrollo que facilitan la mantenibilidad, la legibilidad y el cambio del código.

Objetivos

Objetivo principal de esta guía es que el código sea:

- **Mantenible.**
- **Legible**
- **Entendible**
- **Testeable**
- **Extensible**
- **Resiliente**

Estas guías tratan de ser agnósticas al lenguaje de desarrollo por lo que se proporcionarán acceso a ejemplos de implementación en cada lenguaje.

1.1. Ámbito de aplicación

Ámbito de aplicación obligatorio (*)

Esta arquitectura de referencia será de aplicación obligatoria(*) en los siguientes casos:

- Desarrollo de nuevos sistemas de información corporativos.
- Refactorización de sistemas de información existentes.
- Evolución de sistemas de información que requieran la implementación de nuevos procesos o funcionalidades y que requiera de desarrollo en funciones "core" del sistema.

Se recomienda su aplicación:

- **Pre-release:** Periodo durante el cual la normativa se hace publica aunque no es necesario adherirse inmediatamente. En este periodo aún puede sufrir pequeñas correcciones.
- **Adopción :** Periodo durante el cual la normativa inicia un periodo de tiempo flexible para su aplicación de forma obligatoria al ámbito en el cual está destinado.
- **Activa:** Periodo durante el cual es obligatorio su cumplimiento para el ámbito en el cual está destinado.
- **Retiro:** Periodo durante el cual dejará de aplicarse en el ámbito al cual está destinado en sustitución de nuevas versiones o normas.

(*) Cualquier propuesta que difiera de esta arquitectura deberá ser aprobada por el Área de Gobernanza de la STIC, previa solicitud y justificación en su caso.

- Sistemas comerciales que permitan una arquitectura tecnológica flexible y alineada con este modelo de referencia.
- Desarrollos locales con previsión de escalar a sistemas corporativos.
- Cualquier desarrollo para implementar nuevos procesos sobre los sistemas de información existentes.

(**) En proceso de elaboración de una arquitectura de referencia para sistemas analíticos y big data.

Se recomienda analizar en detalle con la unidad de Arquitectura de la STIC y particularizarla para los siguientes casos:

- Sistemas analíticos o que gestionen grandes volúmenes de datos (**).

2. Clean Code

Definición

Estas pautas no son leyes o reglas inalterables que se tengan que cumplir, pero si son una serie de recomendaciones prácticas que deben ser aplicadas en su conjunto para obtener los beneficios esperados.

Objetivo

Mantener el código **legible, mantenible y libre de errores.**

Validación

Todas las pautas podrán ser validadas a través de revisiones de código y de arquitectura del proyecto

Ámbito: Naming			
Pauta	**Descripción**	**No recomendable**	**Recomendable**
P1	Hacer uso de las convenciones de cada lenguaje		
P2	Hacer uso de los namespace/packages antes que los prefijos	`class Ejemplo{ private m_attr1; }`	`package m; class Ejemplo{ private attr1; }`
P3	El nombre debe revelar las intenciones de la variable, método. clase, módulo	`class A{ private d; }`	`class Ejemplo{ private elapsedTimeInDays; private daysSinceCreation; }`
P4	El nombre debe ser pronunciable	`package m; class Ejemplo{ private genyymmddhhmm; }`	`package m; class Ejemplo { private generateTimeStamp; }`
P5	Se deben usar las palabras necesarias. Hacer uso del namespace y la clase para completar la semántica	`public void saveUserIntoMongoDBDatabase (User user);`	`package database; public void saveUser (User user);`

P6	Utilizar la misma palabra para el mismo concepto		
P7	Las palabras que se utilicen deben ser los más precisas posible para evitar ambigüedades		
P8	El lenguaje utilizado debe ser ubicuo (DDD)		
P9	Usar verbos para la definición de métodos y nombres para la definición de clases y atributos		
P10	Utilizar la misma palabra para el mismo concepto		

