Técnica para mejorar la Protección Modular en Sistemas Legados de Software

Technique to enhance Modular Protection in Legacy Software Systems

Nelida Barón Pérez, René Santaolaya Salgado, Blanca Dina Valenzuela Robles
Centro Nacional de Investigación y Desarrollo Tecnológico
Cuernavaca, México
d18ce064@cenidet.tecnm.mx, rene.ss@cenidet.tecnm.mx, blanca.vr@cenidet.tecnm.mx

Resumen — **Ignorar la protección modular y el principio de ocultamiento de información, en relación con las reglas de acceso, durante el desarrollo de aplicaciones orientadas a objetos, genera una forma de deuda técnica. Esto puede llevar a un diseño incorrecto de las entidades de software y, en última instancia, a su fragilidad. En este trabajo de investigación, se presenta una técnica de refactorización cuyo objetivo es identificar y asignar el modificador de acceso correcto a cada uno de los métodos de las clases de objetos que conforman una aplicación escrita en lenguaje Java. Para la evaluación del proceso de refactorización se aplicaron cinco métricas de calidad, tanto al código fuente original como al código fuente refactorizado. Estas métricas miden el nivel de protección modular en distintos niveles de acceso, incluyendo la protección de métodos privados, protegidos y amistosos, así como la protección modular general y total. La diferencia en los valores obtenidos entre las dos pruebas de calidad revela el impacto en el encapsulamiento en términos de protección de información.**

Palabras Clave – protección modular;ocultamiento de información; reglas de acceso; refactorización.

Abstract — **Ignoring modular protection and the principle of information hiding, in relation to access rules, during the development of object-oriented applications generates a form of technical debt. This can lead to incorrect design of software entities and, ultimately, their fragility. In this research work, a refactoring technique is presented whose objective is to identify and assign the correct access modifier to each of the methods of the object classes that make up a Java application. For the evaluation of the refactoring process, five quality metrics were applied to both the original source code and the refactored source code. These metrics measure the level of modular protection at different levels of access, including the protection of private, protected, and package-friendly methods, as well as general and overall modular protection. The difference in the values obtained between the two quality tests reveals the impact on encapsulation in terms of information protection.**

Keywords - modular protection; information hiding; visibility rules; refactoring.

I. INTRODUCCIÓN

Durante la etapa de desarrollo de software, es común que la falta de experiencia, habilidad y conocimiento en el paradigma de programación orientada a objetos conduzcan a tomar decisiones incorrectas en el diseño e implementación, lo que resulta en la presencia de *code smell* en las aplicaciones. El *code smell* no necesariamente significa que son errores, pero esto dificulta el mantenimiento y la evolución de las aplicaciones, como lo han demostrado las investigaciones [1], [2], [3] y [4]. Las decisiones de diseño incorrectas pueden provocar un mal funcionamiento y errores en un futuro próximo, tal como lo señalan Singh y Kaur en [5]. En este contexto, el code smell genera lo que se conoce como deuda técnica de diseño, un concepto introducido por Ward Cunningham en 1992 [6]. La deuda técnica hace referencia a aquellas aplicaciones que cumplen con sus objetivos, pero no lo hace de manera eficiente, lo que resulta en una reducción de su tiempo de vida útil. Es decir, presenta deficiencias que deben abordarse a tiempo, ya que, al igual que cualquier deuda, si no se atiende con prontitud, los intereses a pagar serán mayores.

En este artículo se presenta un trabajo de investigación que se enfoca a desarrollar una técnica de refactorización, cuyo objetivo es mejorar la protección modular de las arquitecturas de clases de objetos al proteger sus métodos. Para lograr esto, es necesario que todos los métodos integrados en cada clase de objetos, dentro de una arquitectura de software, tengan el modificador de acceso correcto según los cuatro niveles de acceso: 1) atributos y métodos privados (*private*), que son los más protegidos y solo se pueden acceder y utilizar internamente dentro de la misma clase de objetos; 2) atributos y métodos protegidos (*protected*), que solo pueden ser utilizados por métodos de la misma clase o clases de objetos derivadas; 3) atributos y métodos amistosos (*friendly*), que son accedidos y utilizados por métodos de cualquier clase de objetos dentro del mismo paquete; 4) atributos y métodos públicos (*public*), que son los menos restrictivos y pueden ser utilizados por cualquier método de cualquier clase de objetos en el sistema de software. Con el fin de verificar los niveles de acceso, se utilizan cinco métricas de calidad descritas en el presente artículo, que miden el grado de protección modular en los diferentes niveles de acceso.

