Practica de Identificación de Code Smells

Manuel de Jesús Sanabria Montoya
Randall Sánchez Rivera

Universidad CENFOTEC
Curso: Diseño de Sistemas de Software
Profesores: Mario Chacón Rivas, Kevin A. Hernández Rostrán
Fecha: Julio, 2025

Tabla de contenidos

Abstract

Este documento expone el proceso de análisis y refactorización de un método que violaba el principio de responsabilidad única (SRP), uno de los fundamentos del enfoque SOLID en el diseño orientado a objetos. A través de la identificación de múltiples responsabilidades concentradas en una sola función como el cálculo de macronutrientes, la generación de planes de comidas, el manejo de excepciones y la persistencia de datos, se evidenció la necesidad de una reestructuración modular. Se implementó una arquitectura basada en clases especializadas coordinadas por un orquestador central, lo que permitió separar claramente las responsabilidades, mejorar la legibilidad del código, facilitar su mantenimiento y fortalecer la capacidad de prueba.

Esta práctica no solo demuestra los beneficios técnicos de aplicar SRP, sino que también fomenta una mentalidad de diseño sostenible y escalable. La experiencia refuerza la importancia de aplicar principios sólidos desde las primeras fases del desarrollo para construir sistemas robustos y fácilmente adaptables.

Palabras clave: Single Responsibility Principle, SRP, refactorización, SOLID, diseño limpio, arquitectura modular, buenas prácticas, mantenibilidad del software.

1 Refactorización Aplicando Principio SOLID: Single Responsibility

1.1 1. Código Original

El siguiente método actualizar_valoracion_con_macros se encontraba en una clase de servicio. Su objetivo era actualizar una valoración con cálculos nutricionales y generación de planes:

```
def actualizar_valoracion_con_macros(self, valoracion: Valoracion) -> Valoracion:
    """
    Calcula y actualiza los macronutrientes y tabla de equivalencias en una valoración.

    Args:
        valoracion: Instancia de Valoración a actualizar

    Returns:
        Valoración actualizada con macronutrientes y tabla de equivalencias

    Raises:
        Exception: Si hay error en el cálculo
    """
    try:
        # Calcular macronutrientes
        macros = self.calcular_macronutrientes(valoracion)

        # Actualizar valoración con macronutrientes
        valoracion.carbohidratos_g = macros['carbohidratos_g']
        valoracion.proteinas_g = macros['proteinas_g']
        valoracion.grasas_g = macros['grasas_g']
        valoracion.calorias_totales = macros['calorias_totales']
        valoracion.recomendaciones = macros['recomendaciones']

        # Guardar primero los macronutrientes
        valoracion.save()

        # Ahora generar el plan de comidas (necesita los macros ya guardados)
        try:
            plan_comidas = self.calcular_plan_comidas(valoracion)
            valoracion.plan_comidas = plan_comidas
            valoracion.save()
            logger.info(f"Plan de comidas generado para valoración ID {valoracion.id}")
        except Exception as e:
            logger.warning(f"Error generando plan de comidas para valoración ID {valoracion.id}: {str(e)}")

        # Ahora generar la tabla de equivalencias (necesita los macros ya guardados)
        try:
            tabla_equivalencias = self.calcular_tabla_equivalencias(valoracion)
            valoracion.tabla_equivalencias = tabla_equivalencias
            valoracion.save()
            logger.info(f"Tabla de equivalencias generada para valoración ID {valoracion.id}")
        except Exception as e:
            logger.warning(f"Error generando tabla de equivalencias para valoración ID {valoracion.id}: {str(e)}")

        logger.info(f"Valoración ID {valoracion.id} actualizada con macronutrientes")
        return valoracion

    except Exception as e:
        logger.error(f"Error actualizando valoración con macros: {str(e)}")
        raise
```

1.2 2. Problemas Detectados

Este método viola el principio SRP por las siguientes razones:

- Realiza múltiples tareas independientes en una misma función.
- Mezcla lógica de negocio, almacenamiento en base de datos y *logging*.
- Tiene múltiples motivos para cambiar: cambios en cálculo, persistencia, *logging*, formato de recomendaciones, etc.

Cada vez que se desee modificar el cálculo de macros, la forma en que se guarda o se genera el plan, habría que modificar este mismo método.

1.3 3. Refactorización Aplicada

En el proceso de revisión del método actualizar_valoracion_con_macros, se identificó que esta función, ubicada dentro del módulo de servicios de valoración, estaba asumiendo múltiples responsabilidades. Su lógica no solo incluía el cálculo de macronutrientes, sino también la actualización directa del objeto de valoración, la generación del plan de comidas, la creación de una tabla de equivalencias nutricionales, el manejo de errores para cada parte del proceso y la escritura de logs para informar el estado del flujo. Esta combinación de tareas en un solo método violaba claramente el principio de responsabilidad única (*Single Responsibility Principle*, SRP), uno de los pilares fundamentales de la programación orientada a objetos bajo la filosofía SOLID.