Ámbito: Métodos y variables			
Pauta	**Descripción**	**No recomendable**	**Recomendable**
P1	Código estructurado (facilita localizar los elementos de una clase si siempre se sigue la misma estructura) 1. Variables al principio 2. Métodos importantes al principio y métodos secundarios después. 3. Métodos ordenados en función de su uso		

P2	Si una variable sólo se usa en un método, declárala como local	`class Ejemplo{`
		`private m_attr1;`
		`}`
			`package m;`
			`class Ejemplo{`
			`private attr1;`
			`}`

P3	Un cambio en cualquier parte de la funcionalidad de un programa no incluye cambios en partes que no tengan una relación lógica con la funcionalidad cambiada. (Single Responsability y DRY)	`class A{`
		`private d;`
		`}`
			`class Ejemplo{`
			`private elapsedTimeInDays;`
			`private daysSinceCreation;`
			`}`

P4	Métodos con una sola responsabilidad (Single Responsability)	`saveUserAndChargeCard()`	`saveUser();` `chargeCard();`
P5	El tamaño del método debe tener un tamaño relativo a la complejidad funcional que implementa		
P6	Evitar parámetros de salida	`addSignature(Email)`	`email.addSignature()`
P7	En caso de error es preferible hacer uso de excepciones antes que de código de error.		

Ámbito: Comentarios			
Pauta	**Descripción**	**No recomendable**	**Recomendable**
P1	No comentes un mal código, reescríbelo.		
P2	Haz el código auto explicativo	`//Comprueba si un empleado tiene derecho a beneficios completo if(empleado.flags && this.HOURLY_FLAG && empleado.edad >= 65)`	`if(empleado.tieneDerechoABeneficiosCompletos())`
P3	Hacer uso de comentarios para explicar las intenciones del desarrollador no el código	`class A{ private d; }`	`class Ejemplo{ private elapsedTimeInDays; private daysSinceCreation; }`
P4	Hacer uso de comentarios para advertir de consecuencias identificadas	`saveUserAndChargeCard()`	`saveUser(); chargeCard();`
P5	Hacer uso de comentarios para enfatizar un mensaje		
P6	Hay lenguajes que la definición de sus métodos no son explícitos (PHP) usa los doc para aclararlo	`addSignature(Email)`	`email.addSignature()`
P7	Evitar aquellos comentarios repetitivos o que generen confusión (noise comments) Esta arquitectura de referencia será de aplicación obligatoria(*) en los siguientes casos: • Desarrollo de nuevos sistemas de información corporativos. • Refactorización de sistemas de información existentes. • Evolución de sistemas de información que requieran la implementación de nuevos procesos o funcionalidades y que requiera de desarrollo en funciones "core" del sistema.		

P8	No dejar código comentado		

Ámbito: Cosas a Evitar	
Pauta	**Descripción**
P1	No dejar código muerto
P2	Un tamaño excesivo de las clases y métodos que no se justifiquen por la complejidad del negocio
P3	Múltiples lenguajes en un mismo fichero, por ejemplo HTML, Javascript, CSS
P4	No modificar frameworks. Al modificar un framework te ves acoplado a la versión y a la necesidad de realizar un mantenimiento del cambio con cada actualización, Suele ser síntoma de que el framework elegido no es el adecuado ya que no cubre la totalidad de la necesidad del negocio
P5	Usar instrucciones condicionales con grandes condiciones complejas de entender. Substitúyelas por funciones explicativas.
P6	Llamar a la funcionalidad padre (super) al sobrescribir un método
P7	No establecer valores hardcoded, extraer los valores a constantes

3. Principios SOLID

Definición

Estos principios **no son leyes o reglas inalterables** que se tengan que cumplir, pero si son una serie de **re comendaciones prácticas** que de ser aplicadas en su conjunto.

Objetivo

Implementaciones de buena calidad con un **bajo acoplamiento y una alta cohesión**.