Para probar y validar la técnica de refactorización, se aplicó en tres casos de estudio. Como primer paso, se realizó el cálculo del conjunto de métricas propuestas al código fuente original, en el segundo paso se aplicó la técnica de refactorización para la protección modular, y en el tercer paso se volvió a evaluar la protección modular al código refactorizado con objeto de

verificar la mejora. Los resultados obtenidos reflejan mejoras notables en la protección modular en los tres casos de prueba, con un aumento del 22.22% en el primer caso, un 21.15% de mejora en el segundo caso, y un incremento del 29.78% en el tercer caso. Estos resultados indican que la técnica de refactorización es efectiva para mejorar la protección modular en arquitecturas de clases de objetos de sistemas de software.

Este artículo está organizado de la siguiente manera. En la Sección II, se describe en detalle la técnica de refactorización, enfocándose en los criterios considerados para asignar los modificadores de acceso correctos a los métodos de cada clase de objetos. La Sección III presenta la descripción de las métricas desarrolladas en la investigación. En la Sección IV, se muestra la evaluación y los resultados obtenidos de los casos de prueba. Finalmente, en la Sección V, se concluyen las lecciones aprendidas, los hallazgos encontrados derivados de los resultados y se presentan las perspectivas futuras para esta investigación.

II. PROCESO DE REFACTORIZACIÓN

La Figura 1 representa el proceso general de refactorización para mejorar la protección modular, el cual se compone de cuatro actividades principales. Cada una de estas actividades se describe en detalle en la subsección A.

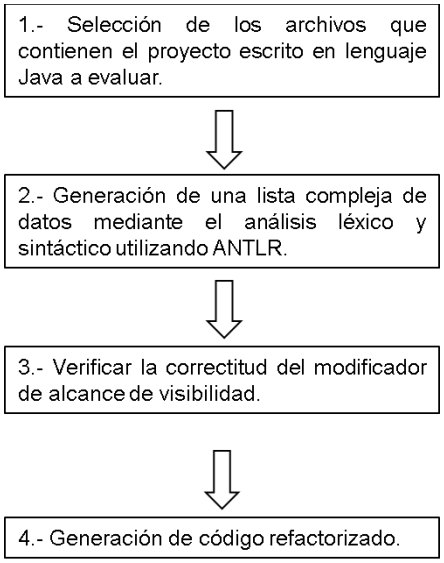

Figura 1. Esquema gráfico general del proceso de refactorización

A. Descripción de actividades del proceso de refactorización

1) Selección de los archivos que contienen el proyecto a evaluar escrito escrito en lenguaje Java: El proceso de refactorización se inicia al recibir como entrada los archivos con el código original escrito en lenguaje Java del proyecto a ser refactorizado. Para cargar estos archivos, se utiliza una interfaz gráfica.

2) Generación de una lista compleja de datos mediante el análisis léxico y sintáctico utilizando ANTLR: En esta etapa, los archivos recibidos en la actividad anterior se someten a un análisis léxico, sintáctico y semántico utilizando la herramienta ANTLR. Esta es una herramienta escrita en Java que genera analizadores o compiladores [7]. Su función principal es generar un analizador léxico, sintáctico y semántico basado en una gramática específica. Para este proyecto de investigación, se utiliza la gramática del lenguaje Java en su versión 8. El analizador léxico produce la clase de objetos Lexer.java, que contiene los elementos léxicos del lenguaje definidos mediante tokens, tales como palabras reservadas, comentarios, signos, etc. El analizador sintáctico crea la clase de objetos Parser.java, que contiene la estructura sintáctica del lenguaje por medio de reglas de producción, en el análisis semántico se crea un AST (Abstract Syntax Tree) que sirve para manejar la información semántica del código de entrada, obteniendo la información que conforma cada una de las clases de objetos que se recibieron como entrada en la actividad 1. Este árbol representa cada una de las clases de objetos recibidas en la actividad anterior y toda la información sobre los atributos y métodos contenidas en éstas. La información de cada clase de objetos del código bajo estudio es registrada en otra clase de objetos de nombre "oClase", para su posterior uso en las siguientes etapas del proceso.

