Cohesión en el Diseño de Software

Edilson Avalos Condori

June 13, 2024

1 Introducción

La cohesión es un concepto fundamental en el diseño de software que mide el grado en que los elementos de un módulo están relacionados funcionalmente. Un módulo con alta cohesión significa que sus elementos están estrechamente relacionados, lo que facilita su mantenimiento y reutilización[1].

2 Definición de Cohesión

La cohesión se refiere a la medida en que las responsabilidades de un módulo forman un todo significativo y unido. Cuanto mayor sea la cohesión, más fácil será para los desarrolladores entender, mantener y reutilizar el módulo.

2.1 Conceptos Clave

- **Cohesión Funcional**: Todos los elementos contribuyen a una sola función bien definida.
- **Cohesión Secuencial**: La salida de un elemento es la entrada de otro.
- **Cohesión Comunicacional**: Los elementos operan sobre los mismos datos.
- **Cohesión Procedural**: Los elementos son parte de un mismo procedimiento.
- **Cohesión Temporal**: Los elementos se ejecutan en el mismo período de tiempo.

[1] Pressman, R. S. (2014). *Software Engineering: A Practitioner's Approach*. McGraw-Hill Education.

- **Cohesión Lógica**: Los elementos realizan tareas lógicamente similares.
- **Cohesión Coincidental**: Los elementos están agrupados arbitrariamente.

3 Importancia de la Alta Cohesión

La alta cohesión en el diseño de software tiene varias ventajas:

3.1 Facilidad de Mantenimiento

Los módulos con alta cohesión son más fáciles de entender y modificar, lo que facilita la localización y corrección de errores.

3.2 Reutilización

Los módulos cohesivos son más fáciles de reutilizar en otros contextos, ya que tienen responsabilidades bien definidas y estrechamente relacionadas.

3.3 Comprensibilidad

El código es más fácil de entender y modificar, lo que reduce el tiempo necesario para formar nuevos desarrolladores y permite un desarrollo más rápido y eficiente.

4 Tipos de Cohesión

Existen varios tipos de cohesión, cada uno con diferentes niveles de eficacia y aplicabilidad:

4.1 Cohesión Funcional

Todos los elementos contribuyen a una sola función bien definida. Es el tipo de cohesión más fuerte y deseable.

4.2 Cohesión Secuencial

La salida de un elemento es la entrada de otro. Es adecuada cuando las tareas deben realizarse en una secuencia específica.

4.3 Cohesión Comunicacional

Los elementos operan sobre los mismos datos. Es útil cuando las operaciones están relacionadas por los datos que manipulan.

4.4 Cohesión Procedural

Los elementos son parte de un mismo procedimiento. Este tipo de cohesión es útil cuando las tareas forman parte de un procedimiento mayor.

4.5 Cohesión Temporal

Los elementos se ejecutan en el mismo período de tiempo. Es adecuada cuando las tareas deben ejecutarse juntas debido a restricciones de tiempo.

4.6 Cohesión Lógica

Los elementos realizan tareas lógicamente similares. Es menos fuerte que la cohesión funcional, pero aún es útil cuando las tareas están relacionadas por su naturaleza lógica.

4.7 Cohesión Coincidental

Los elementos están agrupados arbitrariamente. Es el tipo de cohesión más débil y menos deseable, ya que no hay una relación clara entre los elementos del módulo.

Table 1: Tipos de Cohesión y sus Ejemplos

Tipo de Cohesión	Descripción	Ejemplo
Funcional	Todos los elementos contribuyen a una sola función	Clase Calculadora
Secuencial	La salida de un elemento es la entrada de otro	Pipeline de datos
Comunicacional	Operan sobre los mismos datos	Módulo de base de dat
Procedural	Parte de un mismo procedimiento	Algoritmo de ordenaci
Temporal	Se ejecutan en el mismo período de tiempo	Módulo de inicializació
Lógica	Realizan tareas lógicamente similares	Módulo de utilidades
Coincidental	Agrupados arbitrariamente	Módulo misceláneo

5 Ejemplo de Código

A continuación, se muestra un ejemplo de código en Python para ilustrar cómo una clase con alta cohesión tendría todos sus métodos y atributos relacionados con una única tarea o conjunto de tareas estrechamente relacionadas:

```python
class Calculadora:
    def __init__(self):
        self.resultado = 0

    def sumar(self, valor):
        self.resultado += valor
        return self.resultado

    def restar(self, valor):
        self.resultado -= valor
        return self.resultado

    def multiplicar(self, valor):
        self.resultado *= valor
        return self.resultado

    def dividir(self, valor):
        if valor != 0:
            self.resultado /= valor
        else:
            raise ValueError("No se puede dividir por cero")
        return self.resultado

# Ejemplo de uso de la clase Calculadora
calculadora = Calculadora()
print(calculadora.sumar(10))        # Output: 10
print(calculadora.restar(2))        # Output: 8
print(calculadora.multiplicar(3))   # Output: 24
print(calculadora.dividir(4))       # Output: 6
```

6 Estudios de Caso y Aplicaciones Prácticas

6.1 Caso 1: Desarrollo de un Sistema de Gestión de Inventarios

En el desarrollo de un sistema de gestión de inventarios, la cohesión es crucial para asegurar que los módulos encargados de la gestión de productos, órdenes y clientes estén bien definidos y sean fácilmente mantenibles. Según Fenton

y Bieman (2014), un diseño cohesivo en sistemas de inventarios mejora significativamente la eficiencia y la precisión en la gestión de datos[2].

6.2 Caso 2: Aplicaciones Bancarias

En las aplicaciones bancarias, la cohesión en el diseño de módulos como la gestión de cuentas, transacciones y clientes asegura que el sistema sea robusto y seguro. Como menciona Pressman (2014), los módulos cohesivos permiten una mejor gestión de la seguridad y la integridad de los datos en sistemas críticos[3].

7 Referencias

- Pressman, R. S. (2014). *Software Engineering: A Practitioner's Approach*. McGraw-Hill Education.
- Fenton, N. E., & Bieman, J. (2014). *Software Metrics: A Rigorous and Practical Approach*. CRC Press.
- Chidamber, S. R., & Kemerer, C. F. (1994). *A Metrics Suite for Object-Oriented Design*. IEEE Transactions on Software Engineering. Disponible en: `https://www.eso.org/~tcsmgr/oowg-forum/TechMeetings/Articles/OOMetrics.pdf`
- Rubens, P. (2017). *Quality Control in Agile Projects*. IEEE Software.
- Repositorio de GitHub con ejemplos relacionados: `https://github.com/EdilsonAvalosCondori/Ingenier-a-de-software`

[2]Fenton, N. E., & Bieman, J. (2014). *Software Metrics: A Rigorous and Practical Approach*. CRC Press.

[3]Pressman, R. S. (2014). *Software Engineering: A Practitioner's Approach*. McGraw-Hill Education.