Validación

Todas las pautas podrán ser validadas a través de revisiones de código y de arquitectura del proyecto

3.1 - **S**OLID - SINGLE RESPONSABILITY

El principio de responsabilidad único establece que un modulo, clase, método, etc, debe tener **una y solo una razón para cambiar**. Esto significa que una entidad de software debe tener solo **una responsabilidad y hacer únicamente la tarea para la cual ha sido diseñada**. De lo contrario si asume más de una responsabilidad existirá un **alto acoplamiento** provocando que nuestra lógica de programación sea **frágil ante cualquier cambio**.

Ejemplo: Supongamos que tienes una clase `Report` que puede generar informes y enviarlos por correo electrónico. Según SRP, deberías dividir esta clase en dos: una para manejar la generación de informes (`ReportGenerator`) y otra para manejar el envío de correos electrónicos (`EmailSender`).

Esta arquitectura de referencia será de aplicación obligatoria(*) en los siguientes casos:

- Desarrollo de nuevos sistemas de información corporativos.
- Refactorización de sistemas de información existentes.
- Evolución de sistemas de información que requieran la implementación de nuevos procesos o funcionalidades y que requiera de desarrollo en funciones "core" del sistema.

Ámbito: Módulo	
Definir una correcta granularidad a la hora de establecer la responsabilidad de un módulo.	
Pauta	**Descripción**
P1	Definir las responsabilidades en base a la tecnología/capa (Arquitecturas limpias)
P2	Definir las responsabilidades en base al negocio (DDD - Bounded Context) Esta arquitectura de referencia será de aplicación obligatoria(*) en los siguientes casos:

- Desarrollo de nuevos sistemas de información corporativos.
- Refactorización de sistemas de información existentes.
- Esta arquitectura de referencia será de aplicación obligatoria(*) en los siguientes casos:
  - Desarrollo de nuevos sistemas de información corporativos.
  - Refactorización de sistemas de información existentes.
  - Evolución de sistemas de información que requieran la implementación de nuevos procesos o funcionalidades y que requiera de desarrollo en funciones "core" del sistema.

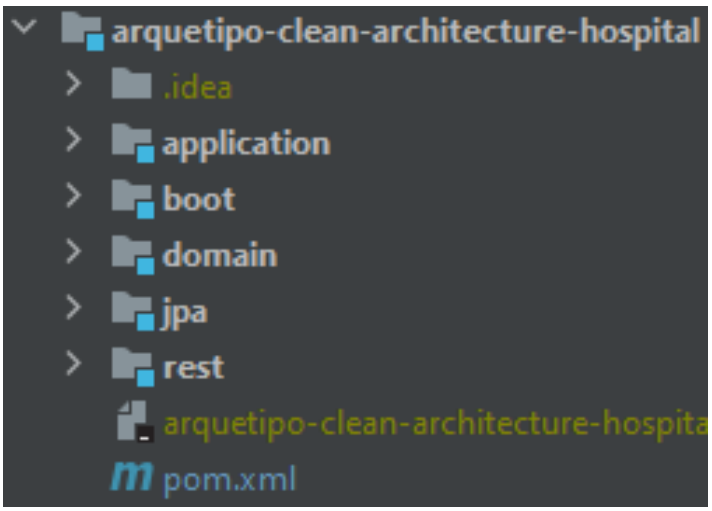

Ilustración - Ejemplo de SR aplicado a módulos

Ámbito: Clases	
Definir una correcta responsabilidad para la clase y asegurarse de que no asume acciones fuera de ella. Se adherirán a una funcionalidad. Su s métodos y datos estarán relacionados con un propósito claro. Esto significa una **alta cohesión ,** así como **robustez, que en conjunto reducen los errores** .	
Pauta	**Descripción**
P1	Cada clase debe tener una responsabilidad, un solo propósito . Esto significa que una clase solo hará un trabajo, lo que nos lleva a concluir que **solo** debe tener **una razón para cambiar.**
P2	Para establecer la responsabilidad el desarrollador debe basar sus criterios en la definición del negocio (DDD), las necesidades funcionales del producto y las necesidades
P3	Para establecer la responsabilidad el desarrollador debe basar sus criterios en las necesidades funcionales del producto
P4	Para establecer la responsabilidad el desarrollador debe basar sus criterios en las necesidades de la arquitectura (Arquitecturas limpias)