Como resultado del análisis de la información, se obtiene una lista compleja conformada por instancias de la "oClase", con los datos recopilados. La Fig. 2 muestra una representación gráfica parcial de la estructura resultante.

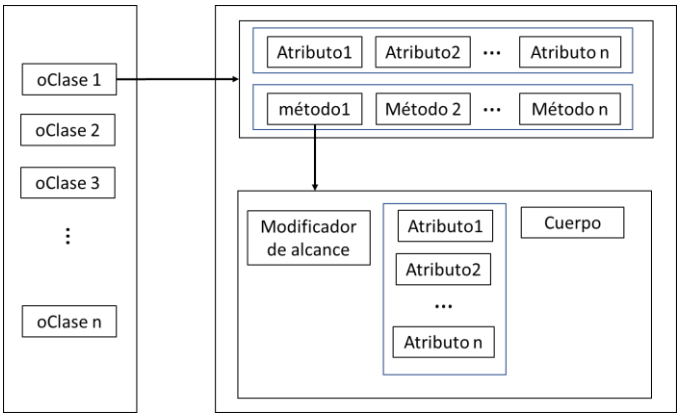

Figura 2. Representación parcial de la lista compleja

3) Verificar la correctitud del modificador de acceso: En esta etapa, se utiliza como entrada la lista compleja obtenida en la actividad 2. Esta lista contiene la información acerca de los métodos presentes en cada clase de objetos de la arquitectura del código original. A través de esta información, se realiza un seguimiento de invocación a cada método en todas las clases de objetos, con el fin de identificar el modificador de acceso para cada método.

La asignación del modificador de acceso correcto se realiza siguiendo los criterios específicos siguientes:

1. Todos los métodos localizados en una interfaz son calificados como *public*.
2. Todos los métodos abstractos localizados en clases de objetos abstractas son calificados como *public*.
3. El método principal (*main*) es calificado como *public*.
4. Los métodos utilizados por métodos de clases de objetos de diferente paquete son calificados como *public*.

5. Los métodos sobre-escritos que se implementan en clases de objetos derivadas heredan el modificador de acceso del método definido en la clase de objetos base.
6. Todos los métodos utilizados en el alcance de la misma clase de objetos son calificados como *private*.
7. Todos los métodos utilizados únicamente por otros métodos y que están localizados en clases de objetos del mismo paquete son calificados como *friendly*.
8. Todos los métodos implementados en una clase de objetos base, que son utilizados únicamente por métodos de clases de objetos derivadas son calificados como *protected*.
9. Todos los métodos constructores son calificados como *public*.
10. Todos los métodos implementados en una clase de objetos base y utilizados por clases de objetos derivadas de diferente paquete son calificados como *protected*.
11. Reglas de precedencia de modificadores de acceso:
    - Los métodos invocados por métodos de otra clase de objetos de diferente paquete, deben ser calificados como *public*.
    - Los métodos que no son invocados por métodos de otra clase de objetos de diferente paquete, pero si son invocados por métodos de otra clase de objetos del mismo paquete, deben ser calificados como *friendly*.
    - Los métodos que no son invocados por métodos de otra clase de objetos de diferente o del mismo paquete, pero que sí son invocados por métodos de clases de objetos derivadas deben ser calificados como *protected*
    - Los métodos que no son invocados por métodos de clase de objetos de diferente paquete o del mismo paquete, ni clases de objetos derivadas, pero que si son invocados por métodos de la misma clase de objetos deben ser calificados como *private*.

