EDUCACIÓN
SECRETARÍA DE EDUCACIÓN PÚBLICA

TECNOLÓGICO NACIONAL DE MÉXICO

Tecnológico Nacional de México

Centro Nacional de Investigación y Desarrollo Tecnológico

Tesis de Maestría

Tratamiento de la Deuda Técnica Originada por la Carencia de Protección de Funciones Plantilla de Software Legado

presentada por

Ing. Elías Alejandro Ramírez García

como requisito para la obtención del grado de

Maestro en Ciencias de la Computación

Director de tesis

Dr. René Santaolaya Salgado

Codirectora de tesis

Dra. Blanca Dina Valenzuela Robles

Cuernavaca, Morelos, México. Marzo de 2023.

cenidet
Centro Nacional de Investigación y Desarrollo Tecnológico

Cuernavaca, Mor., 27/marzo/20

OFICIO No. DCC/056/2023
Asunto: Aceptación de documento de tesis
CENIDET-AC-004-M14-OFICIO

CARLOS MANUEL ASTORGA ZARAGOZA
SUBDIRECTOR ACADÉMICO
PRESENTE

Por este conducto, los integrantes de Comité Tutorial de Elías Alejandro Ramírez García, con número de control M21CE022, de la Maestría en Ciencias en de la Computación, le informamos que hemos revisado el trabajo de tesis de grado titulado **"Tratamiento de la deuda técnica originada por la carencia de protección de funciones plantilla de software legado"**, y hemos encontrado que se han atendido todas las observaciones que se le indicaron, por lo que hemos acordado aceptar el documento de tesis y le solicitamos la autorización de impresión definitiva.

René Santaolaya Salgado
Director de tesis

Blanca Dina Valenzuela Robles
Codirectora de Tesis

Olivia Graciela Fragoso Díaz
Revisor 1

Humberto Hernández García
Revisor 2

EDUCACIÓN
CENTRO NACIONAL DE INVESTIGACIÓN Y DESARROLLO TECNOLÓGICO
SERVICIOS ESCOLARES
RECIBIDO
29 MAR 2023

SEP TecNM CENTRO NACIONAL DE INVESTIGACIÓN Y DESARROLLO TECNOLÓGICO
RECIBIDO
29 MAR 2023
SUBDIRECCIÓN ACADÉMICA

C.c.p. Silvia del Carmen Ortiz Fuentes. Depto. Servicios Escolares
Expediente / Estudiante
MYHP/ibm

Cuernavaca, Mor., 29/marzo/2023
No. De Oficio: SAC/051/2023
Asunto: Autorización de impresión de tesis

ELÍAS ALEJANDRO RAMÍREZ GARCÍA
CANDIDATO AL GRADO DE MAESTRO EN CIENCIAS DE LA COMPUTACIÓN
P R E S E N T E

Por este conducto, tengo el agrado de comunicarle que el Comité Tutorial asignado a su trabajo de tesis titulado **"TRATAMIENTO DE LA DEUDA TÉCNICA ORIGINADA POR LA CARENCIA DE PROTECCIÓN DE FUNCIONES PLANTILLA DE SOFTWARE LEGADO"**, ha informado a esta Subdirección Académica, que están de acuerdo con el trabajo presentado. Por lo anterior, se le autoriza a que proceda con la impresión definitiva de su trabajo de tesis.

Esperando que el logro del mismo sea acorde con sus aspiraciones profesionales, reciba un cordial saludo.

A T E N T A M E N T E
Excelencia en Educación Tecnológica®
"Conocimiento y tecnología al servicio de México"

EDUCACIÓN | TECNOLOGICO NACIONAL DE MÉXICO
CENTRO NACIONAL DE INVESTIGACIÓN Y DESARROLLO TECNOLÓGICO
SUBDIRECCIÓN ACADÉMICA

CARLOS MANUEL ASTORGA ZARAGOZA
SUBDIRECTOR ACADÉMICO

C. c. p. Departamento de Ciencias Computacionales
Departamento de Servicios Escolares

CMAZ/LMZ

DEDICATORIAS

Esta tesis está dedicada a:

A mi padre y madre, por brindarme las herramientas necesarias para ser un hombre libre, autónomo e inalienable. Gracias por todos los momentos en los que me han apoyado para mi formación académica y el amor brindado sin pedir nada a cambio, este trabajo es fruto de lo que ustedes plantaron.

Madre te dedico esta parte de mi vida, gracias por permitirme forjar mis ideales y estar presente en cada éxito y fracaso durante toda mi vida, así como apoyarme en el ámbito académico desde que me llevabas en el carro a la primaria hasta hoy en día brindándome apoyo moral a la distancia durante mi estudio de posgrado. Estoy orgulloso de ser tu hijo.

A ti padre, que has sido mi ejemplo a seguir en la vida, mi inspiración, te agradezco por cada momento en el que me has apoyado en mi formación académica y en la formación de mi persona, por enseñarme a cuidar mis pensamientos, mis palabras, mis actos, mis hábitos y mi carácter, que por esta razón es que soy el arquitecto de mi propio destino. Ser tu hijo es mi mayor orgullo. T:.A:.F:.

También dedico este trabajo a mi abuelita por recordarme siempre que tipo de persona debo ser en sociedad y apoyarme en todo con todo su cariño incondicional.

A mi hermana Regina, por tu paciencia de tenerme como hermano, espero ser motivación para que continues con tu formación profesional como lo haces en este momento.

"Para ser lo que queremos ser…. Además de perseverancia, hay que creer que ya lo somos, nuestra mente acciona al universo cuando vemos al futuro como si fuera el presente" – M. en I. Elías Ramírez Espino.

AGRADECIMIENTOS

Al Tecnológico Nacional de México por ser una institución de excelencia, formación académica y profesional al servicio de México.

Al Centro Nacional de Investigación y Desarrollo Tecnológico por brindarme el espacio y el tiempo necesario para concluir el programa de Maestría en Ciencias de la Computación en dicha Institución.

Al Consejo Nacional de Ciencia y Tecnología (CONACYT) por el apoyo económico brindado durante la realización de mis estudios de Maestría.

Especialmente a mi director de tesis, Dr. René Santaolaya Salgado por su acertada orientación y experiencia científica para que este trabajo de tesis llegará a su culminación.

A mi codirectora, Dra. Blanca Dina Valenzuela Robles por su inestimable ayuda durante mi estadía en CENIDET y por su colaboración para el desarrollo de esta tesis.

A mis revisores, Dra. Olivia Graciela Fragoso Diaz y M.C. Humberto Hernández García por el tiempo y dedicación, observaciones y comentarios en el desarrollo de esta investigación.

A mis profesores en general por su enseñanza profesional y académica.

A mis padres, hermana y abuelita que son pilares de mi vida.

A mis compañeros y amigos que estuvieron conmigo en todo el recorrido de mi estadía en CENIDET por la convivencia que tuvimos, confianza: Enrique, Luis, Merino, Mike, Jonathan, Nancy, Fernando, Sócrates y Alan.

RESUMEN

En el paradigma de programación orientado objetos se hace uso de propiedades como lo son la herencia, la abstracción, la encapsulación y el polimorfismo, así como criterios y principios, tales como "*protección modular*" y "*ocultamiento de información*". Cuando se violan estos criterios o principios en arquitecturas de software que implementan el patrón de diseño "*Template Method*", se tiende a manifestar code smell en el software, debido al bajo grado de protección modular presente en las funciones asociadas al patrón de diseño colocando indiscriminadamente reglas de visibilidad permisivas sobre el alcance de las funciones plantilla, sin respetar el patrón de diseño "*Template Method*" que a su vez generan deuda técnica en el software.

Estas malas decisiones de diseño pueden ocasionar acoplamiento indirecto entre clases lo que produce fragilidad en la arquitectura (originada por la manipulación del código debido a cambios o adición de nuevos requerimientos), los cambios que se propagan en el sistema, pueden ocasionar fallos o datos incorrectos y el sistema se expone a que entidades externas realicen inadecuadamente cambios en el estado de los objetos.

Para evitar estas situaciones problemáticas, en este proyecto de investigación se desarrolló un método[1] de refactorización para mejorar la protección de las funciones plantilla, funciones variantes y funciones invariantes asociadas al patrón de diseño "*Template Method*". Así como su implementación en un sistema escrito en Java para su automatización.

Para cuestión de evaluar del funcionamiento del método de refactorización se desarrollaron cuatro métricas para medir la protección modular del patrón de diseño "*Template Method*": PMMP (Protección Modular de Métodos Plantilla), PMFV (Protección Modular de Funciones Variantes), PMFI (Protección Modular de Funciones Invariantes) y PTTM (Protección Total del Template Method).

[1] Con el fin de no confundir la palabra "método" que se usa tanto en el contexto de métodos de refactorización como en el contexto de los métodos de clases de objetos, tal como el método plantilla que se refiere en el patrón de diseño Template Method, en lo sucesivo usaremos la palabra "función" para referirnos a cualquier método declarado en las clases de objetos y la palabra método en lo que se refiera al método de refactorización, con excepción a las secciones correspondientes a las métricas desarrolladas en esta tesis.

El método de refactorización fue probado en tres sistemas distintos con un tamaño de 30, 67 y 89 clases consecutivamente en los que se obtuvo una mejora del 66.7% en la protección modular de funciones plantilla de las arquitecturas de software.

ABSTRACT

The object-oriented programming paradigm makes use of properties such as inheritance, abstraction, encapsulation and polymorphism, as well as criteria and principles such as "modular protection" and "information hiding". When these criteria or principles are violated in software architectures that implement the "*Template Method*" design pattern, *code smell* tends to manifest itself in the software, due to the low degree of modular protection present in the functions associated to the design pattern, indiscriminately placing permissive visibility rules on the scope of the template functions, without respecting the "*Template Method*" design pattern, which in turn generates technical debt in the software.

These bad design decisions can cause indirect coupling between classes, which produces fragility in the architecture (originated by code manipulation due to changes or addition of new requirements), the changes that propagate in the system, can cause failures or incorrect data and the system is exposed to external entities to inappropriately make changes in the state of the objects.

To avoid these problematic situations, in this research project a refactoring method was developed to improve the protection of template methods, variant functions and invariant functions associated with the "*Template Method*" design pattern. As well as its implementation in a system written in Java for its automation.

In order to evaluate the performance of the refactoring method, four metrics were developed to measure the modular protection of the "*Template Method*" design pattern: PMMP (Modular Protection of Template Methods), PMFV (Modular Protection of Variant Functions), PMFI (Modular Protection of Invariant Functions) and PTTM (Total Protection of the Template Method).

The refactoring method was tested on three different systems with a size of 30, 67 and 89 classes consecutively in which an improvement of 66.7% was obtained in the modular protection of template functions of the software architectures.

Contenido

Capítulo 1.
Introducción

La producción de software de calidad bajo el paradigma orientado a objetos requiere que los desarrolladores implementen una capacidad significativa de comprensión, abstracción y habilidades analíticas para la resolución de problemas prácticos en aplicaciones informáticas (Cervantes O et al., 2016). Para los desarrolladores de software, estas capacidades son difíciles de ejercer, y más aún cuando se aplican en conjunto.

Cuando no se tiene la habilidad para la implementación de estas capacidades, se produce software que carece de calidad y por ende también se produce código mal oliente (code smell) el cual resultará en software difícil de mantener, entender, escalar y también resultará en deuda técnica sobre el código.

El *code smell* se define como *"síntomas de elecciones deficientes de diseño e implementación"* (Tufano et al., 2017) y la deuda técnica son *"las obligaciones futuras del mantenimiento del sistema que son consecuencia de que un desarrollador tome atajos en el ciclo de vida de un sistema de software, violar los estándares establecidos de programación y diseño"* (Ramasubbu & Kemerer, 2021).

Unas de las decisiones mal tomadas por parte de un desarrollador es la falta de un buen encapsulamiento y una buena legibilidad en el código, las cuales están en función del aislamiento y el ocultamiento de un estado exclusivo; es decir el valor de los datos de un objeto de alguna clase, así como la independencia de la clase (independencia para cumplir su meta de valor). La independencia se logra cuando una clase de objetos no requiere de otras clases para cumplir con su meta de valor (alta coherencia de la clase) y cuando una clase tiene sus funciones y datos relacionados entre sí (alta cohesión). Esta mala decisión afecta directamente a la modularidad de unidades de un programa.

En el desarrollo de software existen plantillas que proporcionan soluciones apropiadas a algún problema de diseño de software, estas plantillas se conocen como patrones de diseño. El "*Template Method*" es un patrón de diseño de comportamiento (patrones relacionados con algoritmos y distribución de responsabilidades a objetos), en el cual se definen funciones que son algoritmos comunes a subclases, conocidos como funciones invariantes, y delegando algunos

pasos, en funciones variantes, que pueden cambiar su comportamiento en subclases, y se define la propia función plantilla, cuya tarea es orquestar un proceso algorítmico con la participación de las funciones variantes e invariantes de una superclase abstracta (Gamma et al., 1994).

Cuando existe *code smell* a causa de una mala encapsulación por la carencia de protección de funciones plantilla y sus funciones asociadas en el código legado, resulta un alto grado de dependencia entre clases de un programa (acoplamiento indirecto), por lo que se llega a acortar el tiempo de vida útil del software legado, causando un sistema expuesto a manipulación externa que puede terminar en efectos secundarios imprevistos en el estado de las clases y produciendo datos incorrectos o falsos.

En este trabajo de tesis se presenta el diseño e implementación de un método de refactorización para aumentar la protección modular de funciones plantilla del patrón de diseño "*Template Method*". Para el cumplimiento de este objetivo se implementará el método de refactorización a cada función plantilla ubicada en clases de objetos, para proteger las funciones asociadas con el calificador de alcance correcto de acuerdo a los detalles de implementación del patrón de diseño "*Template Method*" descritos en (Gamma et al., 1994), donde los autores recomiendan que el calificador de alcance utilizado en la función plantilla sea público, ya que, al ser la función orquestadora de un algoritmo, se requiere que pueda ser llamado o accedido por otras clases cliente. Las operaciones variantes (abstractas) deben ser declaradas como miembros protegidos para garantizar que solo sean llamadas por la función plantilla y por funciones de las clases derivadas. Finalmente, el calificador de alcance para las funciones invariantes debe ser privado, para evitar el acceso inadvertido desde el exterior, lo cual produce un acoplamiento indirecto entre las clases participantes. En el contexto particular del lenguaje Java, la función plantilla podrá ser friendly siempre y cuando sus clientes estén en el mismo paquete.

Adicionalmente, también se propuso realizar el diseño y la implementación de métricas para medir la protección modular de las clases que implementan el patrón de diseño Template Method.

Capítulo 2.
Antecedentes

2.1.- Planteamiento del Problema

El problema radica en que el funcionamiento del software legado presenta deuda técnica por su fragilidad. Esta situación se presenta por no aplicar el principio de ocultamiento de información y la inadecuada declaración de los calificadores de alcance para proteger tanto a datos como a funciones de clases del acceso no permitido de entidades externas, quedando el software legado expuesto a manipulación que puede terminar en efectos secundarios imprevistos en el estado de las clases, produciendo datos incorrectos o falsos y, por tanto, se acorta el tiempo de vida útil del software. Una situación particular es la protección de funciones variantes e invariantes asociadas a las funciones plantilla conforme lo estipula el patrón de diseño "*Template Method*".

2.2.- Solución Propuesta

La solución propuesta para este proyecto de tesis, fue desarrollar un método de refactorización para software legado escrito en java, aplicado a las funciones del patrón de diseño "*Template Method*".

Este método identifica la o las clases que implementan el patrón de diseño "*Template Method*" y trabaja haciendo una reestructuración de los calificadores de alcance en las funciones plantilla software legado. El calificador de alcance "public" o "friendly" (según sea el alcance de la llamada efectuada por el cliente en términos de paquetes) es aplicado en las funciones plantilla (funciones orquestadoras del patrón de diseño), el calificador de alcance "private" para las funciones privadas que únicamente son usadas dentro de esa misma clase por la función plantilla, y el calificador "protected" se aplica a las funciones que se tienen que implementar en clases derivadas.

Adicionalmente el proyecto de tesis incluye un análisis para medir el grado de protección modular de las funciones plantillas y asociadas.

El método propuesto fue implementado en un sistema de software para automatizarlo, donde el usuario selecciona la carpeta de ubicación que contiene los

archivos del código fuente, escrito en lenguaje Java, que se requiere evaluar y mejorar el grado de protección modular de sus funciones plantilla.

A petición del usuario se corrigen los defectos de ocultamiento de información con el calificador de alcance correcto en las funciones asociadas al patrón de diseño "*Template Method*", de cada clase plantilla[2] que implemente dicho patrón de diseño, esto es conducido por el "*criterio de protección modular*", el "*principio de ocultamiento de datos*" y el patrón de diseño "*Template Method*". Finalmente se mide nuevamente el grado de estas métricas en las funciones plantilla refactorizadas y se alerta al usuario sobre el grado de mejora en estas dimensiones.

El proceso propuesto para la aplicación del método será:

1. Seleccionar archivos de Java.
2. Realizar un análisis sintáctico y semántico del código fuente del software legado para extraer la información necesaria tanto para el cálculo de las métricas como para el proceso de refactorización.
3. Calcular el grado inicial de protección modular de las funciones plantilla.
4. Aplicar la refactorización al código legado para corregir el ocultamiento de las funciones plantilla y sus funciones asociadas, presentes en la arquitectura del sistema legado. La refactorización consiste en ajustar la estructura de datos contenedora de información (Lista), apropiadamente, para que represente la arquitectura deseada, cambiando los calificadores de alcance de las funciones involucradas en la función plantilla.
5. Generar código refactorizado.
6. Recalcular el grado de protección resultante de las funciones plantilla y sus funciones asociadas para verificar la mejora.

2.3.- Justificación

Una unidad de software con alta protección modular, favorece su grado de encapsulado, mientras que una unidad de software que carece de protección modular demerita el nivel correcto de encapsulado e incrementa su complejidad.

El ocultamiento de información tanto de datos como de funciones de un módulo resulta en un aumento de su protección modular.

[2] En el contexto de este trabajo de tesis en lo sucesivo nos referiremos como "clase plantilla" a aquella clase que implementa el patrón de diseño "Template Method".

Actualmente no se cuenta con un método de refactorización que mejore la protección de funciones plantilla. Cuando una clase de objetos no cuenta con protección en sus funciones plantilla, se crea un acoplamiento indirecto entre las clases, generando situaciones en las que las clases relacionadas necesitan comprender los detalles internos de las demás, los cambios se propagarán en el sistema y se presentará el síntoma de fragilidad en el sistema. Se dice que un sistema es frágil cuando tiende a generar fallas ante los cambios, por lo que hace complicada la actividad de mantenimiento del software.

De igual forma en la actualidad no se cuenta con métricas para medir el grado de protección modular en las funciones plantilla y asociadas del "*Template Method*", por lo que se incluye el diseño y la implementación de métricas sustentadas en la teoría de la medición.

La herramienta SR2-Refactoring en su estado actual no abarca la refactorización para mejorar la protección modular y el encapsulamiento de las funciones plantillas y sus funciones asociadas, por lo que se extendió para dar este servicio de refactorización.

2.4.- Objetivo

2.4.1.- Objetivo General

El objetivo general de este trabajo de tesis es evitar el cambio de estado de los objetos de sistemas legados de software realizado inadecuadamente por entidades externas, por medio del uso de funciones distinguidas de acceso abierto (public o friendly) integradas en la implementación del patrón de diseño "*Template Method*".

2.4.2.- Objetivos Específicos

- Mejorar la protección de funciones plantilla en el software legado escrito en el lenguaje de programación Java.
- Mejorar el encapsulamiento de funciones plantilla del software legado escrito en el lenguaje de programación Java.
- Medir la protección modular de funciones plantilla en el software legado escrito en el lenguaje de programación Java.

2.5.- Estado del Arte

2.5.1.- Antecedentes

En el CENIDET se han desarrollado proyectos de reingeniería y refactorización para mejorar las arquitecturas de software legado escritos en C++ y Java.

"La refactorización es un proceso que modifica a un sistema de software para mejorar la estructura interna del código sin dañar o alterar el comportamiento de éste" (Martin Fowler & Beck, 2018).

El proceso de refactorización es una forma más organizada de mejorar y limpiar código, cuando el código es refactorizado existen pocas probabilidades de introducir defectos.

El SR2-Refactoring (Sistema de reingeniería para reúso) es la herramienta principal de antecedente en la cual se encuentran métodos resultantes de varios proyectos de tesis del CENIDET (Centro Nacional de Investigación y Desarrollo Tecnológico) antes mencionados, esta herramienta fue escrita en el lenguaje de programación Java, pero soporta la mejora de arquitecturas de sistemas legados en lenguajes Java y C++. Actualmente el SR2-Refactoring implementa seis métodos de refactorización.

Los proyectos de tesis que implementan los métodos y métricas implementadas en el SR2-Refactoring son los siguientes:

- Reestructuración de código legado a partir del comportamiento para la generación de componentes reutilizables, desarrollado por Cesar Bustamante Laos (Bustamante Laos, 2003).
- Factorización de funciones hacia métodos de plantilla, desarrollado por Laura Alicia Hernández Moreno (Hernández Moreno, 2003).
- Reestructuración de software escrito por procedimientos conducido por patrones de diseño composicionales, desarrollado por Armando Méndez Morales (Méndez Morales Armando, 2004).
- Refactorización de marcos orientados a objetos para reducir el acoplamiento aplicando el patrón de diseño mediator, desarrollado por Leonor Adriana Cárdenas Robledo (Cárdenas Robledo, 2004).
- Método de refactorización de marcos de aplicaciones orientados a objetos por la separación de interfaces, desarrollado por Manuel Alejandro Valdés Marrero (Valdés Marrero, 2004).
- Método de refactorización de software legado para desacoplar el código funcional de la vista del código funcional de la aplicación, desarrollado por Erika Yesenia Ávila Melgar (Ávila Melgar, 2006).
- Método de refactorización de código java con interfaces y abstracciones incorrectas, desarrollado por Pablo Padilla Salgado (Padilla Salgado, 2019).
- Métodos de refactorización de arquitecturas de software con carencia de abstracciones, desarrollado por Ricardo Federico Tello Díaz (Tello Díaz, 2019).
- Refactorización de código para reducir el acoplamiento entre clases relacionadas por herencia de implementación en arquitecturas orientadas a objetos, desarrollado por Orlando Ortiz Gutiérrez (Ortiz Gutiérrez, 2020).
- Método de refactorización para mejorar la protección modular de arquitecturas orientadas a objetos de sistemas de software existentes, desarrollado por Nélida Barón Pérez (Barón Pérez, 2020).
- Métodos de refactorización de código Java para mejorar su modularidad y reducir las dependencias entre clases de objetos, desarrollada por Marisol Ramírez Cruz (Ramírez Cruz, 2022).

El principal trabajo de investigación desarrollado en el cuerpo académico de Ingeniería de Software del DCC que es antecedente de esta tesis es el desarrollado por Nélida Barón Pérez (Barón Pérez, 2020). Este trabajo de tesis es una extensión al trabajo de tesis desarrollado por Nélida Barón Pérez (Barón Pérez, 2020), en el que se pretende reducir el acoplamiento por el acceso indirecto a datos de módulos por otros módulos externos complementando así el trabajo de investigación de antecedente.

2.5.2.- Trabajos Relacionados

Software metrics: Using Measurement Theory to Describe the Properties and Scales of Static Software Complexity Metrics (Zuse & Bollmann, 1989)

En este artículo se presentan los conceptos básicos de la teoría de la medición aplicada a software. Se hace uso de un enfoque que permite describir las propiedades de las métricas de software, así como describir las métricas como escala Ordinal, Intervalo o de Razón.

El enfoque que se presenta en este artículo es debido a la necesidad de criterios para decidir qué medida de software se tiene que utilizar en una situación dada.

Properties of software measures (Zuse, 1992)

El autor de esta publicación proporciona las propiedades objetivas que tienen las escalas de medición para que las métricas o mediciones al cumplir las condiciones dadas según la clasificación de las escalas puedan ser catalogadas de un tipo de escala.

También se presentan teoremas que describen las propiedades de las medidas de software relacionadas a las operaciones de un cálculo de una medición. Se discuten y explican las propiedades de las medidas de software, según lo requerido en la literatura, mediante enunciados de la teoría de la medición. Los resultados indican que la mayoría de las propiedades necesarias de las medidas de software en la literatura pueden ser explicadas por las condiciones de la teoría de la medición.

Properties of Object-Oriented Software Measures (Zuse, 1996)

Este artículo trata sobre las propiedades de las medidas del software orientado a objetos para clasificar las mediciones en alguna escala de medición.

El autor explica que los criterios para las propiedades de las medidas del software orientado a objetos se caracterizan por varias operaciones binarias entre objetos, clases, métodos, etc. Las operaciones binarias se pueden usar como una herramienta para dar una interpretación a los números por encima del nivel de la escala ordinal.

Evaluating the impact of object-oriented design on software quality (Brito e Abreu & Melo, 1996)

En esta publicación los autores presentan un marco de métricas en las cuales se presenta la métrica Factor de Ocultación del Método (MHF), la cual mide el factor de ocultamiento de los métodos, pero no consideran los calificadores de alcance del lenguaje java.

Los resultados del experimento descrito en este artículo revelan que los mecanismos de diseño orientado a objetos, como la herencia, el polimorfismo, la ocultación de información y el acoplamiento, tienen una influencia significativa en características de calidad importantes, como la fiabilidad y la mantenibilidad.

Refactoring a legacy component for reuse in a software product line: A case study (Kolb et al., 2006)

El enfoque de este artículo es sobre líneas de producción de software (un enfoque que mejora conceptualmente la productividad del proceso de desarrollo de software), y trata una situación que se da a partir de la migración de componentes de software heredados, el problema con la situación es que el componente a migrar no fue diseñado para el reúso. Por lo tanto, el autor propone una serie de procesos de refactorización básicos y avanzados para mejorar la mantenibilidad y la reutilización del componente.

Toward an implementation of the "form template method" Refactoring (Juillerat & Hirsbrunner, 2007)

En esta publicación se presenta un algoritmo que realiza una transformación por medio de refactorización del código legado a funciones plantilla, tomando en cuenta tres pasos estructurados los cuales son:

- Detección de similitudes y diferencias
- Resolución de limitaciones
- Extracción de funciones

Una función plantilla es aquella que contiene la invocación tanto de las distintas funciones variantes como de las funciones invariantes comunes. Por lo que el primer paso del algoritmo es identificar las invocaciones de funciones invariantes que aún no han sido factorizadas (código duplicado).

En el segundo paso se verifica el alcance de las funciones invariantes que no se puedan extraer y se modifican de tal manera que la extracción y factorización se pueda realizar.

El tercer y último paso consiste en extraer subconjuntos de declaraciones consecutivas en nuevas funciones.

Making program refactoring safer (Soares et al., 2010)

En este artículo se analizan las condiciones previas que garantizan una buena refactorización para preservar el comportamiento del sistema. Los autores presentan una herramienta para mejorar la seguridad durante la refactorización que genera automáticamente un conjunto de pruebas adecuado para detectar cambios de comportamiento. Los autores utilizaron esta herramienta para evaluar siete refactorizaciones de estudios de casos reales de 3 a 100 KLOC (miles de líneas de código).

También se describe y se evalúa la herramienta SAFEREFAC-TOR la cual presentan los autores para verificar la seguridad de la refactorización en lenguajes de programación escritos en java.

A security-Aware refactoring tool for Java programs (Maruyama & Omori, 2011)

En este artículo se habla acerca de la refactorización y como ayuda a mantener el software, gracias al mejoramiento del código existente sin cambiar el comportamiento del sistema. También introduce a una problemática existente en la refactorización, que es la vulnerabilidad del código refactorizado. "*Una vulnerabilidad es un problema que puede ser aprovechado por los atacantes o una debilidad que hace posible que ocurra una amenaza*" (Maruyama & Omori, 2011).

Por esta razón, en este artículo se propone una herramienta con soporte para la refactorización que implemente un enfoque en seguridad. Esta herramienta puede detectar posibles vulnerabilidades en el código de algún sistema cuando se refactoriza y presentar advertencias de seguridad sobre estas vulnerabilidades.

Refactoring for software design smells: managing technical debt (Suryanarayana et al., 2014)

En el libro se describe "*code smell*" que produce deuda técnica en la implementación del diseño, para que los desarrolladores o ingenieros de software con este conocimiento puedan comprender los errores implementados en la etapa de diseño, y de qué manera puedan abordar el *code smell* a través de la refactorización.

Parte del *code smell* que se describe son los olores de "Encapsulación con fugas".

"*Encapsulación con fugas, Este olor surge cuando una abstracción "expone" o "filtra" detalles de implementación a través de su interfaz pública. Dado que los detalles de implementación se exponen a través de la interfaz, no solo es más difícil cambiar la implementación, sino que también permite a los clientes acceder directamente a las partes internas del objeto (lo que conduce a una posible corrupción del estado)."* (Suryanarayana et al., 2014).

AutoRefactoring: A platform to build refactoring agents (Santos Neto et al., 2015)

Este artículo trata sobre cómo los efectos no deseados en el software se pueden reducir mediante la refactorización, y los autores proponen puntos a considerar para refactorización segura y una plataforma para este propósito.

Los aspectos a considerar para aplicar una refactorización segura son los siguientes:

- Identificar las partes del código que deben mejorarse.
- Determinar los cambios que se deben aplicar al código para mejorarlo.
- Evaluar los impactos de las correcciones en la calidad del código.
- Comprobar que el comportamiento observable del software se conservará después de aplicar las correcciones.

La plataforma propuesta por los autores, es una plataforma basada en agentes que permite usar un agente con la capacidad de lidiar de manera autónoma con los aspectos anteriormente mencionados de la refactorización.

Architectural refactoring: A task-centric view on software evolution
(Zimmermann, 2015)

Este articulo habla acerca de la refactorización por medio de decisiones arquitectónicas, la clasificación de las tareas que se llevan a cabo según la refactorización y se propone el uso de plantillas donde se identifique el dominio de la refactorización arquitectónica y qué tipos de refactorización se pueden hacer sobre cada punto de la plantilla. El autor explica que la refactorización arquitectónica se ocupa de la documentación de la arquitectura y la manifestación de la arquitectura en el código. Por lo tanto, no existe un solo árbol de sintaxis arquitectónica como lo sería en refactorizaciones de código.

Refactoring for software architecture smells (Samarthyam et al., 2016)

Este artículo expone cómo el *code smell* y la refactorización deben recibir más atención de parte de los ingenieros de software, los autores motivan la necesidad de la refactorización en las arquitecturas.

"*Con la evolución de un sistema de software, la complejidad del sistema aumenta a menos que se realicen esfuerzos para mantenerlo y reducirlo. La refactorización es una técnica bien conocida definida como "comportamiento que preserva las transformaciones del programa" para hacer frente a la creciente complejidad y mantener el software mantenible. La falta de refactorización periódica conduce a la acumulación de deuda técnica y, en última instancia, a la "quiebra técnica". Por lo tanto, la refactorización periódica es esencial para que el software de larga duración mejore y mantenga la calidad estructural del software. En general, los olores y sus correspondientes refactorizaciones se aplican en varios niveles de granularidad."* (Samarthyam et al., 2016).

Adicional a esto, los autores categorizan los *olores* del código en tres categorías:

- Olores de implementación.
- Olores de diseño.
- Olores de arquitectura.

Automated refactoring of legacy java software to default methods (Khatchadourian & Masuhara, 2017)

En este artículo se presenta un enfoque de refactorización basado en restricciones de tipo para hacer la refactorización de los métodos de implementación esqueléticos, que forman parte del patrón de diseño de software de implementación esquelético a interfaces como métodos predeterminados que se implementan en Java 8.

Su enfoque se implementa en Eclipse como código abierto para que se pueda utilizar en el mundo real y se realiza una evaluación experimental donde los resultados arrojados demuestran que las técnicas propuestas son efectivas y prácticas y hacen avanzar el estado del arte del código legado de java a código de java moderno.

Automated refactoring of super-class method invocations to the Template Method design pattern (Zafeiris et al., 2017)

Este artículo trata la herencia de implementación que se implementa en el anti patrón “Call Super” y como es necesaria la refactorización para convertir a un patrón de diseño “Template Method”, esta refactorización contribuye a la sustitución de la herencia de implementación por herencia de interfaz.

Para este trabajo el método que se utilizó fue generar un algoritmo para la identificación de código a refactorizar, así como la forma de transformar código fuente para la refactorización de una instancia Call Super a Template Method.

Entre los resultados obtenidos destaca el potencial del método que utilizaron para identificar y eliminar instancias del Call Super y como el código mejoró gracias a esto.

Dive Into Refactoring (Alexander, 2019)

Este libro presenta y clasifica 21 tipos de *code smell* y 66 técnicas de refactorización correspondientes para corregirlos.

La estructura del libro se compone de dos secciones principales: "*Code Smells*" y "Técnicas de refactorización". En la primera sección se explican diversos indicios y manifestaciones de *code smell*, mientras que la segunda sección presenta distintos

enfoques para abordar el *code smell* y mejorarlo. En este libro se presentan varias técnicas de refactorización pequeñas como mover un método, extraer un método, introducir parámetros, etc.

30 Years of Software Refactoring Research: A Systematic Literature Review (Abid et al., 2020)

En este artículo se presenta una revisión sistemática de 3183 trabajos relacionados con la refactorización durante los últimos 30 años, con el objetivo de proporcionar una revisión exhaustiva y completa de los estudios de investigación existentes. Basándose en los trabajos revisados, se desarrolló una taxonomía para clasificar las investigaciones existentes, identificar tendencias de investigación y destacar lagunas en la literatura que sugieren temas para futuras investigaciones en el ámbito de la refactorización.

Refactoring Graphs: Assessing Refactoring over Time (Brito et al., 2020)

En este artículo se presentan gráficos de refactorización para dar seguimiento a los cambios estructurales que se realizan en el código durante la refactorización. En el artículo nombran ocho operaciones de refactorización atómicas que se presentan en los gráficos de refactorización:

- RENAME.
- MOVE.
- MOVE AND RENAME.
- PULL UP.
- PUSH DOWN.
- EXTRACT.
- INLINE.
- EXTRACT AND MOVE.

Así mismo muestran la manera de en qué se representan dichas operaciones gráficamente.

Unveiling process insights from refactoring practices (Caldeira et al., 2020)

Los autores de este artículo proponen e implementan la evaluación de una tarea de refactorización enfocada a la comprensión y complejidad del software bajo análisis para su refactorización. Esta información es obtenida y recopilada mediante sesiones de trabajo con el IDE utilizado. Los resultados obtenidos de su investigación permitieron predecir con un nivel de precisión del 92.95% el tipo de refactorización utilizado (automatizada o manual) por el equipo de desarrollo, así como el nivel de reducción de la complejidad ciclomática del software con una precisión del 94.36%.

Controlling technical debt remediation in outsourced enterprise systems maintenance: an empirical analysis (Ramasubbu & Kemerer, 2021)

En este artículo los autores hablan acerca de la deuda técnica y cómo ésta también afecta a sistemas empresariales, en un esfuerzo para gestionar los remedios para la deuda técnica que se presenta en dichos sistemas. Las empresas subcontratan el mantenimiento, pero los servicios de refactorización contratados no son los mejores, porque se necesita información y código base, cosa que las empresas no los otorgan por seguridad. Los autores como resultado del análisis realizado proponen implementar procesos de gestión de proyectos de software que permitan mejorar la visibilidad de la deuda técnica.

How Do I Refactor This? An Empirical Study on Refactoring Trends and Topics in Stack Overflow (Peruma et al., 2021)

Este artículo presenta cómo reducir la brecha que existe en la refactorización aplicada en la investigación, con la refactorización aplicada en la práctica. Para tratar esta situación el autor realiza una investigación en Stack Overflow para conocer los principales problemas con los que se encuentra un desarrollador en el tema de la refactorización:

- Optimización de código.
- Herramientas e IDE.
- Patrones de arquitectura y diseño.
- Pruebas unitarias.
- Base de datos.

Capítulo 3. Marco Teórico

3.1.- Paradigma de Programación Orientada a Objetos (Cervantes O et al., 2016)

El paradigma orientado a objetos es una metodología de programación en el cual se modela de forma análoga a la realidad, simplificando el diseño de alto nivel y permitiendo facilidad en el mantenimiento y reúso del software. En la programación orientada a objetos (POO) se definen entidades virtuales llamadas objetos, éstos simulan a otros objetos de la realidad que tienen un estado y un comportamiento (operaciones) dado. A nivel computacional los estados son llamados “atributos” y al comportamiento se le llama “métodos”.

Para crear objetos es necesario hacer uso de clases, una clase es un molde donde se definen las propiedades comunes de un conjunto de objetos. Los objetos son instancias de una clase.

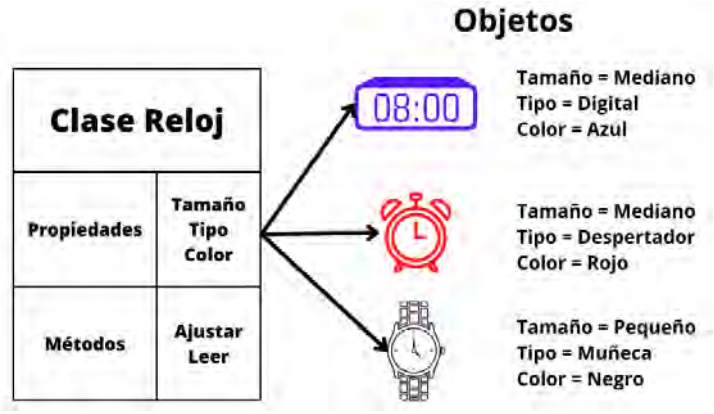


FIGURA 1 MODELO CONCEPTUAL DE CLASE Y OBJETOS

3.2.- Métodos

Un método constituye una parte del comportamiento de un objeto de una clase. Dependiendo del nivel de alcance que tenga un método puede ser una operación de un objeto externo o una operación interna de una clase (Cervantes O et al., 2016).

3.3.- Calificadores de Alcance

La protección de información en lenguajes de programación orientado a objetos, se realiza mediante las reglas de visibilidad o de alcance. En lenguaje java se definen cuatro distintos calificadores de alcance o reglas de visibilidad, los cuales sirven para determinar si una clase de objetos puede hacer uso de un atributo o invocar una función de otra clase (Oracle Corporation, n.d.), en el nivel de control de acceso de funciones. A continuación, se muestran los calificadores de alcance mencionados:

- Public: Una función declarada con este calificador tiene alcance por todas las funciones de todas las clases de objetos de un sistema de software.
- Protected: Una función protegida sólo puede ser accedida por las funciones dentro del mismo paquete o que sean miembros de subclases de objetos.
- Friendly: Este es el calificador de alcance por defecto en el lenguaje Java. Este calificador se aplica a entidades de estado y/o de comportamiento que permiten su visibilidad para todas las funciones de todas las clases integradas en un mismo paquete.
- Private: Una función definida con este calificador de alcance, solamente podrá ser accedida por otras funciones que sean miembros de la misma clase.

3.4.- Patrón de Diseño "*Template Method*"

Un patrón de diseño es una solución conocida para un problema conocido que se presenta recurrentemente (Martin, 2000). En (Gamma et al., 1994) los patrones de diseño son definidos como descripciones de objetos y clases que se comunican y se personalizan para resolver un problema de diseño general en un contexto particular. Un patrón de diseño tiene clases e instancias participantes, roles y distribución de responsabilidades.

Cada patrón de diseño se enfoca en un problema de diseño orientado a objetos en particular. El patrón de diseño en el que se enfoca esta tesis es el patrón de diseño "*Template Method*". Este patrón de diseño se clasifica como de "Comportamiento". Los patrones de comportamiento están relacionados con los algoritmos y con la asignación de responsabilidades. El "*Template Method*" define el esqueleto de un algoritmo en una función y asigna algunos de los pasos a subclases sin que cambie la estructura del algoritmo, en este patrón existen tres roles participantes dentro de

una superclase abstracta, propiamente la función plantilla, las funciones variantes y las funciones invariantes.

La función plantilla es la función principal que orquesta una secuencia algorítmica definida en su cuerpo. Dentro del cuerpo de la función plantilla se encuentran las llamadas a funciones variantes e invariantes, las funciones variantes se implementan en subclases las cuales permiten que su comportamiento pueda variar y las funciones invariantes son partes que no cambian en el algoritmo y solamente se definen en la super clase abstracta (Gamma et al., 1994).

3.5.- Refactorización

El software regularmente debe evolucionar conforme avance la experiencia de los desarrolladores y la refactorización es fundamental para que el código se mantenga legible y fácil de modificar (Martin Fowler & Beck, 2018). La refactorización es una parte esencial para el mantenimiento y la evolución del software, es una técnica que consiste en cambiar la estructura interna del software sin que cambie su comportamiento. Cuando se refactoriza se busca mejorar la calidad interna y reducir la deuda técnica del software (Peruma et al., 2021).

Estas reestructuraciones de código se realizan de forma manual o automatizada, siendo la primera la forma en la que con mayor frecuencia se implementa. La reestructura manual puede ser peligrosa ya que al ser guiada la refactorización por las habilidades de los desarrolladores se pueden introducir defectos inesperados (Caldeira et al., 2020).

3.6.- ANTLR y StringTemplate

ANTLR (Another Tool Language Recognition) es un poderoso generador de analizadores para leer, procesar, ejecutar o traducir texto estructurado. Es ampliamente utilizado para construir lenguajes, herramientas y marcos de trabajo. A partir de una gramática, ANTLR genera un analizador que puede construir y analizar árboles sintácticos (Parr, 2013).

ANTLR hace uso de gramáticas las cuales deben estar en archivos de formato “.g4”. La gramática se divide en dos archivos, la parte léxica (tokens) y las reglas sintácticas, cada archivo se representa en código escrito similar a la notación extendida Backus-Naur.

StringTemplate es un motor para generar código fuente a partir de una plantilla (Terence Parr, n.d.).

3.7.- Escalas de Medición

Se conoce como escala de medición al conjunto de valores ordenados correlativamente que puede tomar una variable. Las escalas de medición permiten identificar diferentes niveles de la variable, aceptan un punto inicial y otro final. Existen cuatro escalas de medición, las cuales tienen una finalidad según las variables en uso, las escalas categóricas comúnmente son usadas para variables cualitativas y las escalas numéricas son utilizadas para la medición de variables cuantitativas (Coronado Padilla, 2007):

Escala categórica:

- Nominal: Su uso es para la medición de variables cualitativas, usando letras o números como identificadores de una unidad de estudio, clasifica las unidades de estudio en categorías según sus características, atributos o propiedades distintivas y observadas.

- Ordinal: El uso de esta escala es para la medición de variables cualitativas. En este tipo de medición las características, atributos, propiedades distintivas y observadas son colocadas en un orden relativo según las cualidades que poseen.

Escala numérica:

- De intervalo: Su uso es para la medición de variables cuantitativas. Esta medición establece intervalos iguales para las distancias entre categorías, independientemente del orden o jerarquía de las categorías. La diferencia entre los objetos medidos se determina usando esta escala. El cero no representa la ausencia de la característica medida, representa un punto conveniente del cual se marcan intervalos para construir la escala.

- De proporción, cociente o razón: Su uso es para la medición de variables cuantitativas. También maneja las propiedades de los intervalos, pero el cero si representa la ausencia de la característica, por lo que permite comparar proporciones e indicar cuántas veces un objeto de estudio es más grande que otro. En esta escala, el cálculo de porcentaje es el más relevante (Zuse, 1992). Las métricas propuestas en esta investigación cumplen con las condiciones que establece la teoría de la medición para pertenecer a esta escala.

3.8.- Teoría de la Medición

La teoría de la medición proporciona un medio para combinar condiciones empíricas con condiciones numéricas, y traduce las propiedades empíricas a propiedades matemáticas, bajo el supuesto de un homomorfismo (Zuse, 1996). Para describir las métricas diseñadas en esta tesis es necesario introducir los conceptos teóricos de la teoría de la medición. En (Zuse & Bollmann, 1989) se consideran dos sistemas relacionados:

Sistema relacional empírico:

Sea el sistema relacional empírico **A** = (P, R_1,…,R_n, Op_1,…,Op_m), donde:

P es un conjunto no vacío de objetos empíricos a ser medidos

R_i son i-esimas relaciones empíricas sobre los objetos empíricos P

Op_j son operaciones binarias sobre los objetos empíricos P

Sistema relacional formal:

Sea el sistema relacional formal **B** = (Q, S_1,…,S_n, Ope_1,…,Ope_m), donde:

Q es un conjunto no vacío de objetos formales, tales como números o vectores

S_i donde i = 1,…,m; son las i-esimas relaciones sobre Q, tales como “mayor que”

Ope_j donde j = 1,…,m; son operaciones binarias cerradas sobre los objetos Q, tales como la adición o la multiplicación

La métrica μ:

Una métrica μ es un mapeo μ: A → B, donde para cada objeto empírico P ∈ A, produce un objeto formal (valor medido) μ(P) ∈ B.

Escala:

Dado un sistema relacional empírico **A** = (P, R_1,…,R_n, Op_1,…,Op_m), un sistema relacional formal **B** =(Q, S_1,…,S_n, Ope_1,…,Ope_m) y una métrica μ. La tripleta (**A**, **B**,

μ) es una escala, sí y sólo si para toda i, j y para toda a, b, $a_i1,\ldots,a_ik \in A$, se conserva que:

$$\mathbf{Ri}(\mathbf{a1}, \ldots, \mathbf{aki}) \Leftrightarrow \mathbf{Si}(\boldsymbol{\mu}(\mathbf{a1}), \ldots, \boldsymbol{\mu}(\mathbf{aki}))$$

$$\boldsymbol{\mu}(\mathbf{a\ Op\ b}) = \boldsymbol{\mu}(\mathbf{a})\ \mathbf{Ope}\ \boldsymbol{\mu}(\mathbf{b})$$

Un mapeo μ de un sistema relacional **A,** a otro sistema relacional **B**, que preserva todas las relaciones y operaciones es llamado un *homomorfismo*.

El tipo de escala que se asumió para las métricas diseñadas en esta investigación es de razón. Para confirmar lo anterior se debe cumplir con los siguientes puntos del axioma de orden débil (Zuse, 1992).

Sea P un conjunto no vacío de objetos empíricos a ser medidos y •≥ una relación binaria definida en P, entonces •≥ es de orden débil si:

P •≥ P'	P es igual o más compleja que P'
P •≥ P' | P' •≥ P	Completitud
(P •≥ P' ^ P' •≥ P'') => P •≥ P''	Transitividad

Para considerar una escala de razón, se deben considerar las condiciones de una escala ordinal. Esta escala dicta que, si existe un sistema relacional empírico de escala ordinal (P, •≥), entonces existe un mapeo μ: P → Q:

P •≥ P' ⇔ μ(P) ≥ μ(P')	Homomorfismo de escala ordinal

En una escala de razón el sistema relacional empírico (P, •≥) evoluciona a una estructura extensa (P, •≥, Op), donde se agrega la notación •>, ≈ y un segundo homomorfismo, donde:

Op	operación binaria cerrada sobre los objetos empíricos P
P •≥ P' ^ ¬(P' •≥ P) => P •> P'	P es más compleja que P'
P •≥ P' ^ P' •≥ P => P ≈ P'	P es igual de complejo que P'
μ(a Op b) = μ(a) + μ(b)	Homomorfismo de razón

Así mismo, la estructura extensa debe cumplir con los siguientes axiomas:

P Op (P' Op P'') ≈ (P Op P') Op P''	Axioma de asociatividad
P Op P' ≈ P' Op P	Axioma de conmutatividad

P •≥ P’ => P Op P’’ •≥ P’ Op P’’ Axioma de monotonicidad

Y el axioma de Arquímedes:

P •> P’ => para cada P’’, P’’’ existe un número natural n > 0, tal que: nP Op P’’ •> nP’ Op P’’’

Capítulo 4.
Materiales y Métodos de Solución

En este capítulo de presenta la siguiente figura que muestra una solución visual y estructurada a la solución abordada en este trabajo de tesis con el fin de ayudar a comprender y visualizar la solución realizada de una manera más clara y concisa.