Mala separación de responsabilidades

```
class Book {
        function getTitle() {Esta arquitectura de referencia será de aplicación obligatoria(*) en los
siguientes casos:

    Desarrollo de nuevos sistemas de información corporativos.
    Refactorización de sistemas de información existentes.
```

```
    Evolución de sistemas de información que requieran la implementación de nuevos procesos o
funcionalidades y que requiera de desarrollo en funciones "core" del sistema.
        return "A Great Book";
    }
        function getAuthor() {
        return "John Doe";
    }
        function turnPage() {
        // pointer to next page
    }
        // Esta no es responsabilidad de un libro
        function printCurrentPage() {
        echo "current page content";
    }
}
```

Buena separación de responsabilidades

```
class Book {
        function getTitle() {
        return "A Great Book";
    }
        function getAuthor() {
        return "John Doe";
    }
        function turnPage() {
        // pointer to next page
    }
}

interface Printer {

    function printPage($page);
}

class PlainTextPrinter implements Printer {

    function printPage($page) {
        echo $page;
    }

}

class HtmlPrinter implements Printer {

    function printPage($page) {
        echo '<div style="single-page">' . $page . '</div>';
    }
}
```

Ámbito: Métodos	
Pauta	**Descripción**
P1	Si no cambia la signatura, un cambio en la implementación de un método no debe implicar un cambio en quien lo invoca
P2	Debe haber **un solo test unitario** para el caso de éxito.
P3	Para el caso de éxito el código no debería hacer uso de instrucciones condicionales tales como **if-else/switch/ternaria**.

	Hay excepciones a la regla en el que la responsabilidad del propio método es precisamente de selector de opciones, por ejemplo la implementación de una factoría.
P4	No debe recibir como **parámetro un booleano.** Suele indicar múltiple responsabilidades
P5	La funcionalidad descrita no debe contener las conjunciones **y/o**. Suele indicar múltiple responsabilidades. *Ejemplo: El método Y hace XXX y/o XXX*
P6	La funcionalidad descrita no debe contener el adverbio **cuando**. Suele indicar múltiple responsabilidades. *Ejemplo: El método Y hace XXX cuando XXX*
P7	La funcionalidad descrita no debe contener las conjunciones condicional **si**. Suele indicar múltiple responsabilidades. *Ejemplo: SI XXX el método Y hace XXX*
P8	Prestar atención a las condiciones definidas en los bucles **for,do-while,while.** Pueden indicarnos múltiples responsabilidades

3.2 - S**O**LID - OPEN/CLOSE PRINCIPLE

Una clase está cerrada a **la modificación pero abierta para extenderla**. Esto significa que una entidad de software debe ser fácilmente **extensible sin necesidad de modificar su código existente**. Si diseñamos sistemas extensibles **serán menos propensos a errores ante cambios en los requerimientos**

Ejemplo: Si tienes una clase `Shape` con un método `draw()`, y quieres añadir una nueva forma, en lugar de modificar la clase `Shape`, puedes extenderla creando una nueva subclase como `Circle` o `Square`.