Los ajustes del modificador de acceso en todos los métodos identificados se realizan directamente en la lista compleja. De esta forma, se realizan los cambios sin generar impacto en el funcionamiento de la aplicación original.

4) Generación de código refactorizado: En esta actividad, se utiliza el código modificado presente en la lista compleja obtenida durante el proceso de refactorización. Utilizando la plantilla "*StringTemplate*" (ST), se generar el código refactorizado que corresponde a la aplicación original. La Fig. 3 muestra un fragmento de la plantilla utilizada para la generación del código.

```
package <paquete>;

<clase.importaciones :{ imp | <imp><\n>}>

  <clase.notacionDeClase>
  public <if(clase.abstracta)>abstract class <endif>
<if(clase.esInterfaz)>interface <endif>
<if(!clase.abstracta && !clase.esInterfaz )>
 class <endif><clase.nombre><clase.parametros>
<if(clase.heredaDeClase)> extends <clase.clasePadre><endif>
<if(clase.implementaClase)> implements <clase.claseImple><endif> {
```

Figura 3. Fragmento de plantilla ST.

Como resultado de todo el proceso de refactorización, se generan nuevos archivos Java, los cuales contienen los ajustes de protección modular aplicados a los métodos de las clases de objetos originales. Estos archivos mantienen el mismo comportamiento que tenían antes de la refactorización, en cumplimiento con la característica principal de la refactorización.

III. MÉTRICAS DE PROTECCIÓN MODULAR

Se emplearon cinco métricas para analizar el efecto en la protección modular de arquitecturas de software orientadas a objetos. Con este fin, se consideraron los distintos niveles de acceso establecidos por el lenguaje Java. Las métricas utilizadas son: PMP (Protección de Métodos Privados), PMPr (Protección de Métodos Protegidos), PMF (Protección de Métodos Amistosos), PM (Protección Modular) y PMT (Protección Modular Total). En estas métricas, un valor cercano a 1 indica un alto grado de protección, mientras que un valor cercano a 0 indica un bajo nivel de protección modular.

La métrica PMFP mide el grado de protección modular con respecto a los métodos que han sido declarados con el modificador de acceso *private*. La expresión matemática de la métrica PMFP se muestra a continuación:

$$\mathrm{PMP} = \frac{\sum_{ci=1}^{ci=n}\left(\frac{\sum_{fi=0}^{fi=m} FP}{FTC}\right)}{NTC} \quad (1)$$

Donde:

FP	Métodos con modificador *private*.
Fi	"i-esimo" método.
FTC	Número total de métodos de la clase de objetos.
Ci	"i-esima" clase de objetos.
NTC	Número total de clases de objetos.

La métrica PMFPr mide el grado de protección modular con respecto a los métodos que han sido declarados con el modificador de acceso *protected*. La expresión matemática de la métrica PMFPr se muestra a continuación:

$$\mathrm{PMPr} = \frac{\sum_{ji=1}^{ji=n}\left(\frac{\sum_{fi=0}^{fi=m} FPr}{NTF}\right)}{NTJ} \quad (2)$$

Donde:

FPr	Métodos con modificador *protected*.
Fi	"i-esimo" método *protected* de la jerarquía.
NTF	Número total de métodos de la jerarquía.
ji	"i-esima" jerarquía.
NTJ	Número total de jerarquías.

PMF mide el grado de protección modular con respecto a los métodos carecen de un modificador de acceso específico, comúnmente conocidos como métodos *friendly*. La expresión matemáticas de la métrica PMFF se muestra a continuación:

$$\mathrm{PMF} = \frac{\sum_{i=0}^{i=n} FF}{NTF} \quad (3)$$

Donde:
FF Métodos con modificador *friendly* o *default*.
i "i-esimo" método *Friendly* o *default*.
NTF Número total de métodos.

La métrica PM mide el grado de protección modular de sistemas de software legado. La expresión matemática de la métrica PM se muestra a continuación:

$$\mathrm{PM} = \frac{\sum_{i=0}^{i=n} \mathrm{FNP}}{\mathrm{TF}} \quad (4)$$

Donde:
FNP Métodos que no son *public*.
N, TF Número total de métodos.

