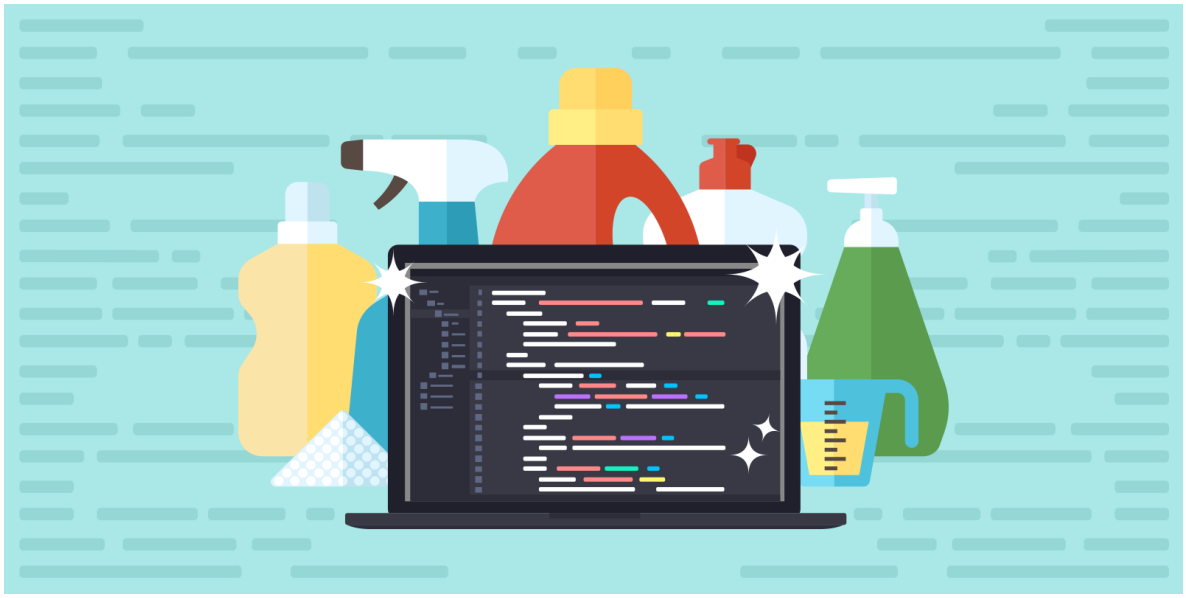
PRINCIPIOS GENERALES DEL CLEAN CODE



Contenido

Principio Kiss (keep It Simple, Stupid) 3
Principio Dry (Don’t Repeat Yourself).....4
Principio YAGNI (You Aren’t Gonna Need It) 6
Legible antes que conciso 6
Conclusión.....7
Bibliografía 7

Principio Kiss (keep It Simple, Stupid)

“Que sea sencillo, estúpido”, este principio dicta a los programadores que el código debe mantenerse *lo más sencillo posible,* evitando complejidades innecesarias. Puesto que siempre habrá más de una forma de resolver un problema. Las tareas, por ende, siempre podrán expresarse de forma distinta o con distintos comandos, por lo tanto, los programadores que sigan el principio *Kiss* deben preguntarse siempre si podrían dar con una solución mas simple a un problema en particular.

Consideraciones para tener en cuenta al seguir el principio *Kiss:*

- La descomposición de algo complejo en componentes simples es un enfoque arquitectónicamente correcto.
- No siempre se necesita precisión matemática absoluta o detalles extremos: Los datos pueden y deben procesarse con la precisión suficiente para una solución de alta calidad del problema, y los detalles se dan en la cantidad necesaria para el usuario, y no en el volumen máximo posible.

Analícese el siguiente ejemplo en el que se solicita calcular la suma y la multiplicación de los 2 números mayores entre 3 números entregados.