Este principio establece que una clase o función debe tener una sola razón para cambiar. En este caso, cualquier modificación en el cálculo de los macronutrientes, en la forma de persistir datos, o en la lógica del plan de comidas, implicaría alterar el mismo método. Este nivel de acoplamiento entre funcionalidades distintas hacía que el código fuera difícil de mantener, de probar de forma aislada, y poco flexible ante futuras extensiones del sistema.

Con el fin de resolver esta situación, se aplicó una refactorización basada en el SRP. El primer paso fue reducir la responsabilidad del método original, convirtiéndolo en un simple punto de entrada que delega el procesamiento a un orquestador. En lugar de realizar todas las operaciones internamente, este método ahora instancia una clase llamada OrquestadorValoracion, cuya única responsabilidad es coordinar el proceso completo de actualización de una valoración. Esta delegación no solo reduce la complejidad del método principal, sino que permite que cada tarea sea abordada de forma separada, organizada y controlada.

El orquestador, a su vez, se apoya en tres clases especializadas. La primera es MacronutrientesCalculator, encargada exclusivamente de calcular los macronutrientes a partir de una instancia de valoración. Esta clase actúa como una capa intermedia entre el orquestador y el servicio nutricional ya existente, encapsulando la lógica del cálculo. La segunda clase, PlanComidasGenerator, tiene la única responsabilidad de generar un plan de comidas, también utilizando el servicio original, pero desacoplada del resto de las operaciones. La tercera clase, TablaEquivalenciasGenerator, se dedica a generar una tabla

de equivalencias nutricionales, necesaria para complementar los datos que se presentan al usuario.

Cada una de estas clases fue diseñada para tener una sola función, y pueden ser fácilmente testeadas, modificadas o reutilizadas en otros contextos. Por ejemplo, si en el futuro se desea ajustar la fórmula para el cálculo de proteínas, se podrá hacer dentro de MacronutrientesCalculator sin afectar en absoluto al flujo general orquestado, ni a las demás funcionalidades del sistema. De igual forma, cualquier cambio en el formato del plan de comidas será manejado dentro de PlanComidasGenerator sin necesidad de tocar la lógica que actualiza la base de datos o el objeto de valoración.

Este diseño modular mejora sustancialmente la mantenibilidad del sistema. El código es ahora más legible, más fácil de extender y mucho más robusto frente a cambios. Además, al contar con componentes más pequeños y específicos, se facilita la creación de pruebas unitarias que verifiquen el correcto funcionamiento de cada parte por separado, sin depender de la ejecución de todo el flujo.

En resumen, esta refactorización demuestra cómo la aplicación del principio de responsabilidad única no solo mejora la calidad del código, sino que permite un desarrollo más ágil y controlado. Se pasó de un método monolítico con múltiples responsabilidades a un sistema estructurado, compuesto por clases altamente cohesivas y débilmente acopladas. Esta transición es un claro ejemplo de cómo los principios SOLID, lejos de ser una teoría abstracta, ofrecen soluciones concretas a problemas reales en el diseño de software.

Se separaron las responsabilidades en clases independientes. El método principal fue simplificado para delegar el proceso al orquestador:

Método refactorizado (services.py):

```
def actualizar_valoracion_con_macros(self, valoracion: Valoracion) -> Valoracion:
    """
    Calcula y actualiza los macronutrientes y tabla de equivalencias en una valoración.
    Ahora usa el OrquestadorValoracion que aplica el principio de Single Responsibility.
    """
    from .valoracion_orquestador import OrquestadorValoracion
    orquestador = OrquestadorValoracion()
    return orquestador.actualizar_valoracion_completa(valoracion)
```

1.4 4. Nuevas Clases con SRP

Se creó un archivo nuevo llamado valoracion_orquestador.py que incluye las siguientes clases especializadas:

```
import logging
logger = logging.getLogger(__name__)
```