La métrica PMT mide el grado de protección modular total que tiene una arquitectura de clases de objetos. La expresión matemática de la métrica TPM se muestra a continuación:

$$\mathrm{PMT} = \frac{((\mathrm{PMP}*1) + (\mathrm{PMPr}*0.75) + (\mathrm{PMF}*0.25))}{NTM} \quad (5)$$

Donde:
PMP Grado de protección modular de métodos *private*.
PMPr Grado de protección modular de métodos *protected*.
PMF Gado de protección modular de métodos *friendly*.
NTM Número total de métodos.

IV. EVALUACIÓN DE LA TÉCNICA DE REFACTORIZACIÓN

En esta sección, se describe la evaluación de la técnica de refactorización de modificadores de acceso. Como ya se mencionó, se utilizaron tres aplicaciones de software como casos de prueba para la evaluación de la técnica de refactorización propuesta. En la Tabla I se muestra la descripción general de las aplicaciones seleccionadas para los casos de prueba y en la Tabla II se presentan los tipos y cantidad de modificadores de acceso encontrados en cada una de las aplicaciones antes de la refactorización.

TABLA I. DESCRIPCIÓN GENERAL DE LAS APLICACIONES UTILIZADAS PARA LA EVALUACIÓN DE LA TÉCNICA DE REFACTORIZACIÓN

Id Aplicación	*Objetivo*	*No. de clases de objetos*
Ap1	Realizar operaciones con listas como son: insertar, eliminar, recorrer, etc.	9
Ap2	Medir los tiempos que una persona utiliza en realizar tareas cotidianas.	52
Ap3	Realizar cálculos estadísticos automáticamente, como son: moda, varianza, mediana, etc.	47

TABLA II. CARACTERÍSTICAS DE AP1, AP2 Y AP3 ANTES DE LA REFACTORIZACIÓN

Id Aplicación	*No. Métodos private*	*No. Métodos protected*	*No. Métodos friendly*	*No. Métodos public*	*No. total de métodos*
Ap1	1	0	19	58	78
Ap2	7	7	0	171	185
Ap3	0	0	0	151	151

Antes de aplicar la técnica de refactorización a las aplicaciones Ap1, Ap2 y Ap3, se llevó a cabo una etapa de compilación y ejecución previa para garantizar la ausencia de errores de sintaxis o defectos en su construcción. Además, se aplicaron dichas métricas a cada uno de los casos de prueba con el fin de comparar los valores obtenidos por las métricas antes y después del proceso de refactorización. Los resultados de las métricas para las aplicaciones Ap1, Ap2 y Ap3 antes de la refactorización se muestran en las Tablas III, IV y V respectivamente.

TABLA III. RESULTADO DE LA APLICACIÓN DE LAS MÉTRICAS DE PROTECCIÓN MODULAR A AP1 ANTES DE LA REFACTORIZACIÓN

Métrica	*Sustitución*	*Resultado*
PMP	PMP= 0.08/9	0.009
PMF	PMPr = (0/19 + 0/22) /2	0
PMPr	PMF = 19/78	0.244
PM	PM = 20/78	0.256
TPM	PMT= ((0.009*1) + (0*0.75) + (0.244*0.25))/78	0.0008

TABLA IV. RESULTADO DE LA APLICACIÓN DE LAS MÉTRICAS DE PROTECCIÓN MODULAR A AP2 ANTES DE LA REFACTORIZACIÓN

Métrica	*Sustitución*	*Resultado*
PMP	PMP= 2.333/52	0.045
PMF	PMPr = 0.022/40	0.022
PMPr	PMF = 0/185	0
PM	PM = 14/185	0.076
TPM	PMT=((0.045*1) + (0.022*0.75) + (0*0.25))/185	0.0003

TABLA V. RESULTADO DE LA APLICACIÓN DE LAS MÉTRICAS DE PROTECCIÓN MODULAR A AP3 ANTES DE LA REFACTORIZACIÓN

Métrica	*Sustitución*	*Resultado*
PMP	PMP= 0/47	0
PMF	PMPr = 0/29	0
PMPr	PMF = 0/151	0
PM	PM = 0/151	0
TPM	PMT=((0*1) + (0*0.75) + (0*0.25))/151	0