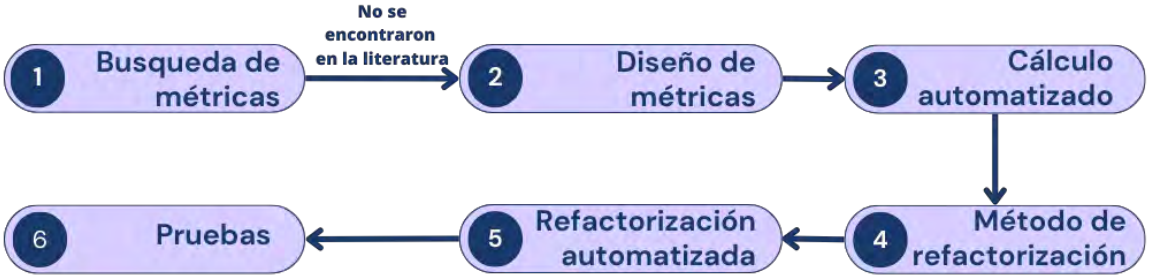


FIGURA 2 DIAGRAMA DE SOLUCIÓN

Mediante una representación Figura 2 se puede identificar la etapa "1", la cual se centró en la búsqueda de métricas adecuadas para medir e identificar la protección de las funciones plantillas. Al no obtenerse los resultados deseados, en la etapa "2" se procedió al diseño de las métricas PMMP, PMFI, PMFV y PTTM con el fin de cumplir el propósito mencionado anteriormente. Posteriormente, en el proceso "3”, se desarrolló un sistema en lenguaje Java para el cálculo automatizado de dichas métricas con el objetivo de que fueran contempladas en la etapa "4", donde se diseñó el método de refactorización y se definieron los procesos que lo conformarían. En la etapa "5", se implementó el método de refactorización mediante

una aplicación escrita en Java para llevar a cabo la refactorización automatizada y mejorar la protección de las funciones plantilla. Finalmente, en la etapa "6" se realizaron las pruebas necesarias para garantizar el correcto funcionamiento del cálculo automatizado de las métricas y de la refactorización automatizada.

4.1.- Diseño de las Métricas

Para identificar *code smell* debido a deficientes elecciones de diseño e implementación es necesario contar con un conjunto de métricas que permitan medir el grado de protección total del "*Template Method*" en la dimensión de protección funcional (Tufano et al., 2017).

En esta investigación se realizó una búsqueda en la literatura acerca de métricas para la protección de información en general y de protección de métodos o funciones plantilla en particular. En esta búsqueda no se encontraron métricas para este propósito, ni en lo general ni en lo particular, salvo en (Barón Pérez, 2020) donde se proponen métricas para medir el grado de protección modular en diferentes niveles de visibilidad funcional, sin embargo, no contempla específicamente a las funciones participantes en el patrón de diseño "*Template Method*". En esta tesis se presenta el diseño de un conjunto de métricas para la medición de la protección modular de funciones plantilla, las operaciones variantes e invariantes que participan en el patrón de diseño "*Template Method*", utilizando la teoría de la medición como sustento. Estas métricas fueron planteadas utilizando el enfoque GQM (Goal-Question-Metric) de la Figura 3.

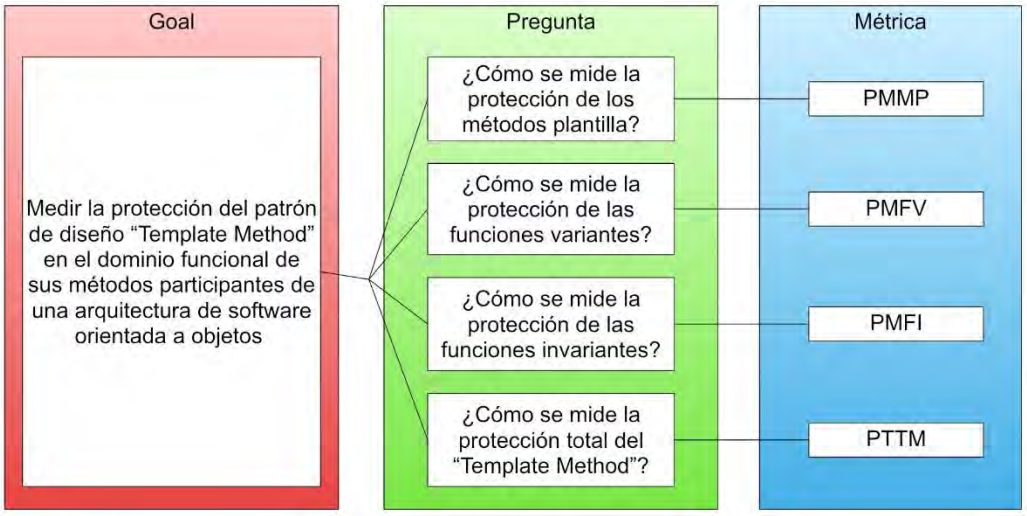


FIGURA 3 MODELO DEL ENFOQUE GQM UTILIZADO

4.2.- Diseño de la Métrica PMMP (Protección Modular de Métodos Plantilla)

La métrica PMMP mide el grado de protección modular de las funciones plantilla en arquitecturas de software orientadas a objetos (estas son las funciones orquestadoras del patrón de diseño "*Template Method*"). La protección modular se realiza mediante el calificador de alcance "public" o "friendly", dependiendo del alcance del código cliente a la función plantilla dentro de la arquitectura.

Si el cliente está en el mismo paquete que la clase plantilla que implementa la función plantilla, el calificador de alcance correcto de dicha función debe ser "friendly". También podría usarse el calificador de alcance "public" sin que el código presente fallas de alcance, sin embargo, ésta no sería la protección adecuada. Finalmente, si el cliente estuviera en otro paquete o no existiera cliente, entonces el calificador debe ser "public", aunque este calificador no protegerá a la función plantilla del acceso desde entidades de software externas a la clase plantilla.

La aplicación de esta métrica consiste en identificar las funciones plantilla en cada una de las clases de la arquitectura bajo estudio e identificar cuáles de ellas tienen el calificador "friendly", posteriormente se divide entre la cantidad total de funciones plantilla identificadas.

$$\mathbf{PMMP} = \frac{\sum_{\mathbf{ci=1}}^{\mathbf{ci=n}}\left(\sum_{\mathbf{fi=0}}^{\mathbf{fi=m}} \boldsymbol{MPP}\right)}{\boldsymbol{MP}} \quad (1)$$

ci = "i-esima" clase.

fi = "i-esima" función plantilla.

MPP = Funciones Plantilla con calificador "friendly".

MP = m = Sumatoria del total de funciones plantilla.

PMMP = Grado de protección modular de funciones plantilla.

A medida que PMMP tiende a 1 significa que la o las funciones plantilla tienen una mejor protección ("friendly").

A medida que PMMP tiende a 0 en la o las funciones plantilla se disminuirá su protección.

En el límite, el mejor resultado de esta métrica es 1 y el peor valor es 0.

4.3.- Diseño de la Métrica PMFV (Protección Modular de Funciones Variantes)

La métrica PMFV mide el grado de protección modular de las funciones variantes (funciones abstractas de las cuales varía su comportamiento implementado en clases derivadas). De conformidad con la sección de Implementación del patrón de diseño "*Template Method*" en (Gamma et al., 1994), la protección modular de estas funciones debería hacerse con el calificador de alcance "protected", para asegurar que estas solamente puedan ser invocados por la función plantilla.

La aplicación de esta métrica consiste en identificar las funciones variantes en cada una de las clases de la arquitectura bajo estudio e identificar cuáles de ellas tienen el calificador "protected". Posteriormente se divide entre la cantidad total de funciones variantes identificadas.

$$\mathbf{PMFV} = \frac{\sum_{\mathbf{ci=1}}^{\mathbf{ci=n}}(\sum_{\mathbf{fi=0}}^{\mathbf{fi=m}} \boldsymbol{FVP})}{\boldsymbol{FV}} \quad (2)$$

ci = "i-esima" clase.

fi = "i-esima" función variante.

FVP = Funciones variantes con calificador "protected".

FV = m = Sumatoria del total de funciones variantes.

PMFV = Grado de protección modular por funciones variantes.

A medida que PMFV tienda a 1 significa que la o las funciones variantes tienen una mejor protección ("protected").

A medida que PMFV tiende a 0 se verá disminuida la protección de la o las funciones variantes.

En el límite, el mejor resultado de esta métrica es 1 y el peor valor es 0.

4.4.- Diseño de la Métrica PMFI (Protección Modular de Funciones Invariantes)

La métrica PMFI mide el grado de protección modular de las funciones invariantes (funciones establecidas que no cambian su comportamiento y que son implementadas en la clase base, propietaria de la función plantilla). La protección modular de estas funciones debería hacerse con el calificador de alcance "private" para evitar que estas funciones sean invocadas directamente desde entidades

externas a la clase plantilla y asegurar que solamente puedan ser invocadas desde la función plantilla.

La aplicación de esta métrica consiste en identificar las funciones invariantes en cada una de las clases de la arquitectura bajo estudio e identificar cuáles de ellas tienen el calificador “private”. Posteriormente se divide entre la cantidad total de funciones invariantes identificadas.

$$\mathbf{PMFI} = \frac{\sum_{ci=1}^{ci=n}(\sum_{fi=0}^{fi=m} \boldsymbol{FIP})}{\boldsymbol{FI}} \quad (3)$$

ci = “i-esima” clase.

fi = “i-esima” función invariante.

FIP = Funciones invariantes con calificador private.

FI = m = Sumatoria del total de funciones invariantes.

PMFI = Grado de protección modular por funciones invariantes.

A medida que PMFI tienda a 1 significa que la o las funciones invariantes tienen una mejor protección (“private”).

A medida que PMFI tiende a 0 se verá disminuida la protección de la o las funciones invariantes.

En el límite, el mejor resultado de esta métrica es 1 y el peor valor es 0.

4.5.- Diseño de la Métrica PTTM (Protección Total del Template Method)

La métrica PTTM es el resultado de una división donde el numerador es la suma de los factores PMMP, PMFV y PMFI, y el denominador es el total de factores que participan en el numerador. Esta métrica mide el grado de protección total de una arquitectura que implementa el patrón de diseño *“Template Method”*. La protección total se realiza mediante los calificadores de alcance correctos de las funciones plantilla, funciones variantes y funciones invariantes.

La aplicación de esta métrica consiste en realizar una sumatoria de los valores resultantes de las métricas PMMP, PMFV y PMFI, posteriormente se divide este resultado entre el número total de factores que participan en el numerador.

$$\mathrm{PTTM} = \frac{\mathrm{PMMP}+\mathrm{PMFV}+\mathrm{PMFI}}{Nf} \quad (4)$$

PMMP = Grado de protección modular por funciones plantilla.

PMFV = Grado de protección modular por funciones variantes.

PMFI = Grado de protección modular por funciones invariantes.

PTTM = Grado de protección total por el patrón de diseño Template Method.

Nf = Número total de factores que participan en el numerador.

A medida que PTTM tienda a 1 significa que las funciones plantilla y sus funciones asociadas tienen una mejor protección.

Si la medida tiende a 0 significa que las funciones plantilla y sus funciones asociadas se verán disminuidas en su protección.

El mejor resultado de esta métrica es 1 y el peor valor es 0.

Cada una de estas métricas fueron sustentadas en la teoría de la medición como escalas de razón, este sustento teórico se encuentra en el anexo A.

A partir del lenguaje orientado a objetos Java, se realizó un sistema de software que automatiza el cálculo en conjunto de las métricas desarrolladas en esta tesis. Este sistema da soporte para la identificación y cálculo de la protección modular de las funciones participantes descritas en la literatura acerca del patrón de diseño "*Template Method*".

El sistema que implementa este marco de métricas fue probado en tres distintos sistemas informáticos escritos en el lenguaje Java y obtenidos de la plataforma de repositorios GitHub:

- Sistema "Sudoku".
- Sistema "SushiRestaurant".
- Sistema "Usta Donerci".

La prueba de estas métricas se describe a detalle en el capítulo 6 "Pruebas".

A continuación, en la sección 4.6 se describe a detalle la parte esencial de la aportación de esta tesis.

4.6.- Método de Refactorización para Protección de Funciones Plantilla

El método de refactorización desarrollado en esta tesis a grandes rasgos consiste de cuatro etapas:

1. Proceso de análisis del código fuente.
2. Evaluación de la protección modular.
3. Proceso de refactorización.
4. Reevaluación de la protección modular.

4.6.1.- Proceso de análisis de código fuente (etapa 1)

El proceso de análisis de código fuente que se somete al estudio, se realiza mediante un analizador generado con la herramienta ANTLR, para cumplir con dos objetivos dentro del método de refactorización independientemente del marco de calidad:

1) Generar una lista como estructura de datos compleja que contiene la información de los metadatos de las clases de objetos que integran la arquitectura del sistema bajo estudio.
2) Pre-identificar las arquitecturas que implementan el patrón de diseño *"Template Method"* para un proceso de refactorización optimizado basado en "Candidateo Semántico".

El proceso de análisis de código fuente comienza seleccionando el proyecto a refactorizar, después el código del proyecto se somete a extraer sus metadatos mediante la implementación de acciones semánticas, bajo las reglas sintácticas del lenguaje java en su versión 8, para generar una lista como de estructura de datos compleja, así como para pre-identificar las arquitecturas que implementan el patrón de diseño *"Template Method",* así como pre-identificar las funciones plantilla y las funciones variantes*.*

La pre-identificación se logra a partir de una técnica creada en el desarrollo de esta investigación a la que se nombró "Candidateo Semántico". Los elementos pre-identificados son llamados "Candidatos". La razón de esto es que se cumplen ciertas condiciones escaladas inmediatas que los vuelven candidatos a partícipes del patrón de diseño "*Template Method*". Las condiciones de que una arquitectura implementa el patrón de diseño son dadas por la definición del patrón de diseño.

En la Figura 4 se observan las condiciones para que las estructuras complejas sean candidatas una vez aplicado el analizador bajo la gramática del lenguaje de programación Java:

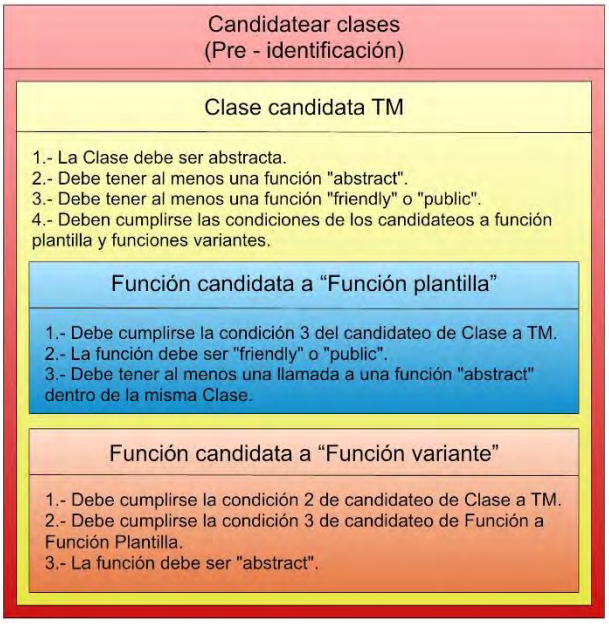


FIGURA 4 CONDICIONES DE CANDIDATEO TM

Mientras se tengan condiciones más específicas de las entidades de software que se desean pre-localizar, mayor será la probabilidad de que se encuentren esas entidades en las candidatas.

4.6.2.- Evaluación de Protección Modular (etapa 2)

La evaluación de la protección modular del patrón de diseño "*Template Method*" es dada por el marco de métricas desarrollado como complemento para esta tesis, este proceso se puede observar en el apartado 4.7.1, "Proceso de Refactorización".

4.6.3.- Proceso de Refactorización (etapa 3)

A la vez, el proceso de refactorización consiste de dos etapas, 1) la etapa de identificación y 2) la etapa de refactorización usando una técnica desarrollada en esta tesis a la que se nombró "tratamientos".

Identificar a los participantes del patrón de diseño resulta ser más sencillo ya que al conocer a los candidatos solamente se les tiene que aplicar las condiciones para corroborar que son funciones plantillas aquellas marcadas como candidatas.

La identificación de arquitecturas que implementan el patrón de diseño "*Template Method*" pasa de ser de un algoritmo comparativo de "fuerza bruta" (uno vs todos) a un algoritmo comparativo de "fuerza parcial" (uno vs algunos).

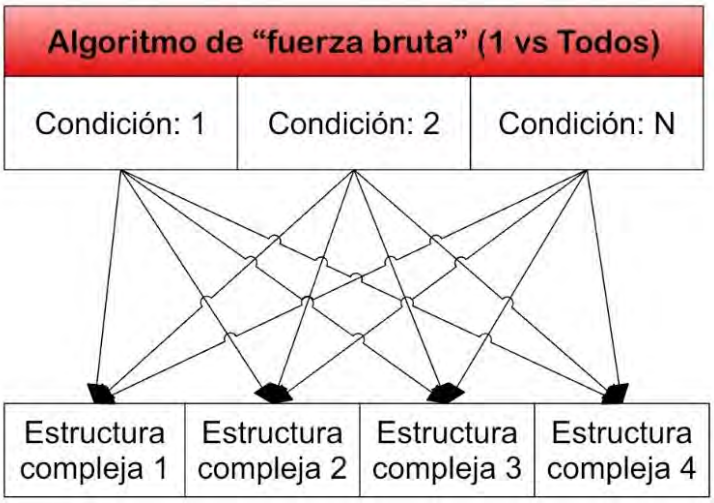


FIGURA 5 ALGORITMO COMPARATIVO DE “FUERZA BRUTA”

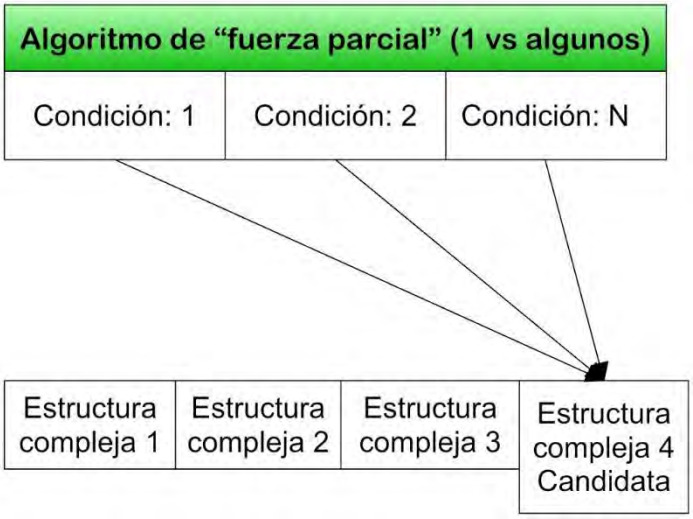


FIGURA 6 ALGORITMO COMPARATIVO DE “FUERZA PARCIAL”

El algoritmo basado en candidateo permite que las comparaciones sean contra estructuras que son altamente probables a ser estructuras que implementen el patrón de diseño “*Template Method*”.

La identificación consiste en una serie de condiciones dadas por los escenarios analizados las cuales van desde:

- Que las funciones variantes candidatas sean invocadas desde la función plantilla candidata, declaradas en la clase base y localizadas en sus clases derivadas.

- Identificar que las funciones invariantes candidatas sean llamadas en la función plantilla y declaradas en la clase plantilla.
- Finalmente, si esto se cumple entonces la función plantilla candidata debe tener llamadas a las funciones anteriormente mencionadas en su cuerpo, sólo entonces se procede a marcar como la función plantilla real, funciones variantes reales y funciones invariantes reales.

Durante la etapa de refactorización, una vez identificado el patrón de diseño y sus participantes en el código, se procede dar un tratamiento específico por cada función asociada a dicho patrón. El objetivo de la refactorización es aumentar la protección funcional de arquitecturas modulares que implementan el patrón de diseño "*Template Method*".

La refactorización hace uso de tratamientos para cada una de las funciones plantilla, la obtención de los tratamientos fue identificando el espejismo de las clases contenedoras de funciones plantilla con las arquitecturas de los escenarios identificados, posteriormente se buscan las funciones plantilla, funciones variantes y funciones invariantes en estas clases para dar el tratamiento según los escenarios encontrados (Figura 7).

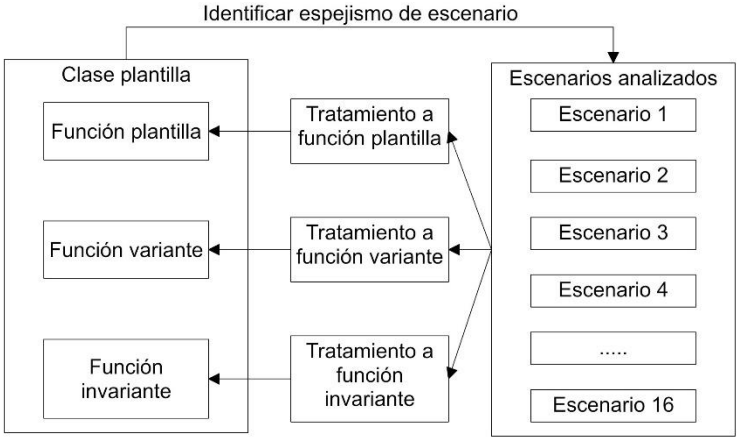


FIGURA 7 GENERACIÓN DE TRATAMIENTOS

Para optimizar el algoritmo de refactorización se seleccionaron tratamientos específicos de los tres tipos de funciones participantes en el patrón de diseño, estos tratamientos se obtuvieron factorizando de cada escenario las soluciones de las situaciones problemáticas sobre las funciones plantilla. Los escenarios problemáticos pueden converger en las funciones plantilla, por lo que, es más eficiente utilizar tratamientos por cada rol de las funciones plantilla que buscar que el código legado encuadre en escenarios sumamente específicos.

El tratamiento para los funciones plantilla (Figura 8) consiste en la identificación de su calificador de alcance actual, después se analizan las llamadas desde los clientes a esta función, si hay por lo menos una llamada a la función externa al paquete en el que se encuentra la clase plantilla o no existe algún cliente que llame al método entonces el calificador correcto para la función plantilla se calibra con “public”, si la o las llamadas se encuentran dentro del mismo paquete entonces el calificador correcto se calibra como “friendly”. En otro escenario donde la función plantilla es llamada solamente por otro método dentro de la misma clase entonces se le da el tratamiento conducido por el “tratamiento por metamorfosis”, se le otorga el tratamiento de método plantilla al método cliente y al método plantilla originalmente identificado se le da un tratamiento de función invariante. Cuando se presentan escenarios donde el patrón de diseño está mal implementado, pero es funcional como en el último caso, se asignan distintos roles a las funciones plantilla para que se logre aumentar la protección modular sin discriminarlas, a estos cambios de roles en las funciones plantilla se les llamó “tratamientos por metamorfosis”. Una vez que el calificador correcto ha sido calibrado, entonces se cambia el calificador actual que tiene la función plantilla y se cambia por el correcto.

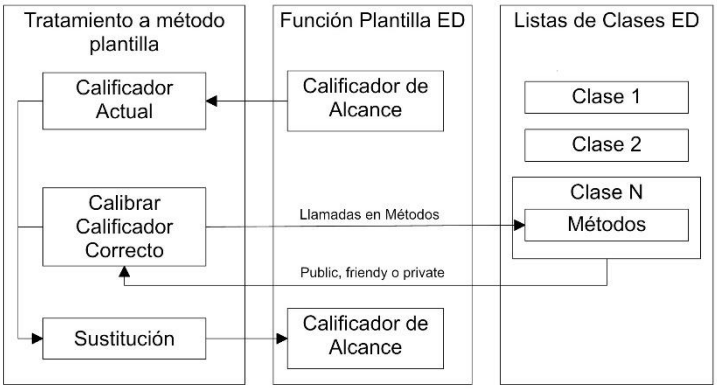


FIGURA 8 TRATAMIENTO DE FUNCIONES PLANTILLA

Una vez finalizado el tratamiento de funciones plantilla se procede a realizar el tratamiento de funciones variantes (Figura 8), este tratamiento consiste en obtener el calificador de alcance actual de las funciones variantes marcadas como candidatas y analizar las jerarquías de clases en las arquitecturas, identificando las clases específicas en las que se heredan las funciones variantes de la clase plantilla, una vez identificadas las funciones variantes en las clases hijas, se cambian sus calificadores de alcance actuales por el calificador de alcance “protected”, así mismo en la clase plantilla se les cambia el calificador de alcance a las funciones variantes (abstractas) por el calificador “protected”.

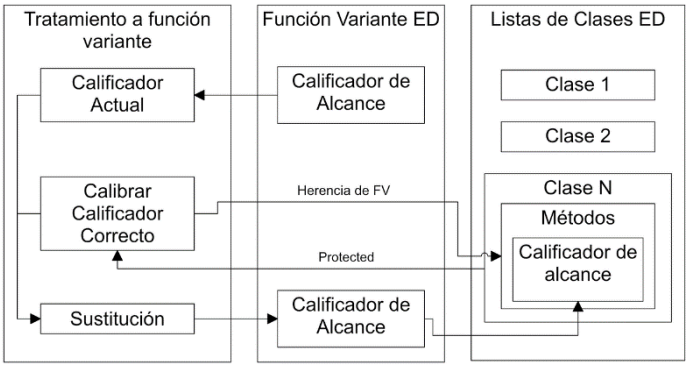


FIGURA 9 TRATAMIENTO DE FUNCIONES VARIANTES

Finalmente, el tratamiento para las funciones invariantes consiste en obtener su calificador de alcance y calibrar el calificador correcto mediante las llamadas a estas funciones. Si las funciones invariantes son llamadas solamente por la función plantilla entonces el calificador correcto debe ser “private”. Cuando una función invariante se define en la clase plantilla y se sobrescribe en una clase derivada se le conoce como función virtual, si se presenta este escenario, por metamorfosis a la función invariante se le clasifica y se le da el tratamiento como si se tratara de una función variante.

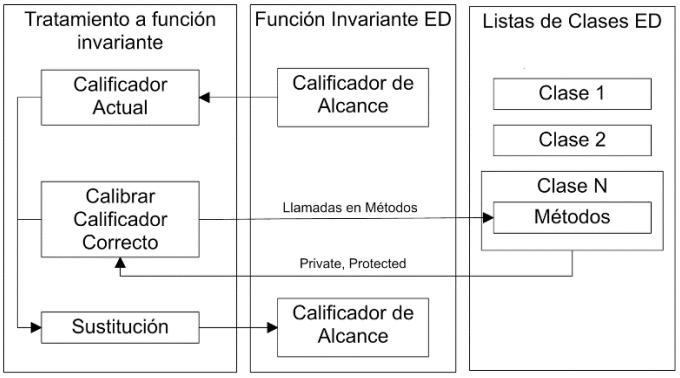


FIGURA 10 TRATAMIENTO DE FUNCIONES INVARIANTES

4.6.4.- Reevaluación de protección modular (etapa 4)

La reevaluación de la protección modular se realiza con el marco de métricas que es extendido en esta tesis, este marco aplica las métricas diseñadas y sustentadas en el anexo A, con la finalidad de observar el grado de mejora en la protección de funciones plantilla.

4.7.- Refactorización Automatizada para la Protección de Funciones Plantilla

Se implementó el método de refactorización en un sistema que automatiza el proceso de refactorización en arquitecturas escritas en el lenguaje de programación orientado a objetos Java.

4.7.1.- Proceso de Refactorización

A continuación, en la Figura 11, se muestra el diagrama BPMN (Business Process Model and Notation) para describir las actividades estructuradas del método de refactorización desarrollado en esta tesis.

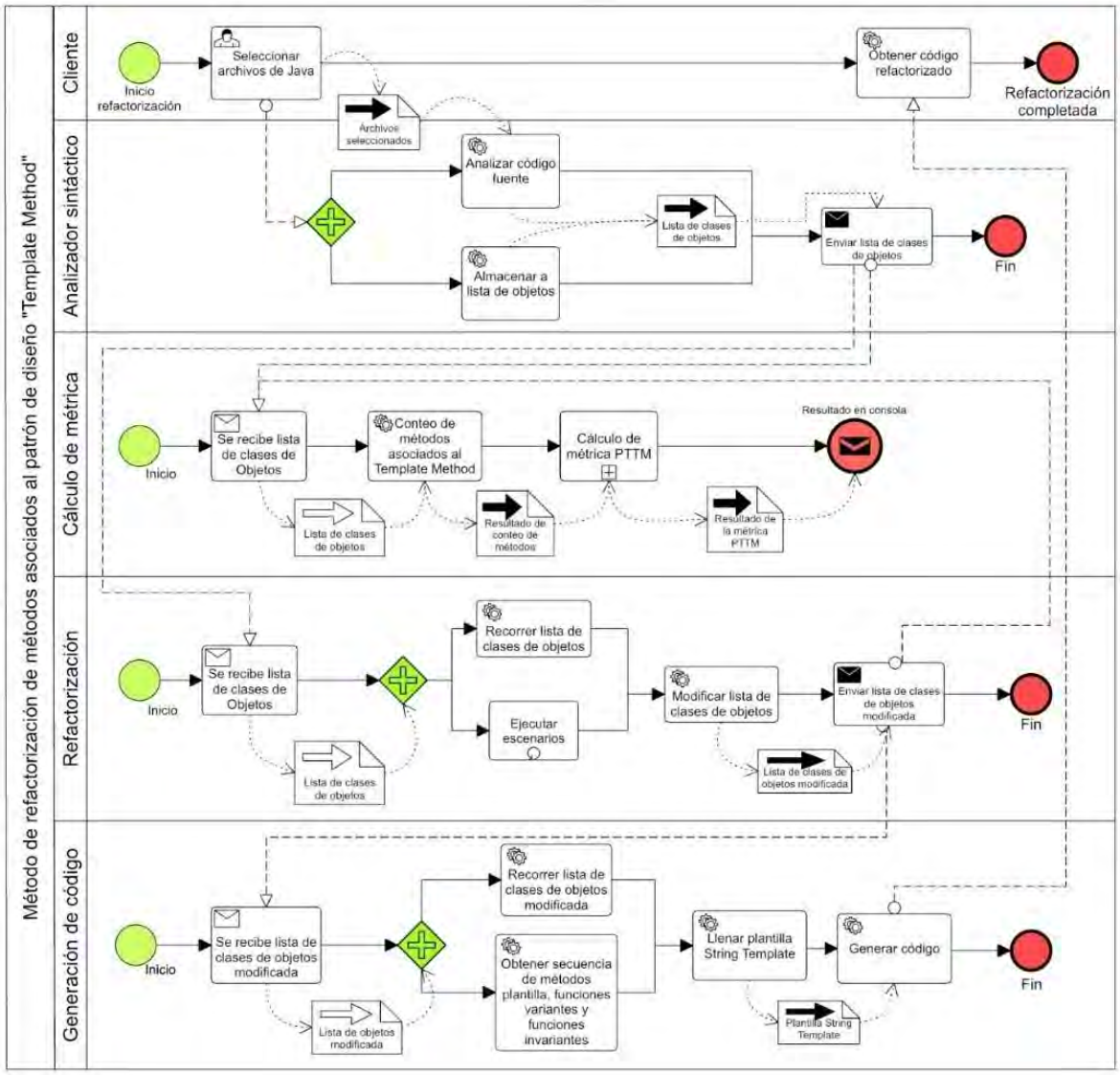


FIGURA 11 MODELO BPMN DEL MÉTODO DE REFACTORIZACIÓN

El primer paso del sistema que automatiza el método de refactorización es que el usuario seleccione manualmente los archivos que integran la aplicación de software legado escrita en Java y que se requiere analizar para refactorizar con la ayuda de un gestor de archivos, como se ilustra en la figura 12. Una vez que se tienen los archivos, se procede a realizar el análisis sintáctico.

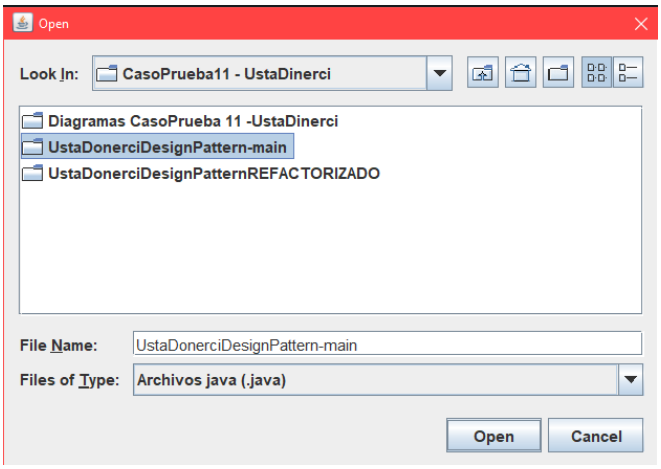


FIGURA 12 SELECCIÓN DE ARCHIVOS

La tarea de análisis sintáctico recibe como entrada los archivos seleccionados con el código fuente escrito en Java, este análisis se realiza mediante el analizador generado por la herramienta ANTLR para extraer la información necesaria que se utiliza en el cálculo de las métricas, la refactorización y la generación de código refactorizado. Esta información se almacena paralelamente en una estructura de datos residente en memoria. Como resultado de esta tarea se obtiene una lista compleja de objetos que contiene la información extraída. Inicialmente el código legado es procesado mediante las acciones sintácticas derivadas de la gramática del lenguaje de programación Java. La gramática se divide en dos archivos, la parte léxica (tokens) y las reglas sintácticas, cada archivo se escribe en un código similar a la notación extendida Backus-Naur, dichos archivos deben llevar un formato “.g4”.

En la Figura 13 se puede observar una parte del contenido del archivo léxico de la gramática Java, y en la Figura 14 parte de las reglas sintácticas de Java:

```
lexer grammar JavaLexer;

// Keywords

ABSTRACT:           'abstract';
ASSERT:             'assert';
BOOLEAN:            'boolean';
BREAK:              'break';
BYTE:               'byte';
CASE:               'case';
CATCH:              'catch';
CHAR:               'char';
CLASS:              'class';
CONST:              'const';
CONTINUE:           'continue';
DEFAULT:            'default';
DO:                 'do';
DOUBLE:             'double';
ELSE:               'else';
ENUM:               'enum';
EXTENDS:            'extends';
FINAL:              'final';
FINALLY:            'finally';
FLOAT:              'float';
FOR:                'for';
IF:                 'if';
GOTO:               'goto';
IMPLEMENTS:         'implements';
IMPORT:             'import';
INSTANCEOF:         'instanceof';
INT:                'int';
INTERFACE:          'interface';
```

FIGURA 13 PARTE DEL CONTENIDO DE JAVALEXER.G4

```
parser grammar JavaParser;

options { tokenVocab=JavaLexer; }

compilationUnit
    : packageDeclaration? importDeclaration* typeDeclaration* EOF
    ;

packageDeclaration
    : annotation* PACKAGE qualifiedName ';'
    ;

importDeclaration
    : IMPORT STATIC? qualifiedName ('.' '*')? ';'
    ;

typeDeclaration
    : classOrInterfaceModifier*
      (classDeclaration | enumDeclaration | interfaceDeclaration |
      annotationTypeDeclaration)
    | ';'
    ;

modifier
    : classOrInterfaceModifier
    | NATIVE
    | SYNCHRONIZED
    | TRANSIENT
    | VOLATILE
    ;
```

FIGURA 14 PARTE DEL CONTENIDO DE JAVAPARSER.G4

Una vez especificada la gramática del lenguaje Java, se ejecuta la herramienta ANTLR en el archivo gramatical “JavaParser.g4”, y la herramienta genera los siguientes archivos:

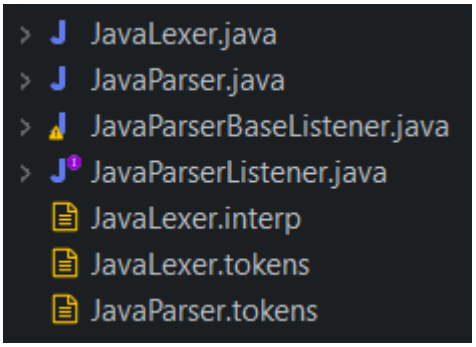


FIGURA 15 ARCHIVOS GENERADOS POR ANTLR

Los archivos más importantes para el proceso de análisis del código fuente son el archivo “JavaParserListener.java” el cual es una interfaz que describe los “eventos” donde podemos disparar acciones semánticas, y el archivo “JavaParserBaseListener.java” implementa la clase anteriormente mencionada para describir las acciones semánticas que deseamos realizar cuando se recorra el árbol sintáctico. La clase “JavaParserBaseListener” se encarga de realizar el análisis sintáctico y el semántico para obtener la estructura de datos con los objetos y métodos de clase marcados como candidatos.

La estructura de datos en la que se registra la información obtenida del proceso de análisis de código fuente es una lista compleja que contiene los metadatos de clases de objetos. Esta lista compleja representa la estructura del modelo de objetos de la Figura 16. Siguiendo este modelo el sistema contiene la estructura arquitectónica que consta de dos clases de objetos, la primera consta de una representación de clases Java y sus metadatos (Clase.java), y la segunda es una representación de métodos de clases Java (Metodo.java) también con sus metadatos.

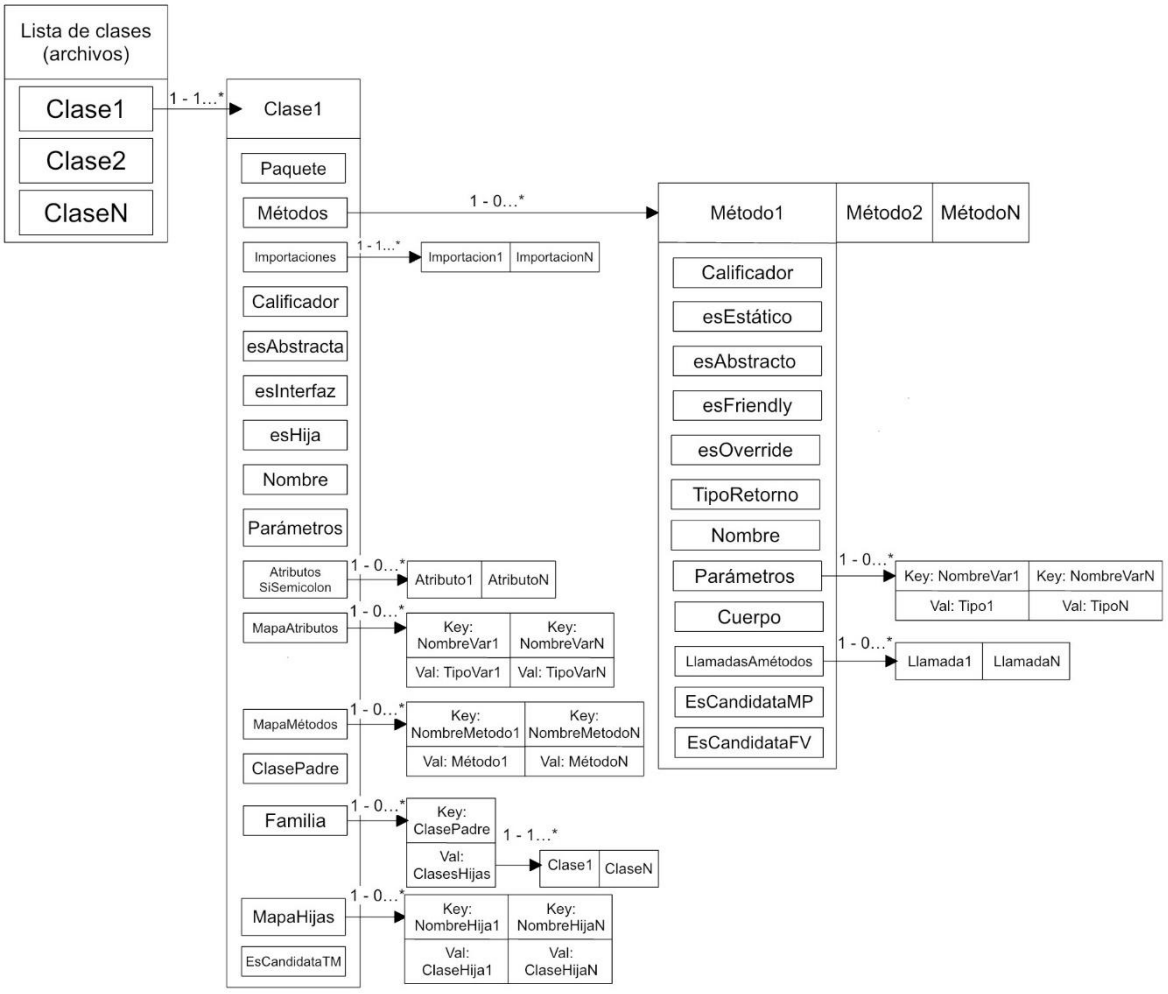


FIGURA 16 MODELO DE LA ESTRUCTURA CONTENEDORA DE INFORMACIÓN

Se ha optado por esta forma de representación ya que todo archivo con la extensión ".java" define por lo menos a una clase, a su vez una clase puede tener de 0 a N métodos, y a que el patrón de diseño "*Template Method*" dentro de su definición hace uso de "operaciones abstractas" (funciones variantes) que se definen en las "subclases concretas" para implementar un algoritmo, y "operaciones primitivas" (funciones invariantes) las cuales forman parte del proceso del algoritmo (Gamma et al., 1994), entonces es necesario utilizar una clase "Método" la cual tiene un rol importante, puesto que los métodos en Java son los principales partícipes en el patrón de diseño "*Template Method*". Otra razón es que el método de refactorización de alto impacto desarrollado, mejora la protección modular y total de las arquitecturas que implementan este patrón de diseño, y esta protección es dada por los calificadores de alcance de los métodos participantes.

Recapitulando, las clases creadas que conforman la arquitectura del sistema, contienen la información de metadatos de clases de objetos y sus métodos asociados, la cual es necesaria para representar el código legado como una estructura de datos que a su vez representa clases de objetos en Java. También contienen la información que servirá para la etapa de identificación en el proceso de refactorización basado en “Candidateo Semántico”.

Las estructuras complejas de información de metadatos son almacenadas en una lista la cual representa a todo un proyecto Java. El registro de las estructuras es dado por las acciones semánticas del archivo “JavaParserBaseListener.java”, estas acciones semánticas son disparadas cuando se recorre el árbol sintáctico generado por el ANTLR. Para que el recorrido sea efectuado es necesaria la implementación de un “caminante de árbol”, el cual es quien dispara los eventos o acciones semánticas durante su recorrido por el árbol sintáctico. La clase donde se ejecutan las acciones semánticas contiene métodos de entrada y salida para cada regla, a medida que el caminante entra o sale de un nodo del árbol se efectúan las acciones.

En orden de ejecución primero se efectúa el analizador léxico para “tokenizar” la entrada (archivo Java), después se ejecuta el analizador sintáctico el cual genera un árbol sintáctico, finalmente se crea un caminante el cual irá disparando acciones semánticas durante el recorrido del árbol sintáctico poblando la estructura contenedora compleja de información.

El proceso de cálculo de métricas que se ilustra en la Figura 17, en una 2da pasada realiza nuevamente el proceso de análisis sintáctico y semántico, el cuál recibe como entrada la lista metadatos de clases de objetos que se crea en el proceso anterior. De esta lista se toman los datos que necesitan las métricas (conteo de métodos y funciones asociados al *“Template Method”*), y se procede a calcular el grado de protección total del *“Template Method”*, como subproceso para el cálculo total, se calcula el grado de protección de las funciones plantilla, evaluando la correctitud del calificador de alcance, tanto de las funciones plantilla, como de sus funciones asociadas (funciones variantes e invariantes).

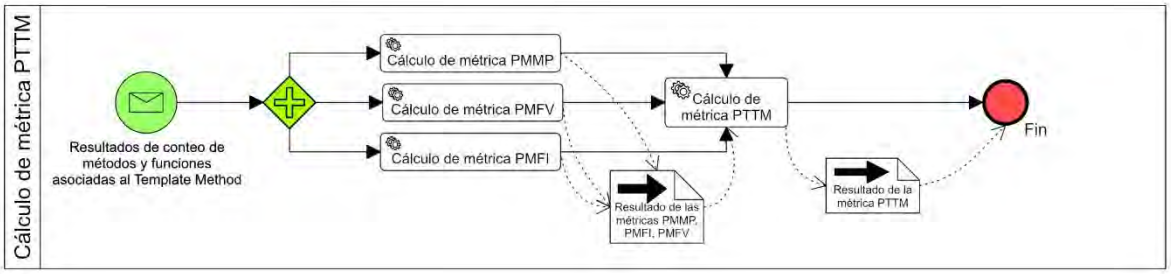


FIGURA 17 SUBPROCESO DEL CÁLCULO DE MÉTRICA PTTM

El proceso de refactorización de los calificadores de alcance de las funciones plantilla y funciones asociadas, consiste en usar la misma lista de metadatos de objetos resultante del proceso de análisis sintáctico, con esa información el sistema hace uso de un método llamado “refactorizarReglasTM”, el cual es el disparador de la refactorización, en este punto la clase “Refactorizar” recorre la lista de metadatos de clases de objetos e identifica los calificadores de alcance de las funciones asociadas a las funciones plantilla. Conforme a los detalles de implementación y escenarios existentes que se pueden dar en el patrón de diseño “*Template Method*”, el sistema ajusta y modifica en la misma lista, los calificadores de alcance correctos según los tratamientos definidos en el apartado 4.6.3 “Proceso de Refactorización”.

La decisión para la asignación de calificador de alcance se toma conforme a los siguientes criterios:

- Para los funciones plantilla (métodos orquestadores del patrón de diseño *“Template Method”)* el calificador de alcance es “public”, en caso que el cliente que hace uso de ellas está en otro paquete, salvo que el cliente esté ubicado en el mismo paquete el calificador de alcance será declarado como “friendly”, en otro caso en el que la función plantilla solamente sea llamada por alguna función miembro de la misma clase en la que se ubica la función plantilla ésta será etiquetada como “private”, el tratamiento es asignado por metamorfosis como si se tratara de una función invariante. En todo caso, la función plantilla deberá ser absoluta, es decir que no puede ni debe ser virtual o abstracta puesto que no será reemplazada ni será implementada en clases derivadas.
- El calificador de alcance “private” se asigna para las **funciones invariantes**. Estas funciones serán usadas únicamente por las funciones que pertenecen a la clase en la que se ubica el método de plantilla.
- El calificador “protected” se aplica a las **funciones variantes**. Estas funciones son aquellas que son definidas en la clase base en que se ubica la función plantilla, pero son implementadas en clases derivadas para garantizar su invocación por jerarquía de herencia.
- Debe considerarse el acceso a estas funciones desde clases externas a la clase plantilla que implementa el “*Template Method*”, para realizar los ajustes pertinentes y recomendando al usuario ejecutar un método de refactorización independiente a éste, que reestructura el código legado en cumplimiento con el principio de “*única responsabilidad*”.