1.4.1 Clase 1: OrquestadorValoracion

```
class OrquestadorValoracion:
    def __init__(self):
        self.macronutrientes_calculator = MacronutrientesCalculator()
        self.plan_comidas_generator = PlanComidasGenerator()
        self.tabla_equivalencias_generator = TablaEquivalenciasGenerator()

    def actualizar_valoracion_completa(self, valoracion):
        try:
            # Calcular macronutrientes
            macros = self.macronutrientes_calculator.calcular(valoracion)
            self._actualizar_valoracion_con_macros(valoracion, macros)

            # Generar plan de comidas
            try:
                plan_comidas = self.plan_comidas_generator.generar(valoracion)
                valoracion.plan_comidas = plan_comidas
                valoracion.save()
                logger.info(f"Plan de comidas generado para valoración ID {valoracion.id}")
            except Exception as e:
                logger.warning(f"Error generando plan de comidas para valoración ID {valoracion.id}: {str(e)}")

            # Generar tabla de equivalencias
            try:
                tabla_equivalencias = self.tabla_equivalencias_generator.generar(valoracion)
                valoracion.tabla_equivalencias = tabla_equivalencias
                valoracion.save()
                logger.info(f"Tabla de equivalencias generada para valoración ID {valoracion.id}")
            except Exception as e:
                logger.warning(f"Error generando tabla de equivalencias para valoración ID {valoracion.id}: {str(e)}")

            logger.info(f"Valoración ID {valoracion.id} actualizada con macronutrientes")
            return valoracion

        except Exception as e:
            logger.error(f"Error actualizando valoración con macros: {str(e)}")
            raise

    def _actualizar_valoracion_con_macros(self, valoracion, macros):
        valoracion.carbohidratos_g = macros['carbohidratos_g']
        valoracion.proteinas_g = macros['proteinas_g']
        valoracion.grasas_g = macros['grasas_g']
        valoracion.calorias_totales = macros['calorias_totales']
        valoracion.recomendaciones = macros['recomendaciones']
        valoracion.save()
```

1.4.2 Clase 2: MacronutrientesCalculator

```
class MacronutrientesCalculator:
    def __init__(self):
        from .services import NutricionCalculatorService
        self.service = NutricionCalculatorService()

    def calcular(self, valoracion):
        return self.service.calcular_macronutrientes(valoracion)
```

1.4.3 Clase 3: PlanComidasGenerator

```
class PlanComidasGenerator:
    def __init__(self):
        from .services import NutricionCalculatorService
        self.service = NutricionCalculatorService()

    def generar(self, valoracion):
        return self.service.calcular_plan_comidas(valoracion)
```

1.4.4 Clase 4: TablaEquivalenciasGenerator

```
class TablaEquivalenciasGenerator:
    def __init__(self):
        from .services import NutricionCalculatorService
        self.service = NutricionCalculatorService()

    def generar(self, valoracion):
        return self.service.calcular_tabla_equivalencias(valoracion)
```

1.5 5. Beneficios del Refactoring

- Separación de responsabilidades: cada clase tiene un único objetivo claro.
- Escalabilidad: si cambia la lógica de equivalencias o plan de comidas, no afecta a los demás módulos.
- Facilidad de prueba: cada clase puede testearse unitariamente.
- Legibilidad: el código principal es ahora más limpio y expresivo.
- Mantenimiento seguro: menor riesgo de introducir errores al modificar código.

1.6 6. Verificación

- El método actualizar_valoracion_con_macros sigue teniendo la misma firma.
- No se alteró la forma en que otros componentes lo consumen.
- Las nuevas clases especializadas están preparadas para testing.
- Las importaciones circulares se resolvieron reubicando los import dentro de métodos o constructores.

2 Conclusiones

La refactorización del método actualizar_valoracion_con_macros permitió constatar la importancia de aplicar correctamente el principio de responsabilidad única (SRP) en el diseño de software. Al dividir las responsabilidades en clases específicas, se logró un sistema más organizado, mantenible y preparado para la evolución. Este ejercicio evidenció que una arquitectura bien estructurada no solo mejora el código en términos técnicos, sino que también facilita su comprensión, extensión y prueba, reduciendo significativamente la complejidad del mantenimiento a largo plazo.

Uno de los aprendizajes más valiosos del proceso fue comprender que los principios SOLID no son meras recomendaciones teóricas, sino guías prácticas que permiten construir software más robusto y sostenible. Aplicar SRP no solo transformó una función compleja en una composición clara de responsabilidades, sino que también reforzó una mentalidad orientada al diseño limpio. Este enfoque promueve el desarrollo de soluciones más flexibles, reutilizables y coherentes, consolidando así las bases para una ingeniería de software profesional y de alta calidad.

Fowler et al. (1999) Gamma et al. (1994) Larman (2004) Martin (2008) Martin (2017) Palomba et al. (2013) Tufano et al. (2015) Yamashita & Moonen (2013)

3 Bibliografía

Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Larman, C. (2004). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development* (3rd ed.). Pearson Education.

Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall.

Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall.

Palomba, F., Bavota, G., De Penta, M., Oliveto, R., & De Lucia, A. (2013). Detecting bad smells in source code using change history information. *2013 IEEE 28th International Conference on Software Maintenance (ICSM)*, 432-435. https://doi.org/10.1109/ICSM.2013.67

Tufano, M., Palomba, F., Bavota, G., De Penta, M., Oliveto, R., & De Lucia, A. (2015). When and why your code starts to smell bad. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*, 403-414. https://doi.org/10.1109/ICSE.2015.58

Yamashita, A., & Moonen, L. (2013). Do code smells reflect important maintainability aspects? *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 303-313. https://doi.org/10.1109/ASE.2013.6693096