Conociendo principios y autores del desarrollo de software

La intención de este documento es dar el puntapié inicial a conocer un conjunto conceptos y autores referentes en la industria del software, con el objetivo de despertar la curiosidad del estudiante y motivar a investigar los mismos en mayor profundidad.

A continuación se presentarán un conjunto de técnicas y principios de desarrollo de software alineados al manifiesto ágil. Por este motivo, los principios en su mayoría son forjados o expuestos por las personas que firmaron dicho manifiesto: Kent Beck (TDD) , Robert C. Martin (SOLID, Clean Code), Jeff Sutherland (SCRUM), Martin Fowler, entre otros.

Esto explica a su vez, por que a pesar de ser distintos autores, todos se alinean entre sí buscando el mismo objetivo y realizan referencias entre ellos constantemente:

"[Valorar] Individuos e interacciones sobre procesos y herramientas
Software funcionando sobre documentación extensiva
Colaboración con el cliente sobre negociación contractual
Respuesta ante el cambio sobre seguir un plan"

https://agilemanifesto.org/iso/es/manifesto.html

CLEAN CODE (Robert C. Martin)

Mencionamos de dicho libro varios tips a tener en cuenta:

- El código que queremos es código limpio que funcione
- Favorecemos la legibilidad ante todo, buscamos lectura similar a un diario o libro
- El código de mala calidad (poco legible o escalable) nos genera olor a podrido (Smell) y conocemos un conjunto de principios para detectarlos y atacarlos
- El software debe ser fácil de cambiar, por ello es software y no hardware
- Aplicaremos la regla del Boy Scout: Dejar el lugar más limpio que cuando llegué
- Los nombres de variables y funciones deben ser representativos
- Las funciones deben hacer una sola cosa y hacerla bien
- Las funciones no deberían recibir más de 2 parámetros (0 es mejor aún, 3 no es deseable)
- Los comentarios no son buenos, lo bueno es el código que no necesita comentarios
- Los nombres de los objetos deberían ser sustantivos y los métodos verbos
- Buscamos cómo objetivo no tener miedo de cambiar el código
- La única forma de lograr refactor rápidos es con una buena batería de test, escritos idealmente con TDD (método para combatir el miedo)

Resumen corto del libro:
https://samuelcasanova.com/2016/09/resumen-clean-code/

Regla del boy Scout:
https://hackernoon.com/object-oriented-tricks-5-boy-scout-rule-cec82aea3b81

Catálogo de code Smell y patrones de diseño:
https://refactoring.guru/es/refactoring/smells

DRY: Don't Repeat Yourself

El principio No te repitas (en inglés Don't Repeat Yourself o DRY, también conocido como Una vez y sólo una) es una filosofía de definición de procesos que promueve la reducción de la duplicación especialmente en computación. Según este principio toda "pieza de información" nunca debería ser duplicada debido a que la duplicación incrementa la dificultad en los cambios y evolución posterior, puede perjudicar la claridad y crear un espacio para posibles inconsistencias. Los términos "pieza de información" son usados en un sentido amplio, abarcando:

datos almacenados en una base de datos;
código fuente de un programa de software;
información textual o documentación.
Cuando el principio DRY se aplica de forma eficiente los cambios en cualquier parte del proceso requieren cambios en un único lugar. Por el contrario, si algunas partes del proceso están repetidas por varios sitios, los cambios pueden provocar fallos con mayor facilidad si todos los sitios en los que aparece no se encuentran sincronizados.

https://en.wikipedia.org/wiki/Don%27t_repeat_yourself

KISS

Del inglés Keep It Simple, Stupid!, «¡Mantenlo sencillo, estúpido!».

Establece que los sistemas funcionan mejor si se mantienen simples. La simplicidad debe ser un objetivo clave del diseño, evitando cualquier complejidad innecesaria.

https://manuel.cillero.es/doc/apuntes-tic/principios-desarrollo-software/

YAGNI

Del inglés You Aren’t Gonna Need It, «No vas a necesitarlo».
Consiste en no agregar nunca una funcionalidad innecesaria o no solicitada.
La tentación de escribir código que no es necesario, pero puede serlo en un futuro, sacrifica tiempo que podría destinarse a funcionalidades básicas, sin olvidar que cualquier característica nueva, necesaria o no, debe ser depurada, documentada y soportada.

https://manuel.cillero.es/doc/apuntes-tic/principios-desarrollo-software/

SLAP

Del inglés Single Level of Abstraction Principle, «Principio de Nivel Único de Abstracción».