Las arquitecturas analizadas a continuación, son los escenarios posibles tomados en cuenta para la implementación del método de refactorización, donde se tiene

remarcado con verde la función plantilla, con amarillo las funciones variante y finalmente con azul la función invariante, todos los escenarios encontrados se pueden ver en “Anexo B”:

Para el escenario de éxito:

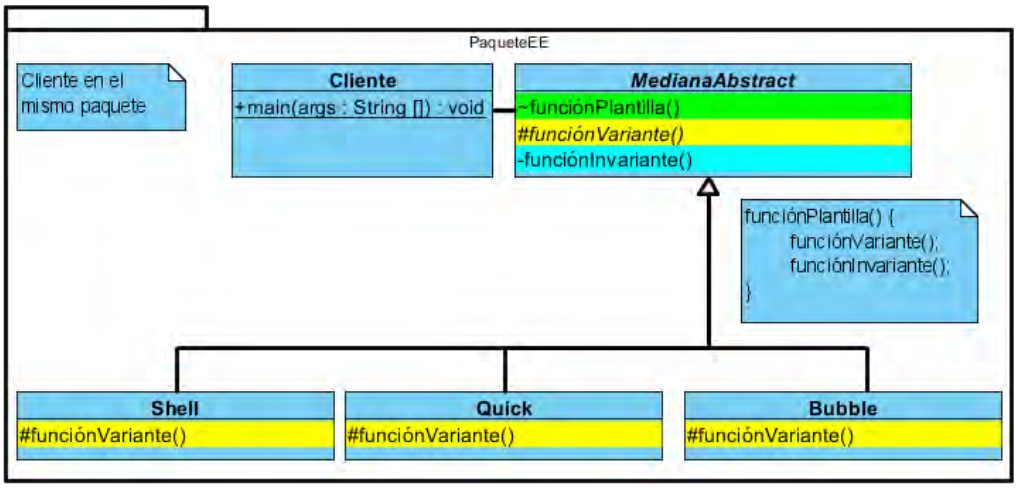


FIGURA 18 ESCENARIO DE ÉXITO 1

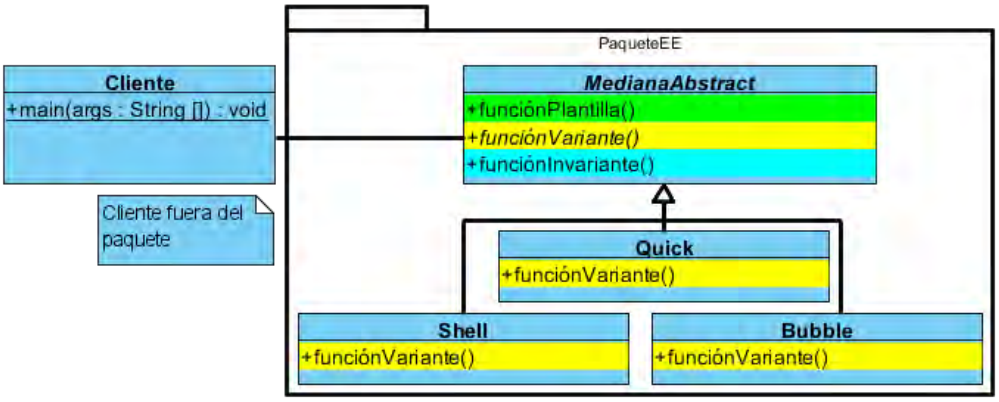


FIGURA 19 ESCENARIO DE ÉXITO 2

Las figuras 18 y 19 presentan dos escenarios posibles de éxito, los dos escenarios tienen sus funciones plantilla, variantes e invariantes con el calificador de alcance correcto. El contenido de la función plantilla en todos los es escenarios es como se observa en la nota de la Figura 18, la clase “Cliente” es quien invoca a la función plantilla, por lo que cuando la clase se encuentra dentro del mismo paquete que la función plantilla, esta última tiene que tener el alcance “Friendly”, y cuando la clase “Cliente” se encuentra fuera del paquete la función plantilla debe tener el calificador “Public” ya que si no fuera de esta forma, se presentarían errores durante la

compilación. Este escenario es el adecuado por lo que no se requiere refactorización y muchas veces las arquitecturas una vez refactorizadas llegarán a presentarse como el escenario de la Figura 19.

Para escenarios de las figuras 20 y 21 con calificadores incorrectos:

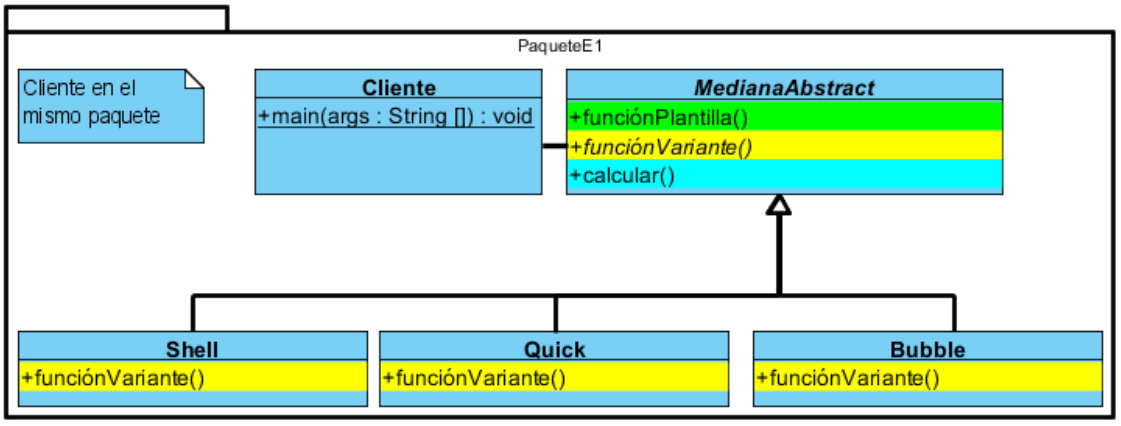


FIGURA 20 ESCENARIO INCORRECTO 1

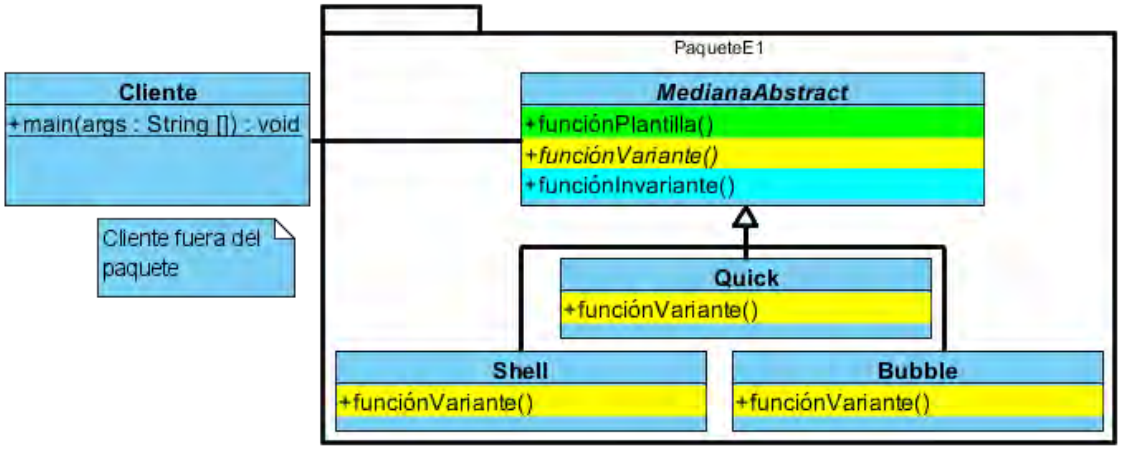


FIGURA 21 ESCENARIO INCORRECTO 2

La situación problemática de los escenarios de las figuras 20 y 21 consiste en que el calificador de alcance de todas las funciones participantes en el patrón de diseño es público y por lo tanto no cumplen con los criterios anteriormente mencionados, mas no obstante la función plantilla donde el cliente está fuera del paquete que implementa el patrón de diseño es correcta. Este escenario desencadena en otros seis escenarios (revisar “Anexo B”) que consisten en distintas combinaciones de los posibles calificadores de alcance que se pueden encontrar en las arquitecturas que

implementan el patrón de diseño "*Template Method*" y en el número de clientes que acceden a la función plantilla. Estos siete escenarios requieren ser refactorizados.

Para escenarios que no tienen funciones invariantes (Figura 22):

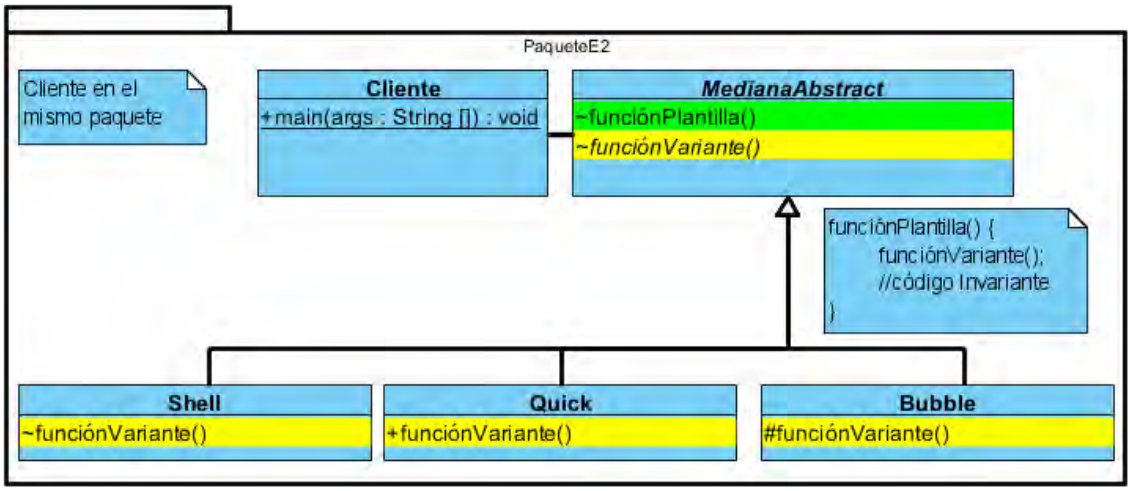


FIGURA 22 ESCENARIO INCORRECTO 3

En el escenario de la Figura 22 se presenta una situación en la que la función invariante puede estar embebida dentro de la función plantilla o simplemente no hay una función invariante presente. En este escenario se contemplan las distintas combinaciones de calificadores inadecuados de la función plantilla y funciones variantes, así como el alcance de la clase "Cliente" quien tiene acceso a la función plantilla. En este caso las funciones variantes tienen el calificador de acceso incorrecto, en la clase abstracta se encuentra la función variante con un alcance a nivel de paquete y las funciones variantes derivadas presentan protección a nivel de paquete, pero no de herencia, la siguiente función variante presenta carencia de protección al ser pública y solamente la tercera presenta una correcta protección. Esta arquitectura y sus distintas variantes deben ser refactorizadas.

Para escenarios que presentan sobrecarga en las funciones plantillas (Figura 23):

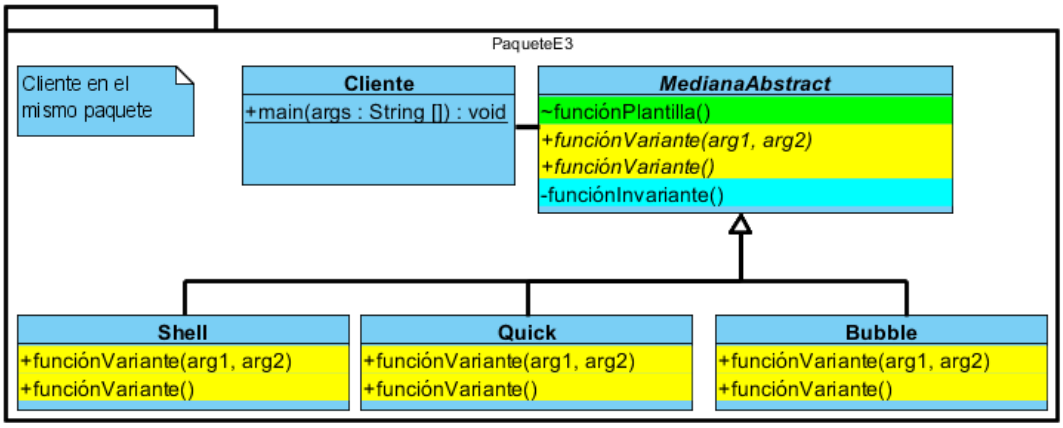


FIGURA 23 ESCENARIO INCORRECTO 4

La sobrecarga de métodos mostrados en la Figura 23 consisten en la definición de métodos con el mismo nombre, pero diferente número de argumentos, la sobrecarga se puede encontrar en las funciones plantilla, en las funciones variantes y funciones invariantes, las funciones plantilla y variantes sobrecargadas son contempladas para su refactorización (ver “Anexo B”), pero las invariantes no los son ya que no se cuenta con la capacidad sintáctica para seguimiento de atributos. En este escenario se muestra la función variante sobrecargada, una con dos argumentos y la siguiente sin argumentos, pero las dos tienen una protección inadecuada al tener un calificador de alcance público, el calificador correcto para las funciones variantes en este caso debe ser el protegido. Se contemplan dos escenarios de refactorización para funciones sobrecargadas del patrón de diseño, así como las distintas combinaciones de calificadores posibles y el respectivo alcance del cliente que accede a la función plantilla.

Para escenario de metamorfosis de función plantilla (Figura 24):

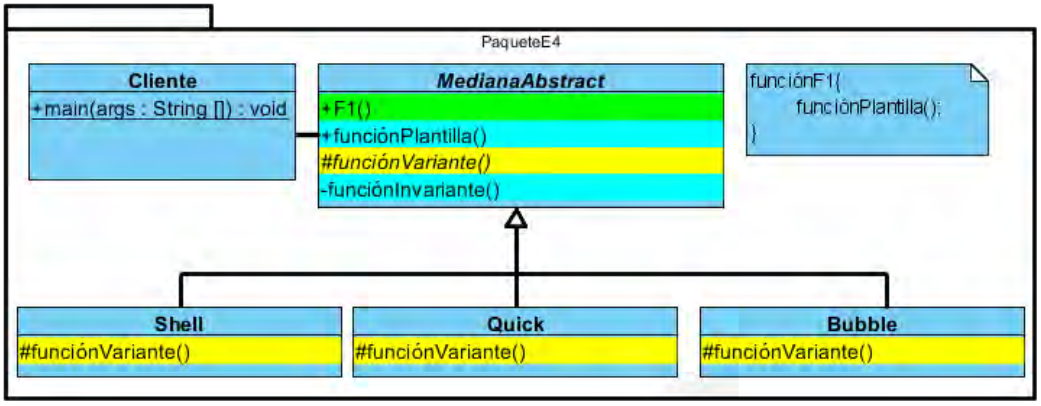


FIGURA 24 ESCENARIO INCORRECTO 5

Este es un escenario mostrado en la Figura 24 en el cual se presenta el patrón de diseño "*Template Method*" pero el acceso del cliente es mediante otra función "F1" independiente al patrón de diseño, a su vez esa función llama a la función plantilla. Este escenario es un claro ejemplo de un patrón de diseño implementado de manera incorrecta pero no por esta situación el patrón de diseño es inadecuado, ya que mantiene su estructura tal cual se dicta en (Gamma et al., 1994), por esta razón se desarrolló una técnica nombrada como "Metamorfosis" que consiste en ajustar los roles de ciertos escenarios problemáticos en cuestión y refactorizarlos de tal manera que se mantenga la correcta protección de la arquitectura. En este escenario se ajustaron los roles de acuerdo a la técnica mencionada para que la función "F1" se le diera tratamiento de función plantilla, la función plantilla original se le ajusta su rol para que se le de tratamiento de función invariante. Se analizaron dos escenarios los cuales presentan una mala implementación del patrón del diseño, pero mantienen la estructura de este, por lo que son contemplados para su refactorización mediante metamorfosis.

Los 16 escenarios analizados se encuentran en el anexo B, cada escenario se repite una vez cambiando el alcance del acceso del cliente, esto es porque dependiendo del alcance del cliente se afecta a la función plantilla en cuestión.

Posteriormente a la refactorización según los criterios y escenarios analizados se ejecuta el proceso de generación de código, este proceso recibe como entrada el código ya refactorizado contenido en la lista compleja de metadatos de clases de objetos, resultado del proceso de refactorización, esta lista recorre las secuencias encontradas en la lista y se aplican a la plantilla *"StringTemplate"* la cual genera el código refactorizado equivalente al código original en funcionalidad.

```
ST

<paquete>

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

  public <if(clase.esAbstracta)>abstract class <endif><if(clase.esInterfaz)>interface <endif>
<if(!clase.esAbstracta && !clase.esInterfaz )>class <endif><clase.nombre><clase.parametros>
<if(clase.esHija && !clase.esHijaIntefaz)> extends <clase.clasePadre><endif>
<if(clase.esHijaIntefaz)> implements <clase.clasePadre><endif> {

   <clase.atributosNoSemicolon :{ atributo | <atributo>;}; separator="\n">

   <clase.metodos :{ metodo |<if(metodo.esOverride)>@Override<endif><\n><metodo.calificador>
<if(metodo.esAbstracto)>abstract <endif><if(metodo.esEstatico)>static <endif>
<metodo.tipoRetorno> <metodo.nombre><metodo.cuerpo> <\n>}; separator="\n\n">

}
```

FIGURA 25 FRAGMENTO DEL STRINGTEMPLATE

Dentro de este proceso se agregó una entidad que solo moverá de manera automatizada el código refactorizado a una ubicación deseada, ya sea para análisis del desarrollador o para sobrescribir las clases problemáticas de la arquitectura que presentan *code smell* debido a la carencia de protección de las funciones plantilla.

El último paso del proceso es el recálculo de métricas de protección de funciones plantilla, este proceso es aplicado al código una vez refactorizado. Se espera que el resultado de la medición muestre una mejora en la propiedad de protección funcional con respecto a la medición del código original.

Capítulo 5. Diseño y Desarrollo de la herramienta

Para efectos de comprobar que el método de refactorización cumple con su cometido, éste fue implementado en un sistema que automatiza el proceso en una herramienta. Esta herramienta de software fue diseñada bajo el estándar UML (ISO/IEC, 2004), donde se utilizaron diagramas de caso de uso, diagramas de secuencias y diagramas de clases.

5.1.- Diagrama General de Casos de Uso para Proteger las "Funciones Plantilla"

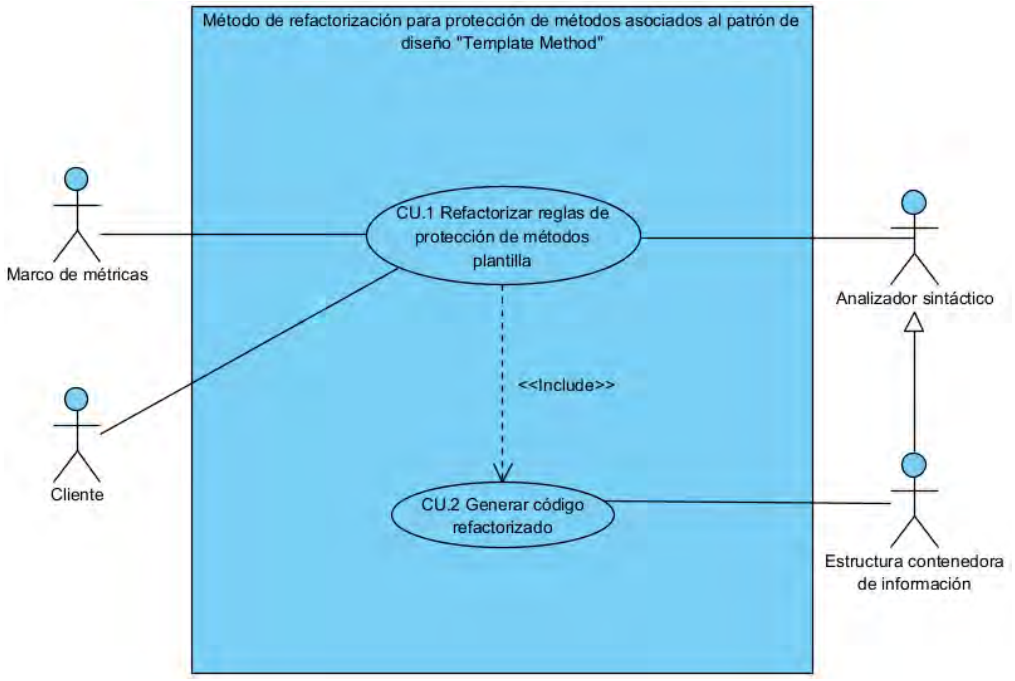


FIGURA 26 DIAGRAMA DE CASO DE USO GENERAL

Este es el principal caso de uso para el desarrollo de la herramienta de refactorización automatizada, en este diagrama de casos de uso los roles que toman los actores “Marco de métricas” y “analizador sintáctico” son de subsistemas. En el análisis se identificaron 17 escenarios alternativos problemáticos y sus mitigaciones que se pueden presentar en arquitecturas orientadas a objetos. En el anexo B, se presentan las plantillas que contienen la información del caso de uso 1 y 2 con sus distintos escenarios.

5.2.- Diagrama de Casos de Uso del Marco de Métricas

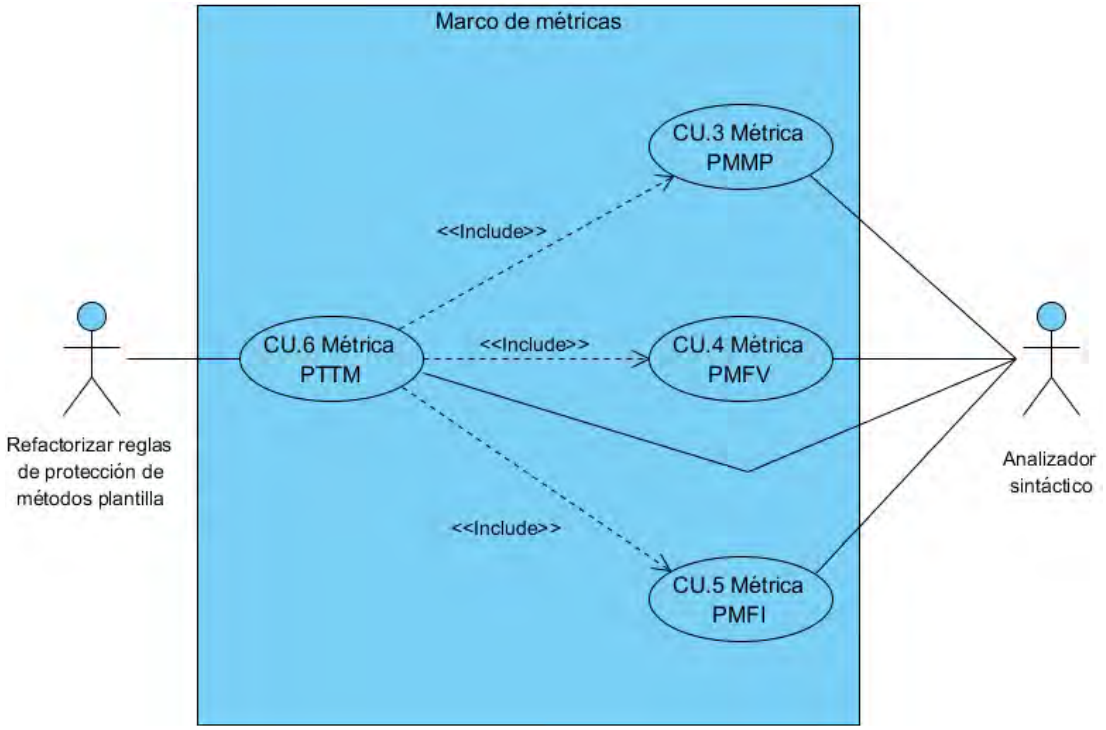


FIGURA 27 DIAGRAMA DEL CASO DE USO DEL MARCO DE MÉTRICAS

El marco de métricas es un subsistema del caso de uso principal cuya tarea es la obtención del resultado del cálculo de las métricas PMMP, PMFV, PMFI y PTTM. Para realizar el cálculo es necesario identificar las funciones y clases plantillas para contabilizarlas y efectuar las operaciones correspondientes, para realizar esta tarea debe participar el analizador sintáctico el cual realizará un análisis sobre el código legado proveniente del CU.1 para identificar las funciones plantilla. En el anexo B se encuentran las plantillas de los casos de uso 3,4,5 y 6.

5.3.- Diagrama de Caso de Uso del Analizador Sintáctico

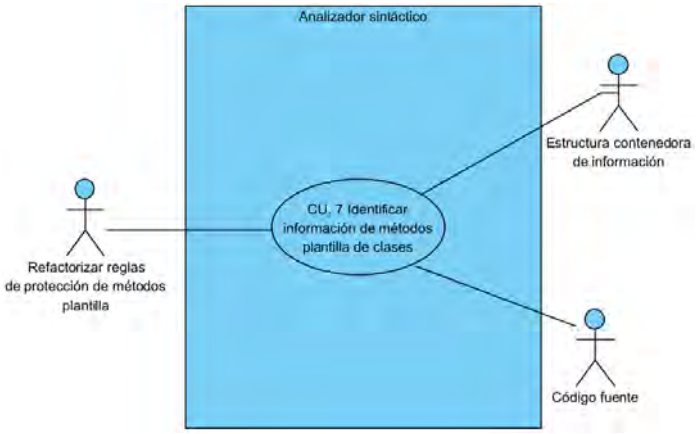


FIGURA 28 DIAGRAMA DE CASO DE USO DEL ANALIZADOR SINTÁCTICO

El analizador sintáctico trabaja como un subsistema al cual se le integran acciones semánticas para obtener información de metadatos del patrón de diseño "*Template Method"*. En el anexo B se encuentra la plantilla del caso de uso 7 el cual consiste en la composición de la estructura contenedora de información y su almacenaje dentro de esta.

5.4.- Diagrama de Secuencia del Marco de Métricas

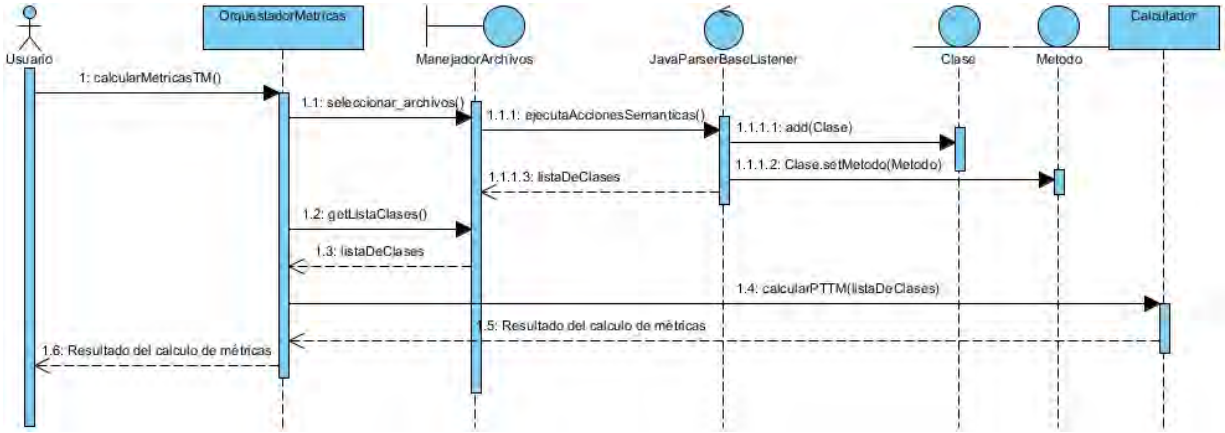


FIGURA 29 DIAGRAMA DE SECUENCIA DEL MARCO DE MÉTRICAS

1: El cliente ejecuta la función para el cálculo de las métricas llamado "calculoMetricas()", esta función se encuentra en la entidad de software llamada "OrquestadorMetricas", ésta se encarga de orquestar el proceso de análisis y cálculo del código a evaluar.

1.1: El orquestador hace llamar a la entidad “ManejadorArchivos” haciendo uso de la función “seleccionar_archivos()” para que el cliente seleccione gráficamente, mediante una ventana, los archivos que contienen el código de la arquitectura a evaluar.

1.1.1: Una vez que se tienen los archivos cargados, se procede a realizar el análisis sintáctico y ejecutar las acciones semánticas, la entidad encargada de realizar estas acciones es la entidad “JavaParserBaseListener”.

1.1.1.1: La entidad “JavaParserBaseListener” almacena los metadatos de clases en una lista compleja de datos (lista de clases).

1.1.1.2: La entidad “JavaParserBaseListener” almacena las representaciones de funciones a las clases correspondientes.

1.1.1.3: Se retorna la lista de clases al manejador de archivos.

1.2: El orquestador realiza una nueva llamada al manejador para solicitar la lista de clases a través de la función “getListaClases()”.

1.3: El manejador de archivos responde con la lista de clases.

1.4: El orquestador solicita a la entidad “Calculador” que se ejecute la métrica PTTM sobre la lista de clases mediante la función “calcularPTTM(listaDeClases)”.

1.5: La entidad “Calculador” responde con el resultado de las métricas PMMP, PMFV, PMFI y PTTM.

1.6: El orquestador presenta los resultados de los cálculos en consola al usuario.

5.5.- Diagrama de Secuencia del Sistema de Refactorización

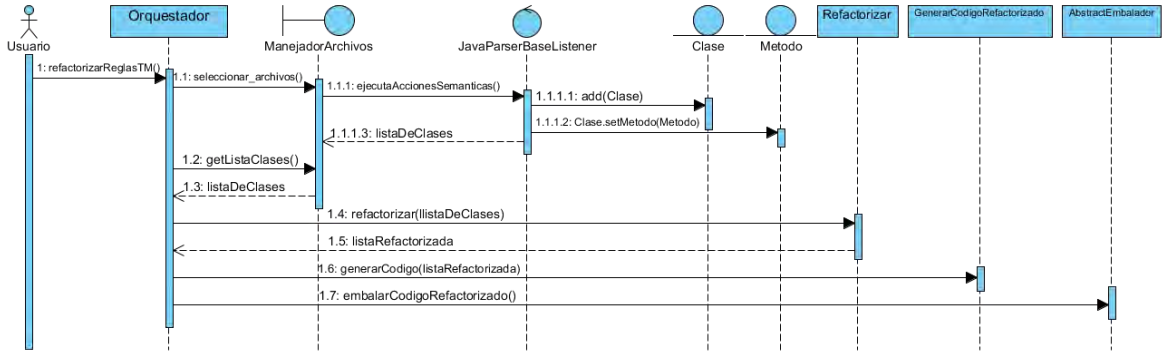


FIGURA 30 DIAGRAMA DE SECUENCIA DEL SISTEMA DE REFACTORIZACIÓN

1: El cliente ejecuta la función “refactorizarReglasTM()” para refactorizar los calificadores de alcance de funciones plantilla, esta función se encuentra en la entidad de software llamada “Orquestador”, esta función se encarga de orquestar el proceso de análisis y de refactorización.

1.1: El orquestador hace llamar a la entidad “ManejadorArchivos” haciendo uso de la función “seleccionar_archivos()” para que el cliente seleccione gráficamente el código de la arquitectura a evaluar mediante una ventana.

1.1.1: Una vez que se tienen los archivos cargados, se procede a realizar el análisis sintáctico y ejecutar las acciones semánticas, la entidad encargada de realizar estas acciones es la entidad “JavaParserBaseListener”.

1.1.1.1: La entidad “JavaParserBaseListener” almacena los metadatos de clases en la lista de datos compleja (lista de clases).

1.1.1.2: La entidad “JavaParserBaseListener” almacena los metadatos de funciones a las clases correspondientes.

1.1.1.3: Se retorna la lista de clases al manejador de archivos.

1.2: El orquestador realiza una nueva llamada al manejador para solicitar la lista de clases a través de la función “getListaClases()”.

1.3: El manejador de archivos responde con la lista de clases.

1.4: El orquestador solicita a la entidad “Refactorizar” que se efectúe la refactorización de calificadores de alcance en las funciones plantilla sobre la lista de clases mediante la función “refactorizar(listaDeClases)”.

1.5: La entidad “Refactorizar” responde con la lista refactorizada.

1.6: El orquestador solicita a la entidad “GenerarCodigoRefactorizado” **generar** en una ubicación específica el código refactorizado equivalente en funcionalidad al código legado, esto a través de librería “*StringTemplate*”.

1.7: El orquestador solicita a la entidad “AbstractEmbalador” **mover** el código generado en una ubicación específica.

5.6.- Diagrama de Clases del Marco de Métricas

La Figura 31 muestra la arquitectura conceptual sin los atributos y métodos del marco de métricas desarrollado en esta tesis, en ella se puede observar la importancia del modelo de objetos utilizado, ya que en este modelo se basa el análisis de las funciones plantilla para la obtención del grado de protección que tienen.

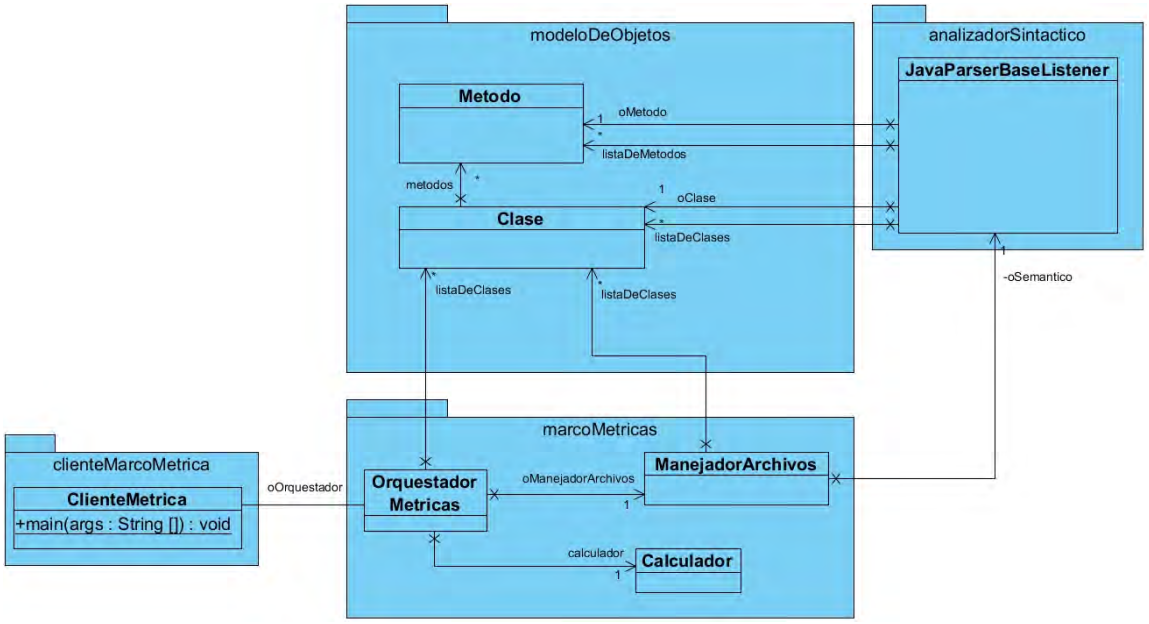


FIGURA 31 DIAGRAMA DE CLASES DEL MARCO DE MÉTRICAS

5.7.- Diagrama de Clases del Sistema de Refactorización

De la misma manera que en el marco de métricas, la lista de clases contiene una representación del código de la arquitectura legada y es procesada múltiples veces sobre las funciones plantilla candidatas. También se observa el uso del patrón de diseño "*Template Method*" en el paquete que distribuye el código generado.

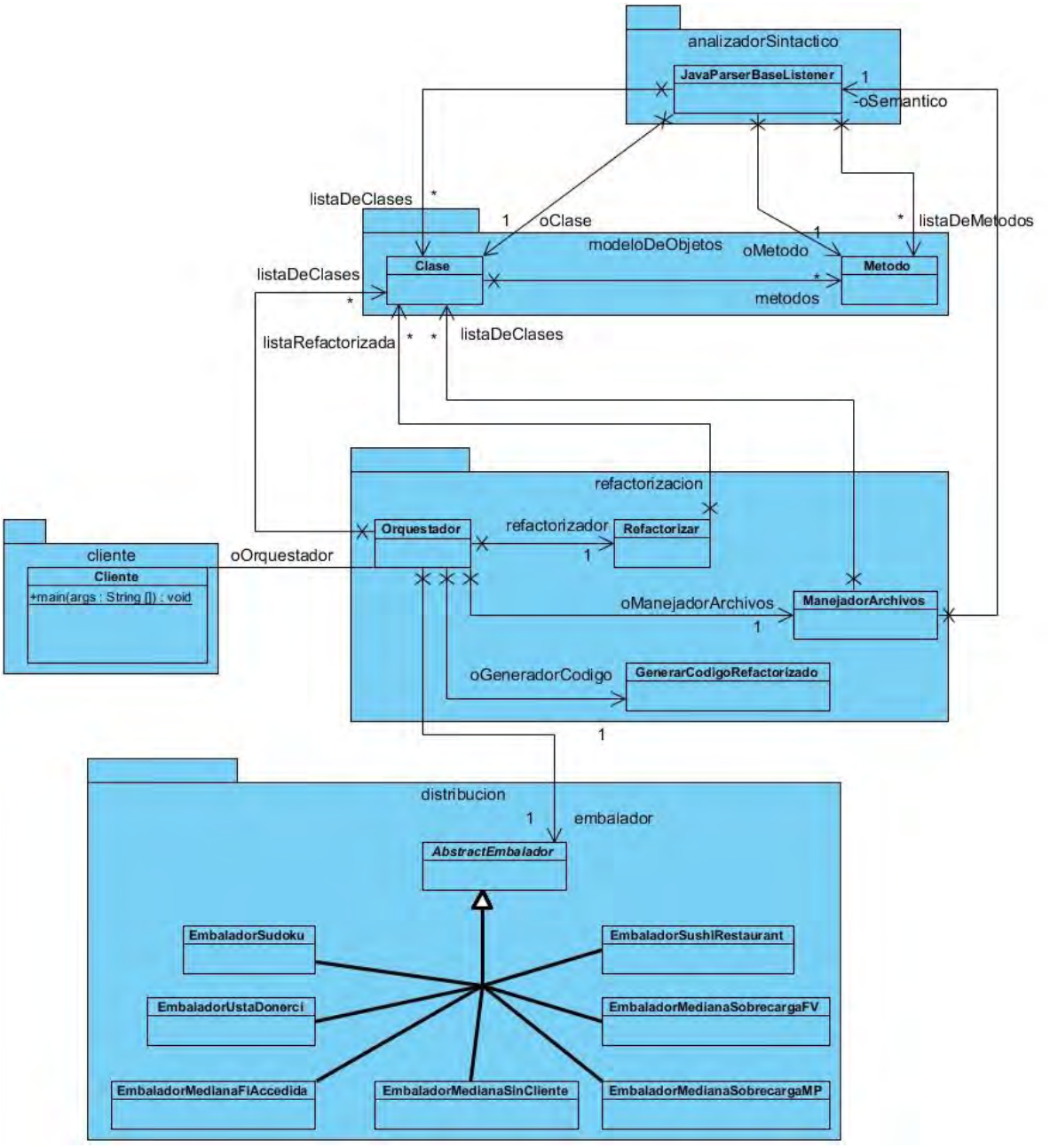


FIGURA 32 DIAGRAMA DE CLASES DEL SISTEMA DE REFACTORIZACIÓN

Capítulo 6. Pruebas

6.1.- Introducción de Pruebas

Se realizó la documentación de pruebas teniendo como base el estándar IEEE Std 829-2008 (Engineering Standards Committee of the IEEE Computer Society, 2008)

6.1.1- Identificador del Documento

ISMRTM0301.

La convención de la nomenclatura de los identificadores es la siguiente:

- IS = Ingeniería de Software.
- MRTM = Método de Refactorización de "*Template Method*".

Tipo de artículo:

- 01 = Módulos de programa.
- 02 = Programas de control.
- 03 = Plan de pruebas.
- 04 = Diseño de pruebas.
- 05 = Casos de pruebas.

Identificador:

- XX = Identificador Numérico.

Ejemplo:

- ISMRTMYYXX

6.1.2.- Alcance

Este plan de pruebas abarca una evaluación completa del método de refactorización de reglas de protección de funciones plantilla y funciones asociadas al patrón de diseño *"Template Method"*, así como la evaluación de las métricas para la protección modular del patrón de diseño *"Template Method".*

6.1.3.- Referencias.

Los siguientes documentos fueron usados como fuente de información para este plan de pruebas:

- Reporte de análisis del sistema de refactorización.
- Reporte de diseño del sistema de refactorización.

6.1.4.- Puntos de Prueba

Los módulos (de tipo 01) que serán probados se muestran a continuación:

Sistema	Función	Identificador
Proceso de análisis estático del código fuente.	Ejecución de un analizador léxico, sintáctico y semántico para la obtención de metadatos del código fuente almacenadas en una estructura compleja de datos de tipo lista.	ISMRTM0100
Marco de métricas de calidad de arquitecturas orientadas a objetos que implementan el patrón de diseño *"Template Method".*	Cálculo de métricas de protección modular del patrón de diseño *"Template Method"*.	ISMRTM0101
Método de refactorización de reglas de protección de funciones plantilla y funciones asociadas al patrón de diseño *"Template Method".*	Subsistema de evaluación y refactorización de reglas de protección de funciones asociadas al patrón de diseño *"Template Method"*.	ISMRTM0102

6.1.5.- Procedimientos de Control de Tareas

Los procedimientos de control para las tareas del programa de aplicación para la refactorización y de las tareas de las métricas para la evaluación de la calidad de la protección de métodos plantilla.

Sistema	Función	Identificador
Programa de aplicación.	Localización del paquete al que pertenece la clase.	ISMRTM0201
Programa de aplicación.	Localización de las importaciones de la clase.	ISMRTM0202

Programa de aplicación.	Localización del nombre de la clase.	ISMRTM0203
Programa de aplicación.	Localización de la clase padre de la clase. (si es que existe).	ISMRTM0204
Programa de aplicación.	Localización de los atributos de la clase.	ISMRTM0205
Programa de aplicación.	Localización de las funciones de la clase.	ISMRTM0206
Programa de aplicación.	Localización de características de las funciones: • Calificador de alcance. • Es tipo abstracto. • Es tipo estático. • Tipo de retorno. • Nombre del método. • Mapa de parámetros. • Cuerpo.	ISMRTM0207
Programa de aplicación.	Localización de funciones plantilla.	ISMRTM0208
Programa de aplicación.	Localización de funciones variantes.	ISMRTM0209
Programa de aplicación.	Localización de funciones invariantes.	ISMRTM0210
Programa de aplicación.	Conteo del número total de funciones plantilla.	ISMRTM0211
Programa de aplicación.	Conteo del número total de funciones variantes.	ISMRTM0212
Programa de aplicación.	Conteo del número total de funciones invariantes.	ISMRTM0213
Programa de aplicación.	Conteo del número total de funciones protegidas asociadas al patrón de diseño *“Template Method”:*	ISMRTM0214

	• Funciones plantilla – “public” o “friendly”. • Funciones variantes – “protected”. • Funciones invariantes – “private”.	
Programa de aplicación.	Cálculo de la métrica *“PMMP”*.	ISMRTM0215
Programa de aplicación.	Cálculo de la métrica *“PMFV”*.	ISMRTM0216
Programa de aplicación.	Cálculo de la métrica *“PMFI”*.	ISMRTM0217
Programa de aplicación.	Cálculo de la métrica *“PTTM”*.	ISMRTM0218
Programa de aplicación.	Validación del calificador de alcance.	ISMRTM0219
Programa de aplicación.	Refactorización de calificadores de alcance y generación de código refactorizado.	ISMRTM0220
Programa de aplicación.	Verificación del funcionamiento del código refactorizado.	ISMRTM0221

6.2.- Características para ser Probadas

Las características que deben ser probadas son:

Diseño de prueba	Identificador
Proceso de análisis del código fuente.	ISMRTM0400
Cálculo de métricas de protección modular del patrón de diseño *“Template Method”*.	ISMRTM0401
Refactorización de acuerdo con las reglas de protección de funciones plantilla y funciones asociadas al patrón de diseño *“Template Method”.*	ISMRTM0402

6.3.- Características para no Probar

A continuación, las características que no serán probadas:

- No se incluirán la totalidad de combinaciones sintácticas del lenguaje java.
- Que el método de refactorización no señale que el código legado esté libre de errores y/o defectos.
- La interfaz al usuario (FrontEnd) no es probada.

6.4.- Enfoque

El enfoque de las pruebas será descrito en las siguientes secciones, siendo el autor de este trabajo de tesis quien realizará las pruebas. De esta manera se puede verificar que las pruebas son de conformidad con lo planificado.

6.4.1.- Pruebas del Proceso de Análisis del Código Fuente

Las pruebas del análisis del código fuente son dadas por la obtención automatizada de los siguientes metadatos del proyecto o sistema legado a refactorizar:

- Clase
- Estructura de una clase:
    - Paquete
    - Importaciones
    - Calificador de alcance o visibilidad
    - Tipo (abstracta, interfaz, subclase)
    - Nombre de la clase
    - Parámetros
    - Atributos
    - Clase padre (Si existen)
    - Mapa de familia (si la clase es padre)
    - Mapa de Subclases (si la clase es padre)
- Funciones
- Estructura de las funciones:
    - Calificador de alcance o visibilidad
    - Es tipo abstracto
    - Es tipo estático
    - Es tipo de retorno
    - Nombre del método
    - Mapa de parámetros
    - Cuerpo

También se verificará el correcto almacenamiento de la lista de clases objeto

6.4.2.- Pruebas de Refactorización

La prueba de refactorización consiste en cambiar el calificador de alcance incorrecto de las funciones plantilla, funciones variantes y funciones invariantes por el calificador de alcance correcto según sea el caso. La validación será dada mediante la aplicación de las métricas de calidad para corroborar el aumento de protección modular, así como la verificación funcional de ejecución del código legado y del código refactorizado, en donde se comprobará que con las mismas entradas se obtiene el mismo comportamiento funcional y los mismos resultados en ambas versiones del código legado.

6.4.3.- Pruebas de Calidad

Las pruebas de calidad son dadas por la aplicación de las métricas de protección modular de funciones plantilla y funciones asociadas al patrón de diseño "*Template Method*" (PMMP, PMFV, PMFI y PTTM), comparando resultados manuales con los resultados automáticos arrojados por el subsistema del marco de métricas.

6.5.- Criterios Aprobado/Desaprobado

6.5.1.- Aprobación/Desaprobación: Pruebas del proceso de análisis del código fuente

El criterio de aprobación o desaprobación del proceso de análisis del código fuente será mediante un análisis comparativo de obtención de la información manual con la obtención automática de la información. El proceso de análisis del código fuente es aprobado si se obtiene la misma información, comprobando la generación correcta de la estructura de datos (lista) que contiene los metadatos de clases de objeto requeridas tanto para el proceso de refactorización como para la aplicación de las métricas de protección modular.

6.5.2.- Aprobación/Desaprobación: Pruebas de refactorización

El criterio de aprobación o desaprobación para las pruebas de refactorización será comparando el comportamiento de la arquitectura de software antes y después del cambio de calificadores de alcance de los métodos y funciones participantes del patrón de diseño "*Template Method*".

6.5.3.- Aprobación/Desaprobación: Pruebas de calidad

Las pruebas de calidad serán aprobadas o desaprobadas mediante la comparación de los resultados obtenidos del cálculo de las métricas manualmente con los resultados obtenidos del cálculo de las métricas automáticamente.

6.6.- Criterios De Suspensión Y Reanudación

Las pruebas no se suspenderán definitivamente, cada vez que se presente una prueba desaprobada se procederá a evaluar y corregir el error.

