Taller: Principios de Diseño de Software

En el desarrollo de software, existen diferentes principios a los cuales podemos apegarnos con el fin de poder tomar mejores decisiones en cuanto a la manera en la que diseñamos nuestros software. Siendo así, aunque existen grandes cantidades de principios con los cuales podríamos trabajar, se estudiarán 4 de estos con el fin de ver que es lo que proponen y el como se aplicaría durante el diseño de nuestro software.

DRY: Don't Repeat Yourself

El principio de diseño *Don't Repeat Yourself*, o DRY por sus siglas, hace referencia a una de las bases de la programación en cuanto a la implementación del mismo. Este principio se refiere a la búsqueda de evitar la repetición de lógica dentro del código. Es decir, aplicando DRY, se busca el evitar casos en los cuales se está escribiendo la misma funcionalidad de algo multiples veces, lo cual incrementaría el costo técnico debido a la necesidad de mantener la lógica en varias secciones del código.

Esta repetición surge por problemas en el diseño en cuanto al como se diseñan cada una de las partes. En este caso, este problema surge por la falta de claridad o división en las funcionalidades de cada una de las partes de la solución planteada.

Un ejemplo de esto puede verse a continuación:

```
class Formatter:
    def format_string(self, unformatted: str):
        formatted_string: str = f"<--> !{title(unformatted)}! <-->"
        json_response: dict = {"newString": formatted_string}
        return json.dumps(json)

    def format_int(self, unformatted: int):
        formatted_int: str = f"<-o-> <{unformatted}> <-o->"
        json_response: dict = {"newIntString": formatted_string}
        return json.dumps(json)
```

En este caso podemos ver el como la funcionalidad de poner cada uno de los valores en un `dict` y retornarlo como json se repite en cada una de estas funciones. En el caso de que tuvieran que agregarse más campos o retornar el valor como otra cosa, tendrían que modificarse ambas funciones para que estas sigan funcionando. Siendo así, podríamos aplicar el principio de DRY y separar esta funcionalidad.

```
class Formatter:
    def to_json(self, key: str, formatted: str):
        json_response: dict = {key: formatted}
        return json.dumps(json)

    def format_string(self, unformatted: str):
        formatted_string: str = f"<--> !{title(unformatted)}! <-->"
        return self.to_json("newString", formatted_string)

    def format_int(self, unformatted: int):
        formatted_int: str = f"<-o-> <{unformatted}> <-o->"
        return self.to_json("newIntString", formatted_int)
```

De esta manera, en el caso de tener que realizar cambios en la lógica del programa, sólo tendría que ser modificada la función `to_json` para resolver el problema.

YAGNI: You Ain't Gonna Need It

YAGNI, o *You Ain't Gonna Need It*, es uno de los principios de diseño más básicos en cuanto a lo que este implica. En esencia este busca que el código que se escriba sea únicamente el código que se necesite en el momento. Este principio está orientado principalmente a la búsqueda de evitar el trabajo innecesario para cualquier momento dado.

Un ejemplo de esto podría verse en donde se está pensando a futuro con el código. Puede que esa funcionalidad extra para la clase sea usada eventualmente pero puede que no lo sea. Siendo así, es mejor escribir únicamente lo que sea necesario para el momento.

KISS: Keep it simple, stupid!

Ahora, KISS, o *Keep it simple, stupid!*, hace referencia a la búsqueda de soluciones sencillas a los problemas. Es decir, buscar escribir menos complejo con el fin de que este sea más sencillo de entender para aquellos que no escribieron el código. Este principio busca el reducir el grado de complejidad innecesario que algunas soluciones de software poseen. Esto está principalmente orientado a la búsqueda de la *mantenibilidad* del código para las futuras generaciones las cuales estén encargadas del código.

Un ejemplo de esto puede verse a continuación:

```
def filter_list(in_list: list[str]):
    return [elem for elem in in_list if len(elem) in [1, 3, 5, 7]]
```

Como podemos ver, en este caso, se nos presenta la función `filter_list`. Esta usa *list comprehension* para poder filtrar cada uno de los elementos, y aunque es corto, la implementación de esta es un poco confusa y poco descriptiva. Esto es especialmente para aquellos que no hayan usado *list comprehension* con anterioridad.

Aplicando KISS, nuestra solución podría verse de la siguiente manera:

```
def filter_list(in_list: list[str]):
    filtered: list[str] = []
    for elem in in_list:
        if elem in [1,3,5,7]:
            filtered.append(elem)

    return filtered
```

Aunque este es más largo que nuestra primera solución, esta es mucho más legible y fácil de comprender. De esta manera, estamos planteando las soluciones más *mantenibles* para el futuro.

Composición Sobre Herencia

La composición sobre la herencia se refiere a, en los casos donde sea posible, usar la composición de clases y no herencia entre ellas. Esto se dice con el propósito principal de evitar la "sobre-herencia" entre clases.

La "sobre-herencia" se refiere principalmente a los casos en los cuales, con el fin de mantener las reglas de herencia, se terminan creando cada vez más clases con el fin de dar cabida a las diferentes variaciones de cada una de las clases.

Un ejemplo extremo puede verse a continuación:

```
class Car {};
```

Tenemos un carro, todo bien hasta aquí. ¿Pero qué pasa si queremos que tenga diferentes colores? Usando únicamente herencia, tendríamos que hacer lo siguiente:

```
class RedCar extends Car {};
class BlueCar extends Car {};
class GreenCar extends Car {};
```

¿Y si queremos llantas blancas?

```
class RedCarWhiteTires extends Car {};
class BlueCarWhiteTires extends Car {};
class GreenCarWhiteTires extends Car {};
```

¿Y si queremos tener un sticker?

```
class RedCarWhiteTiresSticker extends Car {};
class BlueCarWhiteTiresSticker extends Car {};
class GreenCarWhiteTiresSticker extends Car {};
```

¿Y si lo quiero con el sticker pero sin las llantas blancas?

```
class RedCarSticker extends Car {};
class BlueCarSticker extends Car {};
class GreenCarSticker extends Car {};
```

Y así con cada una de las las diferentes variaciones. Esto no es factible. Es más simple el tener composición de cada una de las clases que componen los carros. De esta manera, tendremos el siguiente modelo:

```
class Car {
    private color: ColorType;
    private Tires: TireType;
    private Sticker: StickerType;
}
```

Teniendo esta composición, nos evitamos tener grandes cantidades de clases innecesarias.

Referencias

- https://dzone.com/articles/software-design-principles-dry-and-kiss
- https://medium.com/@derodu/design-patterns-kiss-dry-tda-yagni-soc-828c112b89ee
- https://www.interaction-design.org/literature/topics/keep-it-simple-stupid
- https://www.youtube.com/watch?v=d-KbEQM0724
- https://medium.com/geekculture/composition-over-inheritance-7faed1628595

Daniel David Delgado Cervantes - 2182066