Escribir código tiene que ver con abstracciones, ocultando detalles de bajo nivel de los conceptos de más alto nivel.

Este principio propone dividir el programa en funciones o métodos con una única responsabilidad, pocas líneas de código y un único nivel de abstracción. Sólo deben hacer una cosa, y hacerla bien.

Si es necesario se pueden usar funciones o métodos compuestos, con llamadas a otras funciones o métodos privados, cada uno con un nombre claro que identifique su responsabilidad.

https://manuel.cillero.es/doc/apuntes-tic/principios-desarrollo-software/
https://hackernoon.com/object-oriented-tricks-6-slap-your-functions-a13d25a7d994

CQS (command and query separation)

Command–query separation (CQS), en castellano Separación de comandos y consultas, es un principio de la programación orientada a objetos. Fue ideado por Bertrand Meyer como parte de su trabajo pionero sobre el lenguaje de programación Eiffel.the

El principio afirma que cada método debe ser un comando que realiza una acción, o una consulta que devuelve datos al llamante, pero no ambos. En otras palabras, hacer una pregunta no debe cambiar la respuesta. Más formalmente, los métodos deben devolver un valor sólo si son referencialmente transparentes y, por tanto, no poseen efectos colaterales. Es de destacar que la aplicación rígida de esta especificación hace que el seguimiento del número de veces que las consultas han sido emitidos sea esencialmente imposible; está claramente destinado como una guía de programación en lugar de una regla para una buena codificación, como la de evitar el uso de un GOTO desde un bucle anidado.

https://hackernoon.com/oo-tricks-the-art-of-command-query-separation-9343e50a3de0

https://es.wikipedia.org/wiki/Command%E2%80%93query_separation

TDD (Kent Beck)

Desarrollo guiado por pruebas de software, o Test-driven development (TDD) es una práctica de ingeniería de software que involucra otras dos prácticas: Escribir las pruebas primero (Test First Development) y Refactorización (Refactoring). Para escribir las pruebas generalmente se utilizan las pruebas unitarias (unit test en inglés). En primer lugar, se escribe una prueba y se verifica que la nueva prueba falla. A continuación, se implementa el código que hace que la prueba pase satisfactoriamente y seguidamente se refactoriza el código escrito. El propósito del desarrollo guiado por pruebas es lograr un código limpio que funcione. La idea es que los requisitos sean traducidos a pruebas, de este modo, cuando las pruebas pasen se garantizará que el software cumple con los requisitos que se han establecido.

https://es.wikipedia.org/wiki/Desarrollo_guiado_por_pruebas

SOLID (Robert C. Martin)

En ingeniería de software, SOLID (Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion) es un acrónimo mnemónico introducido por Robert C. Martín a comienzos de la década del 2000 que representa cinco principios básicos de la programación orientada a objetos y el diseño. Cuando estos principios se aplican en conjunto es más probable que un desarrollador cree un sistema que sea fácil de mantener y ampliar con el tiempo.

Los principios SOLID son guías que pueden ser aplicadas en el desarrollo de software para eliminar malos diseños provocando que el programador tenga que refactorizar el código fuente hasta que sea legible y extensible. Puede ser utilizado con el desarrollo guiado por pruebas, y forma parte de la estrategia global del desarrollo ágil de software y desarrollo adaptativo de software.

Single-responsibility Principle (Principio de responsabilidad única)

Cada módulo o clase debe ser responsable de una única cosa, y esa responsabilidad debe estar completamente encapsulada por la clase. (motivo único de cambio)

Open-closed Principle (Principio de abierto/cerrado)

Las clases, módulos o funciones deben estar abiertas para su extensión, pero cerradas para su modificación. O de otra forma, se debería poder extender el comportamiento de una clase, sin modificarla.

Liskov Substitution Principle (Principio de sustitución de Liskov)

Una clase debería ser sustituible por su clase padre. La clase heredada debe complementar, no reemplazar, el comportamiento de la clase base.

Interface Segregation Principle (Principio de segregación de la interfaz)

Ningún cliente debe verse obligado a depender de métodos que no utiliza. Es preferible contar con muchas interfaces con pocos métodos que tener una interface que requiera implementar métodos que no serán usados.

Dependency Inversion Principle (Principio de inversión de la dependencia)

Las dependencias deben recaer sobre abstracciones (interfaces), no sobre clases concretas (implementaciones).

https://es.wikipedia.org/wiki/SOLID
https://manuel.cillero.es/doc/apuntes-tic/principios-desarrollo-software/