6.7.- Liberación De Pruebas

La liberación y aceptación de las pruebas será dada mediante la entrada y salida de datos de las pruebas.

6.8.- Diseño de Pruebas

6.8.1.- Diseño De Prueba ISMRTM0400

1. Diseño de prueba:
   ISMRTM0400 - Proceso de análisis del código fuente.
2. Características a ser aprobadas:
   - Se evaluará la correcta obtención de información de metadatos de representación de un proyecto Java en una estructura de datos mediante pruebas unitarias y de integración.
   - Se evaluará la lista contenedora de estructuras complejas de datos.
3. Refinamiento del enfoque:
   El objetivo es obtener una estructura compleja de datos en la cual se representa una clase de objetos del lenguaje de programación Java.
   Se debe comprobar que en la lista compleja se logre almacenar de una a varias estructuras de metadatos.
4. Aprobación/desaprobación de la evaluación de las características:
   La aprobación de los casos de prueba del análisis del código fuente será dada por la comparativa de las partes de las estructuras de datos obtenidas automáticamente con las esperadas manualmente mediante pruebas unitarias. Para que esta comparación sea aprobatoria deben ser resultados iguales los obtenidos de manera automática con los resultados esperados manualmente.
   Finalmente se realizará la prueba de integración para mostrar en consola las estructuras de datos generadas automáticamente.

6.8.2.- Diseño De Prueba ISMRTM0401

1. Diseño de prueba:
   ISMRTM0401 - Cálculo de métricas de protección modular del patrón de diseño *“Template Method”*.
2. Características a ser aprobadas:

- Se evaluará la correcta aplicación de las métricas para calcular la protección modular de funciones plantilla (PMMP, PMFV, PMFI) mediante pruebas unitarias.
- Se evaluará la correcta aplicación de la métrica para calcular la protección total del "*Template Method*" (PTTM) mediante una prueba de integración, esto sucede ya que se está métrica hace uso de las tres anteriores.

3. Refinamiento del enfoque:
   El objetivo es evaluar la correcta aplicación de las métricas para calcular la protección modular del patrón de diseño *"Template Method"* en el software legado.
   El código legado deberá no presentar fallas por lo que será compilado y ejecutado en un compilador para el lenguaje de programación java, posteriormente el marco orientado a objetos para el cálculo de estas métricas realizará un reconocimiento léxico, sintáctico y semántico del código legado para generar una estructura de datos con la información necesaria para el cálculo de las métricas.
4. Aprobación/desaprobación de la evaluación de las características:
   La aprobación de los casos de prueba del cálculo de las métricas será dada por la comparativa de los resultados obtenidos manualmente con los resultados obtenidos automáticamente, para que esta comparación sea aprobatoria deben ser resultados iguales en cada una de las métricas PMMP, PMFV, PMFI y PTTM.

6.8.3.- Diseño De Prueba ISMRTM0402

1. Diseño de prueba:
   ISMRTM0402 - Método de refactorización de reglas de protección de funciones plantilla y funciones asociadas al patrón de diseño *"Template Method".*
2. Características a ser aprobadas:
   - Se evaluará el funcionamiento correcto del sistema de software que implementa el método de refactorización de funciones plantilla y funciones asociadas al patrón de diseño *"Template Method".*
3. Refinamiento del enfoque:
   El objetivo es evaluar el cambio correcto del calificador de alcance de las funciones plantilla, funciones variantes y funciones invariantes del patrón de diseño *"Template Method"* que implementa un código legado y que requieran de la refactorización.

El código legado que presenta deuda técnica no deberá presentar fallas, por lo que antes del proceso automatizado de refactorización será compilado y ejecutado en un compilador para el lenguaje de programación java, y verificar el comportamiento del software legado. Posteriormente el sistema de refactorización de código tomará el archivo con el código fuente legado para realizar un reconocimiento léxico, sintáctico y semántico con el objetivo de generar las estructuras de datos con la información necesaria para el cálculo de métricas y de la refactorización.

4. Aprobación/desaprobación de la evaluación de las características:
   La aprobación de los casos de prueba del método de refactorización será dada por la comparativa de los resultados obtenidos manualmente y por la aplicación de las métricas obtenidas automáticamente en la arquitectura refactorizada. Para que esta comparación sea aprobatoria ambos resultados deben ser iguales.
   Se verificarán los resultados obtenidos de la medición de calidad en la protección modular antes de la refactorización, con los resultados obtenidos de la medición de calidad en la protección modular después de la refactorización.

6.9.- Especificación de Casos de Prueba

6.9.1.- Caso de Prueba ISMRTM0500

Nombre del Caso de prueba: Mediana.

El programa “Mediana” es un sistema con el objetivo de obtener la mediana implementando el patrón de diseño “*Template Method*” el cual utiliza distintos tipos de ordenamientos dependiendo del comportamiento funcional que se requiera

Características a probar:

Características a probar	Diseño de la prueba
Localización del paquete al que pertenece la clase.	ISMRTM0400
Localización de las importaciones de la clase.	ISMRTM0400
Localización del nombre de la clase.	ISMRTM0400
Localización de la clase padre de la clase. (si es que existe).	ISMRTM0400
Localización de los atributos de la clase.	ISMRTM0400

Localización de las funciones de la clase.	ISMRTM0400
Identificar si la clase es abstracta	ISMRTM0400
Identificar si la clase es una interfaz	ISMRTM0400
Identificar si la clase es hija	ISMRTM0400
Localización de metadatos de las funciones: • Calificador de alcance. • Es tipo abstracto. • Es tipo estático. • Es tipo de retorno. • Nombre del método. • Mapa de parámetros. • Cuerpo.	ISMRTM0400
Generación de la estructura compleja de información completa.	ISMRTM0400
Almacenamiento de la estructura compleja de información en una lista contenedora.	ISMRTM0400

Como entrada se recibe el código legado compilado y probado, el código se analiza para obtener los metadatos de clases de objetos en una estructura de datos compleja (lista) contenedora de dicha información.

Como salida se espera mostrar en consola la lista contenedora de información pobladas con los metadatos representativos de clases de objetos en Java.

6.9.2.- Caso de Prueba ISMRTM0501

Nombre del Caso de prueba: Sudoku.

El programa “Sudoku” fue desarrollado utilizando Java 1.6 sin bibliotecas externas, se compone de 30 clases, su objetivo principal es generar juegos de sudoku aplicando el “*Template Method*” para los distintos niveles que posee. El juego tiene tres dificultades diferentes: FÁCIL, NORMAL y DIFÍCIL. La puntuación se calcula en función del tiempo transcurrido, el número de errores y la dificultad actual.

Características a probar:

Características a probar	**Diseño de la prueba**

Localización de funciones plantilla.	ISMRTM0401
Localización de funciones variantes.	ISMRTM0401
Localización de funciones invariantes.	ISMRTM0401
Conteo del número total de funciones plantilla.	ISMRTM0401
Conteo del número total de funciones variantes.	ISMRTM0401
Conteo del número total de funciones invariantes.	ISMRTM0401
Conteo del número total de funciones protegidas asociadas al patrón de diseño *"Template Method":* • Funciones plantilla – "public" o "friendly". • Funciones variantes – "protected". • Funciones invariantes – "private".	ISMRTM0401
Cálculo de la métrica *"PMMP"*.	ISMRTM0401
Cálculo de la métrica *"PMFV"*.	ISMRTM0401
Cálculo de la métrica *"PMFI"*.	ISMRTM0401
Cálculo de la métrica *"PTTM"*.	ISMRTM0401
Localización de funciones plantilla.	ISMRTM0401

Las clases pertenecientes al sistema "Sudoku" serán la entrada para el cálculo de las métricas. Este sistema debe ser compilado y ejecutado previamente.

Como salida se esperan los resultados calculados de cada una de las métricas.

6.9.3.- Caso de Prueba ISMRTM0502

Nombre del Caso de prueba: Sudoku (Pure Java Implementation).

El programa "Sudoku" fue desarrollado utilizando Java 1.6 sin bibliotecas externas, se compone de 30 clases. Su objetivo principal es generar juegos de sudoku aplicando el "*Template Method*" para los distintos niveles que posee. El juego tiene tres dificultades diferentes: FÁCIL, NORMAL y DIFÍCIL. La puntuación se calcula en función del tiempo transcurrido, el número de errores y la dificultad actual.

Características a probar:

Características a probar	Diseño de la prueba
Función de validación del calificador de alcance.	ISMRTM0402
Función de refactorización de calificadores de alcance y generación de código refactorizado.	ISMRTM0402
Verificación del funcionamiento del código refactorizado.	ISMRTM0402

Las clases pertenecientes al sistema Sudoku serán la entrada para el proceso de refactorización. Este sistema debe ser compilado y ejecutado previamente.

Como salida se esperan las clases que pertenecen al sistema Sudoku refactorizadas libre de la deuda técnica originada por la carencia de protección de funciones plantilla.

6.9.4.- Caso de Prueba ISMRTM0503

Nombre del Caso de prueba: SushiRestaurant.

Este sistema incluye 15 patrones de diseño y cuenta con 67 clases, su finalidad del sistema es para ser usado como base para construir otro sistema más completo para un restaurante de sushi.

Características a probar:

Características a probar	Diseño de la prueba
Localización de funciones plantilla.	ISMRTM0401
Localización de funciones variantes.	ISMRTM0401
Localización de funciones invariantes.	ISMRTM0401
Conteo del número total de funciones plantilla.	ISMRTM0401
Conteo del número total de funciones variantes.	ISMRTM0401
Conteo del número total de funciones invariantes.	ISMRTM0401
Conteo del número total de funciones protegidas asociadas al patrón de diseño *"Template Method":* • Funciones plantilla – "public" o "friendly". • Funciones variantes – "protected".	ISMRTM0401

• Funciones invariantes – “private”.	
Cálculo de la métrica *“PMMP”*.	ISMRTM0401
Cálculo de la métrica *“PMFV”*.	ISMRTM0401
Cálculo de la métrica *“PMFI”*.	ISMRTM0401
Cálculo de la métrica *“PTTM”*.	ISMRTM0401
Localización de funciones plantilla.	ISMRTM0401

Las clases pertenecientes al sistema “SushiRestaurant” serán la entrada para el cálculo de las métricas. Este sistema debe ser compilado y ejecutado previamente.

Como salida se esperan los resultados calculados de cada una de las métricas.

6.9.5.- Caso de Prueba ISMRTM0504

Nombre del Caso de prueba: SushiRestaurant.

Este sistema incluye 15 patrones de diseño y cuenta con 67 clases, su finalidad del sistema es para ser usado como base para construir otro sistema más completo para un restaurante de sushi.

Características a probar:

Características a probar	Diseño de la prueba
Método de validación del calificador de alcance.	ISMRTM0402
Método de refactorización de calificadores de alcance y generación de código refactorizado.	ISMRTM0402
Verificación del funcionamiento del código refactorizado.	ISMRTM0402

Las clases pertenecientes al sistema “SushiRestaurant” serán la entrada para el proceso de refactorización. Este sistema debe ser compilado y ejecutado previamente.

Como salida se esperan las clases que pertenecen al sistema “SushiRestaurant” refactorizadas libre de la deuda técnica originada por la carencia de protección de funciones plantilla.

6.9.6.- Caso de Prueba ISMRTM0505

Nombre del Caso de prueba: Usta Donerci.

El sistema “Usta Donerci” incluye 4 patrones de diseño y cuenta con 89 clases, su finalidad del sistema es para ser usado como base para construir otro sistema más completo para un restaurante que consiste en vender comida rápida, döner, albóndigas y bebidas. Cuando el cliente llega a este restaurante, puede seleccionar su pedido y diseñar el pedido con un producto personalizado que es decorador. Después de crear el pedido, la camarera entrega estos pedidos al cocinero del cliente. El cocinero es responsable de cocinar el orden comestible y preparar la bebida según el pedido del cliente.

Características a probar:

Características a probar	**Diseño de la prueba**
Localización de funciones plantilla.	ISMRTM0401
Localización de funciones variantes.	ISMRTM0401
Localización de funciones invariantes.	ISMRTM0401
Conteo del número total de funciones plantilla.	ISMRTM0401
Conteo del número total de funciones variantes.	ISMRTM0401
Conteo del número total de funciones invariantes.	ISMRTM0401
Conteo del número total de funciones protegidas asociadas al patrón de diseño *“Template Method”:* • Funciones plantilla – “public” o “friendly”. • Funciones variantes – “protected”. • Funciones invariantes – “private”.	ISMRTM0401
Cálculo de la métrica *“PMMP”*.	ISMRTM0401
Cálculo de la métrica *“PMFV”*.	ISMRTM0401
Cálculo de la métrica *“PMFI”*.	ISMRTM0401
Cálculo de la métrica *“PTTM”*.	ISMRTM0401
Localización de funciones plantilla.	ISMRTM0401

Las clases pertenecientes al sistema “Usta Donerci” serán la entrada para el cálculo de las métricas. Este sistema debe ser compilado y ejecutado previamente.

Como salida se esperan los resultados calculados de cada una de las métricas.

6.9.7.- Caso de Prueba ISMRTM0506

Nombre del Caso de prueba: Usta Donerci.

El sistema “Usta Donerci” incluye 4 patrones de diseño y cuenta con 89 clases, su finalidad es ser usado como base para construir otro sistema más completo para un restaurante que consiste en vender comida rápida, döner, albóndigas y bebidas. Cuando el cliente llega a este restaurante, puede seleccionar su pedido y diseñar el pedido con un producto personalizado que es decorador. Después de crear el pedido, la camarera entrega estos pedidos al cocinero del cliente. El cocinero es responsable de cocinar el orden comestible y preparar la bebida según el pedido del cliente.

Características a probar:

Características a probar	Diseño de la prueba
Método de validación del calificador de alcance.	ISMRTM0402
Método de refactorización de calificadores de alcance y generación de código refactorizado.	ISMRTM0402
Verificación del funcionamiento del código refactorizado.	ISMRTM0402

Las clases pertenecientes al sistema “Usta Donerci” serán la entrada para el proceso de refactorización. Este sistema debe ser compilado y ejecutado previamente.

Como salida se esperan las clases que pertenecen al sistema “Usta Donerci” refactorizadas libre de la deuda técnica originada por la carencia de protección de funciones plantilla.

6.10.- Ejecución del Plan de Pruebas

6.10.1.- Caso de Prueba ISMRTM0500

Nombre del Caso de prueba: Mediana.

En la Figura 33 se muestra la arquitectura del sistema para realizar el cálculo de la mediana, este sistema está escrito en lenguaje Java y previamente ha sido compilado y revisado su funcionamiento.

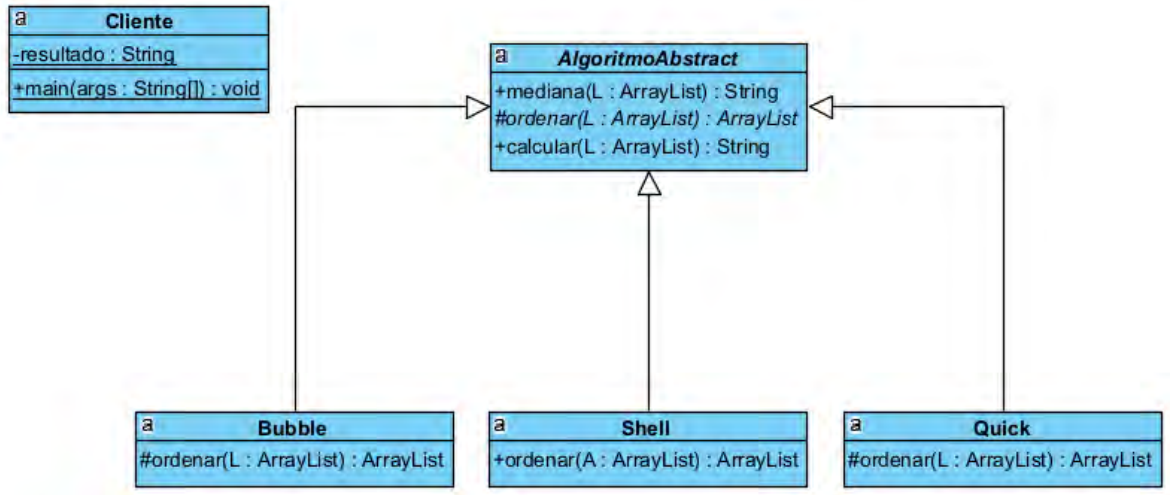
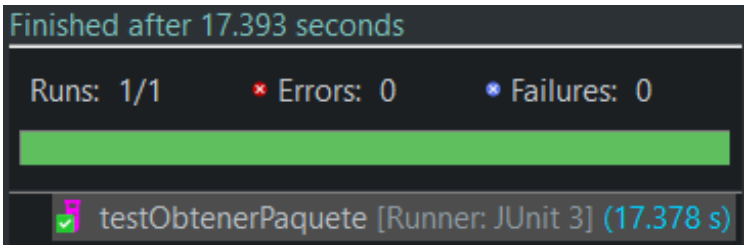
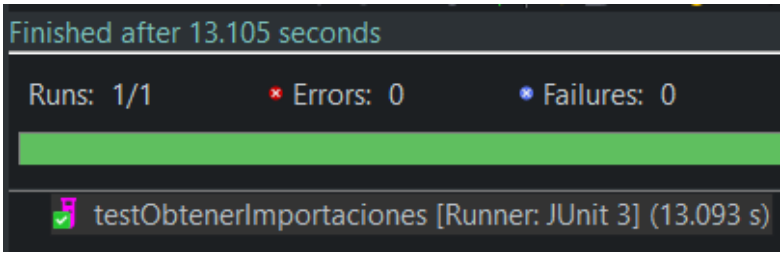


FIGURA 33 CASO DE PRUEBA MEDIANA

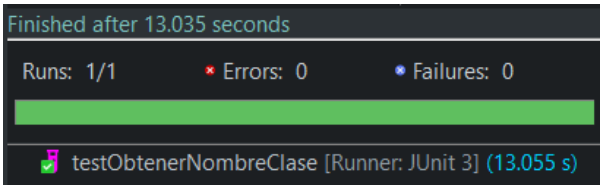
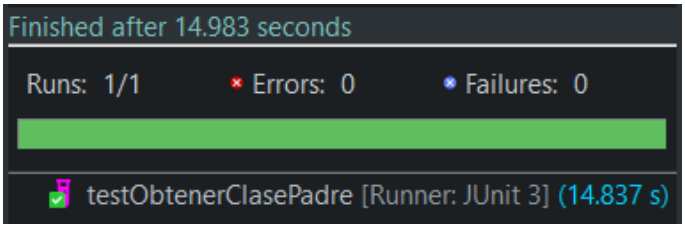
Se procede a ejecutar las pruebas unitarias observando que se cumplan las condiciones comparativas manuales con las automáticas resultantes. Al final la prueba de integración muestra en consola la lista contenedora poblada:

Características a probar	Pruebas unitarias
Localización del paquete al que pertenece la clase.	 

```java
public void testObtenerPaquete() throws IOException {
        escenario();
        for (int i = 0; i < listaDeClases.size(); i++) {
            switch (i) {
            case 0: {
                assertEquals("clienteDentro", listaDeClases.get(i).getPaquete());
                break;
            }
            case 1: {
                assertEquals("clienteDentro", listaDeClases.get(i).getPaquete());
                break;
            }
            case 2: {
                assertEquals("clienteDentro", listaDeClases.get(i).getPaquete());
                break;
            }
            case 3: {
                assertEquals("clienteDentro", listaDeClases.get(i).getPaquete());
                break;
            }
            case 4: {
                assertEquals("clienteDentro", listaDeClases.get(i).getPaquete());
                break;
            }
            }
        }
    }
```

Localización de las importaciones de la clase.	 

```java
public void testObtenerImportaciones() throws IOException {
        escenario();
        ArrayList<String> imports = new ArrayList();
        for (int i = 0; i < listaDeClases.size(); i++) {
            imports.clear();
            switch (i) {
            case 0: {
                imports.add("import java.util.ArrayList ;");
                assertEquals(imports, listaDeClases.get(i).getImportaciones());
                break;
            }
            case 1: {
                imports.add("import java.util.ArrayList ;");
                assertEquals(imports, listaDeClases.get(i).getImportaciones());
                break;
            }
            case 2: {
                imports.add("import java.util.ArrayList ;");
                assertEquals(imports, listaDeClases.get(i).getImportaciones());
                break;
            }
            case 3: {
                imports.add("import java.util.ArrayList ;");
                imports.add("import java.util.Collections ;");
                imports.add("import java.util.Comparator ;");
                assertEquals(imports, listaDeClases.get(i).getImportaciones());
                break;
            }
            case 4: {
                imports.add("import java.util.ArrayList ;");
                assertEquals(imports, listaDeClases.get(i).getImportaciones());
                break;
            }
            }
        }
    }
```

Localización del nombre de la clase.	 
Localización de la clase padre de la clase. (si es que existe).	 

```java
public void testObtenerNombreClase() throws IOException {
        escenario();
        for (int i = 0; i < listaDeClases.size(); i++) {
            switch (i) {
                case 0: {
                    assertEquals("AlgoritmoAbstract", listaDeClases.get(i).getNombre());
                    break;
                }
                case 1: {
                    assertEquals("Bubble", listaDeClases.get(i).getNombre());
                    break;
                }
                case 2: {
                    assertEquals("Cliente", listaDeClases.get(i).getNombre());
                    break;
                }
                case 3: {
                    assertEquals("Quick", listaDeClases.get(i).getNombre());
                    break;
                }
                case 4: {
                    assertEquals("Shell", listaDeClases.get(i).getNombre());
                    break;
                }
            }
        }
    }
```

Finished after 13.035 seconds

Runs: 1/1 Errors: 0 Failures: 0

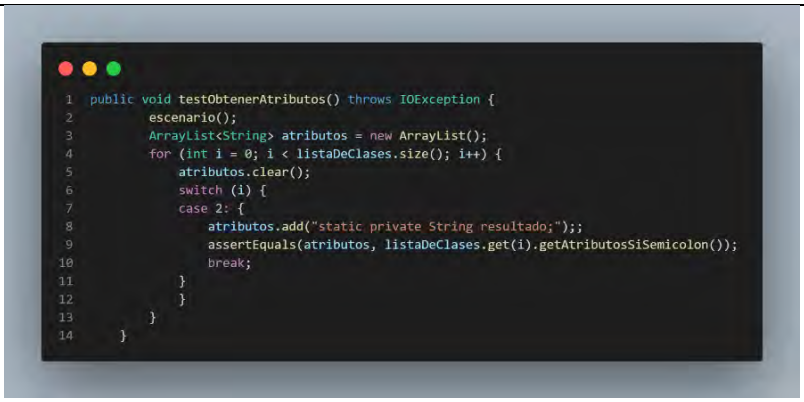
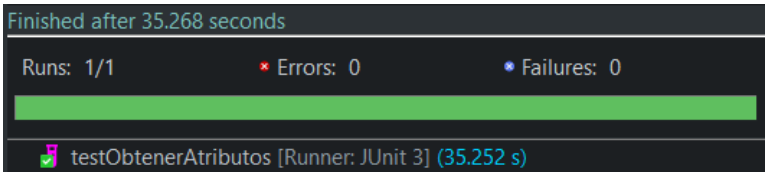
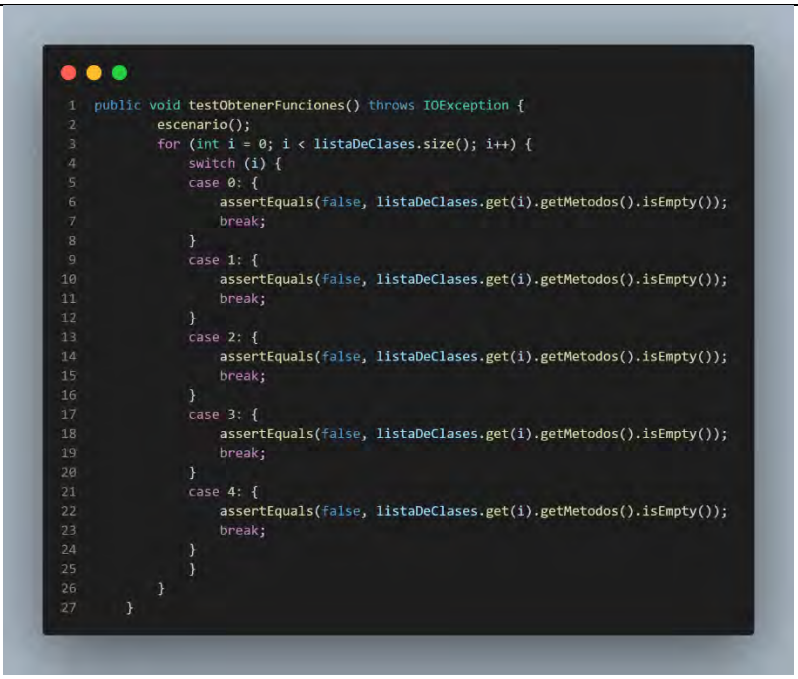
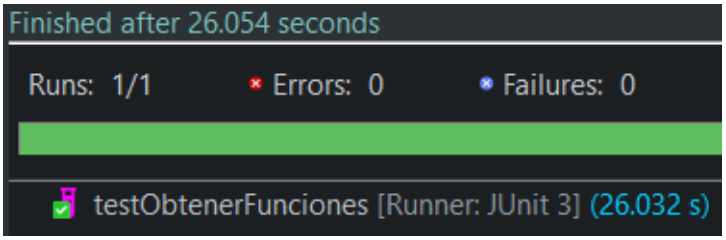
testObtenerNombreClase [Runner: JUnit 3] (13.055 s)

```java
public void testObtenerClasePadre() throws IOException {
        escenario();
        for (int i = 0; i < listaDeClases.size(); i++) {
            switch (i) {
            case 1: {
                assertEquals("AlgoritmoAbstract", listaDeClases.get(i).getClasePadre());
                break;
            }
            case 3: {
                assertEquals("AlgoritmoAbstract", listaDeClases.get(i).getClasePadre());
                break;
            }
            case 4: {
                assertEquals("AlgoritmoAbstract", listaDeClases.get(i).getClasePadre());
                break;
            }
            }
        }
    }
```

Finished after 14.983 seconds

Runs: 1/1 Errors: 0 Failures: 0

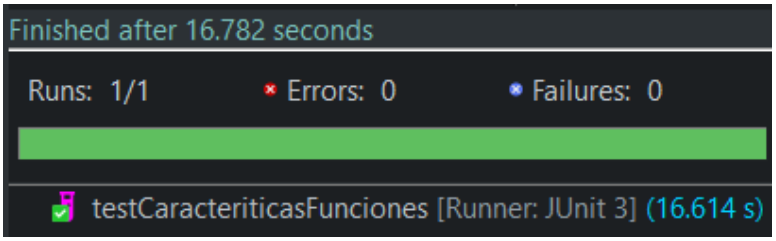
testObtenerClasePadre [Runner: JUnit 3] (14.837 s)

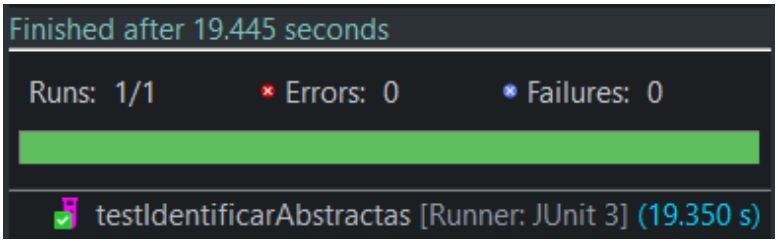
Localización de los atributos de la clase.	 
Localización de las funciones de la clase.	 

```java
public void testObtenerAtributos() throws IOException {
        escenario();
        ArrayList<String> atributos = new ArrayList();
        for (int i = 0; i < listaDeClases.size(); i++) {
            atributos.clear();
            switch (i) {
            case 2: {
                atributos.add("static private String resultado;");;
                assertEquals(atributos, listaDeClases.get(i).getAtributosSiSemicolon());
                break;
            }
            }
        }
    }
```

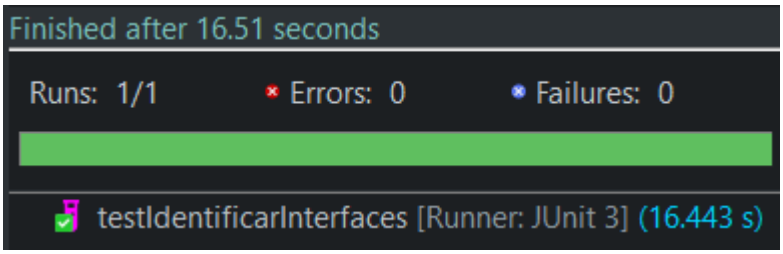


```java
public void testObtenerFunciones() throws IOException {
        escenario();
        for (int i = 0; i < listaDeClases.size(); i++) {
            switch (i) {
            case 0: {
                assertEquals(false, listaDeClases.get(i).getMetodos().isEmpty());
                break;
            }
            case 1: {
                assertEquals(false, listaDeClases.get(i).getMetodos().isEmpty());
                break;
            }
            case 2: {
                assertEquals(false, listaDeClases.get(i).getMetodos().isEmpty());
                break;
            }
            case 3: {
                assertEquals(false, listaDeClases.get(i).getMetodos().isEmpty());
                break;
            }
            case 4: {
                assertEquals(false, listaDeClases.get(i).getMetodos().isEmpty());
                break;
            }
            }
        }
    }
```

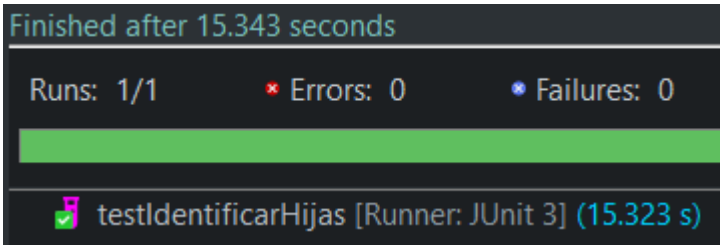
Localización de características de las funciones: • Calificador de alcance. • Es tipo abstracto. • Es tipo estático. • Es tipo de retorno. • Nombre del método. • Mapa de parámetros. • Cuerpo.	 

Identificar si la clase es abstracta	 

```java
public void testIdentificarAbstractas() throws IOException {
        escenario();
        for (int i = 0; i < listaDeClases.size(); i++) {
            switch (i) {
            case 0: {
                assertEquals(true, listaDeClases.get(i).getEsAbstracta());
                break;
            }
            case 1: {
                assertEquals(false, listaDeClases.get(i).getEsAbstracta());
                break;
            }
            case 2: {
                assertEquals(false, listaDeClases.get(i).getEsAbstracta());
                break;
            }
            case 3: {
                assertEquals(false, listaDeClases.get(i).getEsAbstracta());
                break;
            }
            case 4: {
                assertEquals(false, listaDeClases.get(i).getEsAbstracta());
                break;
            }
            }
        }
    }
```

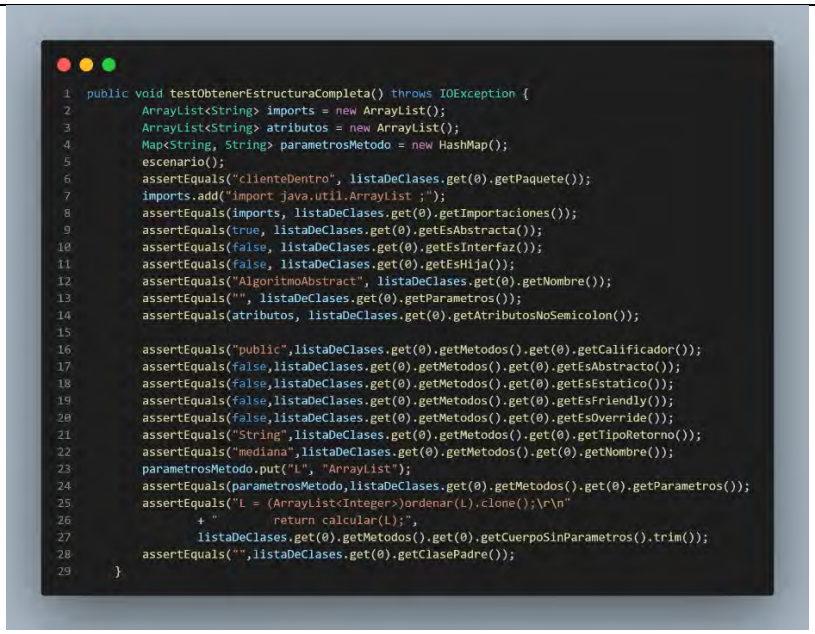
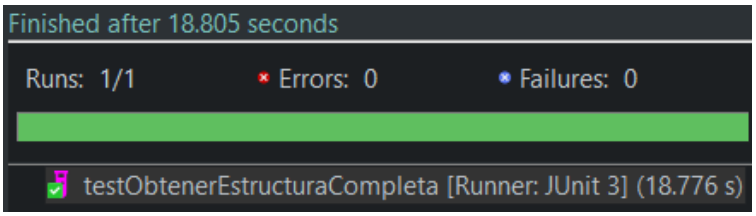
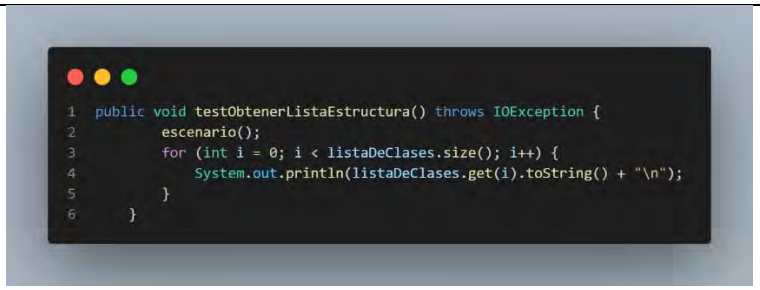
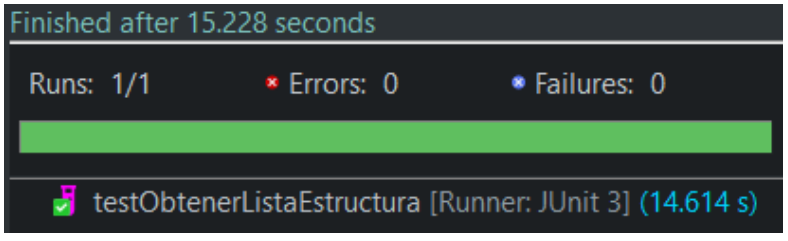
Identificar si la clase es una interfaz	 

```java
public void testIdentificarInterfaces() throws IOException {
        escenario();
        for (int i = 0; i < listaDeClases.size(); i++) {
            switch (i) {
            case 0: {
                assertEquals(false, listaDeClases.get(i).getEsInterfaz());
                break;
            }
            case 1: {
                assertEquals(false, listaDeClases.get(i).getEsInterfaz());
                break;
            }
            case 2: {
                assertEquals(false, listaDeClases.get(i).getEsInterfaz());
                break;
            }
            case 3: {
                assertEquals(false, listaDeClases.get(i).getEsInterfaz());
                break;
            }
            case 4: {
                assertEquals(false, listaDeClases.get(i).getEsInterfaz());
                break;
            }
            }
        }
    }
```

Identificar si la clase es hija	 

```java
public void testIdentificarHijas() throws IOException {
        escenario();
        for (int i = 0; i < listaDeClases.size(); i++) {
            switch (i) {

            case 0: {
                assertEquals(false, listaDeClases.get(i).getEsHija());
                break;
            }
            case 1: {
                assertEquals(true, listaDeClases.get(i).getEsHija());
                break;
            }
            case 2: {
                assertEquals(false, listaDeClases.get(i).getEsHija());
                break;
            }
            case 3: {
                assertEquals(true, listaDeClases.get(i).getEsHija());
                break;
            }
            case 4: {
                assertEquals(true, listaDeClases.get(i).getEsHija());
                break;
            }
            }
        }
    }
```

Generación de estructura compleja de información completa.	 
Almacenamiento de las estructuras complejas de información en una lista contenedora.	 

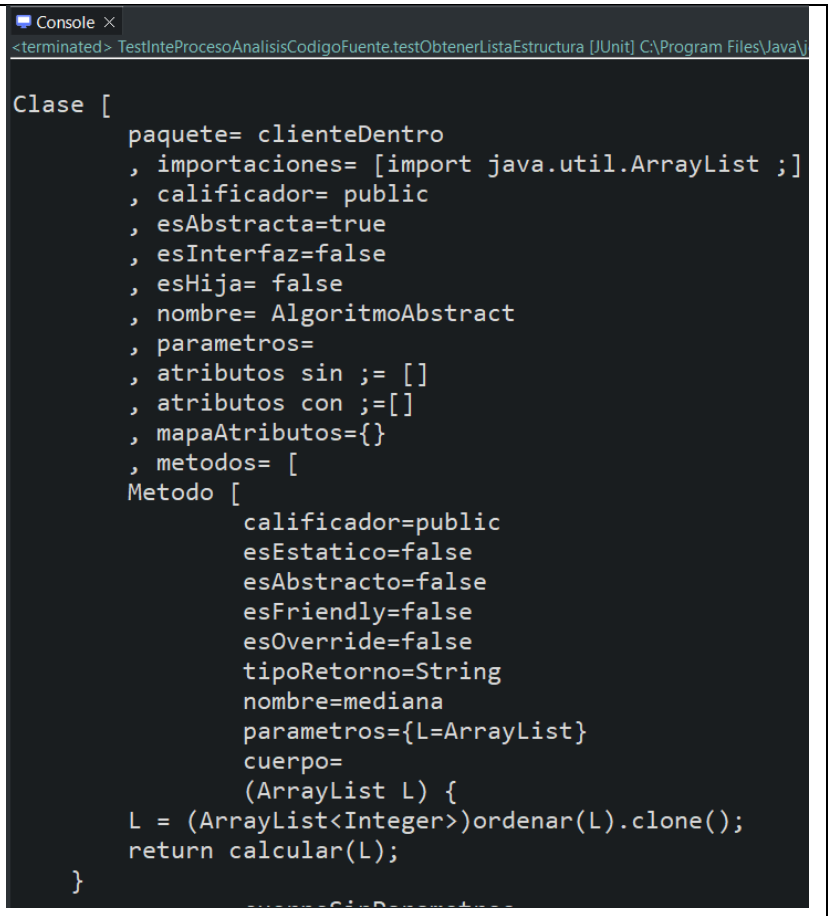
```java
public void testObtenerEstructuraCompleta() throws IOException {
        ArrayList<String> imports = new ArrayList();
        ArrayList<String> atributos = new ArrayList();
        Map<String, String> parametrosMetodo = new HashMap();
        escenario();
        assertEquals("clienteDentro", listaDeClases.get(0).getPaquete());
        imports.add("import java.util.ArrayList ;");
        assertEquals(imports, listaDeClases.get(0).getImportaciones());
        assertEquals(true, listaDeClases.get(0).getEsAbstracta());
        assertEquals(false, listaDeClases.get(0).getEsInterfaz());
        assertEquals(false, listaDeClases.get(0).getEsHija());
        assertEquals("AlgoritmoAbstract", listaDeClases.get(0).getNombre());
        assertEquals("", listaDeClases.get(0).getParametros());
        assertEquals(atributos, listaDeClases.get(0).getAtributosNoSemicolon());

        assertEquals("public",listaDeClases.get(0).getMetodos().get(0).getCalificador());
        assertEquals(false,listaDeClases.get(0).getMetodos().get(0).getEsAbstracto());
        assertEquals(false,listaDeClases.get(0).getMetodos().get(0).getEsEstatico());
        assertEquals(false,listaDeClases.get(0).getMetodos().get(0).getEsFriendly());
        assertEquals(false,listaDeClases.get(0).getMetodos().get(0).getEsOverride());
        assertEquals("String",listaDeClases.get(0).getMetodos().get(0).getTipoRetorno());
        assertEquals("mediana",listaDeClases.get(0).getMetodos().get(0).getNombre());
        parametrosMetodo.put("L", "ArrayList");
        assertEquals(parametrosMetodo,listaDeClases.get(0).getMetodos().get(0).getParametros());
        assertEquals("L = (ArrayList<Integer>)ordenar(L).clone();\r\n"
                + "        return calcular(L);",
                listaDeClases.get(0).getMetodos().get(0).getCuerpoSinParametros().trim());
        assertEquals("",listaDeClases.get(0).getClasePadre());
    }
```

```
Finished after 18.805 seconds
Runs: 1/1   Errors: 0   Failures: 0
testObtenerEstructuraCompleta [Runner: JUnit 3] (18.776 s)
```

```java
public void testObtenerListaEstructura() throws IOException {
        escenario();
        for (int i = 0; i < listaDeClases.size(); i++) {
            System.out.println(listaDeClases.get(i).toString() + "\n");
        }
    }
```

```
Finished after 15.228 seconds
Runs: 1/1   Errors: 0   Failures: 0
testObtenerListaEstructura [Runner: JUnit 3] (14.614 s)
```

	 La consola despliega las clases correctamente

6.10.2.- Caso de Prueba ISMRTM0501

Nombre del Caso de prueba: Sudoku (Pure Java Implementation).

En la Figura 34 se muestra el segmento del diagrama de clases señalado en rojo la carencia de protección y en verde la correcta protección, este segmento es donde se presenta el patrón de diseño "*Template Method*" del sistema Sudoku, el sistema está escrito en lenguaje Java y, previamente, ha sido compilado y revisado su funcionamiento.