Pauta	Descripción
P1	Utiliza interfaces en lugar de superclases para permitir diferentes implementaciones que puede sustituir fácilmente sin cambiar el código que las usa.
P2	Si considera que es beneficioso que dos implementaciones de una interfaz compartan algún código, puede usar herencia o compos ición .
P3	Priorizar el uso de composición por encima de la herencia. De esta forma minimizamos el acoplamiento a una implementación concreta
P4	Evitar el uso de super en las sobre-escritura de los métodos padre. De esta forma minimizamos el acoplamiento y la posibilidad de que un cambio en la funcionalidad en la clase padre impacte en la clase hija.
P5	Existen algunas excepciones en el que la responsabilidad del propio método es precisamente de selector de opciones. *ejemplo: la implementación de una factoría.*
P6	Usar el patrón estrategia para guiar el diseño y el desarrollo de este principio

Principio no aplicado

```
public class Rectangle
{
    public double Width { get; set; }
    public double Height { get; set; }
}
public class Circle
{
```

```
    public double Radius { get; set; }
}
public class AreaCalculator
{
    public double Area(object[] shapes)
        {
                double area = 0;
                foreach (var shape in shapes)
                {
                        if (shape is Rectangle)
                        {
                                Rectangle rectangle = (Rectangle) shape;
                                area += rectangle.Width*rectangle.Height;
                        }
                        else
                        {
                                Circle circle = (Circle)shape;
                                area += circle.Radius * circle.Radius * Math.PI;
                        }
                }

                return area;
        }
}
```

Aplicado Open/Close

```
public interface Shape
{
    public double Area();
}

public class Rectangle : Shape
{
    public double Width { get; set; }
    public double Height { get; set; }
    public override double Area()
    {
        return Width*Height;
    }
}

public class Circle : Shape
{
    public double Radius { get; set; }
    public override double Area()
    {
        return Radius*Radius*Math.PI;
    }
}

public class AreaCalculator
{
    public double Area(Rectangle[] shapes)
    {
        double area = 0;
                foreach (var shape in shapes)
                {
                        area += shape.Area();
                }

                return area;
    }
}
```

3.3 - SO**L**ID - LISKOV SUBTITUTION PRINCIPLE

Cada clase que hereda de otra puede **usarse como su padre sin necesidad de conocer las diferencias entre ellas**. Es decir que, si una clase es un subtipo de otra, los objetos de esta clase subtipo podrían **sustituir a los objetos de la clase padre sin que el programa sufriera ningún cambio de comportamiento**.

Ejemplo: Si tienes una clase base `Bird` con un método `fly()`, y una subclase `Penguin`, la subclase `Penguin` no debería sobrescribir el método `fly()` con una implementación que cause un comportamiento incorrecto.

Pauta	Descripción
P1	Las subclases deben ser sustituibles por sus clases base
P2	No alterar el comportamiento esperado de la clase base. De esta forma minimizamos el acoplamiento y la posibilidad de que un cambio en la funcionalidad en la clase padre impacte en la clase hija.
P3	Asegurar que las subclases cumplen los contratos de la clase base

Principio no aplicado

```
public class Bird {
    public void fly() {
        // Code to fly
    }
}

public class Penguin extends Bird {
    @Override
    public void fly() {
        throw new UnsupportedOperationException("Penguins can't fly");
    }
}
```

Aplicando Liskov Substitution

```
public abstract class Bird {
    public abstract void move();
}

public class Sparrow extends Bird {
    @Override
    public void move() {
        fly();
    }

    private void fly() {
        // Code to fly
    }
}

public class Penguin extends Bird {
    @Override
    public void move() {
        walk();
    }
```

```
    private void walk() {
        // Code to walk
    }
}

// Usage
Bird sparrow = new Sparrow();
sparrow.move();

Bird penguin = new Penguin();
penguin.move();
```

3.4 - SOLID - INTERFACE SEGREGATION

Los clientes no deben estar obligados a depender de interfaces que no utilizan. Es mejor tener varias interfaces específicas del cliente en lugar de una sola interfaz general.

Ejemplo: Si tienes una interfaz `Worker` con métodos `work()` y `eat()`, y una clase `Robot` que solo implementa `work()`, deberías dividir `Worker` en dos interfaces más específicas: `Workable` y `Eatable`.

