**Principio DRY**

El **principio DRY** ("Don't Repeat Yourself" o "No te repitas") es un concepto fundamental en el desarrollo de software que busca evitar la duplicación de código. Este principio establece que cualquier funcionalidad existente en un programa debe estar representada de forma única, evitando así bloques de código repetidos.

Hunt y Thomas (2000) destacan que para desarrollar software de manera confiable y facilitar su mantenimiento, es crucial seguir el principio DRY. Este principio sugiere que cada pieza de conocimiento dentro de un sistema debe tener una única representación autoritativa. Esto significa que cualquier duplicación puede generar dificultades en el mantenimiento y aumentar la posibilidad de errores cuando se realizan cambios en el código​.

Además, el documento detalla las diferentes formas en que la duplicación puede ocurrir, como duplicación impuesta, inadvertida, impaciente y duplicación entre desarrolladores. Abordar estas formas de duplicación es esencial para mantener un código limpio y fácil de mantener​.

**Modelo MVC**

El modelo MVC, que significa "Modelo-Vista-Controlador", es un patrón de diseño utilizado en el desarrollo de software que busca separar las diferentes partes de una aplicación para hacer su mantenimiento y actualización más eficientes. Este enfoque permite que cada componente de la aplicación funcione de manera independiente, facilitando así la colaboración entre desarrolladores y diseñadores.

**Elementos del Patrón MVC**

**Modelo**: Se encarga de la gestión de los datos y la lógica de negocio. Accede a la capa de almacenamiento de datos y es responsable de toda la interacción con la base de datos.

**Vista**: Presenta la información al usuario. Se encarga de mostrar los datos que provienen del modelo, proporcionando una interfaz visual.

**Controlador**: Actúa como intermediario entre el modelo y la vista. Recibe las entradas del usuario, procesa las solicitudes y actualiza el modelo y/o la vista según sea necesario.

**Beneficios del Modelo MVC**

* **Separación de Concerns**: Cada componente tiene una responsabilidad clara, lo que reduce la complejidad del código.
* **Facilidad de Mantenimiento**: Las modificaciones en una parte de la aplicación impactan mínimamente en las otras, lo que simplifica el mantenimiento.
* **Reutilización de Código**: Permite la creación de múltiples vistas para un mismo modelo, lo que favorece la reutilización del código.
* **Especialización**: Facilita la especialización entre desarrolladores (que trabajan en el modelo) y diseñadores (que trabajan en la vista).

**Orígenes y Uso del Modelo MVC**

El patrón MVC fue descrito por primera vez en 1979 por Trygve Reenskaug y se introdujo en el lenguaje de programación Smalltalk-80. Se ha convertido en una norma en el desarrollo de aplicaciones, especialmente en aplicaciones web, donde la vista puede ser una página HTML y el modelo puede ser una base de datos.

**Principio SPR (Single Responsibility Principle)**

El **Principio de Responsabilidad Única (SRP)** establece que "una clase debe tener solo una razón para cambiar". Esto significa que cada entidad en el software debe tener una única responsabilidad o función, lo que facilita su mantenimiento y comprensión. A continuación, se detallan los aspectos clave de este principio:

**Definición e Importancia**

1. **Una sola tarea**: Cada clase o módulo debe describirse con una única oración que resuma su tarea principal.
2. **Cohesión**: Se busca que las funciones de una clase estén altamente relacionadas, lo que se conoce como cohesión.
3. **Evitar clases grandes**: Si una clase tiene demasiados métodos, probablemente está manejando más de una responsabilidad, lo que puede llevar a complicaciones en el código.
4. **Facilidad de cambio**: Un diseño que sigue el SRP permite que los cambios en una parte del sistema no afecten a otras partes, lo que reduce el riesgo de errores.

**Principio ISP (Interface Segregation Principle)**

El Principio de Segregación de Interfaces (ISP) es uno de los cinco principios SOLID de diseño de software, formulado por Robert C. Martin. Este principio establece que "no se debe obligar a los clientes a depender de interfaces que no utilizan". Su objetivo es minimizar los efectos secundarios y la dependencia de interfaces más grandes al dividirlas en otras más pequeñas y específicas.

**Detalles Clave del ISP**

1. **Definición**: ISP sugiere que las interfaces deben ser diseñadas de tal manera que los clientes solo necesiten conocer los métodos que realmente utilizan, evitando así la dependencia de métodos innecesarios.
2. **Reducción de Complejidad**: Al crear interfaces más pequeñas, se simplifica el diseño y se mejora la comprensión del código, lo que facilita su mantenimiento y evolución.
3. **Ejemplo de Aplicación**: Si una interfaz grande proporciona múltiples métodos, se puede refactorizar en varias interfaces pequeñas que representen funcionalidades específicas, permitiendo que las clases implementen solo aquellas interfaces que realmente necesitan.
4. **Relación con Otros Principios SOLID**: ISP está relacionado con el Principio de Responsabilidad Única, ya que ambos buscan que cada componente del software tenga un propósito claro y definido.
5. **Beneficios**:
   * **Menor Acoplamiento**: Las clases que implementan interfaces específicas tienen menos dependencia de otras partes del sistema.
   * **Mejor Mantenibilidad**: Cambios en una funcionalidad específica no impactan a otras partes del sistema, reduciendo el riesgo de introducir errores.
   * **Facilidad de Pruebas**: Al tener interfaces más pequeñas, se pueden probar más fácilmente las interacciones de clase.
6. **Implementación**: Para aplicar el ISP, es fundamental realizar un análisis cuidadoso de las necesidades de las clases y crear interfaces que reflejen solo las operaciones requeridas.
7. **Ejemplo Visual**: En una refactorización típica, se puede ilustrar cómo una interfaz se divide en múltiples interfaces más específicas, mostrando claramente las relaciones entre ellas y las clases que las implementan.
8. **Desafíos**: Aunque implementar el ISP puede requerir más tiempo y esfuerzo en la fase de diseño, los beneficios a largo plazo en términos de código limpio y mantenible justifican esta inversión inicial.

**Conclusión**

El Principio de Segregación de Interfaces es esencial para el diseño de software eficiente y bien estructurado. Al reducir la complejidad y la dependencia de interfaces, se logra un código más claro y fácil de mantener, lo que resulta en un desarrollo de software más ágil y efectivo 1.

**Bibliografía**

Sucasaca Surco, W. Uso de los principios DRY y SOLID en un proceso de refactoring de una aplicación web java para mejorar su calidad interna. Tomado de: <http://repositorio.unap.edu.pe/bitstream/handle/20.500.14082/9267/Sucasaca_Willington_Surco.pdf?sequence=1&isAllowed=y>

González, Y. D., & Romero, Y. F. (2012). Patrón Modelo-Vista-Controlador. *Telemática*, *11*(1), 47-57. Tomado de: <https://revistatelematica.cujae.edu.cu/index.php/tele/article/view/15/10>

Jennings, Richard, and Fabiola De la Cueva. 2020. “SOLID Principles of Object-Oriented Design.” Chap. 4.2 in *LabVIEW Graphical Programming*. 5th ed. New York: McGraw-Hill Education. <https://www.accessengineeringlibrary.com/content/book/9781260135268/toc-chapter/chapter4/section/section37>

Sánchez, G. (2023). Principios SOLID en la Automatización de Pruebas de Software para Interfaces de Usuario Web con Selenium WebDriver y Java. *Revista Politécnica de Aguascalientes*, *2*. Tomado de: <https://revistapolitecnicaags.upa.edu.mx/wp-content/uploads/2023/12/V2116-.pdf>