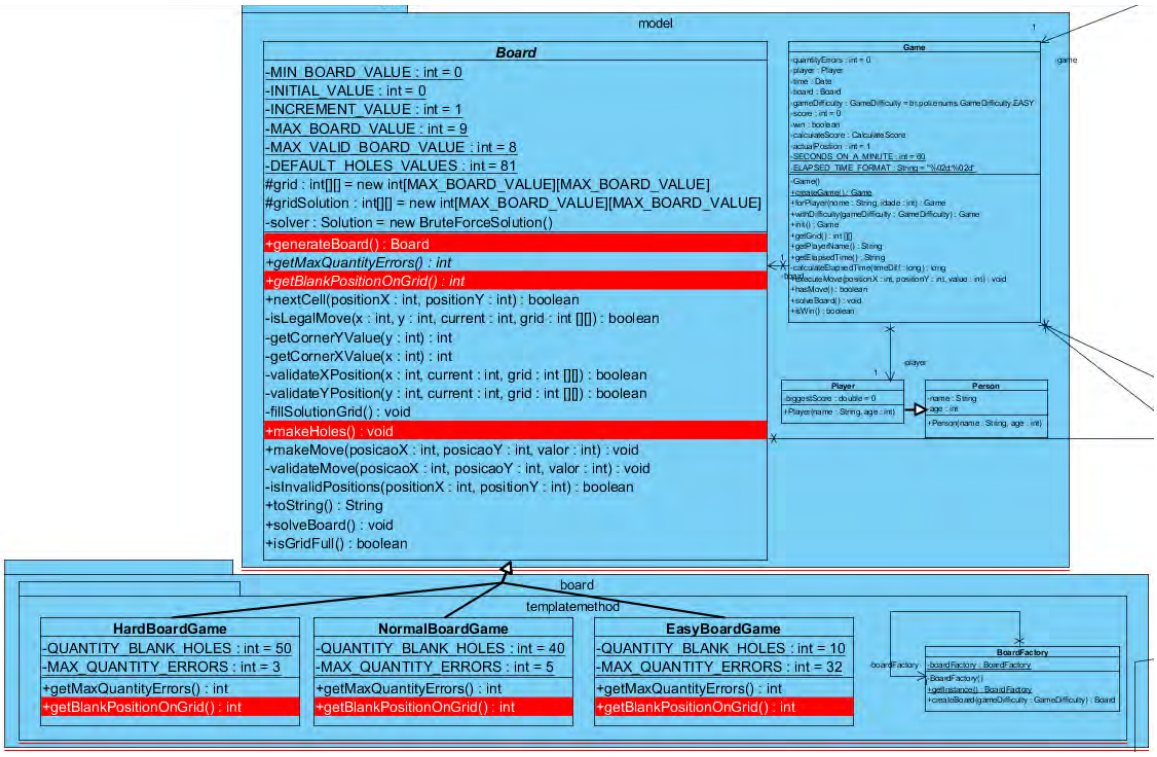
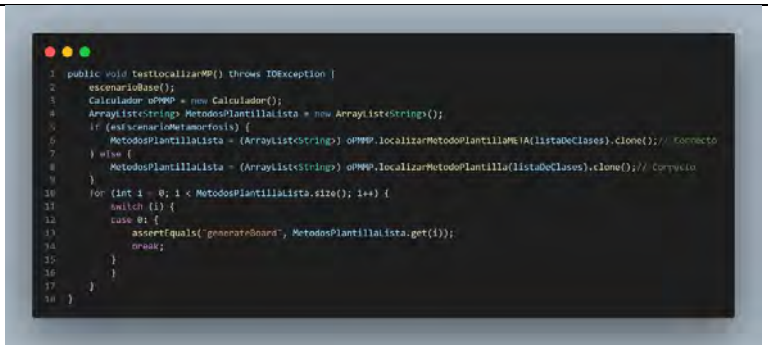
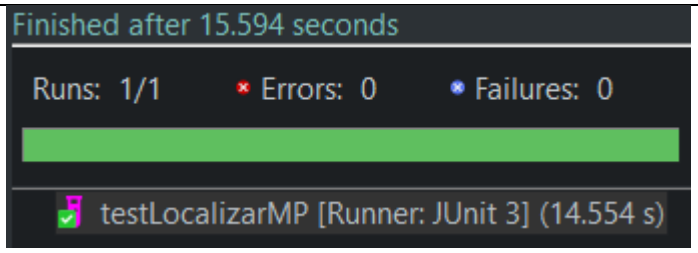
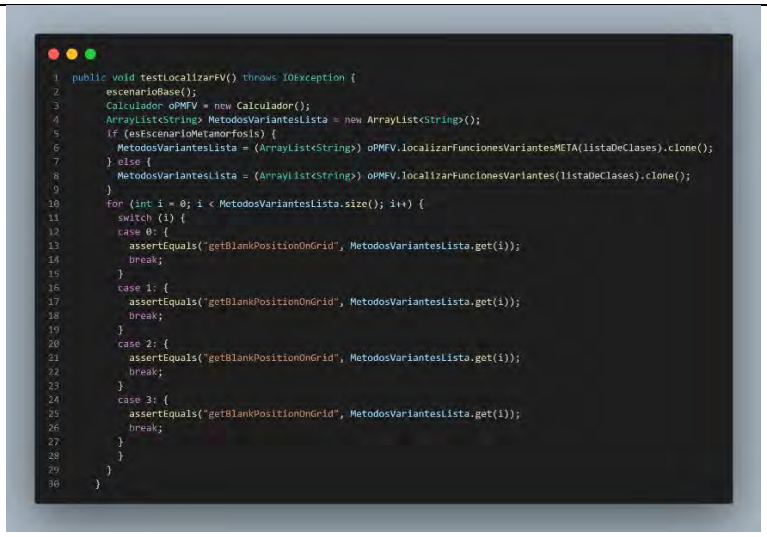
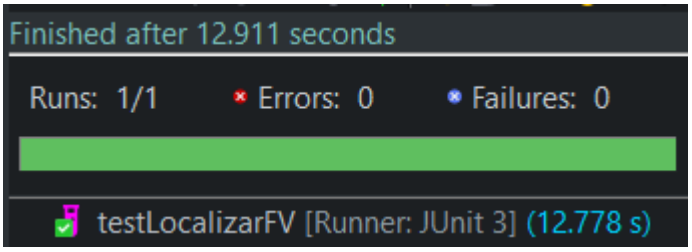
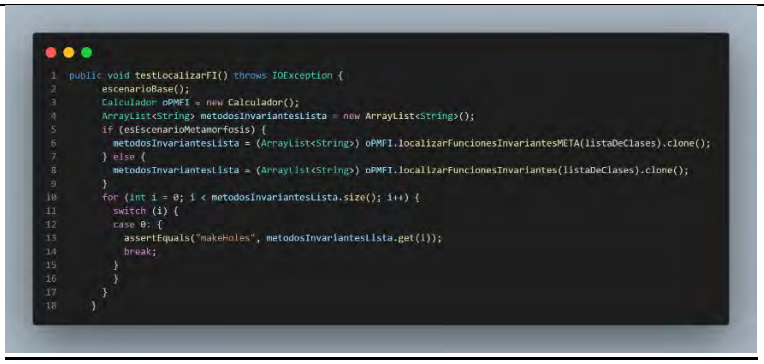
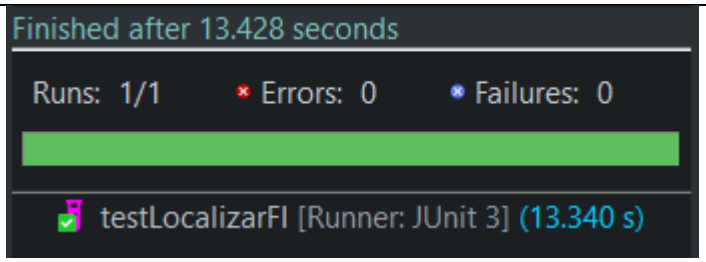
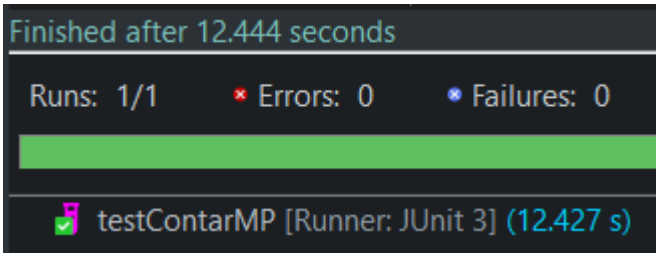
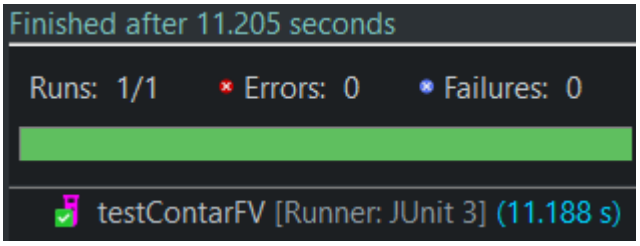


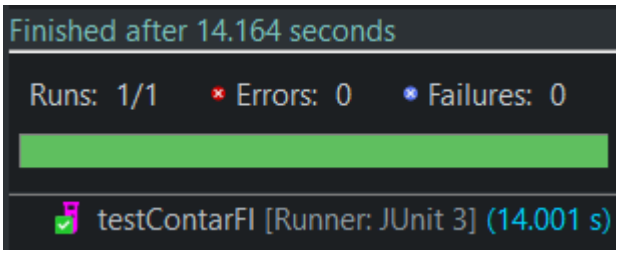
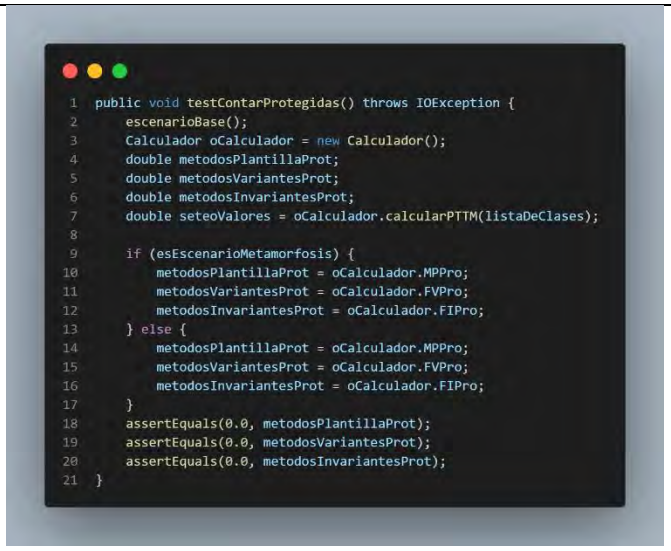
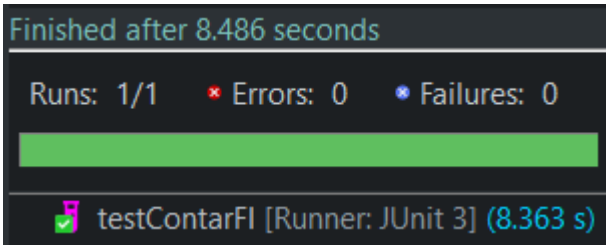
FIGURA 34 CASO DE PRUEBA SUDOKU ANTES DE LA REFACTORIZACIÓN

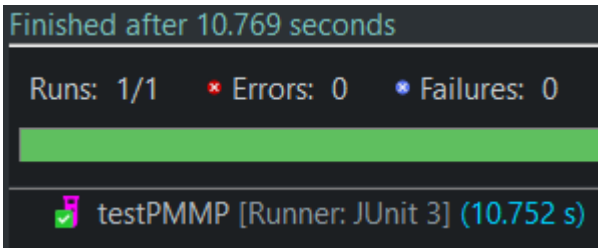
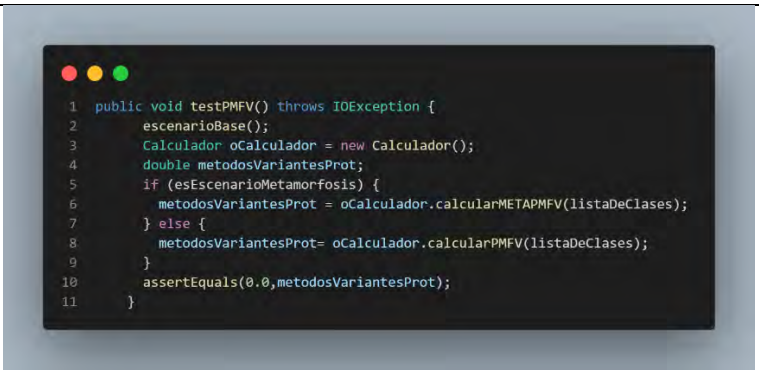
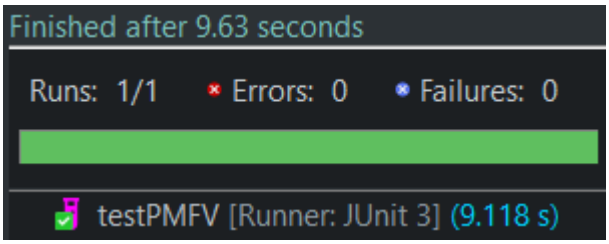
Como se puede observar en la Figura 34, a simple vista las reglas de ocultamiento de información están severamente mal aplicadas en las funciones: “generateBoard()”, “getBlankPositionOnGrid()”, “makeHoles()”. Se procede a ejecutar las pruebas unitarias observando que se cumplan las condiciones comparativas manuales con las automáticas resultantes y, finalmente, como prueba de integración el cálculo de la métrica PTTM:

Características a probar	**Pruebas Unitarias**
Localización de funciones plantilla.	

	
Localización de funciones variantes.	 
Localización de funciones invariantes.	

	
Conteo del número total de funciones plantilla.	 
Conteo del número total de funciones variantes.	 

Conteo del número total de funciones invariantes.	 
Conteo del número total de funciones protegidas asociadas al patrón de diseño *"Template Method":* • Funciones plantilla – "public" o "friendly". • Funciones variantes – "protected". • Funciones invariantes – "private".	 

Cálculo de la métrica *"PMMP"*.	 
Cálculo de la métrica *"PMFV"*.	 

```
public void testPMMP() throws IOException {
    escenarioBase();
    Calculador oCalculador = new Calculador();
    double metodosPlantillaProt;
    if (esEscenarioMetamorfosis) {
        metodosPlantillaProt = oCalculador.calcularMETAPMMP(listaDeClases);
    } else {
        metodosPlantillaProt = oCalculador.calcularPMMP(listaDeClases);
    }
    assertEquals(0.0, metodosPlantillaProt);
}
```

Finished after 10.769 seconds

Runs: 1/1 Errors: 0 Failures: 0

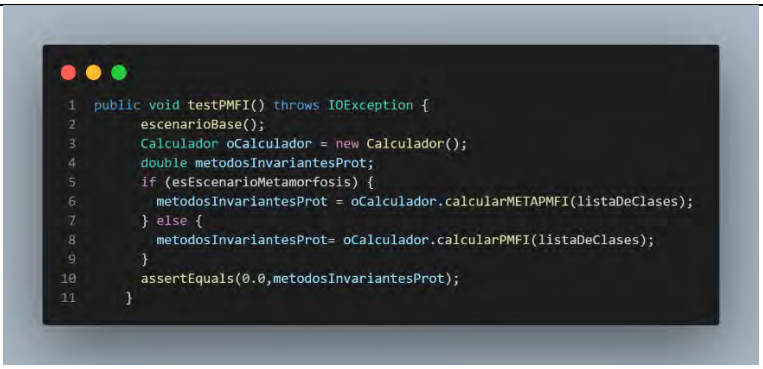
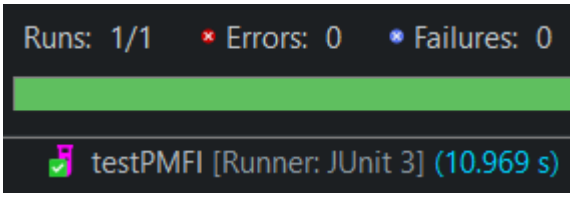
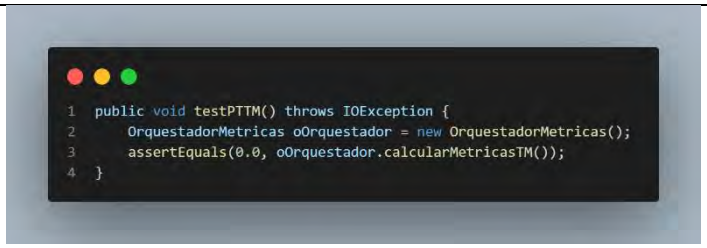
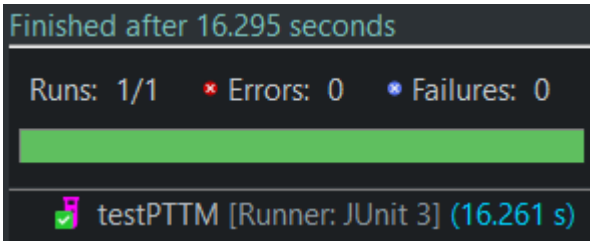
testPMMP [Runner: JUnit 3] (10.752 s)

```
public void testPMFV() throws IOException {
      escenarioBase();
      Calculador oCalculador = new Calculador();
      double metodosVariantesProt;
      if (esEscenarioMetamorfosis) {
        metodosVariantesProt = oCalculador.calcularMETAPMFV(listaDeClases);
      } else {
        metodosVariantesProt= oCalculador.calcularPMFV(listaDeClases);
      }
      assertEquals(0.0,metodosVariantesProt);
    }
```

Finished after 9.63 seconds

Runs: 1/1 Errors: 0 Failures: 0

testPMFV [Runner: JUnit 3] (9.118 s)

Cálculo de la métrica *“PMFI”*.	 
Cálculo de la métrica *“PTTM”*.	 

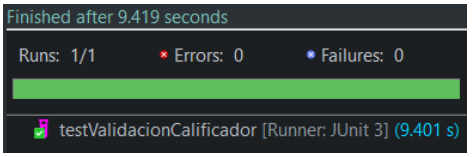
Las pruebas unitarias fueron aprobadas al comparar la igualdad de los resultados esperados con los reales. La prueba de integración hace uso de las métricas aprobadas en las pruebas unitarias, y es aprobada ya que el resultado que se obtiene es el esperado manualmente. Por lo anterior las pruebas del cálculo de métricas de protección modular del patrón de diseño *“Template Method”* son aprobadas y aceptadas.

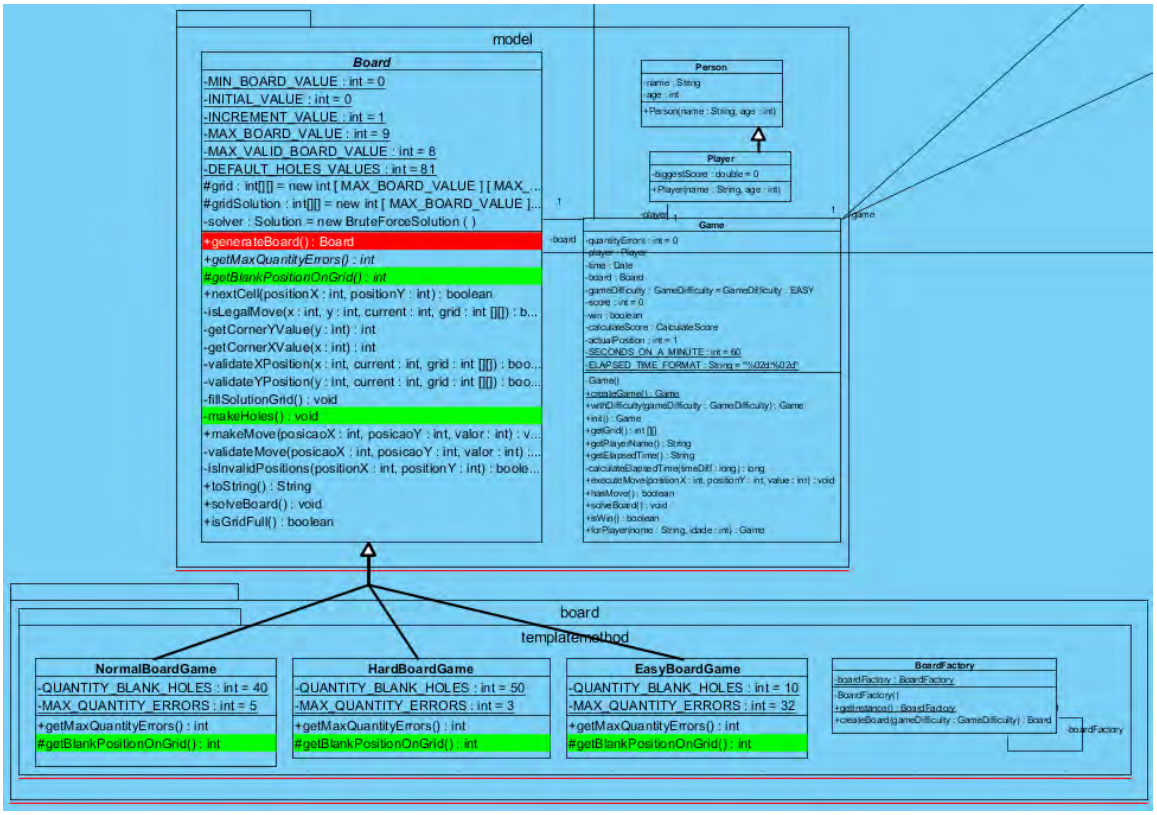
6.10.3.- Caso de Prueba ISMRTM0502

Nombre del Caso de prueba: Sudoku (Pure Java Implementation).

Se procede a ejecutar la prueba unitaria observando que se cumplan las condiciones comparativas manuales con las automáticas resultantes, posteriormente se prueba la mejora en la protección del patrón de diseño *"Template Method"* mediante el cálculo de las métricas y, finalmente, se prueba que el comportamiento del sistema siga siendo el mismo.

En la siguiente tabla se muestran las características a probar para el proceso de refactorización:

Características a probar	**Pruebas unitarias**
Método de validación del calificador de alcance.	 

Método de refactorización de calificadores de alcance y generación de código refactorizado.
El método de refactorización de código legado con carencia de protección en funciones plantilla y la generación de código son probados utilizando los diagramas de clases como referencia. Una vez ejecutado el método de refactorización se deben generar las clases que conforman a la aplicación. Para corroborar la correcta refactorización se genera el diagrama de clases de la Figura 35 y se comparan los calificadores de alcance de los métodos partícipes del patrón de diseño "*Template Method*" identificados con los de la Figura 34. A continuación, se muestra el diagrama de clases del sistema refactorizado donde se protegieron todas las funciones satisfactoriamente tomando en cuenta los distintos escenarios previstos:    FIGURA 35 CASO DE PRUEBA SUDOKU DESPUÉS DE LA REFACTORIZACIÓN En el sistema se presenta un escenario particular, en el cual la función plantilla "makeHoles" es llamado solamente por otro método de la misma clase llamado "generateBoard", cuando se presenta este escenario la función plantilla pasa a

ser tratada como una función invariante y la función que llama a la función plantilla se vuelve la función plantilla real. En este sentido, inicialmente, el sistema contaba con carencia de protección en el único método plantilla “generateBoard” de la arquitectura a causa del calificador de alcance “public”, pero al ser accedido por un cliente externo al paquete este método debe de ser así, por lo que el grado de protección modular de la función plantilla es de 0 antes y después de la refactorización.

Las funciones variantes del sistema aumentaron su grado de protección de 0 a 1, por lo que antes de la refactorización se presentaba ausencia de protección en todas las funciones variantes.

La protección modular en funciones invariantes aumentó de 0 a 1, de igual forma se presentaba ausencia de protección en la única función invariante antes de la refactorización.

Finalmente, la protección total del patrón de diseño aumentó de un 0 a 0.666666666, mejorando la protección del patrón de diseño “*Template Method*” para este escenario.

Esto es:

Sudoku	PMMP		PMFV		PMFI		PTTM
	Funciones plantilla protegidos	Funciones plantilla Totales	Funciones variantes protegidas	Funciones variantes totales	Funciones invariantes protegidas	Funciones invariantes totales	
Código Legado sin refactorizar	0	1	0	4	0	1	0
Código Legado Refactorizado	0	1	4	4	1	1	0.666667

Comparando gráficamente:

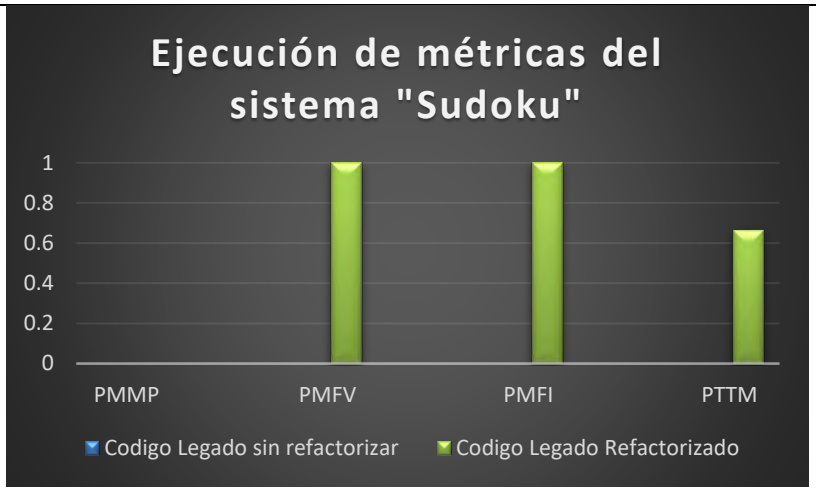


FIGURA 36 GRADO DE MEJORA DEL SISTEMA SUDOKU

Con los datos resultantes al aplicar las métricas nuevamente se comprueba que el método de refactorización cumple con la identificación y refactorización de las funciones plantilla, funciones variantes y funciones invariantes participantes del patrón de diseño "*Template Method*". Como se muestra en la Figura 36, antes de la refactorización no hay presencia de valores en las métricas, esto es debido a que no existe protección alguna en la arquitectura. De forma contraria, después de refactorizar se puede observar el aumento de los valores en las métricas menos en la métrica PMMP ya que la o las funciones plantillas se mantienen públicas.

Verificación del funcionamiento del código refactorizado.	
Ejecución de sistema antes de refactorizar 1.- Ingresar información del jugador: Inform your name: Elías Alejandro Inform the number a number of the difficulty game (1) EASY, (2) NORMAL, (3) HARD: 1 Inform your age: 24 Digit 1 for initiate or 0 for exit: 1 2.- Despliegue del sudoku y selección de coordenadas para ingresar valor:	Ejecución de sistema después de refactorizar 1.- Ingresar información del jugador: Inform your name: Elías Alejandro Inform the number a number of the difficulty game (1) EASY, (2) NORMAL, (3) HARD: 1 Inform your age: 24 Digit 1 for initiate or 0 for exit: 1 2.- Despliegue del sudoku y selección de coordenadas para ingresar valor:

```
    (0) (1) (2) (3) (4) (5) (6) (7) (8)
   |===.===.===|===.===.===|===.===.===|
(0)| 4 |   | 6 |   | 5 | 9 | 7 | 8 |   |
   |---.---.---|---.---.---|---.---.---|
(1)|   |   | 7 | 4 |   |   | 5 | 1 | 9 |
   |---.---.---|---.---.---|---.---.---|
(2)| 9 | 5 |   | 7 | 1 |   | 3 |   | 4 |
   |===.===.===|===.===.===|===.===.===|
(3)|   | 4 | 8 |   | 9 | 7 | 1 | 3 | 5 |
   |---.---.---|---.---.---|---.---.---|
(4)| 5 | 6 | 9 | 1 |   |   | 8 | 4 |   |
   |---.---.---|---.---.---|---.---.---|
(5)| 7 |   | 1 | 5 |   |   |   | 2 | 6 |
   |===.===.===|===.===.===|===.===.===|
(6)| 8 |   | 4 | 2 |   |   | 6 | 7 | 1 |
   |---.---.---|---.---.---|---.---.---|
(7)|   |   | 3 |   | 4 |   |   | 5 | 8 |
   |---.---.---|---.---.---|---.---.---|
(8)|   | 2 |   | 8 |   | 6 |   | 9 |   |
   |===.===.===|===.===.===|===.===.===|
Inform the position X: 3
Inform the position Y: 0
```

3.-Ingresar valor nuevo al sudoku, resultado de correctitud del dato ingresado y redespliegue del sudoku:

```
Please digit a number between 1 and 9: 2
Correct move!
Quantity of errors: 0
Elapsed time: 00:33
Press any key to continue

    (0) (1) (2) (3) (4) (5) (6) (7) (8)
   |===.===.===|===.===.===|===.===.===|
(0)| 4 |   | 6 |   | 5 | 9 | 7 | 8 |   |
   |---.---.---|---.---.---|---.---.---|
(1)|   |   | 7 | 4 |   |   | 5 | 1 | 9 |
   |---.---.---|---.---.---|---.---.---|
(2)| 9 | 5 |   | 7 | 1 |   | 3 |   | 4 |
   |===.===.===|===.===.===|===.===.===|
(3)| 2 | 4 | 8 |   | 9 | 7 | 1 | 3 | 5 |
   |---.---.---|---.---.---|---.---.---|
(4)| 5 | 6 | 9 | 1 |   |   | 8 | 4 |   |
   |---.---.---|---.---.---|---.---.---|
(5)| 7 |   | 1 | 5 |   |   |   | 2 | 6 |
   |===.===.===|===.===.===|===.===.===|
(6)| 8 |   | 4 | 2 |   |   | 6 | 7 | 1 |
   |---.---.---|---.---.---|---.---.---|
(7)|   |   | 3 |   | 4 |   |   | 5 | 8 |
   |---.---.---|---.---.---|---.---.---|
(8)|   | 2 |   | 8 |   | 6 |   | 9 |   |
   |===.===.===|===.===.===|===.===.===|
```

```
    (0) (1) (2) (3) (4) (5) (6) (7) (8)
   |===.===.===|===.===.===|===.===.===|
(0)| 2 | 5 | 1 |   |   | 9 |   | 4 | 6 |
   |---.---.---|---.---.---|---.---.---|
(1)|   |   | 9 | 6 | 4 | 1 |   |   | 5 |
   |---.---.---|---.---.---|---.---.---|
(2)| 4 |   | 6 |   | 2 |   |   | 1 | 9 |
   |===.===.===|===.===.===|===.===.===|
(3)|   |   | 2 | 9 | 8 |   | 4 | 3 |   |
   |---.---.---|---.---.---|---.---.---|
(4)|   | 4 | 3 | 2 | 1 | 5 | 6 | 7 |   |
   |---.---.---|---.---.---|---.---.---|
(5)| 7 | 1 |   | 4 | 3 | 6 |   | 5 |   |
   |===.===.===|===.===.===|===.===.===|
(6)| 8 |   | 5 |   | 6 |   | 1 |   | 7 |
   |---.---.---|---.---.---|---.---.---|
(7)|   | 9 | 4 |   | 5 | 8 |   |   | 3 |
   |---.---.---|---.---.---|---.---.---|
(8)| 6 |   |   | 1 |   | 2 |   | 8 | 4 |
   |===.===.===|===.===.===|===.===.===|
Inform the position X: 5
Inform the position Y: 6
```

3.- Ingresar valor nuevo al sudoku, resultado de correctitud del dato ingresado y redespliegue del sudoku:

```
Please digit a number between 1 and 9: 9
Correct move!
Quantity of errors: 0
Elapsed time: 00:46
Press any key to continue

    (0) (1) (2) (3) (4) (5) (6) (7) (8)
   |===.===.===|===.===.===|===.===.===|
(0)| 2 | 5 | 1 |   |   | 9 |   | 4 | 6 |
   |---.---.---|---.---.---|---.---.---|
(1)|   |   | 9 | 6 | 4 | 1 |   |   | 5 |
   |---.---.---|---.---.---|---.---.---|
(2)| 4 |   | 6 |   | 2 |   |   | 1 | 9 |
   |===.===.===|===.===.===|===.===.===|
(3)|   |   | 2 | 9 | 8 |   | 4 | 3 |   |
   |---.---.---|---.---.---|---.---.---|
(4)|   | 4 | 3 | 2 | 1 | 5 | 6 | 7 |   |
   |---.---.---|---.---.---|---.---.---|
(5)| 7 | 1 |   | 4 | 3 | 6 | 9 | 5 |   |
   |===.===.===|===.===.===|===.===.===|
(6)| 8 |   | 5 |   | 6 |   | 1 |   | 7 |
   |---.---.---|---.---.---|---.---.---|
(7)|   | 9 | 4 |   | 5 | 8 |   |   | 3 |
   |---.---.---|---.---.---|---.---.---|
(8)| 6 |   |   | 1 |   | 2 |   | 8 | 4 |
   |===.===.===|===.===.===|===.===.===|
```

Al comparar la igualdad de los resultados esperados con los reales con el método comparador “assertEquals()” de la librería JUnit, las pruebas unitarias son aprobadas. Así mismo, el método de refactorización y generación de código refactorizado es aprobado al comparar el cambio de calificador de alcance y por consiguiente el grado de mejora en la protección de los métodos participantes del “*Template Method*”. Finalmente, se aprueba la comparación del funcionamiento del sistema antes y después de la refactorización, por lo que se concluye que la refactorización no altera el comportamiento del sistema.

6.10.4.- Caso de Prueba ISMRTM0503

Nombre del Caso de prueba: SushiRestaurant.

En las figuras 37 y 38 se muestra la arquitectura de clases con los dos segmentos del diagrama señalado en rojo la carencia de protección y en verde la correcta protección. Estos son los segmentos donde se presenta el patrón de diseño “*Template Method*” del sistema SushiRestaurant. El sistema está escrito en lenguaje Java y, previamente, ha sido compilado y revisado su funcionamiento.

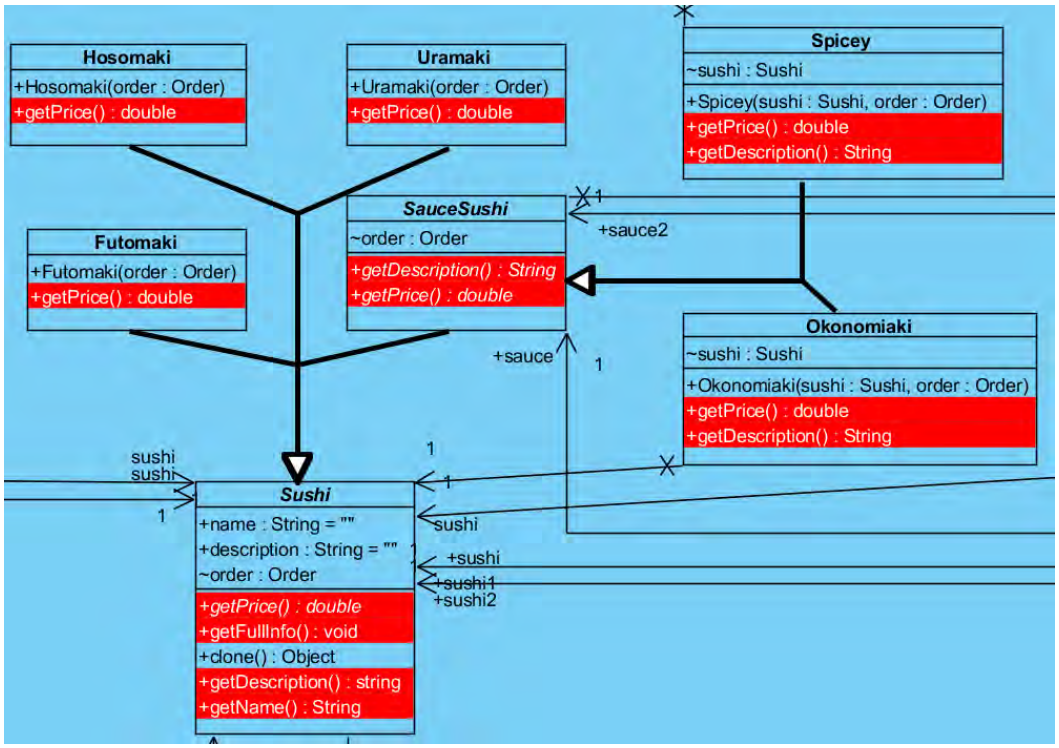


FIGURA 37 CASO DE PRUEBA SUSHIRESTAURANT 1 ANTES DE LA REFACTORIZACIÓN

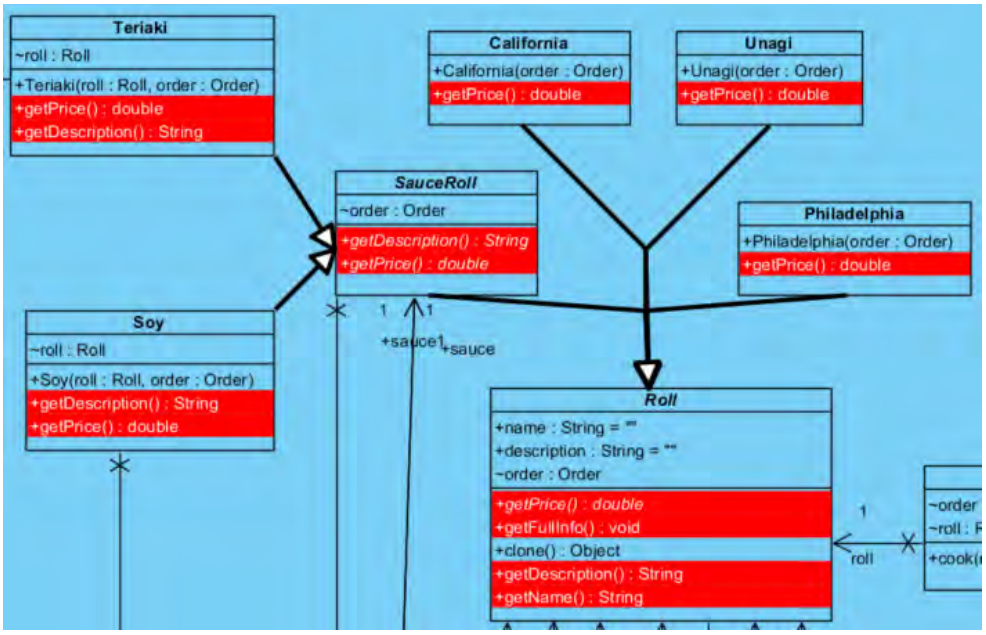
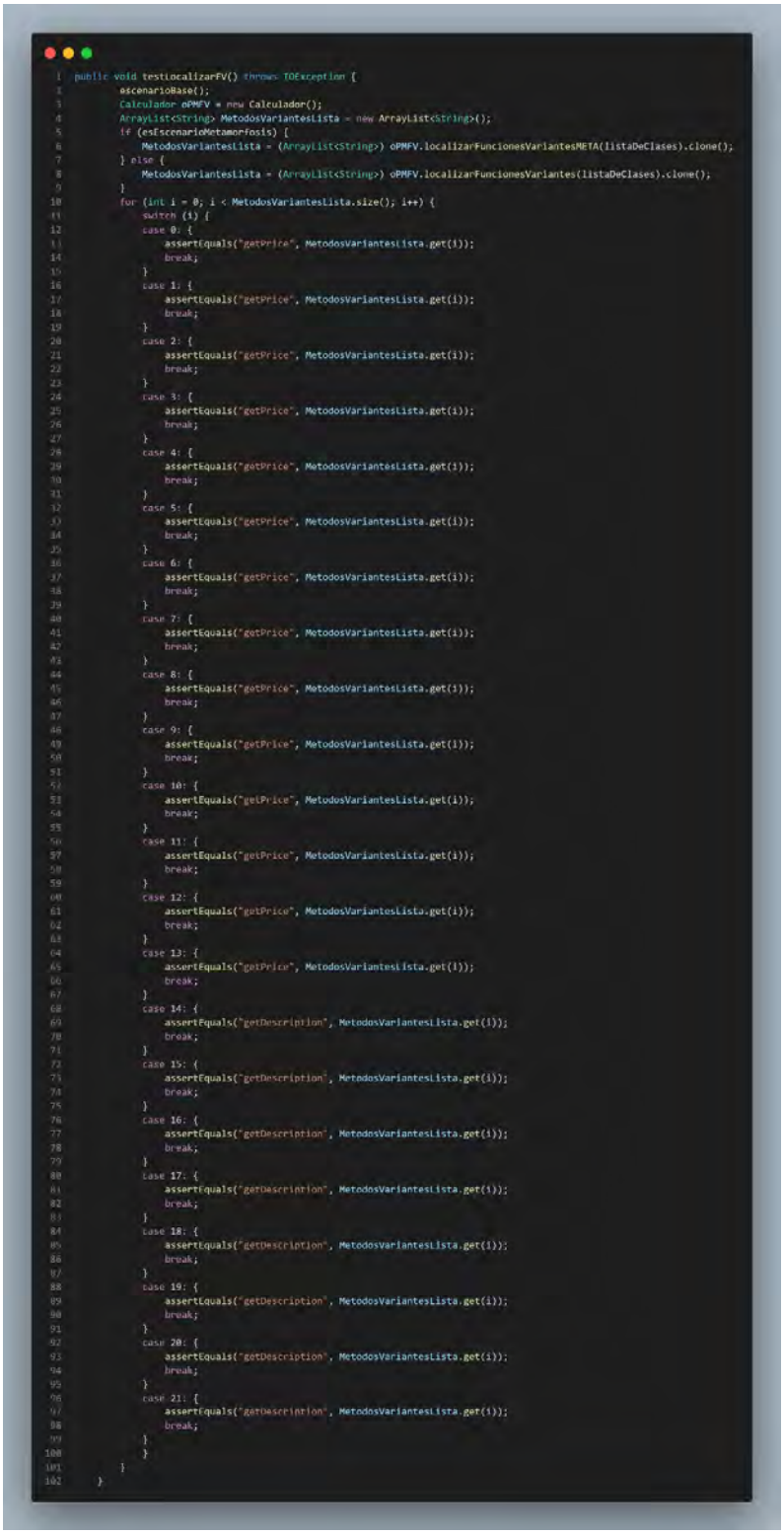


FIGURA 38 CASO DE PRUEBA SUSHIRESTAURANT 2 ANTES DE LA REFACTORIZACIÓN

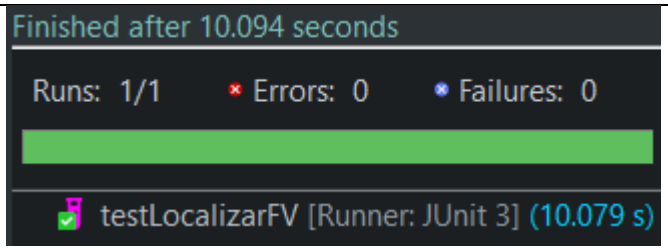
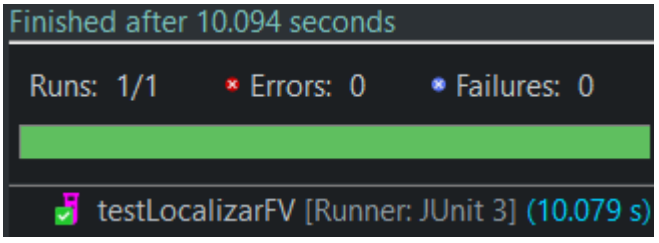
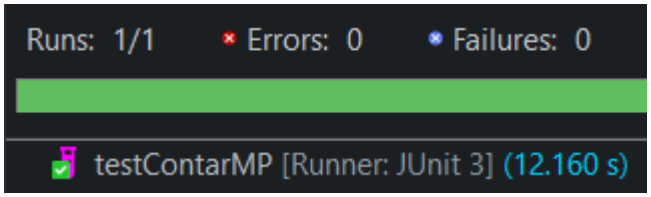
Como se puede observar a simple vista en las figuras 37 y 38 las reglas de ocultamiento de información están severamente mal aplicadas en todas las funciones señalas con rojo. Se procede a ejecutar las pruebas unitarias observando que se cumplan las condiciones comparativas manuales con las automáticas resultantes y finalmente como prueba de integración el cálculo de la métrica PTTM:

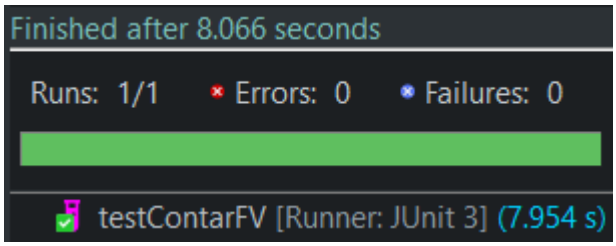
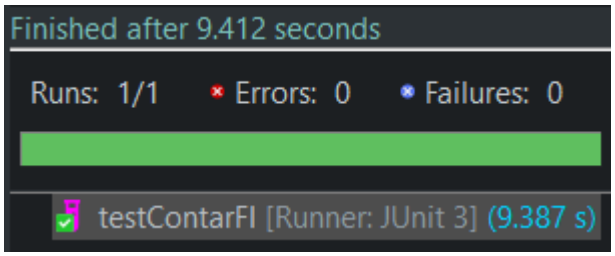
Características a probar	**Pruebas Unitarias**
Localización de funciones plantilla.	

```
public void testlocalizarMP() throws IOException {
        escenarioBase();
        Calculador oPMMP = new Calculador();
        ArrayList<String> MetodosPlantillaLista = new ArrayList<String>();
        if (esEscenarioMetamorfosis) {
            MetodosPlantillaLista = (ArrayList<String>) oPMMP.localizarMetodoPlantillaMETA(listaDeClases).clone();// Correcto
        } else {
            MetodosPlantillaLista = (ArrayList<String>) oPMMP.localizarMetodoPlantilla(listaDeClases).clone();// Correcto
        }
        for (int i = 0; i < MetodosPlantillaLista.size(); i++) {
            switch (i) {
            case 0: {
                assertEquals("getFullInfo", MetodosPlantillaLista.get(i));
                break;
            }
            case 1: {
                assertEquals("getFullInfo", MetodosPlantillaLista.get(i));
                break;
            }
            }
        }
    }
```

Localización de funciones variantes.	

```java
public void testLocalizarFV() throws IOException {
        escenarioBase();
        Calculador oPMFV = new Calculador();
        ArrayList<String> MetodosVariantesLista = new ArrayList<String>();
        if (esEscenarioMetamorfosis) {
            MetodosVariantesLista = (ArrayList<String>) oPMFV.localizarFuncionesVariantesMETA(listaDeClases).clone();
        } else {
            MetodosVariantesLista = (ArrayList<String>) oPMFV.localizarFuncionesVariantes(listaDeClases).clone();
        }
        for (int i = 0; i < MetodosVariantesLista.size(); i++) {
            switch (i) {
            case 0: {
                assertEquals("getPrice", MetodosVariantesLista.get(i));
                break;
            }
            case 1: {
                assertEquals("getPrice", MetodosVariantesLista.get(i));
                break;
            }
            case 2: {
                assertEquals("getPrice", MetodosVariantesLista.get(i));
                break;
            }
            case 3: {
                assertEquals("getPrice", MetodosVariantesLista.get(i));
                break;
            }
            case 4: {
                assertEquals("getPrice", MetodosVariantesLista.get(i));
                break;
            }
            case 5: {
                assertEquals("getPrice", MetodosVariantesLista.get(i));
                break;
            }
            case 6: {
                assertEquals("getPrice", MetodosVariantesLista.get(i));
                break;
            }
            case 7: {
                assertEquals("getPrice", MetodosVariantesLista.get(i));
                break;
            }
            case 8: {
                assertEquals("getPrice", MetodosVariantesLista.get(i));
                break;
            }
            case 9: {
                assertEquals("getPrice", MetodosVariantesLista.get(i));
                break;
            }
            case 10: {
                assertEquals("getPrice", MetodosVariantesLista.get(i));
                break;
            }
            case 11: {
                assertEquals("getPrice", MetodosVariantesLista.get(i));
                break;
            }
            case 12: {
                assertEquals("getPrice", MetodosVariantesLista.get(i));
                break;
            }
            case 13: {
                assertEquals("getPrice", MetodosVariantesLista.get(i));
                break;
            }
            case 14: {
                assertEquals("getDescription", MetodosVariantesLista.get(i));
                break;
            }
            case 15: {
                assertEquals("getDescription", MetodosVariantesLista.get(i));
                break;
            }
            case 16: {
                assertEquals("getDescription", MetodosVariantesLista.get(i));
                break;
            }
            case 17: {
                assertEquals("getDescription", MetodosVariantesLista.get(i));
                break;
            }
            case 18: {
                assertEquals("getDescription", MetodosVariantesLista.get(i));
                break;
            }
            case 19: {
                assertEquals("getDescription", MetodosVariantesLista.get(i));
                break;
            }
            case 20: {
                assertEquals("getDescription", MetodosVariantesLista.get(i));
                break;
            }
            case 21: {
                assertEquals("getDescription", MetodosVariantesLista.get(i));
                break;
            }
            }
        }
    }
```

	
Localización de funciones invariantes.	 
Conteo del número total de funciones plantilla.	 

Conteo del número total de funciones variantes.	 
Conteo del número total de funciones invariantes.	 
Conteo del número total de funciones protegidas asociadas al patrón de diseño *"Template Method":* • Funciones plantilla – "public" o "friendly".	

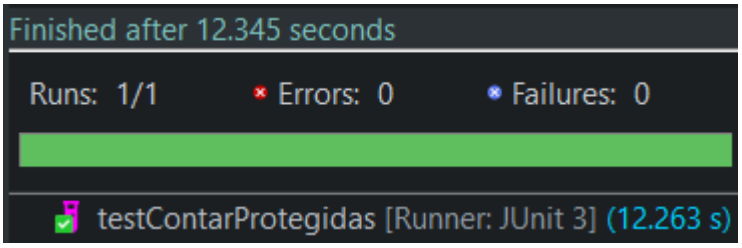
<table>
<tr>
<td>

- Funciones variantes – “protected”.
- Funciones invariantes – “private”.

</td>
<td>

```java
public void testContarProtegidas() throws IOException {
    escenarioBase();
    Calculador oCalculador = new Calculador();
    double metodosPlantillaProt;
    double metodosVariantesProt;
    double metodosInvariantesProt;
    double seteoValores = oCalculador.calcularPTTM(listaDeClases);

    if (esEscenarioMetamorfosis) {
        metodosPlantillaProt = oCalculador.MPPro;
        metodosVariantesProt = oCalculador.FVPro;
        metodosInvariantesProt = oCalculador.FIPro;
    } else {
        metodosPlantillaProt = oCalculador.MPPro;
        metodosVariantesProt = oCalculador.FVPro;
        metodosInvariantesProt = oCalculador.FIPro;
    }
    assertEquals(0.0, metodosPlantillaProt);
    assertEquals(0.0, metodosVariantesProt);
    assertEquals(0.0, metodosInvariantesProt);
}
```

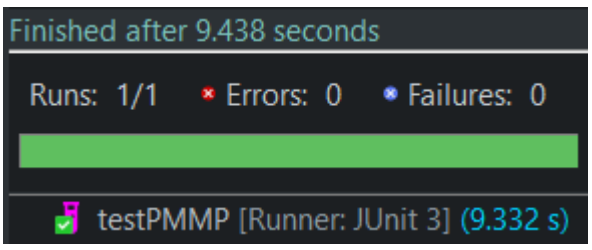


</td>
</tr>
<tr>
<td>

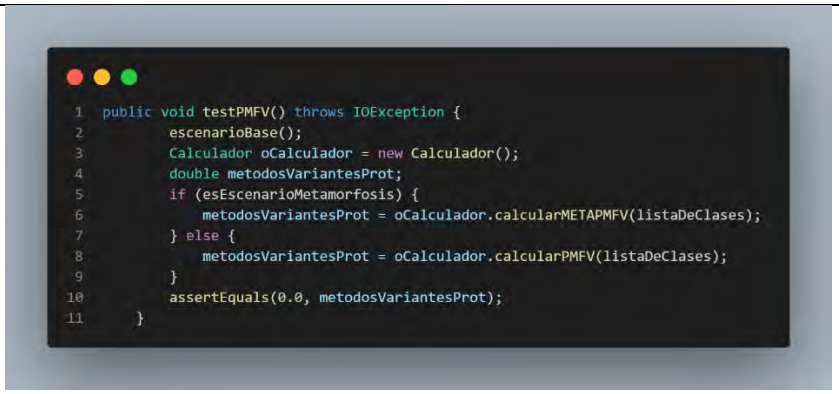
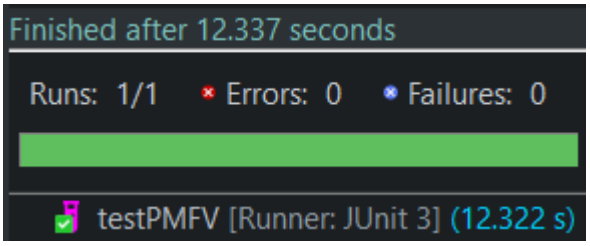
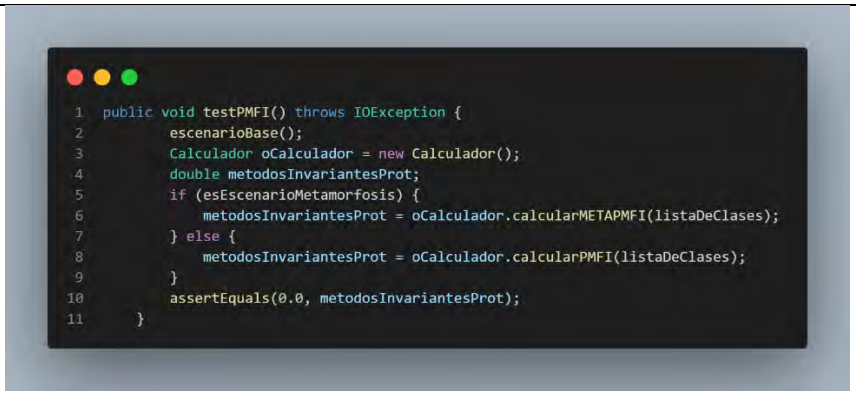
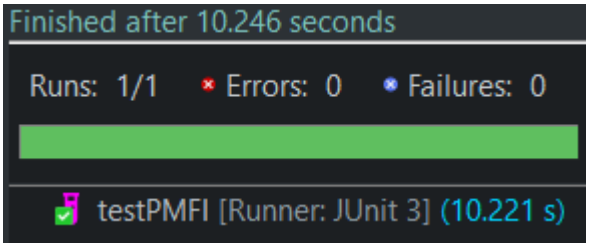
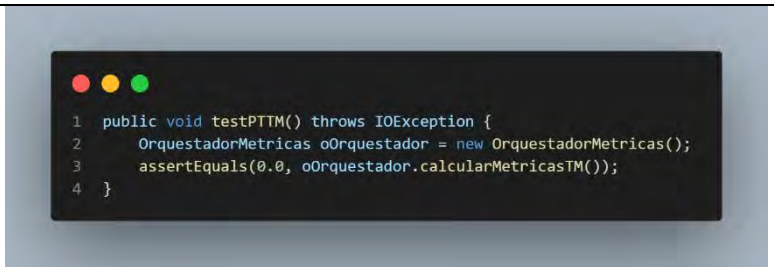
Cálculo de la métrica *“PMMP”*.