Obsérvese que esta solución si bien cumple el objetivo, no sigue el principio KISS de mantener las cosas simples, puesto que no hay descomposición de algo complejo en componentes mas simples, tampoco es un código sencillo de leer, dado que se encuentran condicionales anidados en muchos casos que hace fácil poder perderse en la interpretación.

```cpp
#include <iostream>

using namespace std;

int Calcu(int a, int b,int c,int t);

int main(){

    cout<<Calcu(10,2,5,0)<<endl;
    cout<<Calcu(10,2,5,1)<<endl;

    return 0;
}
```

```
50
15


--------------------------------
Process exited after 0.03529 secon
Presione una tecla para continuar
```

```cpp
int Calcu(int a, int b, int c,int t){
    if(t==0){
        if(a>b){
            if(c>b){
                return a*c;
            }else{
                return a*b;
            }
        }else{
            if(c>a){
                return b*c;
            }else{
                return a*b;
            }
        }
    }else if(t==1){
        if(a>b){
            if(c>b){
                return a+c;
            }else{
                return a+b;
            }
        }else{
            if(c>a){
                return b+c;
            }else{
                return a+b;
            }
        }
    }
}
```

Obsérvese ahora la siguiente solución que cumple con los requerimientos del problema, implementando el principio KISS, dado que si presenta descomposición en módulos que permitían mantener las cosas más sencillas, más mantenibles y reutilizables. Además, es un código más sencillo de entender.

```cpp
#include <iostream>

using namespace std;

int sumarMayores(int a,int b,int c);
int multiplicarMayores(int a, int b, int c);
int encontrarMayor(int a,int b);

int main(){

    cout<<sumarMayores(10,2,5)<<endl;
    cout<<multiplicarMayores(10,2,5)<<endl;

    return 0;
}
```

```
15
50


--------------------------------
Process exited after 0.03804 secor
Presione una tecla para continuar
```

```cpp
int encontrarMayor(int a,int b){
    if(a>b){
        return a;
    }else{
        return b;
    }
}

int sumarMayores(int a, int b,int c){
    int mayor_uno = encontrarMayor(a,b);
    if(mayor_uno==a){
        int mayor_dos = encontrarMayor(b,c);
        return mayor_uno+mayor_dos;
    }else{
        int mayor_dos = encontrarMayor(a,c);
        return mayor_uno+mayor_dos;
    }
}
int multiplicarMayores(int a,int b, int c){
    int mayor_uno = encontrarMayor(a,b);
    if(mayor_uno==a){
        int mayor_dos = encontrarMayor(b,c);
        return mayor_uno*mayor_dos;
    }else{
        int mayor_dos = encontrarMayor(a,c);
        return mayor_uno*mayor_dos;
    }
}
```

Principio Dry (Don't Repeat Yourself)

"no te repitas" es una concreción de ***KISS***. De acuerdo con el principio ***Dry***, cada función debe tener una representación única, y, por lo tanto, inequívoca. 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.

Cuando el principio ***Dry*** es aplicado 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.

Cada vez que se repite un trozo de código (reglas, procesos, rutinas, funciones…etc.) es de vital importancia acordarse donde yacen estas. Puesto que, si posteriormente cambian las especificaciones, es encontrada una mejor forma de hacer las cosas, o se encuentra que algo está mal implementado, hay que cambiar el código en todos los lugares donde existen estos trozos de código, De no hacerse, se tendrá un sistema que se comporta de una forma en ocasiones, y de otra forma en otras, puesto que se utilizan representaciones inconsistentes, por ende, se tendrá un sistema inconsistente.

Obsérvese el siguiente ejemplo:
Se pide un programa que realice la suma de 5 y 8 y la muestra por pantalla, luego debe hacer lo mismo, pero con los números 10 y 20, observemos como se resuelve este problema, basado en el principio ***DRY*** y sin basarse en este mismo:

Nótese como esta solución resuelve el problema, puesto que cumple con realizar ambas sumas y mostrarlas por pantalla, sin embargo, esta misma no cumple con el principio DRY de no repetirse a uno mismo, puesto que se está repitiendo código que hace exactamente lo mismo.

```cpp
#include <iostream>

using namespace std;

int main(){

    int resultado=0;

    resultado = 5+8;

    cout<<"5 + 8 = "<<resultado<<endl;

    resultado = 10+20;

    cout<<"10 + 20 = "<<resultado<<endl;

    return 0;
}

```

```
5 + 8 = 13
10 + 20 = 30

--------------------------------
Process exited after 0.04913 secon
Presione una tecla para continuar
```