Una vez confirmada la compilación sin errores de las aplicaciones y tras aplicar las métricas se llevó a cabo el proceso de refactorización de las aplicaciones Ap1, Ap2 y Ap3. Al finalizar la refactorización, se realizó nuevamente el conteo de métodos en cada aplicación, considerando sus diferentes niveles de protección. Los resultados de este conteo se encuentran presentados en la Tabla VI. También, se aplicaron las métricas al código refactorizado de las aplicaciones Ap1, Ap2 y Ap3. Los resultados correspondientes se muestran en las Tablas VII, VIII y IX, respectivamente.

TABLA VI. CARACTERÍSTICAS DE AP1, AP2 Y AP3 DESPUÉS DE LA REFACTORIZACIÓN

Id Aplicación	*No. Métodos private*	*No. Métodos protected*	*No. Métodos friendly*	*No. Métodos public*	*No. total de métodos*
Ap1	6	0	14	58	78
Ap2	13	43	6	123	185
Ap3	30	11	31	79	151

TABLA VII. RESULTADO DE LA APLICACIÓN DE LAS MÉTRICAS DE PROTECCIÓN MODULAR A AP1 DESPUÉS DE LA REFACTORIZACIÓN

Métrica	*Sustitución*	*Resultado*
PMP	PMP= 0.08/9	0.085
PMF	PMPr = (0/19 + 0/22)/2	0
PMPr	PMF = 19/78	0.179
PM	PM = 20/78	0.256
TPM	PMT=((0.085*1)+(0*0.75)+(0.179* 0.25))/78	0.0016

TABLA VIII. RESULTADO DE LA APLICACIÓN DE LAS MÉTRICAS DE PROTECCIÓN MODULAR A AP2 DESPUÉS DE LA REFACTORIZACIÓN

Métrica	*Sustitución*	*Resultado*
PMP	PMP= 2.496/52	0.048
PMF	PMPr = 5.28/40	0.132
PMPr	PMF = 6/185	0.032
PM	PM = 62/185	0.335
TPM	PMT= ((0.048*1) +(0.132*0.75) +(0.032*0.25))/185	0.0008

TABLA IX. RESULTADO DE LA APLICACIÓN DE LAS MÉTRICAS DE PROTECCIÓN MODULAR A AP3 DESPUÉS DE LA REFACTORIZACIÓN

Métrica	*Sustitución*	*Resultado*
PMP	PMP= 3.272/47	0.07
PMF	PMPr = 0.029/29	0.001
PMPr	PMF = 40/151	0.265
PM	PM = 72/151	0.477
TPM	PMT= ((0.07*1) + (0.001*0.75) + (0.265*0.25))/151	0.0009

Los resultados de la aplicación de las métricas proporcionan información relevante sobre las aplicaciones de software antes y después de la refactorización.

Ap1: En esta aplicación hubo un cambio en la protección de cinco métodos con modificadores de acceso *friendly* a *private*, lo que disminuyó la protección a nivel *friendly* de la arquitectura de 24.4% a 17.9%, sin embargo, se observa un aumento en la protección a nivel privado (*private*), con un valor más cercano a 1, lo que indica una mejora en la protección privada de toda la arquitectura, pasando del 0.9% al 8.5%. Por otro lado, la protección a nivel *protected* se mantuvo en 0% antes y después de la refactorización, ya que esta aplicación Ap1 no cuenta con métodos con modificador de acceso *protected*. La protección modular (PM) se mantuvo sin cambios puesto que no hubo aumentos ni disminuciones en los métodos *public*. La métrica de protección modular total (PMT) aumentó en un 0.08%, pasando de 0.08% a 0.16%, lo cual indica que la arquitectura de Ap1 aumentó su grado total de protección modular debido al incremento de métodos privados después de la refactorización. La Fig. 4 muestra una comparativa de antes y después de la refactorización de AP1.