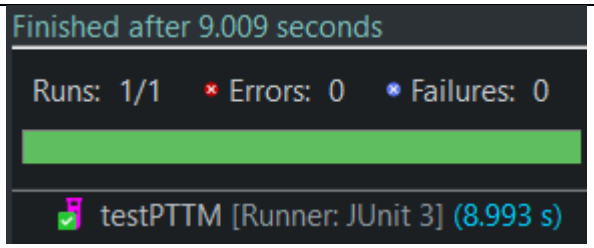
</td>
<td>

```java
public void testPMMP() throws IOException {
    escenarioBase();
    Calculador oCalculador = new Calculador();
    double metodosPlantillaProt;
    if (esEscenarioMetamorfosis) {
        metodosPlantillaProt = oCalculador.calcularMETAPMMP(listaDeClases);
    } else {
        metodosPlantillaProt = oCalculador.calcularPMMP(listaDeClases);
    }
    assertEquals(0.0, metodosPlantillaProt);
}
```



</td>
</tr>
</table>

Cálculo de la métrica *"PMFV"*.	 
Cálculo de la métrica *"PMFI"*.	 
Cálculo de la métrica *"PTTM"*.	

Las pruebas unitarias del cálculo de métricas fueron aprobadas al comparar la igualdad de los resultados esperados con los reales mediante el método “asserEquals()” de la librería JUnit. La prueba de integración hace uso de las métricas aprobadas en las pruebas unitarias, y es aprobada ya que el resultado que se obtiene es el esperado manualmente. Por lo anterior las pruebas del cálculo de métricas de protección modular del patrón de diseño “*Template Method*” son aprobadas y aceptadas.

6.10.5.- Caso de Prueba ISMRTM0504

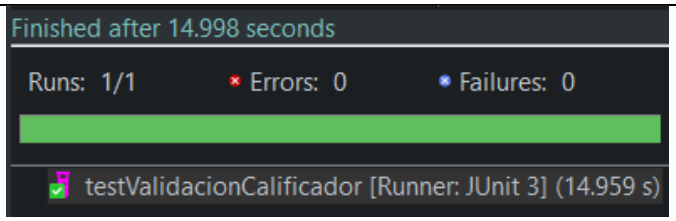
Nombre del Caso de prueba: SushiRestaurant

Se procede a ejecutar la prueba unitaria observando que se cumplan las condiciones comparativas manuales con las automáticas resultantes, posteriormente se prueba la mejora en la protección del patrón de diseño *“Template Method”* mediante el cálculo de las métricas y finalmente se prueba que el comportamiento del sistema siga siendo el mismo.

En la siguiente tabla se muestran las características a probar para el proceso de refactorización:

Características a probar	**Pruebas unitarias**

Método de validación del calificador de alcance.	



Método de refactorización de calificadores de alcance y generación de código refactorizado.

El método de refactorización de código legado con carencia de protección en funciones plantilla y la generación de código son probados utilizando los diagramas de clases como referencia.

Una vez ejecutado el método de refactorización se deben generar las clases que conforman a la aplicación, para corroborar la correcta refactorización se generan los diagramas de clases de la Figura 39 y Figura 40, y se comparan los calificadores de alcance de los métodos partícipes del patrón de diseño "*Template Method*" identificados con los de la Figura 37 y Figura 38.

A continuación, se muestran los dos diagramas de clases del sistema refactorizado donde se protegieron todas las funciones satisfactoriamente tomando en cuenta los distintos escenarios previstos:

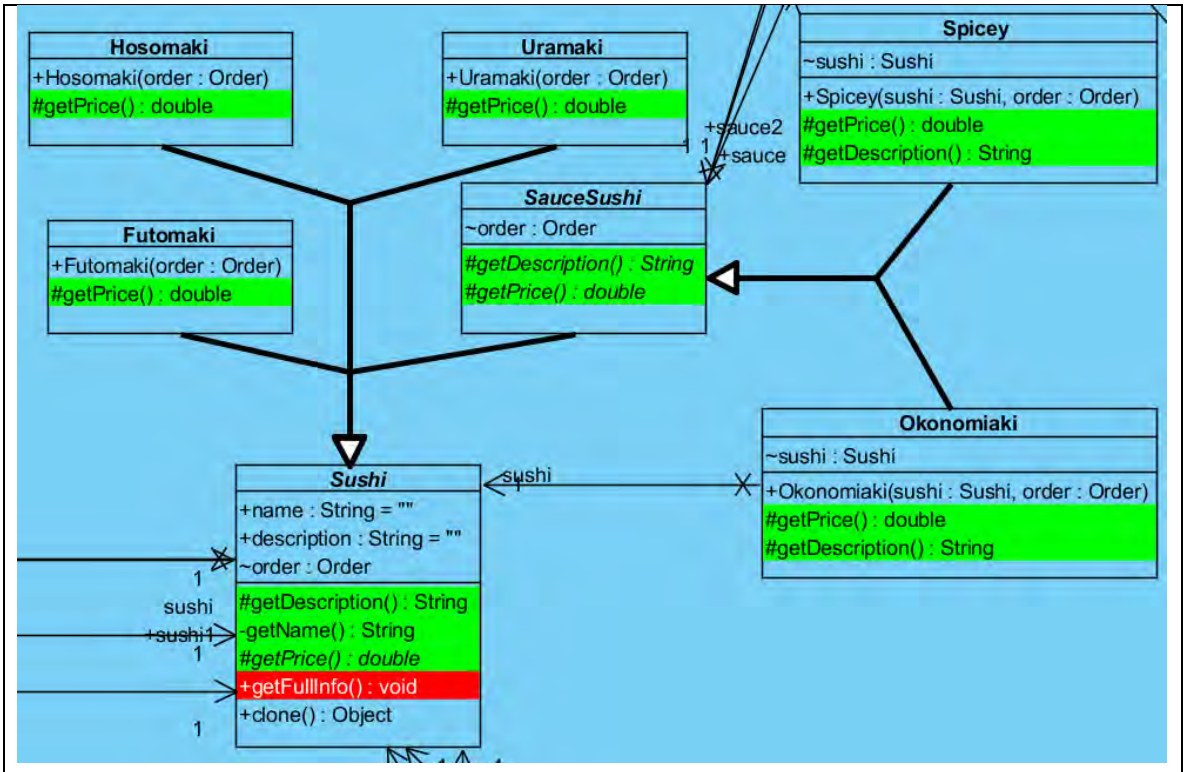


FIGURA 39 CASO DE PRUEBA SUSHIRESTAURANT 1 DESPUÉS DE LA REFACTORIZACIÓN

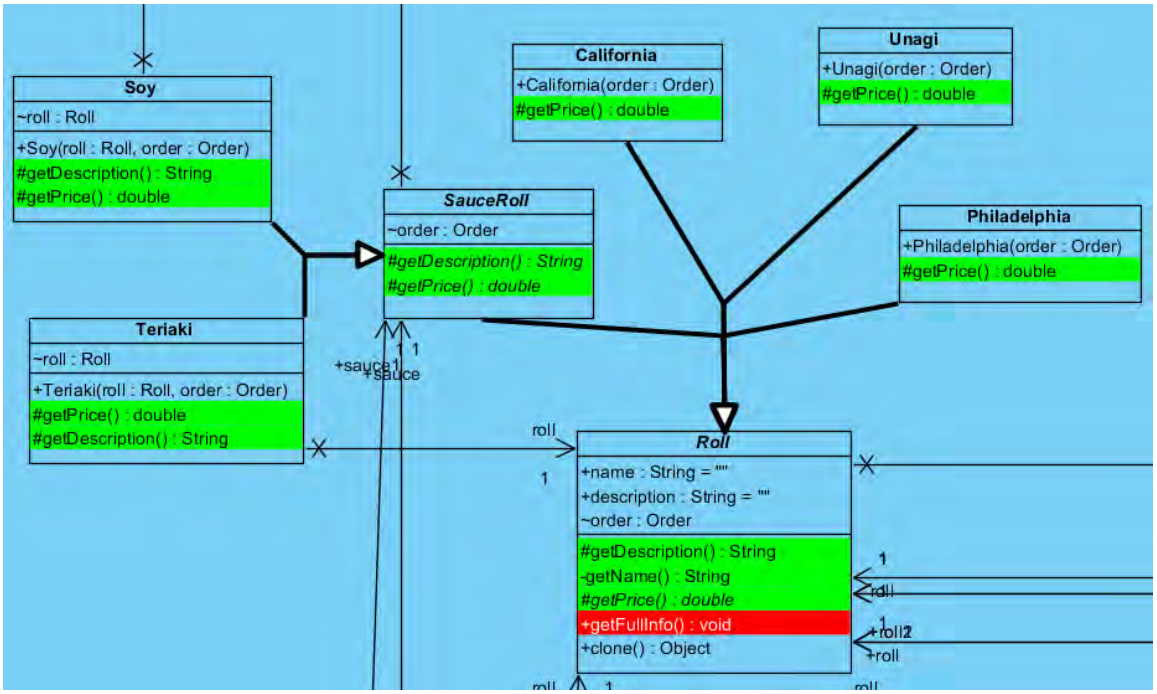


FIGURA 40 CASO DE PRUEBA SUSHIRESTAURANT 2 DESPUÉS DE LA REFACTORIZACIÓN

Inicialmente el sistema contaba con ausencia de protección en las dos funciones plantilla “getFullInfo” de las arquitecturas, ya que estos son accedidos por un cliente externo al paquete, no se hace necesario restringir el acceso de estas funciones, por lo que el grado de protección modular de la función plantilla es de 0 antes y después de la refactorización.

En la arquitectura se presentó un escenario en donde existen 8 funciones invariantes virtuales que se sobreescriben en las clases derivadas (clases hijas y nietas), por lo que a estas funciones virtuales se les da el tratamiento de funciones variantes. Entendido lo anterior, las funciones variantes del sistema aumentaron el grado de protección de 0 a 1, por lo que antes de la refactorización se presentaba una ausencia total de protección en las 22 funciones variantes.

La protección modular en funciones invariantes aumentó de 0 a 1, de igual forma se presentaba una ausencia de protección en todas las funciones invariantes antes de la refactorización.

Finalmente, la protección total del patrón de diseño aumentó de un 0 a 0.666666666, mejorando la protección del patrón de diseño “*Template Method*” para este escenario.

Esto es:

SushiRestaurant	PMMP		PMFV		PMFI		PTTM
	Funciones plantilla protegidos	Funciones plantilla Totales	Funciones variantes protegidas	Funciones variantes totales	Funciones invariantes protegidas	Funciones invariantes totales	
Código Legado sin refactorizar	0	2	0	22	0	2	0
Código Legado Refactorizado	0	2	22	22	2	2	0.666667

Comparando gráficamente:

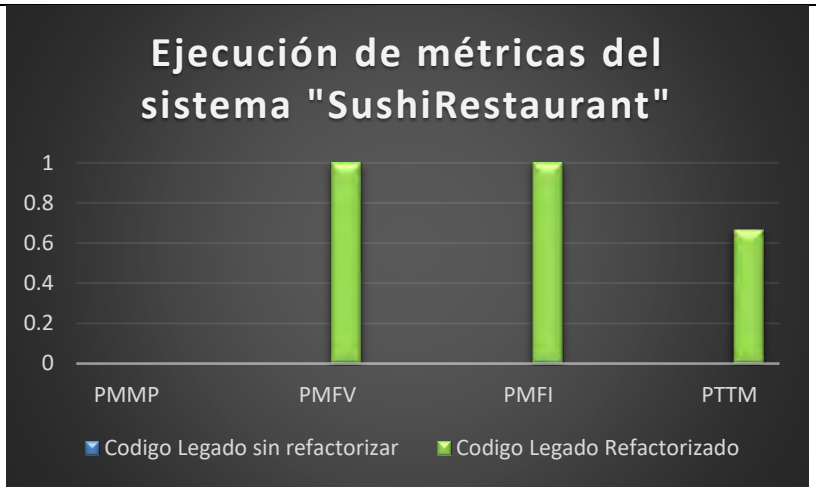


FIGURA 41 GRADO DE MEJORA DEL SISTEMA SUSHIRESTAURANT

Con los datos resultantes al aplicar las métricas se comprueba que el método de refactorización cumple con la identificación y refactorización de las funciones plantilla, funciones variantes y funciones invariantes participantes del patrón de diseño "*Template Method*". Como se muestra en la Figura 41, antes de la refactorización no hay presencia de valores en las métricas, esto es debido a que no existe protección alguna en la arquitectura. De forma contraria, después de refactorizar se puede observar el aumento de los valores en las métricas menos en la métrica PMMP ya que la o las funciones plantillas se mantienen públicas.

Verificación del funcionamiento del código refactorizado.	
Ejecución de sistema antes de refactorizar 1.- Despliegue de mensaje de bienvenida y selección del tipo de pescado: Good morning! Accountant is coming... Cleaning woman is coming... Cook is coming... Dishwasher is coming... Waiter is coming... Assistan cook is coming... Let's make an order. Here we go! Please enter the type of fish (1 - fresh; 2 - marinated): 1	Ejecución de sistema después de refactorizar 1.- Despliegue de mensaje de bienvenida y selección del tipo de pescado: Good morning! Accountant is coming... Cleaning woman is coming... Cook is coming... Dishwasher is coming... Waiter is coming... Assistan cook is coming... Let's make an order. Here we go! Please enter the type of fish (1 - fresh; 2 - marinated): 1

2.- Selección del tipo de plato: `Please enter the kind of dish (1 - simple; 2 - special):` `2` 3.-Tipo de platillo especial: `Please enter the type of special dish` `(1 - Two plates of sushi and sauce;` `2 - Two plates of roll and sauce;` `3 - Sushi, roll and two types of sauce):` `3` 4.- Seleccionar preferencias del sushi e información extra: `Please enter the type of sauce (1 - spicey; 2 - okonomiaki; 0 - no sauce):` `1` `Please enter the type of sushi (1 - futomaki; 2 - hosomaki):` `1` `Please enter the type of sauce (1 - soy; 2 - teriaki; 0 - no sauce):` `2` `Please enter the type of roll (1 - philadelphia):` `1` `Delivery to home? (1 - yes; 2 - no):` `1` `Choose the current day? (1 - simple day; 2 - birthday; 3 - holliday):` `1` 5.- Generación de orden, cuenta, propina y mensaje de finalización: `Yout order: Fresh fish, Eel, salmon, Philadelphia cheese, avocado, cucumber, rice, caviar Masago., Eel, salmon, Philadelphia cheese, avocado, cucumber, rice, caviar Masago.Sauce: Spicey., Rice, seaweed, cucumber, avocado, salmon, Philadelphia cheese, lemon, caviar Tobiko., Rice, seaweed, cucumber, avocado, salmon, Philadelphia cheese, lemon, caviar Tobiko.Sauce: Teriaki., Delivery to home., Tips: 92`  `Yout price: 924.0`  `Cook cooks meal` `Assistan cook cooks meal` `The client is waiting an order..` `Waiter distributes meal` `The client is eating..` `Accountant gives the bill` `The client is waiting a bill..` `Accountant takes the money` `Dishwasher washes dishes` `Cleaning woman cleans the rastaurant`	2.- Selección del tipo de plato: `Please enter the kind of dish (1 - simple; 2 - special):` `2` 3.-Tipo de platillo especial: `Please enter the type of special dish` `(1 - Two plates of sushi and sauce;` `2 - Two plates of roll and sauce;` `3 - Sushi, roll and two types of sauce):` `3` 4.- Seleccionar preferencias del sushi e información extra: `Please enter the type of sauce (1 - spicey; 2 - okonomiaki; 0 - no sauce):` `1` `Please enter the type of sushi (1 - futomaki; 2 - hosomaki):` `1` `Please enter the type of sauce (1 - soy; 2 - teriaki; 0 - no sauce):` `2` `Please enter the type of roll (1 - philadelphia):` `1` `Delivery to home? (1 - yes; 2 - no):` `1` `Choose the current day? (1 - simple day; 2 - birthday; 3 - holliday):` `1` 5.- Generación de orden, cuenta, propina y mensaje de finalización: `Yout order: Fresh fish, Eel, salmon, Philadelphia cheese, avocado, cucumber, rice, caviar Masago., Eel, salmon, Philadelphia cheese, avocado, cucumber, rice, caviar Masago.Sauce: Spicey., Rice, seaweed, cucumber, avocado, salmon, Philadelphia cheese, lemon, caviar Tobiko., Rice, seaweed, cucumber, avocado, salmon, Philadelphia cheese, lemon, caviar Tobiko.Sauce: Teriaki., Delivery to home., Tips: 92`  `Yout price: 924.0`  `Cook cooks meal` `Assistan cook cooks meal` `The client is waiting an order..` `Waiter distributes meal` `The client is eating..` `Accountant gives the bill` `The client is waiting a bill..` `Accountant takes the money` `Dishwasher washes dishes` `Cleaning woman cleans the rastaurant`

Las pruebas unitarias son aprobadas al comparar la igualdad de los resultados esperados con los reales con el método comparador “assertEquals()” de la librería JUnit. El método de refactorización y generación de código refactorizado es aprobado al comparar el cambio de calificador de alcance y en consecuencia el grado de mejora en la protección de los métodos participantes del “*Template Method*”. Finalmente, se aprueba la comparación del funcionamiento del sistema antes y después de la refactorización, por lo que se concluye que la refactorización no altera el comportamiento del sistema.

6.10.6.- Caso de Prueba ISMRTM0505

Nombre del Caso de prueba: Usta Donerci.

En la Figura 42 se muestra el segmento del diagrama de clases señalado con rojo la carencia de protección y con verde la correcta protección, este segmento es donde se presenta el patrón de diseño "*Template Method*" del sistema Usta Donerci, el sistema está escrito en lenguaje Java y ha sido, previamente, compilado y revisado su funcionamiento.

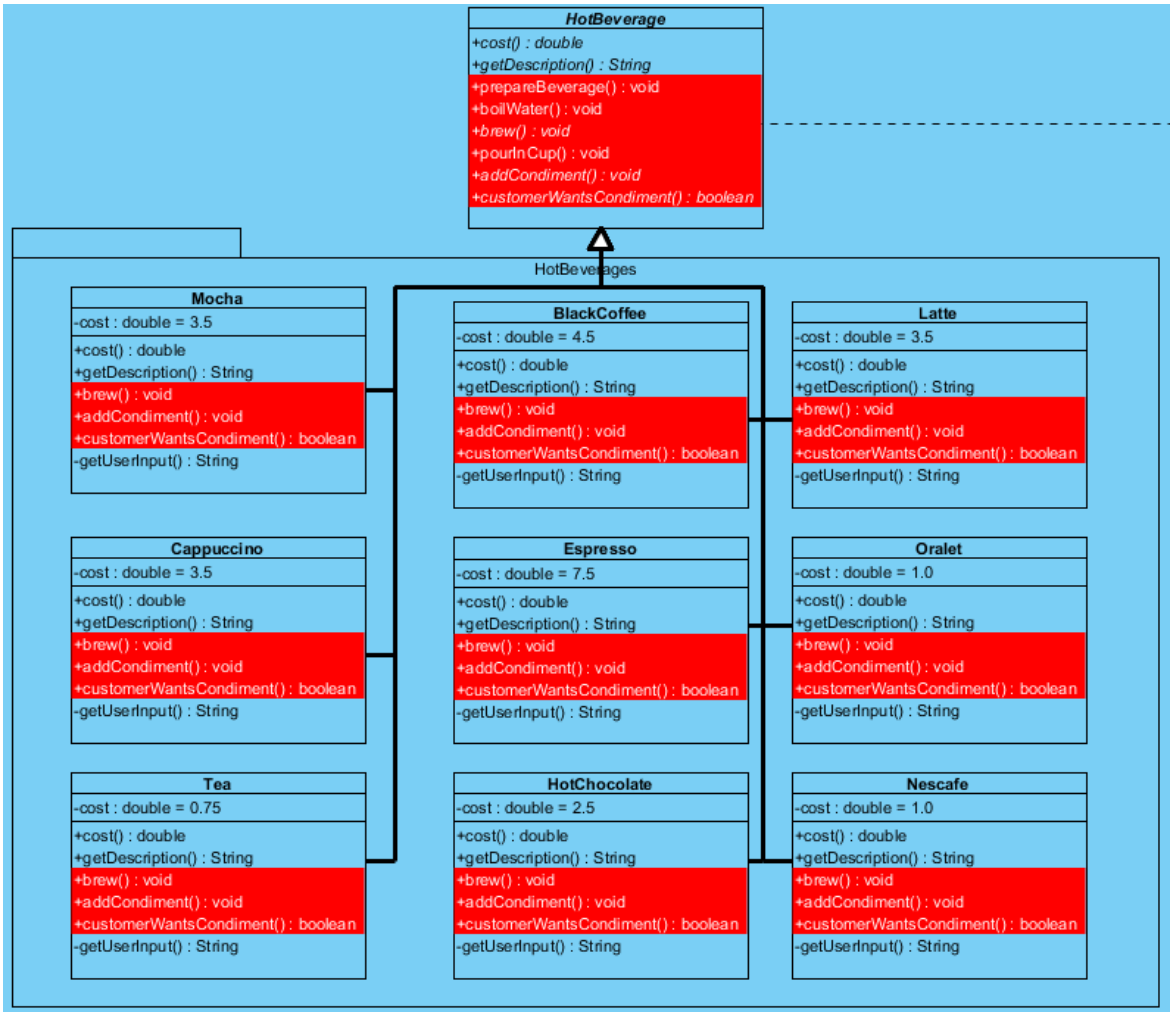
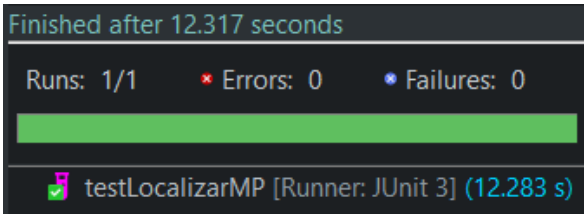


FIGURA 42 CASO DE PRUEBA USTA DONERCI ANTES DE LA REFACTORIZACIÓN

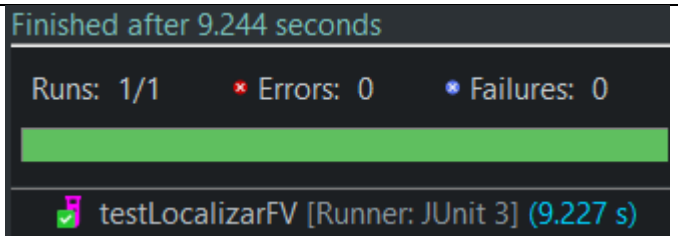
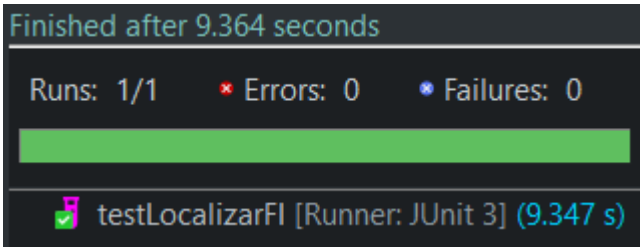
Como se puede observar a simple vista en la Figura 42 las reglas de ocultamiento de información están severamente mal aplicadas en todas las funcione plantilla marcadas con rojo. Se procede a ejecutar las pruebas unitarias observando que se cumplan las condiciones comparativas manuales con las automáticas resultantes y finalmente como prueba de integración el cálculo de la métrica PTTM:

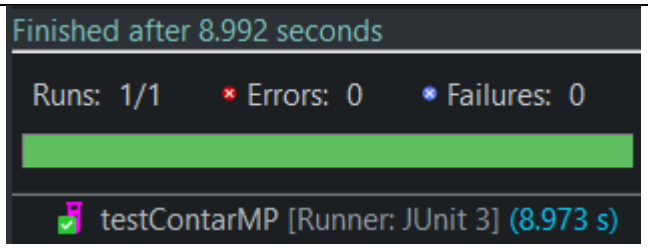
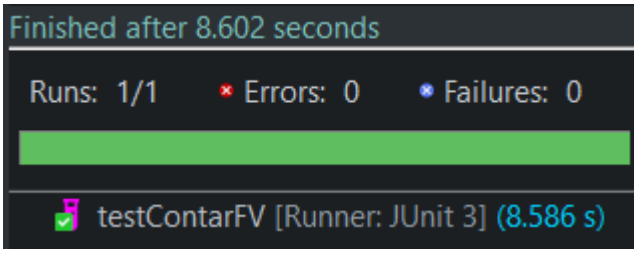
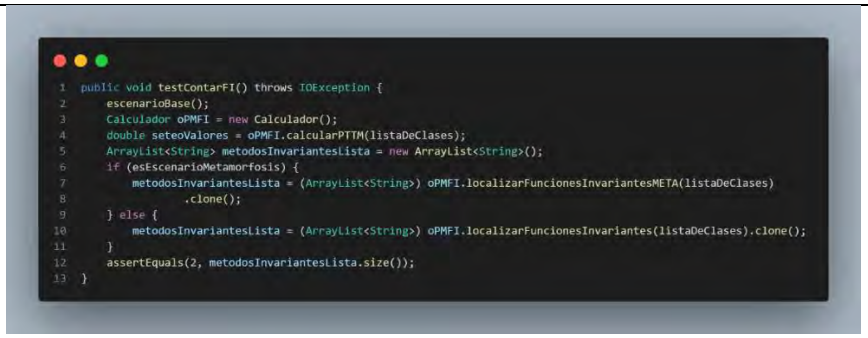
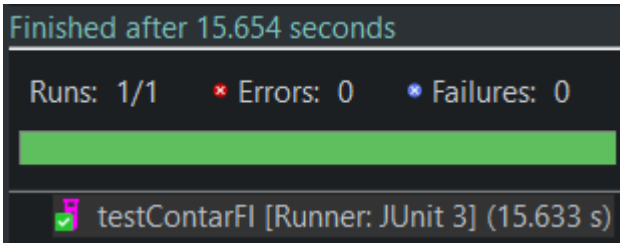
Características a probar	Pruebas Unitarias
Localización de funciones plantilla.	 
Localización de funciones variantes.	

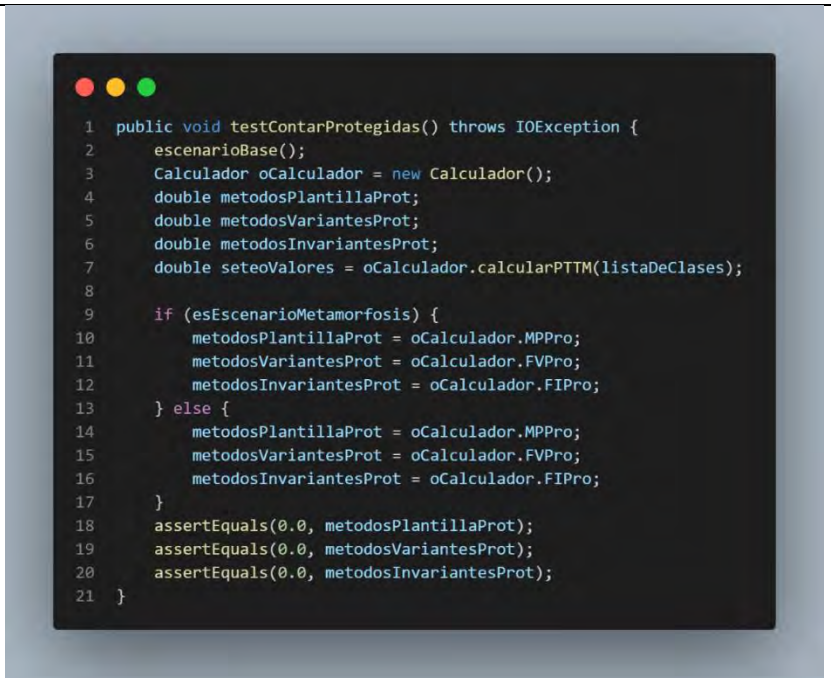
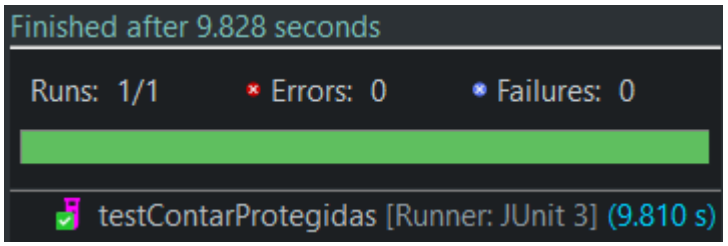
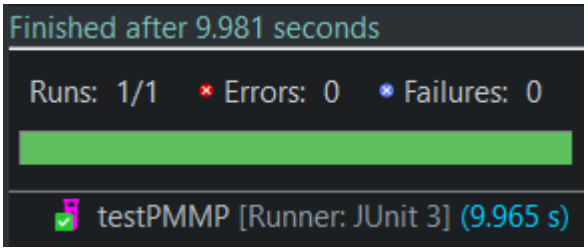
```java
public void testLocalizarMP() throws IOException {
    escenarioBase();
    Calculador oPMMP = new Calculador();
    ArrayList<String> MetodosPlantillaLista = new ArrayList<String>();
    if (esEscenarioMetamorfosis) {
        MetodosPlantillaLista = (ArrayList<String>) oPMMP.localizarMetodoPlantillaMETA(listaDeClases).clone();
    } else {
        MetodosPlantillaLista = (ArrayList<String>) oPMMP.localizarMetodoPlantilla(listaDeClases).clone();
    }
    for (int i = 0; i < MetodosPlantillaLista.size(); i++) {
        switch (i) {
        case 0: {
            assertEquals("prepareBeverage", MetodosPlantillaLista.get(i));
            break;
        }
        }
    }
}
```

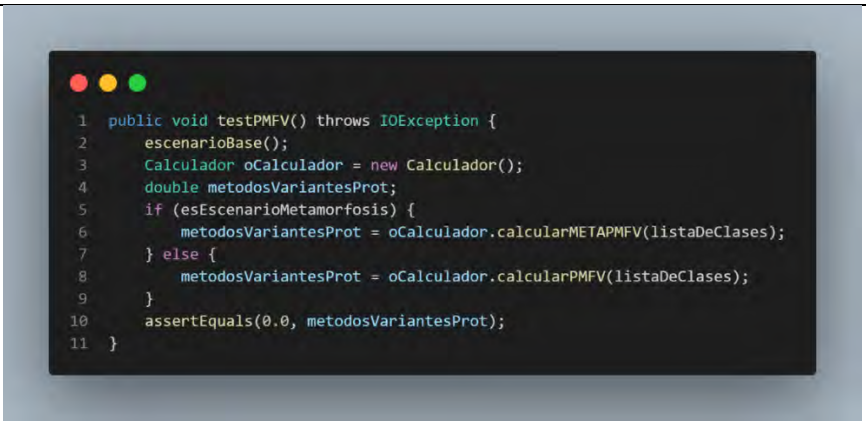
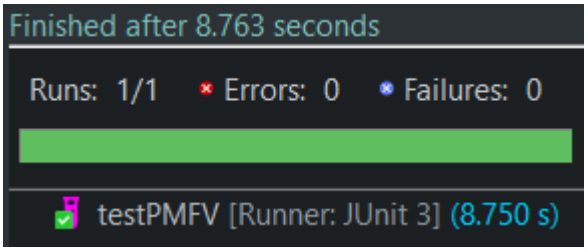
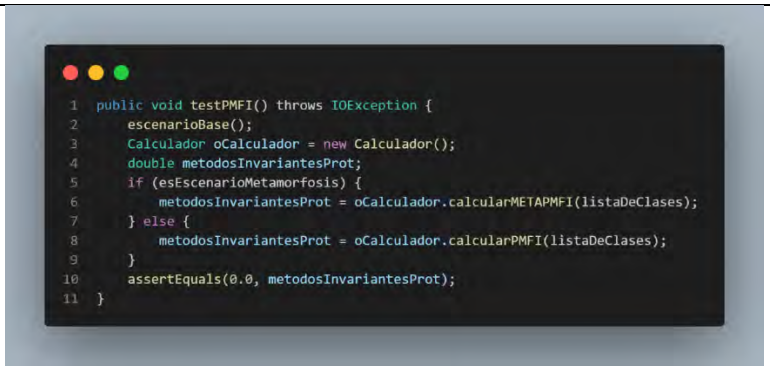
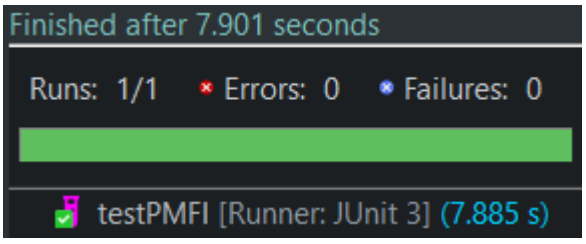
```
Finished after 12.317 seconds
Runs: 1/1   Errors: 0   Failures: 0
testLocalizarMP [Runner: JUnit 3] (12.283 s)
```

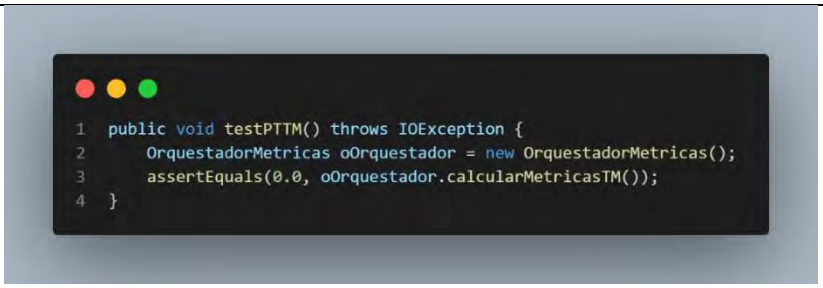
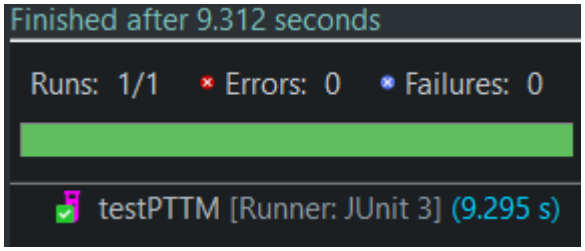
```java
public void testLocalizarFV() throws IOException {
    escenarioBase();
    Calculador oPMFV = new Calculador();
    ArrayList<String> MetodosVariantesLista = new ArrayList<String>();
    if (esEscenarioMetamorfosis) {
        MetodosVariantesLista = (ArrayList<String>) oPMFV.localizarFuncionesVariantesMETA(listaDeClases).clone();
    } else {
        MetodosVariantesLista = (ArrayList<String>) oPMFV.localizarFuncionesVariantes(listaDeClases).clone();
    }
    for (int i = 0; i < MetodosVariantesLista.size(); i++) {
        switch (i) {
        case 0: {
            assertEquals("brew", MetodosVariantesLista.get(i));
            break;
        }
        case 1: {
            assertEquals("customerWantsCondiment", MetodosVariantesLista.get(i));
            break;
        }
        case 2: {
            assertEquals("addCondiment", MetodosVariantesLista.get(i));
            break;
        }
        case 3: {
            assertEquals("brew", MetodosVariantesLista.get(i));
            break;
        }
        case 4: {
            assertEquals("customerWantsCondiment", MetodosVariantesLista.get(i));
            break;
        }
        case 5: {
            assertEquals("addCondiment", MetodosVariantesLista.get(i));
            break;
        }
        case 6: {
            assertEquals("brew", MetodosVariantesLista.get(i));
            break;
        }
        case 7: {
            assertEquals("customerWantsCondiment", MetodosVariantesLista.get(i));
            break;
        }
        case 8: {
            assertEquals("addCondiment", MetodosVariantesLista.get(i));
            break;
        }
        case 9: {
            assertEquals("brew", MetodosVariantesLista.get(i));
            break;
        }
        case 10: {
            assertEquals("customerWantsCondiment", MetodosVariantesLista.get(i));
            break;
        }
        case 11: {
            assertEquals("addCondiment", MetodosVariantesLista.get(i));
            break;
        }
        case 12: {
            assertEquals("brew", MetodosVariantesLista.get(i));
            break;
        }
        case 13: {
            assertEquals("customerWantsCondiment", MetodosVariantesLista.get(i));
            break;
        }
        case 14: {
            assertEquals("addCondiment", MetodosVariantesLista.get(i));
            break;
        }
        case 15: {
            assertEquals("brew", MetodosVariantesLista.get(i));
            break;
        }
        case 16: {
            assertEquals("customerWantsCondiment", MetodosVariantesLista.get(i));
            break;
        }
        case 17: {
            assertEquals("addCondiment", MetodosVariantesLista.get(i));
            break;
        }
        case 18: {
            assertEquals("brew", MetodosVariantesLista.get(i));
            break;
        }
        case 19: {
            assertEquals("customerWantsCondiment", MetodosVariantesLista.get(i));
            break;
        }
        case 20: {
            assertEquals("addCondiment", MetodosVariantesLista.get(i));
            break;
        }
        case 21: {
            assertEquals("brew", MetodosVariantesLista.get(i));
            break;
        }
        case 22: {
            assertEquals("customerWantsCondiment", MetodosVariantesLista.get(i));
            break;
        }
        case 23: {
            assertEquals("addCondiment", MetodosVariantesLista.get(i));
            break;
        }
        }
    }
}
```

	
Localización de funciones invariantes.	 
Conteo del número total de funciones plantilla.	

	
Conteo del número total de funciones variantes.	 
Conteo del número total de funciones invariantes.	 

Conteo del número total de funciones protegidas asociadas al patrón de diseño *"Template Method":* • Funciones plantilla – "public" o "friendly". • Funciones variantes – "protected". • Funciones invariantes – "private".	 
Cálculo de la métrica *"PMMP"*.	 

Cálculo de la métrica *"PMFV"*.	 
Cálculo de la métrica *"PMFI"*.	 

Cálculo de la métrica *"PTTM"*.	 

Las pruebas unitarias fueron aprobadas al comparar la igualdad de los resultados esperados con los reales mediante el método "asserEquals()" de la librería JUnit. La prueba de integración hace uso de las métricas aprobadas en las pruebas unitarias, y es aprobada ya que el resultado que se obtiene es el esperado manualmente. Por lo anterior las pruebas del cálculo de métricas de protección modular del patrón de diseño "*Template Method*" son aprobadas y aceptadas.

6.10.7.- Caso de Prueba ISMRTM0506

Nombre del Caso de prueba: Usta Donerci.

Se procede a ejecutar las pruebas unitarias observando que se cumplan las condiciones comparativas manuales con las automáticas resultantes, posteriormente se prueba la mejora en la protección del patrón de diseño *"Template Method"* mediante el cálculo de las métricas y finalmente se prueba que el comportamiento del sistema siga siendo el mismo.

En la siguiente tabla se muestran las características a probar para el proceso de refactorización:

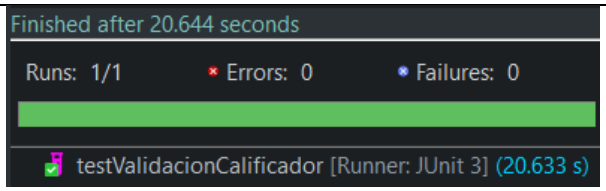
Características a probar	**Pruebas unitarias**

Método de validación del calificador de alcance.	

```
public void testValidacionCalificador() throws IOException {
    escenarioBase();
    Refactorizar refactor = new Refactorizar();
    receptor = (ArrayList<Clase>) refactor.refactorizar(listaDeClases).clone();
    ArrayList<String> calificadorCorrectoMP = refactor.calificadoresCorrectosMP;
    ArrayList<String> calificadorCorrectoFV = refactor.calificadoresCorrectosFV;
    ArrayList<String> calificadorCorrectoFI = refactor.calificadoresCorrectosFI;

    for (int i = 0; i < calificadorCorrectoMP.size(); i++) {
        switch (i) {
        case 0: {
            assertEquals("public", calificadorCorrectoMP.get(i));
            break;
        }
        }
    }
    for (int i = 0; i < calificadorCorrectoFV.size(); i++) {
        switch (i) {
        case 0: {
            assertEquals("protected", calificadorCorrectoFV.get(i));
            break;
        }
        case 1: {
            assertEquals("protected", calificadorCorrectoFV.get(i));
            break;
        }
        case 2: {
            assertEquals("protected", calificadorCorrectoFV.get(i));
            break;
        }
        case 3: {
            assertEquals("protected", calificadorCorrectoFV.get(i));
            break;
        }
        case 4: {
            assertEquals("protected", calificadorCorrectoFV.get(i));
            break;
        }
        case 5: {
            assertEquals("protected", calificadorCorrectoFV.get(i));
            break;
        }
        case 6: {
            assertEquals("protected", calificadorCorrectoFV.get(i));
            break;
        }
        case 7: {
            assertEquals("protected", calificadorCorrectoFV.get(i));
            break;
        }
        case 8: {
            assertEquals("protected", calificadorCorrectoFV.get(i));
            break;
        }
        case 9: {
            assertEquals("protected", calificadorCorrectoFV.get(i));
            break;
        }
        case 10: {
            assertEquals("protected", calificadorCorrectoFV.get(i));
            break;
        }
        case 11: {
            assertEquals("protected", calificadorCorrectoFV.get(i));
            break;
        }
        case 12: {
            assertEquals("protected", calificadorCorrectoFV.get(i));
            break;
        }
        case 13: {
            assertEquals("protected", calificadorCorrectoFV.get(i));
            break;
        }
        case 14: {
            assertEquals("protected", calificadorCorrectoFV.get(i));
            break;
        }
        case 15: {
            assertEquals("protected", calificadorCorrectoFV.get(i));
            break;
        }
        case 16: {
            assertEquals("protected", calificadorCorrectoFV.get(i));
            break;
        }
        case 17: {
            assertEquals("protected", calificadorCorrectoFV.get(i));
            break;
        }
        case 18: {
            assertEquals("protected", calificadorCorrectoFV.get(i));
            break;
        }
        case 19: {
            assertEquals("protected", calificadorCorrectoFV.get(i));
            break;
        }
        case 20: {
            assertEquals("protected", calificadorCorrectoFV.get(i));
            break;
        }
        case 21: {
            assertEquals("protected", calificadorCorrectoFV.get(i));
            break;
        }
        case 22: {
            assertEquals("protected", calificadorCorrectoFV.get(i));
            break;
        }
        case 23: {
            assertEquals("protected", calificadorCorrectoFV.get(i));
            break;
        }
        case 24: {
            assertEquals("protected", calificadorCorrectoFV.get(i));
            break;
        }
        case 25: {
            assertEquals("protected", calificadorCorrectoFV.get(i));
            break;
        }
        case 26: {
            assertEquals("protected", calificadorCorrectoFV.get(i));
            break;
        }
        case 27: {
            assertEquals("protected", calificadorCorrectoFV.get(i));
            break;
        }
        case 28: {
            assertEquals("protected", calificadorCorrectoFV.get(i));
            break;
        }
        case 29: {
            assertEquals("protected", calificadorCorrectoFV.get(i));
            break;
        }
        }

    }
```

Método de refactorización de calificadores de alcance y generación de código refactorizado.

El método de refactorización de código legado con carencia de protección en funciones plantilla y la generación de código son probados utilizando los diagramas de clases como referencia.

Una vez ejecutado el método de refactorización se deben generar las clases que conforman a la aplicación, para corroborar la correcta refactorización se genera el diagrama de clases de la Figura 43 y se comparan los calificadores de alcance de los métodos partícipes del patrón de diseño "*Template Method*" identificados con los de la Figura 42.

A continuación, en la Figura 43, se muestra el diagrama de clases del sistema refactorizado donde se protegieron todas las funciones variantes e invariantes satisfactoriamente tomando en cuenta los distintos escenarios previstos:

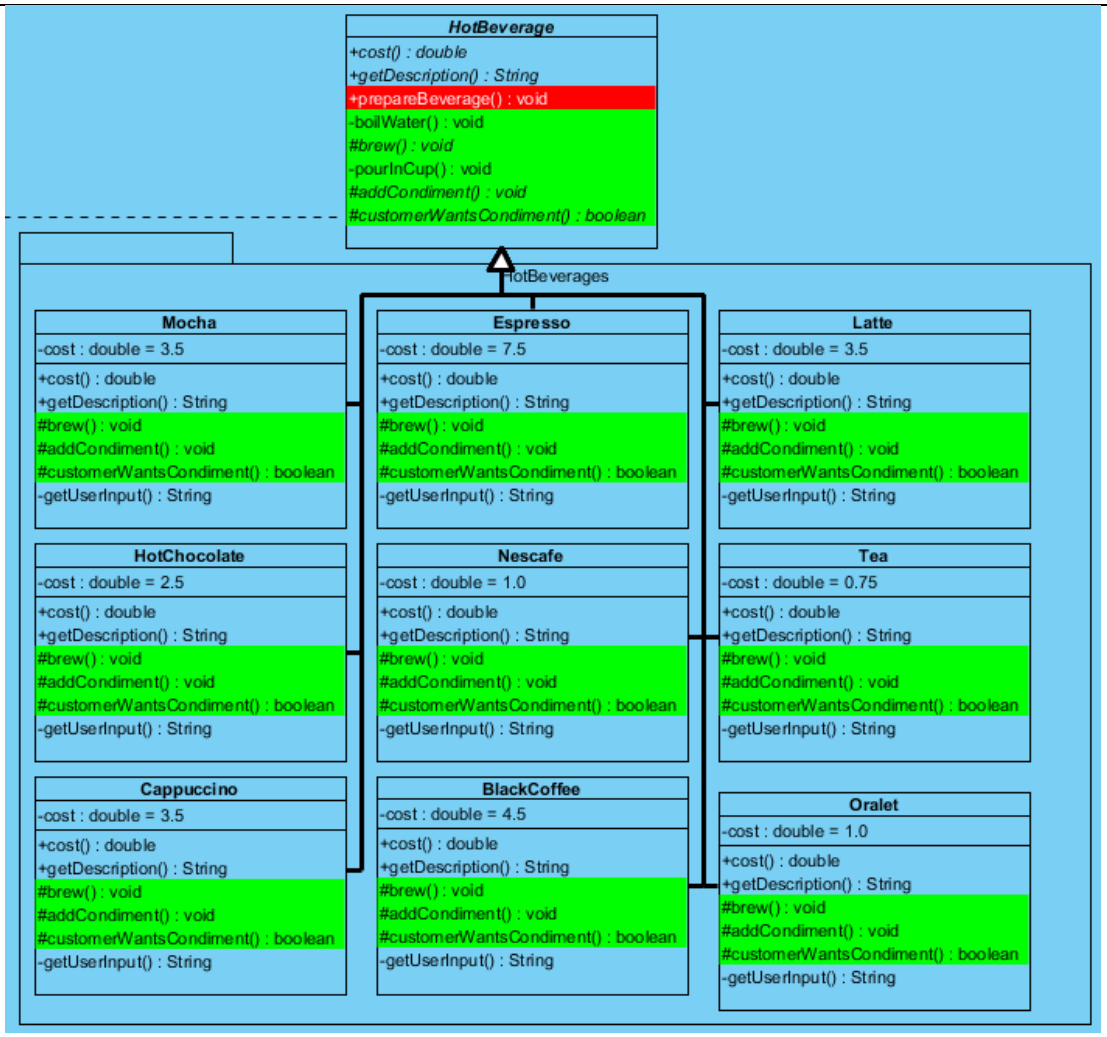


FIGURA 43 CASO DE PRUEBA USTA DONERCI DESPUÉS DE LA REFACTORIZACIÓN

Inicialmente el sistema contaba con carencia de protección en el único método plantilla “prepareBeverage” de la arquitectura, esto es causa del acceso del cliente fuera del paquete, por lo que el grado de protección modular de la función plantilla es de 0 antes y después de la refactorización.