Pauta	Descripción
P1	Crear interfaces específicas para cada tipo de cliente
P2	Evitar interfaces con métodos innecesarios
P3	Segregar interfaces según responsabilidades específicas
P4	Utilizar la inyección de dependencias para gestionar interfaces

Principio no aplicado

```
public interface Worker {
    void work();
    void eat();
}

public class Robot implements Worker {
    @Override
    public void work() {
        // Code to work
    }

    @Override
    public void eat() {
        throw new UnsupportedOperationException("Robots don't eat");
    }
}
```

Aplicando Interface Segregation

```
public interface Workable {
    void work();
}
```

```
public interface Eatable {
    void eat();
}

public class Human implements Workable, Eatable {
    @Override
    public void work() {
        // Code to work
    }

    @Override
    public void eat() {
        // Code to eat
    }
}

public class Robot implements Workable {
    @Override
    public void work() {
        // Code to work
    }
}

// Usage
Workable worker = new Robot();
worker.work();

Eatable eater = new Human();
eater.eat();
```

3.5 - SOL**ID** - DEPENDENCY INVERSION

Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones. Las abstracciones no deben depender de los detalles; los detalles deben depender de las abstracciones. Significa que una clase concreta, no debe depender directamente de otra clase sino de una abstracción (interfaz) de esta. Nos ayuda a reducir la dependencia en implementaciones específicas y así lograr que el código sea más reutilizable.

Ejemplo: Si tienes una clase `Keyboard` que se conecta directamente a una clase `Computer`, deberías usar una interfaz `IKeyboard` que `Keyboard` implemente, y `Computer` debería depender de `IKeyboard`.

Pauta	Descripción
P1	Depender de abstracciones en lugar de implementaciones
P2	Invertir las dependencias mediante inyección de dependencias
P3	Mantener las clases de alto nivel independientes de las clases de bajo nivel
P4	Utilizar contenedores de inyección de dependencias

Principio no aplicado

```
public class Keyboard {
    // Code for keyboard
}

public class Computer {
    private final Keyboard keyboard;

    public Computer() {
        this.keyboard = new Keyboard();
    }
}
```

Aplicando Dependency Inversion

```
public interface IKeyboard {
    // Abstract methods for keyboard
}

public class Keyboard implements IKeyboard {
    // Code for keyboard
}

public class Computer {
    private final IKeyboard keyboard;

    public Computer(IKeyboard keyboard) {
        this.keyboard = keyboard;
    }
}

// Usage
IKeyboard keyboard = new Keyboard();
Computer computer = new Computer(keyboard);
```

4. DRY - Don’t repeat yourself.

Definición

DRY, acrónimo de "Don't Repeat Yourself" (No te repitas), es un principio de desarrollo de software que enfatiza la reducción de la duplicación de código y lógica dentro de un sistema.

Objetivo

Evitar la redundancia, asegurando que cada pieza de conocimiento o lógica esté representada de manera única y centralizada. Mejora la mantenibilidad, facilita la actualización del código y reduce el riesgo de inconsistencias y errores al modificar el código.

Validación

Code Review, SONAR.

Este principio enfatiza la importancia de evitar la duplicación de código o lógica en un sistema. La idea es que cualquier pieza de conocimiento o lógica debe tener una única representación en el sistema. Esto ayuda a mantener el código limpio, reducir errores y facilitar el mantenimiento.

Pauta	Descripción	Validación
P1	Cada funcionalidad debe ser única, específica y representativa identidad dentro del sistema. (Single Responsability)	Revisiones de código
P2	Centralizar la lógica compartida en módulos reutilizables	Uso de módulos/bibliotecas compartidas en múltiples partes del código
P3	Evitar se duplique el código, si no su mantenimiento será mucho más difícil.	Sonar
P4	Utilizar funciones o métodos para operaciones repetitivas	Verificar llamadas repetidas a las mismas funciones/métodos
P5	Mantener la información en una sola fuente de verdad	Revisiones de diseño y arquitectura
P6	Refactorizar el código regularmente	Revisiones de código y herramientas de refactorización

5. KISS - Keep It Simple, Stupid.

Definición

Es un principio de diseño que aboga por la simplicidad y la claridad en la creación de sistemas y soluciones software.