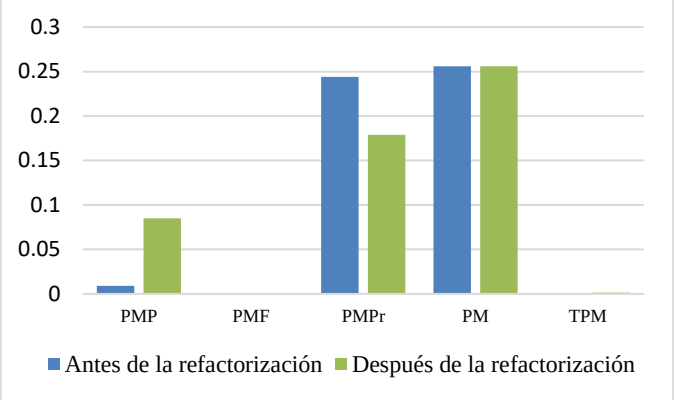

Figura 4. Comparativa de métricas de protección modular antes y después de la refactorización de AP1.

Ap2: Los resultados muestran un incremento en los niveles de protección en todas las categorías, acercándose a valores óptimos de 1. La protección a nivel *private* en la arquitectura aumentó de 4.5% a 4.8%, la protección a nivel *protected* aumentó de 2.2% a 13.2%, y la protección a nivel *friendly* aumentó de 0% a 3.2%. Asimismo, la protección modular experimentó un aumento significativo, pasando de 7.6% a 33.5%. La métrica PMT también tuvo un incremento de 0.05%, pasando de 0.03% a 0.08%. La Fig. 5 muestra una comparativa de antes y después de la refactorización de AP2.

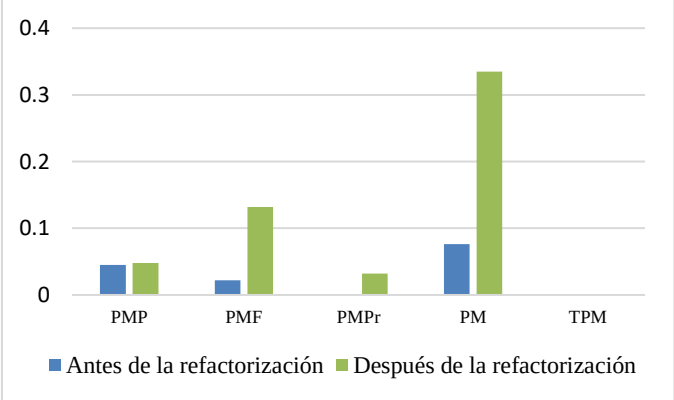

Figura 5. Comparativa de métricas de protección modular antes y después de la refactorización de AP2.

Ap3: Los niveles de protección en esta arquitectura presentaron un aumento en todos los niveles de protección, acercándose al valor óptimo de 1. La protección a nivel *private* tuvo un incremento de 0% a 7%, la protección a nivel *protected* tuvo un aumento de 0% a 0.1%, y la protección a nivel *friendly* aumentó de 0% a 26.5%. Estos incrementos en protección se reflejan a nivel modular incrementado su protección de 0% a 47.7%. Este incremento se debe a que se han cambiado los

modificadores de acceso a 72 de los 151 métodos que originalmente estaban declarados como *public*. Como es de esperarse, la métrica PMT, ha experimentado un aumento de 0.09%, pasando de 0% a 0.09%. La Fig. 6 muestra una comparativa de antes y después de la refactorización de AP3.

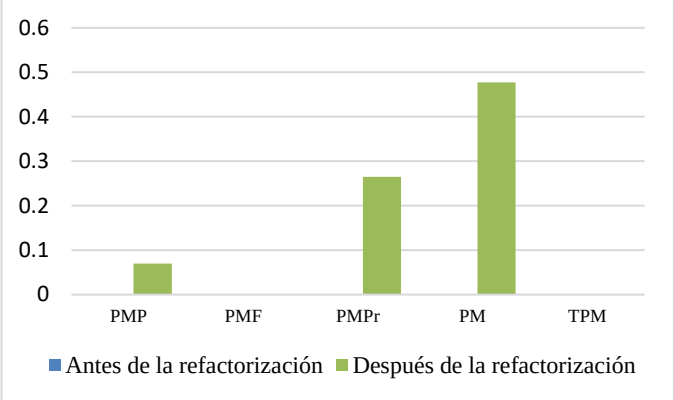

Figura 6. Comparativa de métricas de protección modular antes y después de la refactorización de AP3.