Las funciones variantes del sistema aumentaron el grado de protección de 0 a 1, por lo que antes de la refactorización se presentaba una ausencia de protección en todas las funciones variantes.

La protección modular en funciones invariantes aumentó de 0 a 1, de igual forma se presentaba una ausencia de protección en las funciones invariantes antes de la refactorización.

Finalmente, la protección total del patrón de diseño aumentó de un 0 a 0.666666666, mejorando la protección del patrón de diseño "*Template Method*" para este escenario.

Esto es:

Usta Donerci	PMMP		PMFV		PMFI		PTTM
	Funciones plantilla protegidos	Funciones plantilla Totales	Funciones variantes protegidas	Funciones variantes totales	Funciones invariantes protegidas	Funciones invariantes totales	
Código Legado sin refactorizar	0	1	0	30	0	2	0
Código Legado Refactorizado	0	1	30	30	2	2	0.666667

Comparando gráficamente:

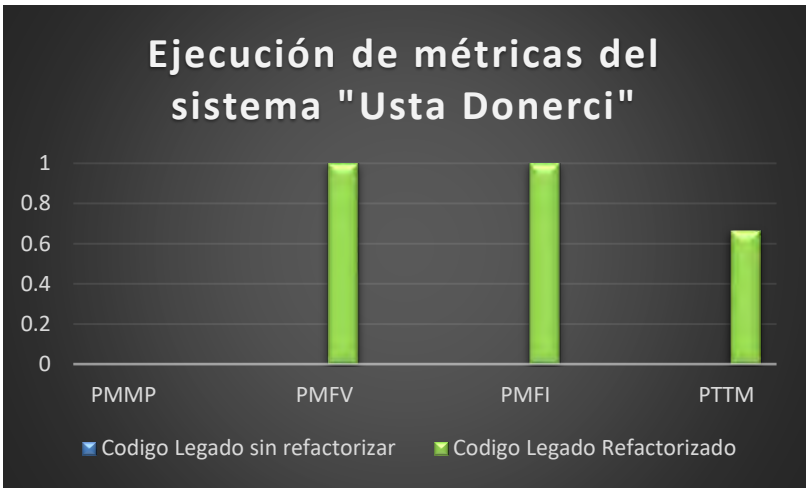


FIGURA 44 GRADO DE MEJORA DEL SISTEMA USTA DONERCI

Con los datos resultantes aplicando las métricas se comprueba que el método de refactorización cumple con la identificación y refactorización de las funciones plantilla, funciones variantes y funciones invariantes participantes del patrón de diseño "*Template Method*". Como se muestra en la Figura 44, antes de la refactorización no hay presencia de valores en las métricas, esto es debido a que no existe protección alguna en la arquitectura. De forma contraria, después de

refactorizar se puede observar el aumento de los valores en las métricas menos en la métrica PMMP ya que la o las funciones plantillas se mantienen públicas.

Verificación del funcionamiento del código refactorizado.

Ejecución de sistema antes de refactorizar	Ejecución de sistema después de refactorizar
1.- Despliegue del menú y selección de la orden:	1.- Despliegue del menú y selección de la orden:
2.- Despliegue y selección de tipo de bebida:	2.- Despliegue y selección de tipo de bebida:
3.- Despliegue, selección de bebida y aviso de orden recibida:	3.- Despliegue, selección de bebida y aviso de orden recibida:
3.- Finalizar orden:	3.- Finalizar orden:

```
********* Welcome to The Usta Donerci *********
What do you want to order?
1 --> Doner
2 --> Meatball
3 --> Kid Menu
4 --> Beverage
0 --> I have finished ordering. Prepare please.
4
```

```
Which type of beverage you want to drink?
1 --> Hot
2 --> Cold
0 --> I don't want to beverage. Back.
1
```

```
What do you want to drink?
1 --> Black Coffee
2 --> Cappuccino
3 --> Hot Chocolate
4 --> Latte
5 --> Mocha
6 --> Espresso
7 --> Oralet
8 --> Tea
9 --> Nescafe
0 --> I don't want to beverage. Back.
6
Espresso order reveived.
```

```
Which type of beverage you want to drink?
1 --> Hot
2 --> Cold
0 --> I don't want to beverage. Back.
0
What do you want to order?
1 --> Doner
2 --> Meatball
3 --> Kid Menu
4 --> Beverage
0 --> I have finished ordering. Prepare please.
0
```

3.- Aceptar oferta de doble expreso y recibir la cuenta:

```
Boiling water
Dripping espresso through filter
Pouring into cup
Would you like double espresso? --- 2.5$ (y/n)
y
Dripping double espresso
Espresso cost is --> 9.0


Total cost is 9.0
```


```
Which type of beverage you want to drink?
1 --> Hot
2 --> Cold
0 --> I don't want to beverage. Back.
0
What do you want to order?
1 --> Doner
2 --> Meatball
3 --> Kid Menu
4 --> Beverage
0 --> I have finished ordering. Prepare please.
0
```

3.- Aceptar oferta de doble expreso y recibir la cuenta:

```
Boiling water
Dripping espresso through filter
Pouring into cup
Would you like double espresso? --- 2.5$ (y/n)
y
Dripping double espresso
Espresso cost is --> 9.0


Total cost is 9.0
```


Las pruebas unitarias son aprobadas al comparar la igualdad de los resultados esperados con los reales con el método comparador “assertEquals()” de la librería JUnit. El método de refactorización y generación de código refactorizado es aprobado al comparar el cambio de calificador de alcance y en consecuencia el grado de mejora en la protección de los métodos participantes del “*Template Method*”. Finalmente, se aprueba la comparación del funcionamiento del sistema antes y después de la refactorización, por lo que se concluye que la refactorización no altera el comportamiento del sistema.

Capítulo 7. Conclusiones y Trabajos Futuros

Conforme a los resultados de las pruebas realizadas en el capítulo 6 “Pruebas” se concluye el cumplimiento del objetivo general del apartado 2.4.1 del capítulo 2.

“*Evitar el cambio de estado de los objetos de sistemas legados de software, realizado inadecuadamente por entidades externas, por medio del uso de funciones distinguidas de acceso abierto (public o friendly) integradas en la implementación del patrón de diseño “Template Method”.*”

De la misma manera se concluye el cumplimiento de los objetivos específicos del apartado 2.4.2 del capítulo 2.

La naturaleza del patrón de diseño “*Template Method*” está en la eliminación de código duplicado factorizando en una super clase abstracta los pasos de algoritmos que se repiten en las subclases, así como en la capacidad de orquestación de los pasos a seguir sobre el algoritmo, permitiendo que la secuencia de los pasos no se vea afectada o manipulada por el cliente o entidades externas.

La naturaleza propia del patrón de diseño requiere de la protección sobre sus funciones participantes para una adecuada implementación del patrón, cuando no existe protección sobre el “*Template Method*”, el acoplamiento indirecto de la arquitectura resultante de esta situación ocasionará que cuando se realice mantenimiento o modificaciones sobre el software, éste podrá propagar fallos a diferentes módulos del sistema, resultando en errores en tiempo de ejecución o en resultados falsos o incorrectos.

En este trabajo de tesis como principal aportación se presentó un método de refactorización para la aumentar la protección en funciones plantilla de software legado orientado a objetos, la implementación del método en un sistema para refactorizar código legado en Java utilizando las técnicas desarrolladas de “Candidateo semántico”, “Refactorización conducida por tratamientos” y “Ajustes de roles por metamorfosis”. Así mismo como aportación secundaria se presentó un

marco de métricas que permite medir el grado de protección en arquitecturas que implementan este patrón de diseño.

En el análisis de escenarios del método de refactorización se encontraron escenarios donde existe código duplicado el cual es denominado como "*code smell*" (Figura 45), esto sobre las arquitecturas que implementan el patrón de diseño "*Template Method*" a causa de una mala implementación de dicho patrón. Estos escenarios no lograron ser refactorizados ya que requieren un estudio más profundo, esto se debe a que el código duplicado en el patrón de diseño puede ser estructuralmente similar o idéntico a otro, cuando se presenta el primer caso es necesario corroborar que se trata del mismo código por lo que se propone un método o técnica de refactorización que permita identificar este código.

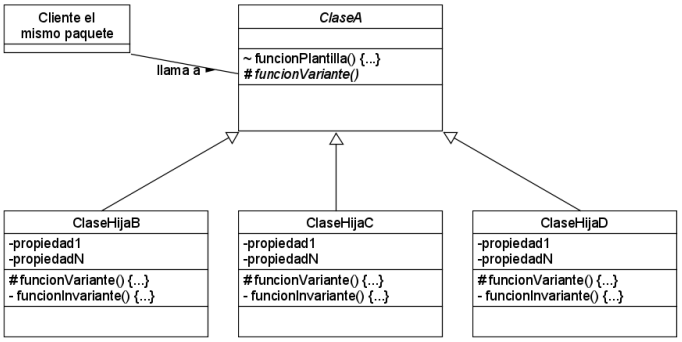


FIGURA 45 ARQUITECTURA CON CÓDIGO DUPLICADO

Otro de los hallazgos en los escenarios fue en un escenario donde se implementa sobrecarga de funciones invariantes (Figura 46). El método para diferenciar estas funciones es mediante la firma del método, esto consiste en obtener el nombre de la función, el número de parámetros y el tipo de cada parámetro de la función. El problema con este escenario radica en que el analizador sintáctico no permite dar seguimiento a la información necesaria de los parámetros de las funciones sobrecargadas, por lo que no se puede identificar cual es la función invariante a la que llama la función plantilla cuando las funciones sobrecargadas tienen el mismo número de parámetros. Como solución a lo anterior, se propone la creación de un marco de analizadores sintácticos que permita que las estructuras de datos que representan proyectos de software escritos en java, puedan ser pobladas con distinta información según lo requiera la técnica de refactorización a realizar.

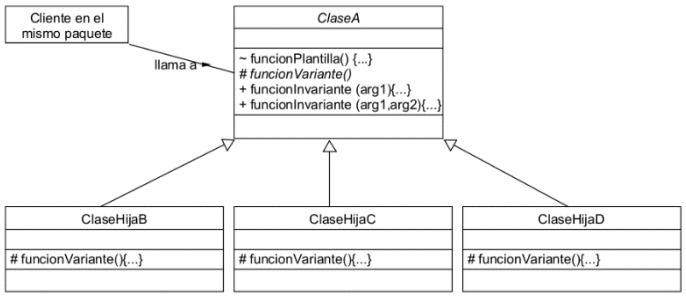


FIGURA 46 ARQUITECTURA CON SOBRECARGA EN FI

Durante la construcción de los tratamientos para la técnica de refactorización se encontraron distintos escenarios para cubrir en posteriores trabajos de investigación.

Para la refactorización de funciones invariantes se encontraron escenarios donde estas funciones son virtuales, por lo que el código que se sobrescribe pasa a ser "dispensable". El *code smell* de tipo "dispensable" como se clasifica en (Alexander, 2019), es código innecesario el cual su ausencia haría que el código fuera más limpio y fácil de mantener, es decir que se puede eliminar sin que este afecte el funcionamiento. En otros casos algunas arquitecturas de software revisadas que no implementan concretamente el "*Template Method*", se encontró "herencia especulativa", que consisten en clases concretas no usadas que también forman parte del código "dispensable" (Alexander, 2019).
En la refactorización de funciones invariantes se encontró código que es accedido por otros clientes además del método plantilla, por lo que no pueden ser privados y violentan el principio de única responsabilidad. Aunque el código se protege, este no se ve reflejado a causa de que no está bien implementado el patrón de diseño. Por estos casos se proponen dos métodos de refactorización para trabajos futuros, un método de refactorización que refactorice el *code smell* de tipo "dispensable" y otro que refactorice el código legado para el cumplimiento del principio de única responsabilidad en un dominio funcional.
También se observa un gran índice de carencia de protección en todos los casos de prueba revisados, algunos de los casos de prueba que implementan el patrón de diseño "*Template Method*" y cuentan con una carencia total de protección en el patrón de diseño, habrían sido realizados por expertos con múltiples certificaciones en el lenguaje de desarrollo Java, por lo que ningún desarrollador queda exento de inyectar *code smell*.

Existen diversos escenarios de *code smell* en un dominio de protección funcional que se pueden encontrar, no obstante, hay una delgada línea entre que el patrón de diseño presente *code smell* y en que esté mal implementado y por lo tanto no

sea un patrón de diseño. En el segundo caso se utiliza la propiedad de metamorfosis en los tratamientos para tratar el código como funciones plantilla bien implementadas, de otra forma serían escenarios donde no se implementa bien el patrón de diseño y por lo tanto no existe el "*Template Method*" en la arquitectura.

Finalmente se observó en el tratamiento de la función plantilla durante la etapa de calibrado del calificador, que regularmente se tendrá el calificador público, ya que, aunque este calificador manifiesta la ausencia de protección es necesario que se mantenga público cuando un cliente externo al paquete hace uso del método.

En el diseño y marco de métricas cabe mencionar que generalmente no se obtendrá un valor de 1 en la métrica de protección modular de funciones plantilla (PMMP), puesto que, en el patrón de diseño "*Template Method*" se considera adecuado que el método principal sea calificado como "public", y este calificador manifiesta ausencia total de protección. El valor de 1 en la métrica PMMP se puede dar para el caso de que la parte cliente o el usuario esté integrado en el mismo paquete, siendo el calificador "friendly" el que tiene un poco más de protección que los métodos públicos. Así mismo, se puede dar el caso, en que exista una ausencia total de protección en el "*Template Method*", cuando suceda que todas las funciones, variantes, invariantes y la función plantilla, sean etiquetados con el calificado de alcance "public". Otra cosa que puede suceder es que las funciones variantes sean etiquetadas con alcance "public" o "friendly", lo cual mermará sustancialmente la protección total, puesto que estas funciones son las que permiten la flexibilidad a cambios de comportamiento, propiedad muy usada en programas orientados a objetos. Los casos de prueba utilizados obtienen el mismo grado de protección en la métrica PTTM de 0.6666 después de la refactorización, esto sucede debido a que el mejor valor de cada métrica PMMP, PMFV y PMFI forma $\frac{1}{3}$ del mejor valor de la métrica PTTM, es decir, cuando el resultado de PMFV es igual a 1, entonces la métrica PTTM se representa fraccionalmente como $\frac{1}{3}$, de la misma manera si la métrica de PMFI tiene como resultado 1 y añadimos este resultado a la métrica PTTM, ahora la representación fraccional de PTTM sería $\frac{2}{3}$, esto es, en representación numérica y como sucede en los resultados de los casos de prueba aplicando las métricas esto es un 0.6666. Estas métricas fueron publicadas en (RAMÍREZ GARCÍA et al., 2023) como producto resultante de esta tesis.

Referencias

Abid, C., Alizadeh, V., Kessentini, M., Ferreira, T. do N., & Dig, D. (2020). 30 Years of Software Refactoring Research:A Systematic Literature Review. *IEEE TRANSACTIONS OF SOFTWARE ENGINEERING*, *1*(1). https://doi.org/10.48550/arxiv.2007.02194

Alexander, S. (2019). *Dive Into Refactoring* (v2019-1.3 ed.). https://www.goodreads.com/book/show/53147841-dive-into-refactoring

Ávila Melgar, E. Y. (2006). *Método de refactorización de software legado para desacoplar el código funcional de la vista del código funcional de la aplicación*. cenidet.

Barón Pérez, N. (2020). *Método de refactorización para mejorar la protección modular de arquitecturas orientadas a objetos de sistemas de software existente* [cenidet]. https://rinacional.tecnm.mx/jspui/bitstream/TecNM/3015/1/Tesis.pdf

Brito, A., Hora, A., & Valente, M. T. (2020). Refactoring Graphs: Assessing Refactoring over Time. *SANER 2020 - Proceedings of the 2020 IEEE 27th International Conference on Software Analysis, Evolution, and Reengineering*, 367–377. https://doi.org/10.1109/SANER48275.2020.9054864

Brito e Abreu, F., & Melo, W. (1996). Evaluating the impact of object-oriented design on software quality. *International Software Metrics Symposium, Proceedings*, 90–99. https://doi.org/10.1109/METRIC.1996.492446

Bustamante Laos, C. (2003). *Reestructuración de código legado a partir del comportamiento para la generación de componentes reutilizables* [cenidet]. https://www.cenidet.edu.mx/subplan/biblio/seleccion/Tesis/MC%20Cesar%20Bustamante%20Laos%202003.pdf

Caldeira, J., Abreu, F. B. e, Cardoso, J., & Reis, J. (2020). Unveiling process insights from refactoring practices. *Arxiv*. https://doi.org/10.1016/j.csi.2021.103587

Cárdenas Robledo, L. A. (2004). *Refactorización de marcos orientados a objetos para reducir el acoplamiento aplicando el patrón de diseño mediator* [cenidet]. https://www.cenidet.edu.mx/subplan/biblio/seleccion/Tesis/MC%20Leonor%20Adriana%20Cardenas%20Robledo%202004.pdf

Cervantes O, J., Gómez F, M. del C., González P, P. P., & García N, A. (2016). *Introducción a la programación orientada a objetos* (1st ed.). Universidad Autónoma Metropolitana.

Coronado Padilla, J. (2007). Escalas de medición. *Paradigmas: Una Revista Disciplinar de Investigación*, *2*(2), 104–125.

Engineering Standards Committee of the IEEE Computer Society, S. (2008). *IEEE Std 829™-2008 IEEE Standard for Software and System Test Documentation IEEE Computer Society*. https://doi.org/10.1109/IEEESTD.2008.4578383

Fowler, M. (2003). *UML Distilled: A Brief Guide to the Standard Object Modeling Language* (Third Edition). Addison-Wesley Professional.

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

Hernández Moreno, L. A. (2003). *Factorización de funciones hacia métodos de plantilla*. cenidet.

ISO/IEC. (2004). *ISO - ISO/IEC 19501:2005 - Information technology — Open Distributed Processing — Unified Modeling Language (UML) Version 1.4.2*. https://www.iso.org/standard/32620.html

Juillerat, N., & Hirsbrunner, B. (2007). Toward an implementation of the "form template method" refactoring. *SCAM 2007 - Proceedings 7th IEEE International Working Conference on Source Code Analysis and Manipulation*. https://doi.org/10.1109/SCAM.2007.11

Khatchadourian, R., & Masuhara, H. (2017). Automated Refactoring of Legacy Java Software to Default Methods. *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering, ICSE 2017*, 82–93. https://doi.org/10.1109/ICSE.2017.16

Kolb, R., Muthig, D., Patzke, T., & Yamauchi, K. (2006, March). Refactoring a legacy component for reuse in a software product line: A case study. *Journal of Software Maintenance and Evolution*, *18*(2), 109–132. https://doi.org/10.1002/smr.329

Martin Fowler, & Beck, K. (2018). *Refactoring: Improving the Design of Existing Code* (2nd ed.). Addison-Wesley Professional.

Martin, R. C. (2000). *Design Principles and Design Patterns*. www.objectmentor.com

Maruyama, K., & Omori, T. (2011). A security-Aware refactoring tool for Java programs. *WRT 2011 - Proceedings of the 4th Workshop on Refactoring Tools, Co-Located with ICSE 2011*, 22–28. https://doi.org/10.1145/1984732.1984737

Morales Méndez Armando. (2004). *Reestructuración de software escrito por procedimientos conducido por patrones de diseño composicionales*. cenidet.

Oracle Corporation. (n.d.). *The Java™ Tutorials*. https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

Ortiz Gutiérrez, O. (2020). *Re-factorización de código para reducir el acoplamiento entre clases relacionadas por herencia de implementación en arquitecturas orientadas a objetos* [cenidet]. https://rinacional.tecnm.mx/jspui/handle/TecNM/4431

Padilla Salgado, P. (2019). *Método de re-factorización de código java con interfaces y abstracciones incorrectas* [cenidet]. https://rinacional.tecnm.mx/jspui/handle/TecNM/1368

Parr, T. (2013). *The Definitive ANTLR 4 Reference* (P1.0). The Pragmatic Bookshelf. http://pragprog.com.

Peruma, A., Simmons, S., AlOmar, E. A., Newman, C. D., Mkaouer, M. W., & Ouni, A. (2021). How Do I Refactor This? An Empirical Study on Refactoring Trends and Topics in Stack Overflow. *Arxiv*. https://doi.org/10.1007/s10664-021-10045-x

Ramasubbu, N., & Kemerer, C. F. (2021). Controlling technical debt remediation in outsourced enterprise systems maintenance: an empirical analysis. *Journal of Management Information Systems*, *38*(1), 4–28. https://doi.org/10.1080/07421222.2021.1870377

Ramírez Cruz, M. (2022). *Métodos de re-factorización de código Java para mejorar su modularidad y reducir las dependencias entre clases de objetos* [CENIDET]. https://rinacional.tecnm.mx/jspui/handle/TecNM/4965

RAMÍREZ GARCÍA, E. A., SANTAOLAYA SALGADO, R., VALENZUELA ROBLES, B. D., & FRAGOSO DIAZ, O. G. (2023, January 27). METRICS FOR THE PROTECTION OF TEMPLATE METHODS IN OBJECT-ORIENTED ARCHITECTURES. *Dyna New Technologies*, *10*. https://doi.org/doi.org/10.6036/NT10751

Samarthyam, G., Suryanarayana, G., & Sharma, T. (2016). Refactoring for software architecture smells. *IWoR 2016 - Proceedings of the 1st International Workshop on Software Refactoring, Co-Located with ASE 2016*, 1–4. https://doi.org/10.1145/2975945.2975946

Santos Neto, B. F. dos, Ribeiro, M., da Silva, V. T., Braga, C., de Lucena, C. J. P., & de Barros Costa, E. (2015). AutoRefactoring: A platform to build refactoring agents. *Expert Systems with Applications*, *42*(3), 1652–1664. https://doi.org/10.1016/j.eswa.2014.09.022

Soares, G., Gheyi, R., Serey, D., & Massoni, T. (2010). Making program refactoring safer. *IEEE Software*, *27*(4), 52–57. https://doi.org/10.1109/MS.2010.63

Suryanarayana, G., Suryanarayana, G., & Sharma, T. (2014). *Refactoring for Software Design Smells: Managing Technical Debt* (Edición Illustrated). Morgan Kaufmann Publishers.

Tello Díaz, R. F. (2019). *MÉTODOS DE REFACTORIZACIÓN DE ARQUITECTURAS DE SOFTWARE CON CARENCIA DE ABSTRACCIONES*. CENIDET.

Terence Parr. (n.d.). *StringTemplate*. Retrieved November 26, 2022, from https://www.stringtemplate.org/

Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Penta, M. di, de Lucia, A., & Poshyvanyk, D. (2017). When and Why Your Code Starts to Smell Bad (and Whether the Smells Go Away). *IEEE Transactions on Software Engineering*, *43*(11), 1063–1088. https://doi.org/10.1109/TSE.2017.2653105

Valdés Marrero, M. A. (2004). *Método de Refactorización de Marcos de Aplicaciones Orientados a Objetos por la Separación de Interfaces*.

Zafeiris, V. E., Poulias, S. H., Diamantidis, N. A., & Giakoumakis, E. A. (2017). Automated refactoring of super-class method invocations to the Template Method design pattern. *Information and Software Technology*, *82*, 19–35. https://doi.org/10.1016/j.infsof.2016.09.008

Zimmermann, O. (2015). Architectural refactoring: A task-centric view on software evolution. *IEEE Software*, *32*(2). https://doi.org/10.1109/MS.2015.37

Zuse, H. (1992). Properties of software measures. *Software Quality Journal*, *1*, 225–260. https://doi.org/https://doi.org/10.1007/BF01885772

Zuse, H. (1996). Foundations of object-oriented software measures. *International Software Metrics Symposium, Proceedings*. https://doi.org/10.1109/METRIC.1996.492445

Zuse, H., & Bollmann, P. (1989). Software metrics: Using Measurement Theory to Describe the Properties and Scales of Static Software Complexity Metrics. *ACM SIGPLAN Notices*, *24*(8), 23–33. https://doi.org/10.1145/70470.70473

Anexo A.- Sustento de métricas PMMP, PMFV, PMFI, PTTM

Sustento Teórico de la Métrica PMMP (Protección Modular de Funciones plantilla) como Escala de Razón

Sea el sistema relacional empírico **A** = (P, R, Op), donde:

P	es una arquitectura de software a ser medida la protección en sus funciones plantilla
R	es la relación •≥ (“igual o más protegido que”) de funciones plantilla sobre P
Op	es una operación binaria cerrada sobre los objetos empíricos P

Sea el sistema relacional formal **B** = (Q, S, Ope), donde:

Q	arquitectura con un conjunto de funciones plantilla
S	es la relación ≥ (“igual o más protegido que”) sobre Q
Ope	es la operación binaria cerrada de adición (+) sobre los objetos Q

La métrica PMMP:

Sea la métrica μ = PMMP un mapeo μA → B, que para cada objeto empírico P ∈ A, produce un valor formal medido μ(P) ∈ B.

Entonces, la métrica PMMP será clasificada como escala de razón si:

•≥	es una relación de orden débil (Transitividad y Completitud)
El sistema relacional empírico (P, •≥, Op)	es una estructura extensa
P1 •≥ P2 ⇔ PMMP(P1) ≥ PMMP(P2)	se cumple el Homomorfismo de escala ordinal
PMMP(P1 + P2) = PMMP(P1) + PMMP(P2)	se cumple el Homomorfismo de razón

Para la comprobación de las condiciones enunciadas se utiliza la ecuación del capítulo 4.1 que representa a la métrica PMMP para evaluar el grado de protección modular de funciones plantilla sobre las arquitecturas presentadas en las figuras P1, P3, P4 y P5 donde la clase cliente se encuentra en el mismo paquete en todas las arquitecturas y el calificador de alcance es representado en UML (Fowler, 2003) con los siguientes indicadores de visibilidad:

- +: public.
- -: private.
- ~: friendly.
- #: protected.

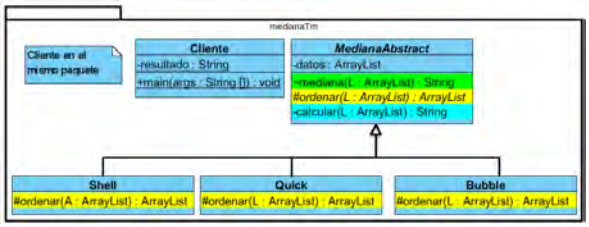
Observación empírica:
En estas figuras se muestra la representación empírica para el cumplimiento de los homomorfismos, donde la protección en las funciones plantilla es P1 •> P2 •≥ P3 •≥ P4. También se tiene identificado el patrón de diseño "*Template Method*" y los funciones plantilla ("mediana() : double"), marcados en color verde:

$$\mathrm{PMMP(P1)} = \frac{\sum_{ci=1}^{ci=n}(\sum_{fi=0}^{fi=m} MPP)}{MP} = \frac{1}{1} = 1$$

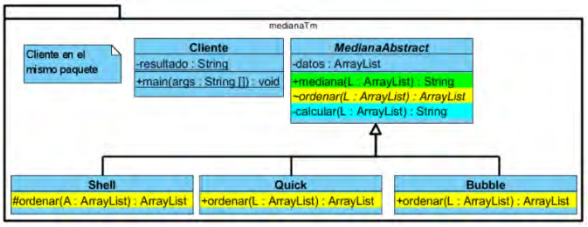
$$\mathrm{PMMP(P2)} = \frac{\sum_{ci=1}^{ci=n}(\sum_{fi=0}^{fi=m} MPP)}{MP} = \frac{0}{1} = 0$$

$$\mathrm{PMMP(P3)} = \frac{\sum_{ci=1}^{ci=n}(\sum_{fi=0}^{fi=m} MPP)}{MP} = \frac{0}{1} = 0$$

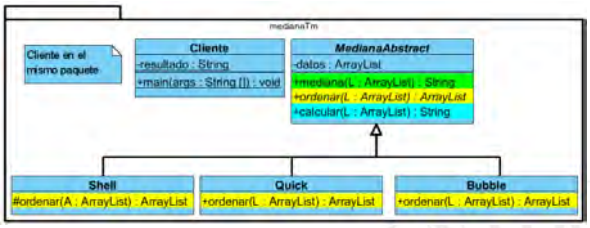
$$\mathrm{PMMP(P4)} = \frac{\sum_{ci=1}^{ci=n}(\sum_{fi=0}^{fi=m} MPP)}{MP} = \frac{0}{1} = 0$$



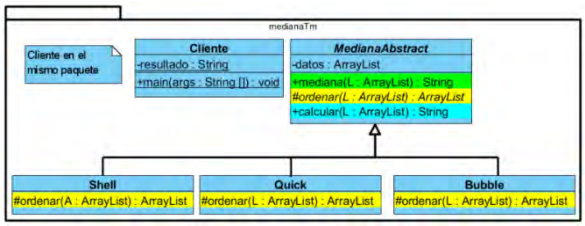
ARQUITECTURA 1 P1



ARQUITECTURA 3 P2



ARQUITECTURA 2 P3



ARQUITECTURA 4 P4

Se comprueba que la relación binaria es de orden débil:

Transitividad:

Dadas las arquitecturas P1, P2 y P3, se tiene la siguiente observación empírica:

- P1 •≥ P2 ^ P2 •≥ P3 => P1 •≥ P3

Utilizando las métricas se obtiene la siguiente observación formal:

- PMMP(P1) ≥ PMMP(P2) ^ PMMP(P2) ≥ PMMP(P3) => PMMP(P1) ≥ PMMP(P3)

La representación numérica es:

- 1 ≥ 0 ^ 0 ≥ 0 => 1 ≥ 0

La arquitectura P1 presenta mayor grado de protección en sus funciones plantillas, ya que la función plantilla que contiene está protegido correctamente. La arquitectura P1 está mayormente protegida que la arquitectura P2 y a su vez P2 tiene el mismo grado de protección que P3.

Por lo que P1 tiene mayor grado de protección que P3, cumpliéndose que la relación binaria "•≥" es transitiva.

Completitud:

Dadas las arquitecturas P1 y P2, se podría decir que P1 tiene mayor o igual grado de protección modular en funciones plantilla que P2 o viceversa. Observando empíricamente:

- P1 •≥ P2 | P2 •≥ P1

Calculando la métrica PMMP en las arquitecturas P1 y P2, se obtiene el siguiente resultado formal:

- 1 ≥ 0.

Dado que la arquitectura P1 tiene un valor en la métrica PMMP de P1 y P2 tiene un valor 0, se tiene que:

- PMMP(P1) ≥ PMMP(P2).

Por lo tanto, la métrica PMMP es una relación binaria completa, ya que siempre fue posible determinar el mayor o igual grado de protección de las funciones plantilla en las distintas arquitecturas.

Asociatividad:

Dadas las arquitecturas P1, P2 y P3, se puede decir que la asociación de las arquitecturas para operarlas es posible sin alterar el resultado. Observando empíricamente:

- P1 Op (P2 Op P3) ≈ (P1 Op P2) Op P3

Calculando la métrica PMMP en las arquitecturas P1, P2 y P3, y sumándolas se obtienen los siguientes resultados formales:

- 1 + (0 + 0) ≈ (1 + 0) + 0
- 1 ≈ 1

Dado que los valores de la métrica PMMP aplicada en las arquitecturas pueden asociarse y operarse sin alterar el resultado, se tiene que:

- PMMP(P1) + ((PMMP(P2) + PMMP(P3)) ≈ ((PMMP(P1) + PMMP(P2)) + PMMP(P3)

Por lo tanto, la métrica PMMP cumple con el axioma de asociatividad, ya que siempre fue posible determinar el resultado de la operación binaria cerrada asociando diferentes elementos de la operación.

Conmutatividad:

Dadas las arquitecturas P1 y P2, se puede decir que el orden de las arquitecturas en la operación binaria cerrada no altera el resultado. Observando empíricamente:

- P1 Op P2 ≈ P2 Op P1

Calculando la métrica PMMP en las arquitecturas P1, P2, y sumándolas se obtienen los siguientes resultados:

- 1 + 0 ≈ 0 + 1
- 1 ≈ 1

Dado que el orden de los valores de la métrica PMMP aplicada en las arquitecturas no alteran el resultado, se tiene que:

- PMMP(P1) + PMMP(P2) ≈ PMMP(P2) + PMMP(P1).

Por lo tanto, la métrica PMMP cumple con el axioma de conmutatividad, ya que siempre fue posible determinar el resultado de la operación binaria cerrada alternando los diferentes elementos de la operación.

Monotonicidad:

Dadas las arquitecturas P1, P2 y P3, se pueden extender hechos a partir de una relación binaria. Observando empíricamente:

- P1 •≥ P2 => P1 Op P3 •≥ P2 Op P3

Calculando la métrica PMMP en las arquitecturas P1, P2 y P3, y asumiendo hechos a partir de la comparación de la arquitectura P1 con la P2 se obtienen los siguientes resultados formales:

- 1 ≥ 0 => 1 + 0 ≥ 0 + 0

Dado que los hechos a partir de la relación binaria de orden débil son ciertos, se tiene que:

- PMMP(P1) ≥ PMMP(P2) => PMMP(P1) + PMMP(P3) ≥ PMMP(P2) + PMMP(P3).

Por lo tanto, la métrica PMMP cumple con el axioma de monotonicidad, ya que siempre fue posible determinar los hechos aditivos derivados a partir de la relación binaria de orden débil.

Axioma de Arquímedes:

El axioma dicta que dadas las arquitecturas P1, P2, P3 y P4, guardan una razón si al ser multiplicadas por un número natural n pueden extenderse mutuamente. Observando empíricamente:

- P1 •> P2 => para cada P3, P4, existe un número natural n > 0, de tal manera que: nP1 Op P3 •> nP2 Op P4

Calculando la métrica PMMP en las arquitecturas P1, P2, P3 y P4, donde n = 2, se tiene la siguiente observación formal:

- PMMP(P1) > PMMP(P2) => (2)PMMP(P1) + PMMP(P3) > (2)PMMP(P2) + PMMP(P4)

Calculando:

- 1 > 0 => (2)1 + 0 > (2)0 + 1
- 1 > 0 => 2 > 1

Las magnitudes guardan una razón, por lo tanto, la métrica PMMP cumple con el axioma de Arquímedes, ya que existe una razón entre las magnitudes.

Homomorfismo de escala ordinal

El Homomorfismo P1 •≥ P2 ⇔ PMMP(P1) ≥ PMMP(P2).

Observando empíricamente:

- P1 •≥ P2 •≥ P3 •≥ P4 ⇔ PMMP(P1) ≥ PMMP(P2) ≥ PMMP(P3) ≥ PMMP(P4).

Calculando la métrica PMMP en las arquitecturas P1, P2, P3 y P4, se obtiene la siguiente observación formal:

- 1 ≥ 0 ≥ 0 ≥ 0

Se demuestra que la métrica PMMP es un Homomorfismo ya que al aplicar los valores de la métrica se obtiene el mismo comportamiento.

Homomorfismo de razón

El Homomorfismo PMMP(P1 + P2) = PMMP(P1) + PMMP(P2)

Sería denotado como:

Dadas las arquitecturas P1 y P2, la métrica PMMP con la característica de ser aditiva, la cual dicta que los resultados de las métricas son sumables:

- PMMP ($\frac{1}{1} + \frac{0}{1}$) = PMMP($\frac{1}{1}$) + PMMP($\frac{0}{1}$).
- PMMP ($\frac{1}{1}$)= PMMP($\frac{1}{1}$) + PMMP ($\frac{0}{1}$).
- 1 = 1 + 0.
- 1 = 1.

Se demuestra que la métrica PMMP es un homomorfismo de razón, ya que al aplicar los valores de la métrica y sumarlos se comprueba la propiedad de adición y la propiedad de decidir si operar antes o después de la ejecución de la métrica y que el comportamiento sea el mismo.

Conclusión:

La métrica PMMP cumple con las condiciones:

•≥	es una relación de orden débil (Transitividad y Completitud)
El sistema relacional (P, •≥, Op)	es una estructura cerrada extensa
P1 •≥ P2 ⇔ PMMP(P1) ≥ PMMP(P2)	se cumple el Homomorfismo de escala ordinal
PMMP(P1 + P2) = PMMP(P1) + PMMP(P2)	se cumple el Homomorfismo de razón

Por lo tanto, se concluye que la métrica PMMP es de razón.

Sustento Teórico de la Métrica PMFV (Protección Modular de Funciones Variantes) como Escala de Razón

Sea el sistema relacional empírico **A** = (P, R, Op), donde:

P	es una arquitectura de software a ser medida la protección en sus funciones variantes
R	es la relación •≥ (“igual o más protegido que”) de funciones variantes sobre P
Op	es una operación binaria cerrada sobre los objetos empíricos P

Sea el sistema relacional formal **B** = (Q, S, Ope), donde:

Q	arquitectura con un conjunto de funciones variantes
S	es la relación ≥ (“igual o más protegido que”) sobre Q
Ope	es la operación binaria cerrada de adición (+) sobre los objetos Q

La métrica PMFV:

Sea la métrica μ = PMFV un mapeo μA → B, que para cada objeto empírico P ∈ A, produce un valor formal medido μ(P) ∈ B.

Entonces, la métrica PMFV será clasificada como escala de razón si:

•≥	es una relación de orden débil (Transitividad y Completitud)
El sistema relacional empírico (P, •≥, Op)	es una estructura extensa
P1 •≥ P2 ⇔ PMFV(P1) ≥ PMFV(P2)	se cumple el Homomorfismo de escala ordinal
PMFV(P1 + P2) = PMFV(P1) + PMFV(P2)	se cumple el Homomorfismo de razón

Aplicando la métrica para la obtención del grado de protección modular de funciones variantes del apartado 4.2 para efectuar el cálculo de la métrica PMFV sobre las arquitecturas presentadas en las figuras P1 a P4. En estas arquitecturas se tiene identificado el patrón de diseño “*Template Method*” y sus funciones variantes (“*ordenar(L : ArrayList) : ArrayList*”) marcadas con el color amarillo:

$$\mathrm{PMFV(P1)} = \frac{\sum_{ci=1}^{ci=n}(\sum_{fi=0}^{fi=m} FVP)}{FV} = \frac{4}{4} = 1$$

$$\mathrm{PMFV(P2)} = \frac{\sum_{ci=1}^{ci=n}(\sum_{fi=0}^{fi=m} FVP)}{FV} = \frac{0}{4} = 0$$

$$\text{PMFV(P3)} = \frac{\sum_{ci=1}^{ci=n}(\sum_{fi=0}^{fi=m} FVP)}{FV} = \frac{0}{4} = 0$$

$$\text{PMFV(P4)} = \frac{\sum_{ci=1}^{ci=n}(\sum_{fi=0}^{fi=m} FVP)}{FV} = \frac{4}{4} = 1$$

Se comprueba que la relación binaria es de orden débil:

Transitividad:

Dadas las arquitecturas P1, P2 y P3, se tiene la siguiente observación empírica:

- P1 •≥ P2 ^ P2 •≥ P3 => P1 •≥ P3

Utilizando las métricas se obtiene la siguiente observación formal:

- PMFV(P1) ≥ PMFV(P2) ^ PMFV(P2) ≥ PMFV(P3) => PMFV(P1) ≥ PMFV(P3)

La representación numérica es:

- 1 ≥ 0 ^ 0 ≥ 0 => 1 ≥ 0

La arquitectura P1 presenta mayor grado de protección en sus funciones variantes, ya que las funciones variantes que contiene están protegidas correctamente. La arquitectura P1 está mayormente protegida que la arquitectura P2 y a su vez P2 tiene el mismo grado de protección que P3.

Por lo que P1 tiene mayor grado de protección que P3, cumpliéndose que la relación binaria "•≥" es transitiva.

Completitud:

Dadas las arquitecturas P1 y P2, se podría decir que P1 tiene mayor o igual grado de protección modular en sus funciones variantes que P2 o viceversa. Observando empíricamente:

- P1 •≥ P2 | P2 •≥ P1

Calculando la métrica PMFV en las arquitecturas P1 y P2, se obtiene el siguiente resultado formal:

- 1 ≥ 0.

Dado que la arquitectura P1 tiene un valor de 1 en la métrica PMFV y P2 tiene un valor 0, se tiene que:

- PMFV(P1) ≥ PMFV(P2).

Por lo tanto, la métrica PMFV es una relación binaria completa, ya que siempre fue posible determinar el mayor o igual grado de protección de las funciones variantes en las distintas arquitecturas.

Asociatividad:

Dadas las arquitecturas P1, P2 y P3, se puede decir que la asociación de las arquitecturas para operarlas es posible sin alterar el resultado. Observando empíricamente:

- P1 Op (P2 Op P3) ≈ (P1 Op P2) Op P3

Calculando la métrica PMFV en las arquitecturas P1, P2 y P3, y sumándolas se obtienen los siguientes resultados formales:

- 1 + (0 + 0) ≈ (1 + 0) + 0
- 1 ≈ 1

Dado que los valores de la métrica PMFV aplicada en las arquitecturas pueden asociarse y operarse sin alterar el resultado, se tiene que:

- PMFV(P1) + ((PMFV(P2) + PMFV(P3)) ≈ ((PMFV(P1) + PMFV(P2)) + PMFV(P3)

Por lo tanto, la métrica PMFV cumple con el axioma de asociatividad, ya que siempre fue posible determinar el resultado de la operación binaria cerrada asociando diferentes elementos de la operación.

Conmutatividad:

Dadas las arquitecturas P1 y P2, se puede decir que el orden de las arquitecturas en la operación binaria cerrada no altera el resultado. Observando empíricamente:

- P1 Op P2 ≈ P2 Op P1

Calculando la métrica PMFV en las arquitecturas P1, P2, y sumándolas se obtienen los siguientes resultados:

- 1 + 0 ≈ 0 + 1
- 1 ≈ 1

Dado que el orden de los valores de la métrica PMFV aplicada en las arquitecturas no alteran el resultado, se tiene que:

- PMFV(P1) + PMFV(P2) ≈ PMFV(P2) + PMFV(P1).

Por lo tanto, la métrica PMFV cumple con el axioma de conmutatividad, ya que siempre fue posible determinar el resultado de la operación binaria cerrada alternando los diferentes elementos de la operación.

Monotonicidad:

Dadas las arquitecturas P1, P2 y P3, se pueden extender hechos a partir de una relación binaria. Observando empíricamente:

- P1 •≥ P2 => P1 Op P3 •≥ P2 Op P3

Calculando la métrica PMFV en las arquitecturas P1, P2 y P3, y asumiendo hechos a partir de la comparación de la arquitectura P1 con la P2 se obtienen los siguientes resultados formales:

- 1 ≥ 0 => 1 + 0 ≥ 0 + 0

Dado que los hechos a partir de la relación binaria de orden débil son ciertos, se tiene que:

- PMFV(P1) ≥ PMFV(P2) => PMFV(P1) + PMFV(P3) ≥ PMFV(P2) + PMFV(P3).

Por lo tanto, la métrica PMFV cumple con el axioma de monotonicidad, ya que siempre fue posible determinar los hechos aditivos derivados a partir de la relación binaria de orden débil.

Axioma de Arquímedes:

El axioma dicta que dadas las arquitecturas P1, P2, P3 y P4, guardan una razón si al ser multiplicadas por un número natural n pueden extenderse mutuamente. Observando empíricamente:

- P1 •> P2 => para cada P3, P4, existe un número natural n > 0, de tal manera que: nP1 Op P3 •> nP2 Op P4

Calculando la métrica PMFV en las arquitecturas P1, P2, P3 y P4, donde n = 2, se tiene la siguiente observación formal:

- PMFV(P1) > PMFV(P2) => (2)PMFV(P1) + PMFV(P3) > (2)PMFV(P2) + PMFV(P4)

Calculando:

- 1 > 0 => (2)1 + 0 > (2)0 + 1
- 1 > 0 => 2 > 1

Las magnitudes guardan una razón, por lo tanto, la métrica PMFV cumple con el axioma de Arquímedes, ya que existe una razón entre las magnitudes.

Homomorfismo de escala ordinal

El Homomorfismo P1 •≥ P2 ⇔ PMFV(P1) ≥ PMFV(P2).

Observando empíricamente:

- P1 •≥ P4 •≥ P2 •≥ P3 ⇔ PMFV(P1) ≥ PMFV(P4) ≥ PMFV(P2) ≥ PMFV(P3).

Calculando la métrica PMFV en las arquitecturas P1, P2, P3 y P4, se obtiene la siguiente observación formal:

- 1 ≥ 1 ≥ 0 ≥ 0

Se demuestra que la métrica PMFV es un Homomorfismo ya que al aplicar los valores de la métrica se obtiene el mismo comportamiento.

Homomorfismo de razón

El Homomorfismo PMFV(P1 + P2) = PMFV(P1) + PMFV(P2)

Seria denotado como:

Dadas las arquitecturas P1 y P2, la métrica PMFV con la característica de ser aditiva, la cual dicta que los resultados de las métricas son sumables:

- PMFV($\frac{4}{4} + \frac{0}{4}$) = PMFV($\frac{4}{4}$) + PMFV($\frac{0}{4}$).
- PMFV($\frac{4}{4}$)= PMFV($\frac{4}{4}$) + PMFV($\frac{0}{4}$).
- 1 = 1 + 0.
- 1 = 1.

Se demuestra que la métrica PMFV es un homomorfismo de razón, ya que al aplicar los valores de la métrica y sumarlos se comprueba la propiedad de adición y la propiedad de decidir si operar antes o después de la ejecución de la métrica y que el comportamiento sea el mismo.

Conclusión:

La métrica PMFV cumple con las condiciones:

•≥	es una relación de orden débil (Transitividad y Completitud)
El sistema relacional (P, •≥, Op)	es una estructura cerrada extensa
P1 •≥ P2 ⇔ PMFV(P1) ≥ PMFV(P2)	se cumple el Homomorfismo de escala ordinal
PMFV(P1 + P2) = PMFV(P1) + PMFV(P2)	se cumple el Homomorfismo de razón

Por lo tanto, se concluye que la métrica PMFV es de razón.