Objetivo

Soluciones simples y libres de complejidades innecesarias, lo que mejora su comprensión, mantenibilidad, eficiencia y robustez.

Validación

Code Review, SONAR.

El principio aboga por mantener el código lo más simple y claro posible. La simplicidad es clave para facilitar la comprensión, el mantenimiento y la extensión del código. La complejidad innecesaria debe ser evitada, y las soluciones deben ser directas y sin complicaciones.

Pauta	Descripción	Validación
P1	Mantener el código lo más simple y claro posible	Revisiones de requisitos y validaciones de especificaciones
P2	Evitar la complejidad innecesaria	Análisis de complejidad ciclomática y revisiones de diseño
P3	Usar nombres de variables y funciones descriptivos	Revisiones de código y Sonar
P4	Evitar la sobre-ingeniería	Revisiones de diseño y análisis de requisitos
P5	Dividir los problemas grandes en subproblemas simples	Revisiones de código y herramientas de refactorización

6. YAGNI

Definición

"You Aren't Gonna Need It" es un principio de desarrollo ágil que aconseja no implementar funcionalidades hasta que realmente sean necesarias.

Objetivo

Se promueve la eficicienca y la simplificación del código, evitando el trabajo extra y la compeljidad que surge de preveer y construir features que podrían no ser utilizadas en el funturo. Impacta positivamente en la pertinencia funcional y la adaptabilidad.

Validación

Reviews

El principio sostiene que no se deben implementar funcionalidades hasta que realmente se necesiten. Esto ayuda a evitar el desperdicio de tiempo y recursos en características que no se usarán, y mantiene el código más limpio y enfocado en los requisitos actuales.

Estos principios son fundamentales en el desarrollo de software ágil y ayudan a mantener un código base limpio, manejable y eficiente.

Pauta	Descripción	Validación
P1	Implementar solo las funcionalidades necesarias	Revisiones de requisitos y validaciones de especificaciones
P2	Evitar la anticipación de necesidades futuras	Revisiones de código y validaciones de alcance
P3	Eliminar el código muerto o no utilizado	Análisis de cobertura de código y revisiones de código
P4	Mantener el enfoque en los requisitos actuales	Revisiones de requisitos y gestión de cambios
P5	Refactorizar el código regularmente para mantenerlo limpio	Revisiones de diseño y arquitectura

No se puede representar {include}. No se ha encontrado la página incluida.

Terminología	
Mantenible	"Capacidad de un sistema o componente de software para ser modificado de manera efectiva y eficiente, incluyendo correcciones, mejoras o adaptaciones a cambios en el entorno o requisitos."
Legible	"Facilidad con la que el código fuente del software puede ser comprendido, interpretado y revisado por los desarrolladores y otros interesados."
Entendible	

	"Capacidad de un sistema o componente de software para ser comprendido en términos de sus conceptos y la representación lógica de sus funciones y procesos."
Testeable	"Capacidad de un sistema o componente de software para permitir la ejecución de pruebas que demuestren que cumple con sus requisitos especificados."
Extensible	"Capacidad de un sistema o componente de software para ser modificado y ampliado con nuevas funcionalidades sin afectar su estructura original."
Resiliente	"Capacidad de un sistema o componente de software para operar correctamente y recuperarse rápidamente ante condiciones adversas o fallos."
Acoplamiento	"Grado en que un módulo, clase, método o cualquier otra entidad de software, está directamente vinculada a las demás. Este grado de acoplamiento también puede ser visto como un grado de dependencia."
Cohesión	"Medida en que se relacionan dos o más partes de un sistema y cómo trabajan en conjunto para lograr mejores objetivos que las partes individuales."