En resumen, los resultados revelan que Ap1, Ap2 y Ap3 experimentaron una mejora en su nivel de protección modular total debido a la reducción de métodos públicos.

V. CONCLUSIONES

El pertinente uso de los modificadores de acceso y el principio de ocultamiento de información tiene un impacto significativo en el nivel de encapsulamiento en términos de protección modular. Al restringir el acceso a la información privada solo desde el interior de los objetos instanciados y evitar su acceso por parte de agentes externos, previene el mal uso de objetos y se reduce la fragilidad del sistema. La fragilidad puede dar lugar a la propagación de defectos, provocando resultados falsos o incorrectos en las entidades de software que aparentemente funcionan bien. El encapsulamiento permite agrupar el comportamiento interno de un objeto sin afectar el funcionamiento de otros objetos, incluso en presencia de cambios o modificaciones, aún cuando sean del mismo tipo, lo que fortalece la robustez del código.

En resumen, el correcto uso de estos principios y técnicas promueve la integridad y calidad del software, y se recomienda su aplicación en arquitecturas orientadas a objetos para una mayor protección modular del acceso no autorizado entre las diferentes unidades de los sistemas de software y un desarrollo de software más confiable.

Finalmente, la técnica de refactorización para mejorar la protección modular que se describe en este artículo, mejoró aspectos de diseño correspondientes a los niveles de acceso de los métodos de clases de objetos. Esta mejora tiene el potencial de reducir la deuda técnica que se acumula al ignorar la protección modular y el principio de ocultamiento de información, tal como se establece en las reglas de acceso del lenguaje de programación Java. En consecuencia, la técnica de refactorización propuesta contribuye a la correcta aplicación del principio de ocultamiento de información, evitando la manipulación inadvertida de los detalles internos de las entidades de software por parte de agentes externos y mejorando la calidad general del software. Como trabajo futuro se sugiere aplicar la misma técnica de protección modular a las estructuras de datos y atributos presentes en las clases de objetos que conforman arquitecturas orientadas a objetos, incluyendo variables de instancia, variables de clase de objetos y variables de referencia.

VI. REFERENCIAS

[1] B. Walter, F. A. Fontana, and V. Ferme, “Code smells and their collocations: A large-scale experiment on open-source systems,” *Journal of Systems and Software*, vol. 144, pp. 1–21, 2018, doi: 10.1016/j.jss.2018.05.057.

[2] A. Kaur, S. Jain, S. Goel, and G. Dhiman, “Prioritization of code smells in object-oriented software: A review,” *Mater Today Proc*, no. xxxx, 2021, doi: 10.1016/j.matpr.2020.11.218.

[3] V. G. Kumar, *ICICCT 2019 – System Reliability, Quality Control, Safety, Maintenance and Management*. 2019.

[4] G. Lacerda, F. Petrillo, M. Pimenta, and Y. G. Guéhéneuc, “Code smells and refactoring: A tertiary systematic review of challenges and observations,” *Journal of Systems and Software*, vol. 167, 2020, doi: 10.1016/j.jss.2020.110610.

[5] G. Kaur and B. Singh, “Improving the quality of software by refactoring,” *Proceedings of the 2017 International Conference on Intelligent Computing and Control Systems, ICICCS 2017*, vol. 2018-Janua, pp. 185–191, 2017, doi: 10.1109/ICCONS.2017.8250707.

[6] W. Cunningham, “The WyCash portfolio management system,” *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA*, vol. Part F1296, no. October, pp. 29–30, 1992, doi: 10.1145/157709.157715.

[7] T. Parr, “The Definitive ANTLR 4 Reference,” 2013. https://www.antlr.org/