Observemos ahora como una manera mas limpia de resolver este problema, en la solución presentada observamos que se modulariza el proceso de sumar en una función que recibe 2 números y retorna la suma de estos mismos, observamos como se cumple el principio de “no te repitas” puesto que para realizar varias veces la misma tarea no se reescribe código, simplemente se llama al modulo encargado y se le pasan los valores

```cpp
#include <iostream>

using namespace std;

int sumar(int a, int b);

int main(){

    cout<<"5 + 8 = "<<sumar(5,8)<<endl;
    cout<<"10 + 20 = "<<sumar(10,20)<<endl;

    return 0;
}
int sumar(int a, int b){
    return a+b;
}
```

```
5 + 8 = 13
10 + 20 = 30

--------------------------------
Process exited after 0.03751 secc
Presione una tecla para continuar
```

Principio YAGNI (You Aren't Gonna Need It)

"no lo vas a necesitar" o ***Yagni***, se basa en una idea, la idea de que el desarrollador solo debe ***añadir funciones*** al código ***cuando sea estrictamente necesario.*** YAGNI está íntimamente relacionado con los métodos del desarrollo ágil del software. De acuerdo con este principio, en lugar de comenzar a programar partiendo de un concepto general, la arquitectura del software se desarrolla ***paso a paso*** para poder reaccionar a cada problema de forma dinámica. En otras palabras, se crea clean code cuando los problemas subyacentes se resuelven de la manera ***más eficiente posible.***

Por ejemplo, si se necesitara realizar la validación de un correo electrónico y una contraseña, no se debería validar el nombre, puesto que es posible que nunca se necesite

Nótese que el ejemplo anterior no es implementado usando código, puesto que YAGNI va mas allá de como hacer algo en código, trata de mostrar como codificar de manera efectiva al escribir solamente el código requerido.

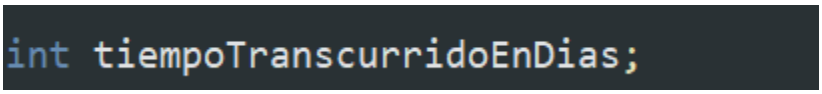
Legible antes que conciso

El código no solo debe funcionar y ser interpretado por la maquina que lo ejecuta, sino que también debe ser comprensible para otros desarrolladores, especialmente si se trabaja en proyectos colaborativos. Por lo tanto, en el ámbito del desarrollo de software, la legibilidad del código siempre es mas importante que su concisión: no tiene sentido escribir código conciso si el resto de los desarrolladores no lo entienden. Un buen ejemplo de creación de código legible sería nombrar las variables.

Los nombres de las variables siempre deben ser comprensibles. Por ejemplo, no es posible entender la siguiente variable:

```cpp
int d;  //Tiempo transcurrido en dia
```

Sin embargo, con el siguiente nombre, la misma variable se explica por sí sola:



```
int tiempoTranscurridoEnDias;
```

Para el caso de las funciones se recomienda el uso de nombres descriptivos, nombres adecuados que describan el contenido de la función, no debería presentar temor a los nombres extensos. Un nombre descriptivo extenso es mucho mejor que uno breve pero enigmático. Úsese una convención de nombres que permita leer varias palabras en los nombres de funciones y procure el uso de esas palabras para asignar a la función un nombre que describa su cometido.

Obsérvese los siguientes ejemplos:

```
int CalcS(int a, int b);  //mal nombre, no existe descripción de la funcionalidad
int calcularSuma(int a, int b); //buen nombre, existe una descripción de la funcionalidad y el cometido

int igual(int a, int b); //mal nombre, no se encuentra un sentido descriptivo de la función

int esIgual(int a, int b); //buen nombre, regleja la funcionalidad y la intención de esta misma
```

Conclusión

Es de completo agrado cerrar este material citando una breve parte de la introducción del libro clean code de Robert C Martin:

> “Aprender a crear código limpio es complicado. Requiere algo más que conocer principios y patrones. Tiene que sudar. Debe practicarlo y fallar. Debe ver como otros practican y fallan. Debe observar cómo se caen y recuperan el paso. Debe ver como agonizan en cada decisión y el precio que pagan por tomar decisiones equivocadas”.

Bibliografía

Código Limpio Manual para el desarrollo ágil de software Robert C. Martin

https://medium.com/@derodu/design-patterns-kiss-dry-tda-yagni-soc-828c112b89ee#:~:text=YAGNI%3A%20You%20ain't%20gonna%20need%20it&text=In%20general%20it%20says%20that,we%20may%20never%20need%20it.