Sustento Teórico de la Métrica PMFI (Protección Modular de Funciones Invariantes) como Escala de Razón

Sea el sistema relacional empírico **A** = (P, R, Op), donde:

P	es una arquitectura de software a ser medida la protección en sus funciones variantes
R	es la relación •≥ (“igual o más protegido que”) de funciones invariantes sobre P
Op	es una operación binaria cerrada sobre los objetos empíricos P

Sea el sistema relacional formal **B** = (Q, S, Ope), donde:

Q	arquitectura con un conjunto de funciones invariantes
S	es la relación ≥ (“igual o más protegido que”) sobre Q
Ope	es la operación binaria cerrada de adición (+) sobre los objetos Q

La métrica PMFI:

Sea la métrica μ = PMFI un mapeo μA → B, que para cada objeto empírico P ∈ A, produce un valor formal medido μ(P) ∈ B.

Entonces, la métrica PMFI será clasificada como escala de razón si:

•≥	es una relación de orden débil (Transitividad y Completitud)
El sistema relacional empírico (P, •≥, Op)	es una estructura extensa
P1 •≥ P2 ⇔ PMFI(P1) ≥ PMFI(P2)	se cumple el Homomorfismo de escala ordinal
PMFI(P1 + P2) = PMFI(P1) + PMFI(P2)	se cumple el Homomorfismo de razón

Aplicando la métrica para la obtención del grado de protección modular de funciones invariantes del apartado 4.3 para efectuar el cálculo de la métrica PMFI sobre las arquitecturas presentadas en las figuras P1 a P4. En estas arquitecturas se tiene identificado el patrón de diseño "*Template Method*" y su función invariante ("calcular (L : ArrayList) : String") marcada con el color azul celeste:

$$\text{PMFI(P1)} = \frac{\sum_{ci=1}^{ci=n}(\sum_{fi=0}^{fi=m} FIP)}{FI} = \frac{1}{1} = 1$$

$$\text{PMFI(P2)} = \frac{\sum_{ci=1}^{ci=n}(\sum_{fi=0}^{fi=m} FIP)}{FI} = \frac{1}{1} = 1$$

$$\text{PMFI(P3)} = \frac{\sum_{ci=1}^{ci=n}(\sum_{fi=0}^{fi=m} FIP)}{FI} = \frac{0}{1} = 0$$

$$\text{PMFI(P4)} = \frac{\sum_{ci=1}^{ci=n}(\sum_{fi=0}^{fi=m} FIP)}{FI} = \frac{0}{1} = 0$$

Se comprueba que la relación binaria es de orden débil:

Transitividad:

Dadas las arquitecturas P1, P2 y P3, se tiene la siguiente observación empírica:

- P1 •≥ P2 ^ P2 •≥ P3 => P1 •≥ P3

Utilizando las métricas se obtiene la siguiente observación formal:

- PMFI(P1) ≥ PMFI(P2) ^ PMFI(P2) ≥ PMFI(P3) => PMFI(P1) ≥ PMFI(P3)

La representación numérica es:

- 1 ≥ 1 ^ 1 ≥ 0 => 1 ≥ 0

La arquitectura P1 presenta mayor grado de protección en sus funciones invariantes, ya que la función invariante que contiene está protegida correctamente. La arquitectura P1 está igualmente protegida que la arquitectura P2 y a su vez P2 tiene mayor grado de protección que P3.

Por lo que P1 tiene mayor grado de protección que P3, cumpliéndose que la relación binaria "•≥" es transitiva.

Completitud:

Dadas las arquitecturas P1 y P2, se podría decir que P1 tiene mayor o igual grado de protección modular en sus funciones variantes que P2 o viceversa. Observando empíricamente:

- P1 •≥ P2 | P2 •≥ P1

Calculando la métrica PMFI en las arquitecturas P1 y P2, se obtiene el siguiente resultado formal:

- 1 ≥ 1.

Dado que la arquitectura P1 tiene un valor de 1 en la métrica PMFI y P2 tiene un valor 1, se tiene que:

- PMFI(P1) ≥ PMFI(P2).

Por lo tanto, la métrica PMFI es una relación binaria completa, ya que siempre fue posible determinar el mayor o igual grado de protección de las funciones invariantes en las distintas arquitecturas.

Asociatividad:

Dadas las arquitecturas P1, P2 y P3, se puede decir que la asociación de las arquitecturas para operarlas es posible sin alterar el resultado. Observando empíricamente:

- P1 Op (P2 Op P3) ≈ (P1 Op P2) Op P3

Calculando la métrica PMFI en las arquitecturas P1, P2 y P3, y sumándolas se obtienen los siguientes resultados formales:

- 1 + (1 + 0) ≈ (1 + 1) + 0
- 2 ≈ 2

Dado que los valores de la métrica PMFI aplicada en las arquitecturas pueden asociarse y operarse sin alterar el resultado, se tiene que:

- PMFI(P1) + ((PMFI(P2) + PMFI(P3)) ≈ ((PMFI(P1) + PMFI(P2)) + PMFI(P3)

Por lo tanto, la métrica PMFI cumple con el axioma de asociatividad, ya que siempre fue posible determinar el resultado de la operación binaria cerrada asociando diferentes elementos de la operación.

Conmutatividad:

Dadas las arquitecturas P1 y P2, se puede decir que el orden de las arquitecturas en la operación binaria cerrada no altera el resultado. Observando empíricamente:

- P1 Op P2 ≈ P2 Op P1

Calculando la métrica PMFI en las arquitecturas P1, P2, y sumándolas se obtienen los siguientes resultados:

- 1 + 1 ≈ 1 + 1
- 2 ≈ 2

Dado que el orden de los valores de la métrica PMFI aplicada en las arquitecturas no alteran el resultado, se tiene que:

- PMFI(P1) + PMFI(P2) ≈ PMFI(P2) + PMFI(P1).

Por lo tanto, la métrica PMFI cumple con el axioma de conmutatividad, ya que siempre fue posible determinar el resultado de la operación binaria cerrada alternando los diferentes elementos de la operación.

Monotonicidad:

Dadas las arquitecturas P1, P2 y P3, se pueden extender hechos a partir de una relación binaria. Observando empíricamente:

- P1 •≥ P2 => P1 Op P3 •≥ P2 Op P3

Calculando la métrica PMFI en las arquitecturas P1, P2 y P3, y asumiendo hechos a partir de la comparación de la arquitectura P1 con la P2 se obtienen los siguientes resultados formales:

- 1 ≥ 1 => 1 + 0 ≥ 1 + 0

Dado que los hechos a partir de la relación binaria de orden débil son ciertos, se tiene que:

- PMFI(P1) ≥ PMFI(P2) => PMFI(P1) + PMFI(P3) ≥ PMFI(P2) + PMFI(P3).

Por lo tanto, la métrica PMFI cumple con el axioma de monotonicidad, ya que siempre fue posible determinar los hechos aditivos derivados a partir de la relación binaria de orden débil.

Axioma de Arquímedes:

El axioma dicta que dadas las arquitecturas P1, P2, P3 y P4, guardan una razón si al ser multiplicadas por un número natural n pueden extenderse mutuamente. Observando empíricamente:

- P1 •> P3 => para cada P2, P4, existe un número natural n > 0, de tal manera que: nP1 Op P4 •> nP3 Op P2

Calculando la métrica PMFI en las arquitecturas P1, P2, P3 y P4, donde n = 2, se tiene la siguiente observación formal:

- PMFI(P1) > PMFI(P3) => (2)PMFI(P1) + PMFI(P4) > (2)PMFI(P3) + PMFI(P2)

Calculando:

- 1 > 0 => (2)1 + 0 > (2)0 + 1
- 1 > 0 => 2 > 1

Las magnitudes guardan una razón, por lo tanto, la métrica PMFI cumple con el axioma de Arquímedes, ya que existe una razón entre las magnitudes.

Homomorfismo de escala ordinal

El Homomorfismo P1 •≥ P2 ⇔ PMFI(P1) ≥ PMFI(P2).

Observando empíricamente:

- P1 •≥ P2 •≥ P3 •≥ P4 ⇔ PMFI(P1) ≥ PMFI(P2) ≥ PMFI(P3) ≥ PMFI(P4).

Calculando la métrica PMFI en las arquitecturas P1, P2, P3 y P4, se obtiene la siguiente observación formal:

- 1 ≥ 1 ≥ 0 ≥ 0

Se demuestra que la métrica PMFI es un Homomorfismo ya que al aplicar los valores de la métrica se obtiene el mismo comportamiento.

Homomorfismo de razón

El Homomorfismo PMFI(P1 + P2) = PMFI(P1) + PMFI(P2)

Sería denotado como:

Dadas las arquitecturas P1 y P2, la métrica PMFI con la característica de ser aditiva, la cual dicta que los resultados de las métricas son sumables:

- PMFI($\frac{1}{1} + \frac{1}{1}$) = PMFI($\frac{1}{1}$) + PMFI($\frac{1}{1}$).
- PMFI($\frac{1+1}{1}$)= PMFI($\frac{1}{1}$) + PMFI($\frac{1}{1}$).
- 2 = 1 + 1.
- 2 = 2.

Se demuestra que la métrica PMFI es un homomorfismo de razón, ya que al aplicar los valores de la métrica y sumarlos se comprueba la propiedad de adición y la propiedad de decidir si operar antes o después de la ejecución de la métrica y que el comportamiento sea el mismo.

Conclusión:

La métrica PMFI cumple con las condiciones:

•≥	es una relación de orden débil (Transitividad y Completitud)
El sistema relacional (P, •≥, Op)	es una estructura cerrada extensa
P1 •≥ P2 ⇔ PMFI(P1) ≥ PMFI(P2)	se cumple el Homomorfismo de escala ordinal
PMFI(P1 + P2) = PMFI(P1) + PMFI(P2)	se cumple el Homomorfismo de razón

Por lo tanto, se concluye que la métrica PMFI es de razón.

Sustento Teórico de la Métrica PTTM (Protección Total del Template Method) como Escala de Razón

Sea el sistema relacional empírico **A** = (P, R, Op), donde:

P	es una arquitectura de software a ser medida la protección total de las funciones plantillas asociadas al patrón de diseño "*Template Method*"
R	es la relación •≥ ("igual o más protegido que") de arquitecturas describe que una arquitectura tiene mayor grado de protección en P
Op	es una operación binaria cerrada sobre los objetos empíricos P

Sea el sistema relacional formal **B** = (Q, S, Ope), donde:

Q	es una arquitectura con un conjunto de funciones plantilla asociadas al patrón de diseño "*Template Method*"
S	es la relación ≥ ("igual o más protegido que") sobre Q
Ope	es la operación binaria cerrada de adición (+) sobre los objetos Q

La métrica PTTM:

Sea la métrica μ = PTTM un mapeo μA → B, que para cada objeto empírico P ∈ A, produce un valor formal medido μ(P) ∈ B.

Entonces, la métrica PTTM será clasificada como escala de razón si:

•≥	es una relación de orden débil (Transitividad y Completitud)
El sistema relacional empírico (P, •≥, Op)	es una estructura extensa
P1 •≥ P2 ⇔ PTTM(P1) ≥ PTTM(P2)	se cumple el Homomorfismo de escala ordinal
PTTM(P1 + P2) = PTTM(P1) + PTTM(P2)	se cumple el Homomorfismo de razón

Aplicando la métrica para la obtención del grado de protección total del "*Template Method*" del apartado 4.4 para efectuar el cálculo de la métrica PTTM sobre las arquitecturas presentadas en las figuras P1 a P4. En estas arquitecturas se tiene identificado el patrón de diseño "*Template Method*" junto con sus respectivas funciones plantilla:

$$\mathrm{PTTM(P1)} = \frac{\mathrm{PMMP} + \mathrm{PMFV} + \mathrm{PMFI}}{Nf} = \frac{1+1+1}{3} = 1$$

$$\mathrm{PTTM(P2)} = \frac{\mathrm{PMMP} + \mathrm{PMFV} + \mathrm{PMFI}}{Nf} = \frac{0+0+1}{3} = 0.3333333333$$

$$\mathrm{PTTM(P3)} = \frac{\mathrm{PMMP} + \mathrm{PMFV} + \mathrm{PMFI}}{Nf} = \frac{0+0+0}{3} = 0$$

$$\mathrm{PTTM(P4)} = \frac{\mathrm{PMMP} + \mathrm{PMFV} + \mathrm{PMFI}}{Nf} = \frac{0+1+0}{3} = 0.3333333333$$

Se comprueba que la relación binaria es de orden débil:

Transitividad:

Dadas las arquitecturas P1, P2 y P3, se tiene la siguiente observación empírica:

- P1 •≥ P2 ^ P2 •≥ P3 => P1 •≥ P3

Utilizando las métricas se obtiene la siguiente observación formal:

- PTTM(P1) ≥ PTTM(P2) ^ PTTM(P2) ≥ PTTM(P3) => PTTM(P1) ≥ PTTM(P3)

La representación numérica es:

- 1 ≥ 0.3333 ^ 0.3333 ≥ 0 => 1 ≥ 0

La arquitectura P1 presenta mayor grado de protección en sus funciones plantillas, ya que las funciones plantilla que contiene están protegidas correctamente. La arquitectura P1 es más protegida que la arquitectura P2 y a su vez P2 tiene mayor grado de protección que P3.

Por lo que P1 tiene mayor grado de protección que P3, cumpliéndose que la relación binaria "•≥" es transitiva.

Completitud:

Dadas las arquitecturas P1 y P2, se podría decir que P1 tiene mayor o igual grado de protección modular en sus funciones variantes que P2 o viceversa. Observando empíricamente:

- P1 •≥ P2 | P2 •≥ P1

Calculando la métrica PTTM en las arquitecturas P1 y P2, se obtiene el siguiente resultado formal:

- 1 ≥ 0.3333.

Dado que la arquitectura P1 tiene un valor de 1 en la métrica PTTM y P2 tiene un valor 0.3333, se tiene que:

- PTTM(P1) ≥ PTTM(P2).

Por lo tanto, la métrica PTTM es una relación binaria completa, ya que siempre fue posible determinar el mayor o igual grado de protección de las funciones invariantes en las distintas arquitecturas.

Asociatividad:

Dadas las arquitecturas P1, P2 y P3, se puede decir que la asociación de las arquitecturas para operarlas es posible sin alterar el resultado. Observando empíricamente:

- P1 Op (P2 Op P3) ≈ (P1 Op P2) Op P3

Calculando la métrica PTTM en las arquitecturas P1, P2 y P3, y sumándolas se obtienen los siguientes resultados formales:

- 1 + (0.3333 + 0) ≈ (1 + 0.3333) + 0
- 1.3333 ≈ 1.3333

Dado que los valores de la métrica PTTM aplicada en las arquitecturas pueden asociarse y operarse sin alterar el resultado, se tiene que:

- PTTM(P1) + ((PTTM(P2) + PTTM(P3)) ≈ ((PTTM(P1) + PTTM(P2)) + PTTM(P3)

Por lo tanto, la métrica PTTM cumple con el axioma de asociatividad, ya que siempre fue posible determinar el resultado de la operación binaria cerrada asociando diferentes elementos de la operación.

Conmutatividad:

Dadas las arquitecturas P1 y P2, se puede decir que el orden de las arquitecturas en la operación binaria cerrada no altera el resultado. Observando empíricamente:

- P1 Op P2 ≈ P2 Op P1

Calculando la métrica PTTM en las arquitecturas P1, P2, y sumándolas se obtienen los siguientes resultados:

- 1 + 0.3333 ≈ 0.3333 + 1
- 1.3333 ≈ 1.3333

Dado que el orden de los valores de la métrica PTTM aplicada en las arquitecturas no alteran el resultado, se tiene que:

- PTTM(P1) + PTTM(P2) ≈ PTTM(P2) + PTTM(P1).

Por lo tanto, la métrica PTTM cumple con el axioma de conmutatividad, ya que siempre fue posible determinar el resultado de la operación binaria cerrada alternando los diferentes elementos de la operación.

Monotonicidad:

Dadas las arquitecturas P1, P2 y P3, se pueden extender hechos a partir de una relación binaria. Observando empíricamente:

- P1 •≥ P2 => P1 Op P3 •≥ P2 Op P3

Calculando la métrica PTTM en las arquitecturas P1, P2 y P3, y asumiendo hechos a partir de la comparación de la arquitectura P1 con la P2 se obtienen los siguientes resultados formales:

- 1 ≥ 0.3333 => 1 + 0 ≥ 0.3333 + 0

Dado que los hechos a partir de la relación binaria de orden débil son ciertos, se tiene que:

- PTTM(P1) ≥ PTTM(P2) => PTTM(P1) + PTTM(P3) ≥ PTTM(P2) + PTTM(P3).

Por lo tanto, la métrica PTTM cumple con el axioma de monotonicidad, ya que siempre fue posible determinar los hechos aditivos derivados a partir de la relación binaria de orden débil.

Axioma de Arquímedes:

El axioma dicta que dadas las arquitecturas P1, P2, P3 y P4, guardan una razón si al ser multiplicadas por un número natural n pueden extenderse mutuamente. Observando empíricamente:

- P1 •> P3 => para cada P2, P4, existe un número natural n > 0, de tal manera que: nP1 Op P4 •> nP3 Op P2

Calculando la métrica PTTM en las arquitecturas P1, P2, P3 y P4, donde n = 2, se tiene la siguiente observación formal:

- PTTM(P1) > PTTM(P3) => (2)PTTM(P1) + PTTM(P4) > (2)PTTM(P3) + PTTM(P2)

Calculando:

- 1 > 0 => (2)1 + 0.3333 > (2)0 + 0.3333
- 1 > 0 => 2.3333 > 0.3333

Las magnitudes guardan una razón, por lo tanto, la métrica PTTM cumple con el axioma de Arquímedes, ya que existe una razón entre las magnitudes.

Homomorfismo de escala ordinal

El Homomorfismo P1 •≥ P2 ⇔ PTTM(P1) ≥ PTTM(P2).

Observando empíricamente:

- P1 •≥ P2 •≥ P4 •≥ P3 ⇔ PTTM(P1) ≥ PTTM(P2) ≥ PTTM(P4) ≥ PTTM(P3).

Calculando la métrica PTTM en las arquitecturas P1, P2, P3 y P4, se obtiene la siguiente observación formal:

- 1 ≥ 0.3333 ≥ 0.3333 ≥ 0

Se demuestra que la métrica PTTM es un Homomorfismo ya que al aplicar los valores de la métrica se obtiene el mismo comportamiento.

Homomorfismo de razón

El Homomorfismo PTTM(P1 + P2) = PTTM(P1) + PTTM(P2)

Sería denotado como:

Dadas las arquitecturas P1 y P2, la métrica PTTM con la característica de ser aditiva, la cual dicta que los resultados de las métricas son sumables:

- PTTM($\frac{1+1+1}{3} + \frac{0+0+1}{3}$) = PTTM($\frac{1+1+1}{3}$) + PTTM($\frac{0+0+1}{3}$).
- PTTM($\frac{3}{3} + \frac{1}{3}$)= PTTM($\frac{3}{3}$) + PTTM($\frac{1}{3}$).
- 1.3333 = 1 + 0.3333.
- 1.3333 = 1.3333.

Se demuestra que la métrica PTTM es un homomorfismo de razón, ya que al aplicar los valores de la métrica y sumarlos se comprueba la propiedad de adición y la propiedad de decidir si operar antes o después de la ejecución de la métrica y que el comportamiento sea el mismo.

Conclusión:

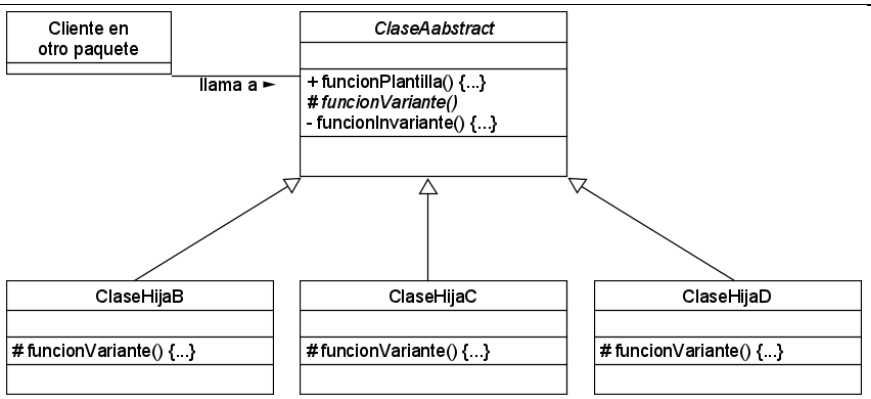
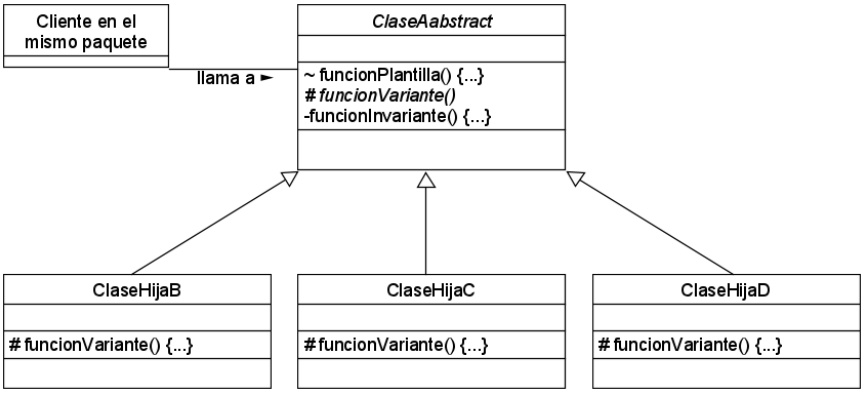
La métrica PTTM cumple con las condiciones:

•≥	es una relación de orden débil (Transitividad y Completitud)
El sistema relacional (P, •≥, Op)	es una estructura cerrada extensa
P1 •≥ P2 ⇔ PTTM(P1) ≥ PTTM(P2)	se cumple el Homomorfismo de escala ordinal
PTTM(P1 + P2) = PTTM(P1) + PTTM(P2)	se cumple el Homomorfismo de razón

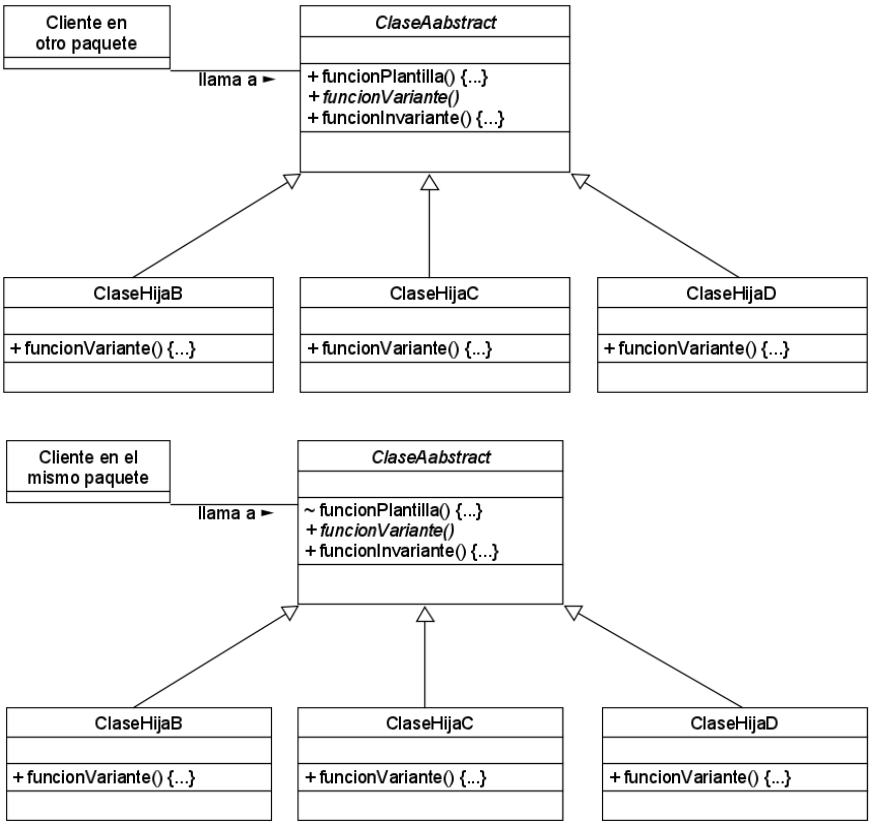
Por lo tanto, se concluye que la métrica PTTM es de razón.

Anexo B.- Plantillas de Casos de Uso

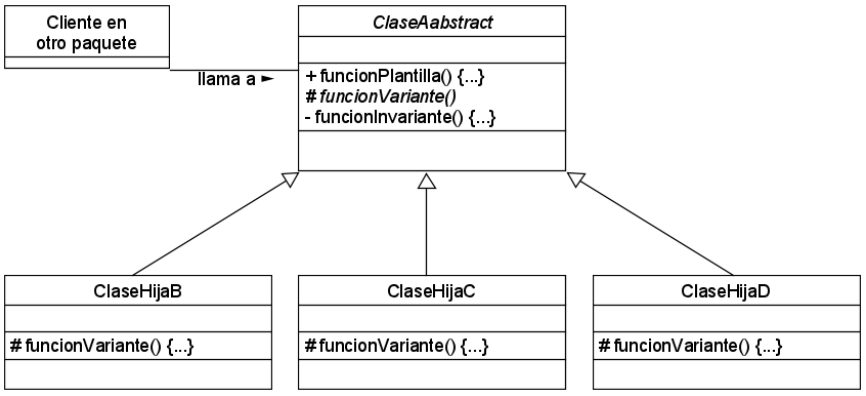
Nombre de caso de Uso:	CU.1 Refactorizar código.
Descripción:	El sistema realiza la refactorización de código legado para aumentar la protección de las funciones plantillas, asociadas al patrón de diseño "*Template Method*".
Actor primario:	Cliente, Marco de métricas, Estructura contenedora de información y Analizador sintáctico.
Precondiciones:	• El código del archivo debe ser escrito en el lenguaje de programación Java. • El código fuente debe compilar sin errores.
Escenario de éxito principal:	1. El código se somete al analizador sintáctico y el analizador sintáctico identifica las clases y métodos al extraer la información. 2. La información obtenida del código analizado se almacena en una estructura contenedora de información (Lista de objetos) y se calculan las métricas. 3. El marco de métricas detecta que el código de entrada cuenta con una buena arquitectura donde la clase que implementa el patrón de diseño "*Template Method*" es una clase base o una clase derivada, el cliente se encuentra en otro paquete o en el mismo paquete donde accede a la clase que implementa el patrón de diseño "*Template Method*" y la arquitectura cuenta con la protección correcta de acuerdo a la documentación del patrón de diseño "*Template Method*".

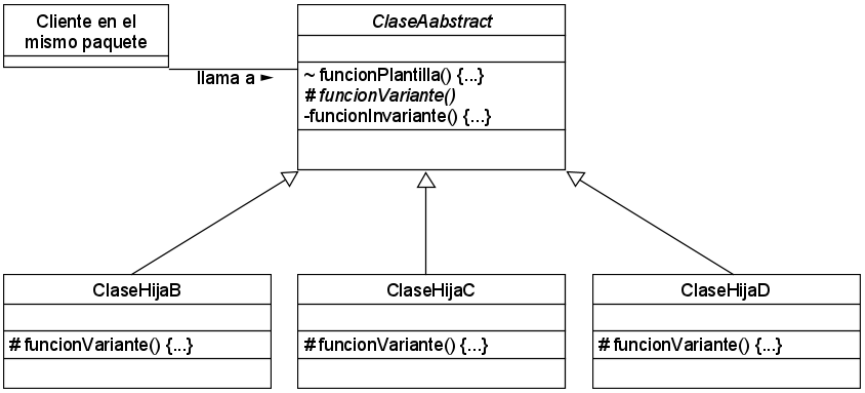
	  4. No se requiere refactorización. 5. Termina el caso de uso.
Escenario alternativo 0:	1. El código se somete al analizador sintáctico y el analizador sintáctico identifica las clases y métodos al extraer la información. 2. La información obtenida del código analizado se almacena en una estructura contenedora de información (Lista de objetos) y se calculan las métricas. 3. Durante el cálculo de las métricas, no se identifican funciones plantillas, funciones variantes y funciones invariantes, se concluye que no es una arquitectura que implementa el patrón de diseño *"Template Method"*.

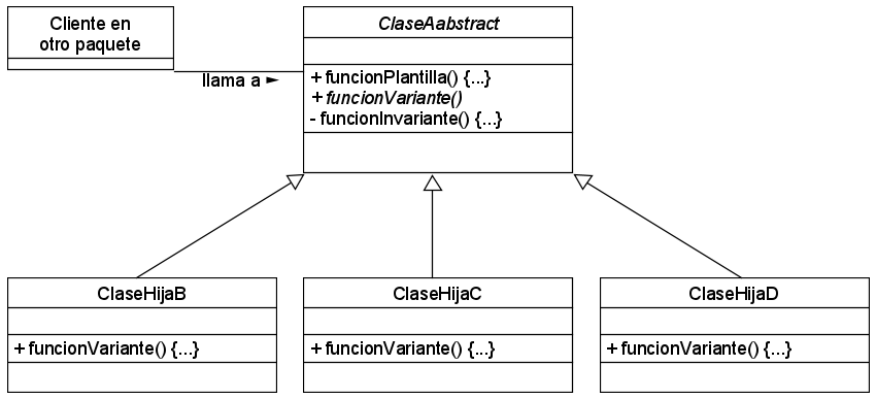
	4. No se ejecuta el método de refactorización. 5. Termina el caso de uso.
Escenario alternativo 1:	1. El código se somete al analizador sintáctico y el analizador sintáctico identifica las clases y métodos al extraer la información. 2. La información obtenida del código analizado se almacena en una estructura contenedora de información (Lista de objetos) y se calculan las métricas. 3. El sistema detecta que el código de entrada cuenta con una arquitectura donde la clase que implementa el patrón de diseño "*Template Method*" es una clase base o una clase derivada, el cliente se encuentra en el mismo paquete o en otro paquete donde accede a la clase que implementa el patrón de diseño "*Template Method*", pero presenta una mala protección en las funciones de una o varias clases que implementan el patrón de diseño "*Template Method*" junto con las clases derivadas.

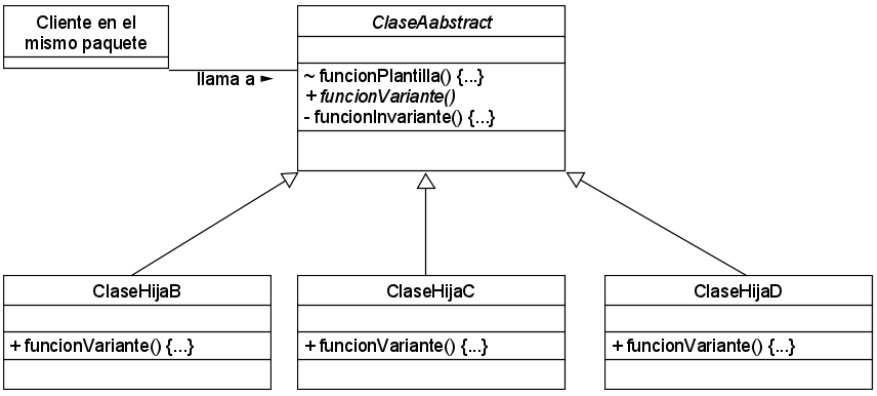


4. Las clases que implementan el "*Template Method*" son refactorizadas junto con las funciones invariantes y las clases derivadas que implementan las funciones variantes.

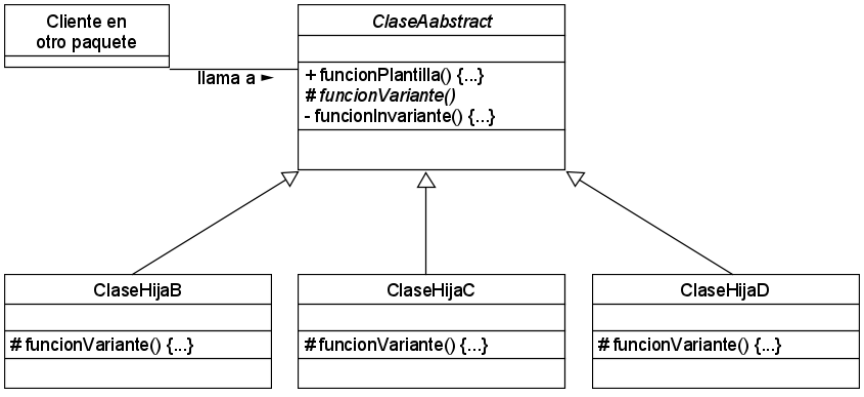


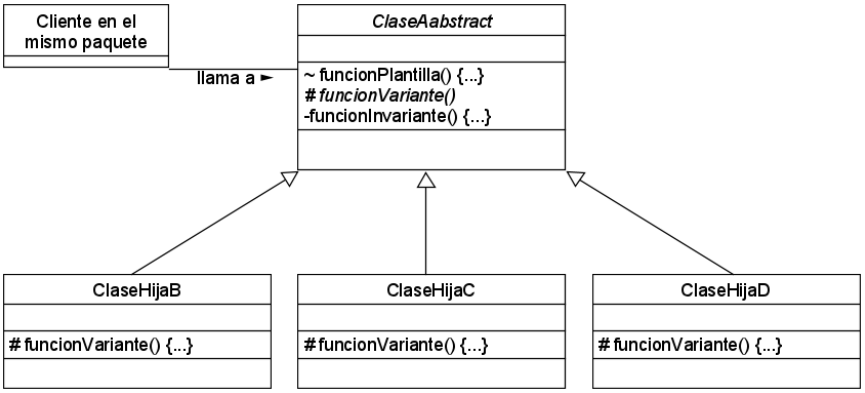
	 5. Se genera el código refactorizado. 6. Se aplican las métricas nuevamente para calcular el grado de protección modular en las clases que implementan el patrón de diseño "*Template Method*". 7. Termina el caso de uso.
Escenario alternativo 2:	1. El código se somete al analizador sintáctico y el analizador sintáctico identifica las clases y métodos al extraer la información. 2. La información obtenida del código analizado se almacena en una estructura contenedora de información (Lista de objetos) y se calculan las métricas. 3. El sistema detecta que el código de entrada cuenta con una arquitectura donde la clase que implementa el patrón de diseño es una clase base o una clase derivada, el cliente se encuentra en el mismo paquete o en otro paquete donde accede a la clase que implementa el patrón de diseño "*Template Method*", pero presenta una mala protección en las funciones variantes de una o varias clases que implementan el patrón de diseño "*Template Method*" y por ende esta mala protección se presenta junto con las clases derivadas.

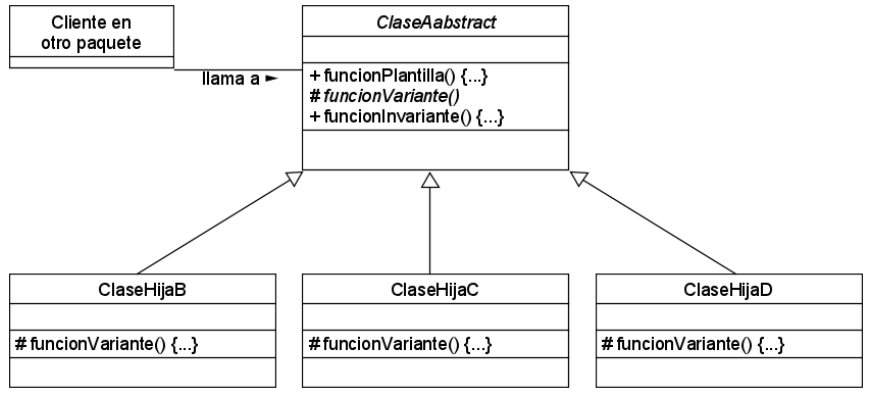


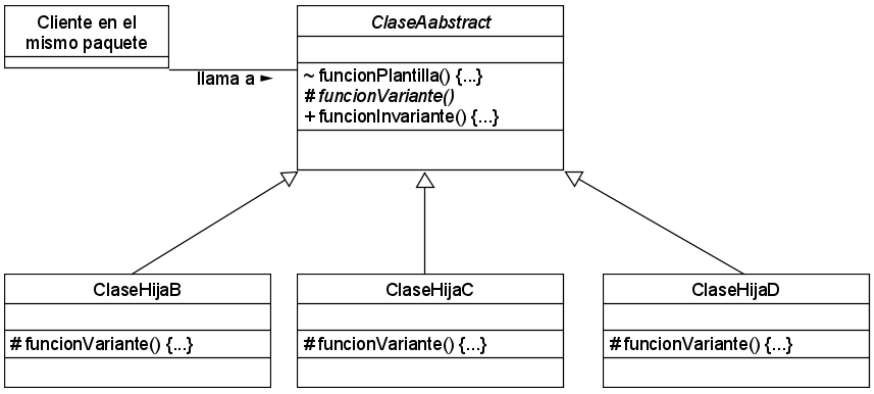


4. Las clases que implementan el "*Template Method*" son refactorizadas junto con las clases derivadas que implementan las funciones variantes.

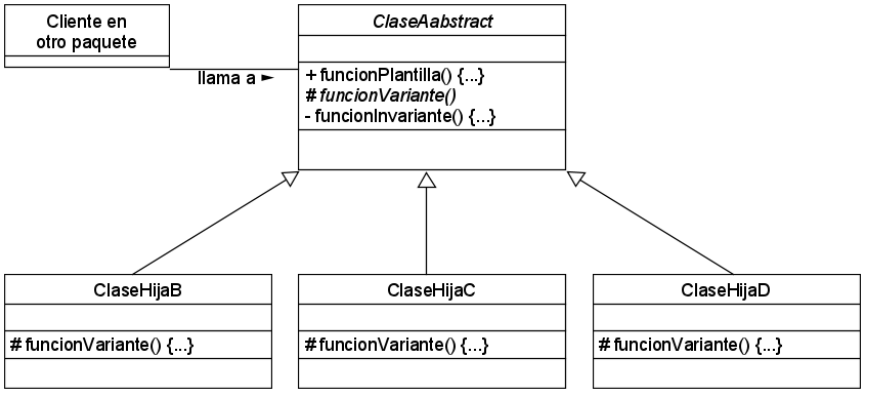


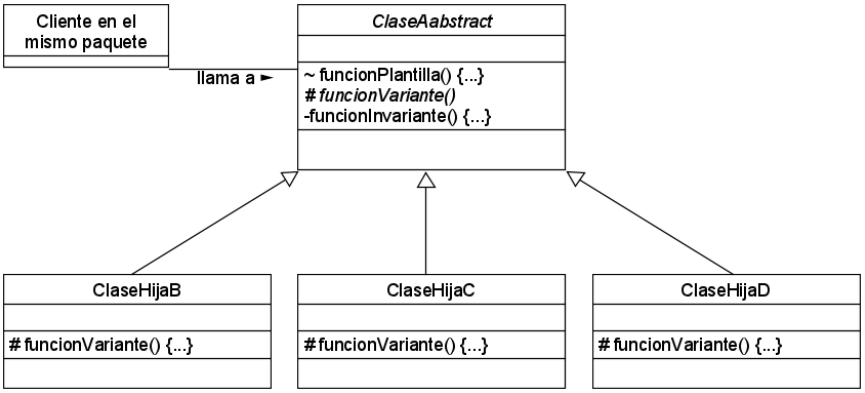
	 5. Se genera el código refactorizado. 6. Se aplican las métricas nuevamente para calcular el grado de protección modular en las clases que implementan el patrón de diseño "*Template Method*". 7. Termina el caso de uso.
Escenario alternativo 3:	1. El código se somete al analizador sintáctico y el analizador sintáctico identifica las clases y métodos al extraer la información. 2. La información obtenida del código analizado se almacena en una estructura contenedora de información (Lista de objetos) y se calculan las métricas. 3. El sistema detecta que el código de entrada cuenta con una arquitectura donde la clase que implementa el patrón de diseño es una clase base o una clase derivada, el cliente se encuentra en el mismo paquete o en otro paquete donde accede a la clase que implementa el patrón de diseño "*Template Method*", pero presenta una mala protección en las funciones invariantes de una o varias clases que implementan el patrón de diseño "*Template Method*".

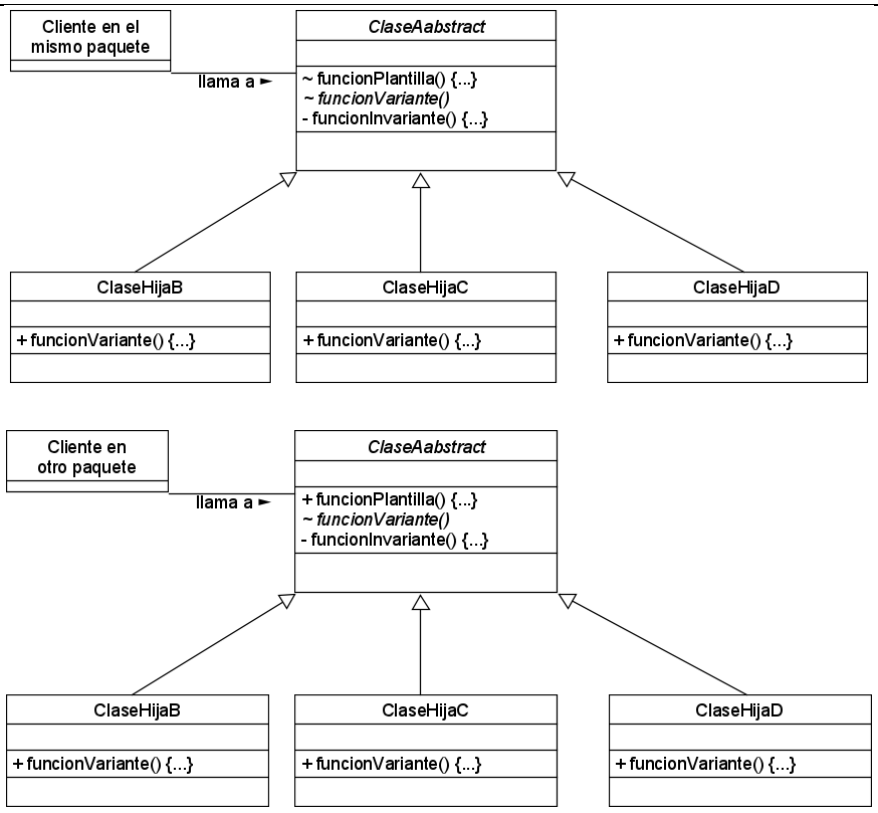




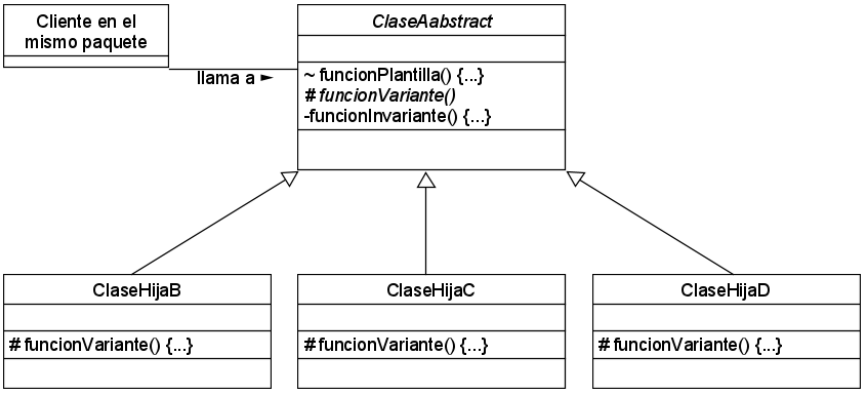
4. Las clases que implementan el "*Template Method*" son refactorizadas junto con las funciones invariantes.

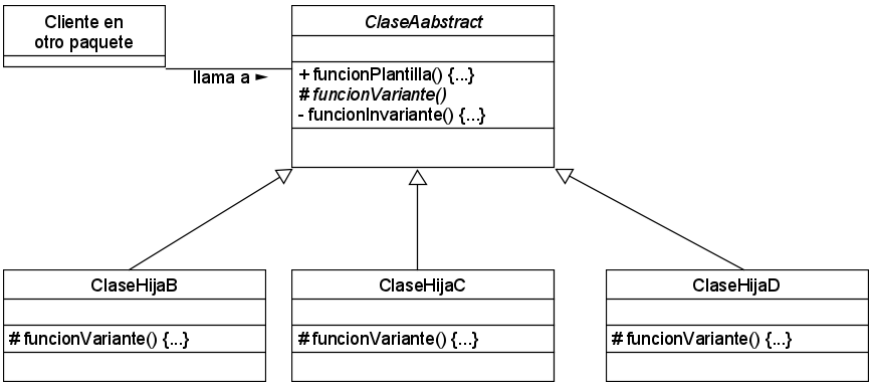


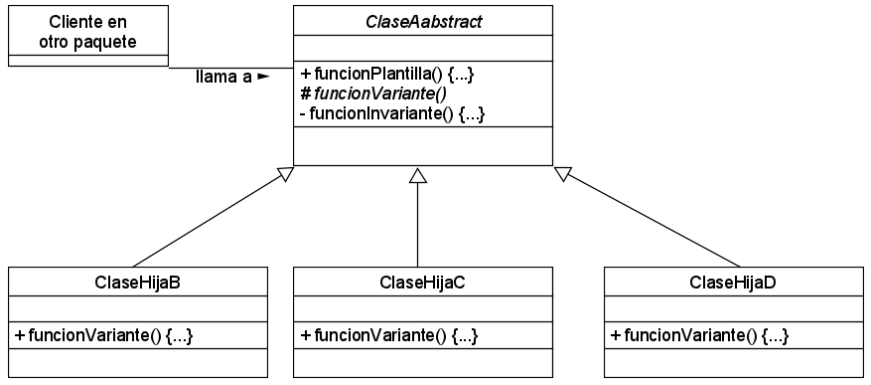
	 5. Se genera el código refactorizado. 6. Se aplican las métricas nuevamente para calcular el grado de protección modular en las clases que implementan el patrón de diseño "*Template Method*". 7. Termina el caso de uso.
Escenario alternativo 4:	1. El código se somete al analizador sintáctico y el analizador sintáctico identifica las clases y métodos al extraer la información. 2. La información obtenida del código analizado se almacena en una estructura contenedora de información (Lista de objetos) y se calculan las métricas. 3. El sistema detecta que el código de entrada cuenta con una arquitectura donde la clase que implementa el patrón de diseño es una clase base o una clase derivada, el cliente se encuentra en el mismo paquete donde accede a la clase que implementa el patrón de diseño "*Template Method*", pero presenta una mala protección en las funciones variantes y las clases derivadas, esta mala protección se presenta en una o varias clases que implementan el patrón de diseño "*Template Method*".

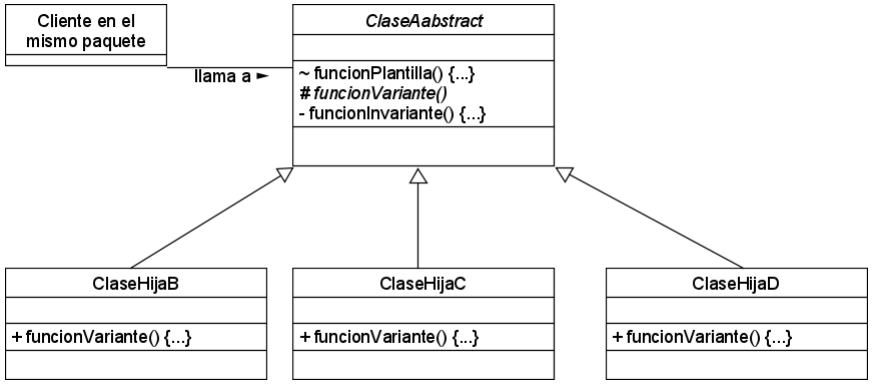


4. Las clases que implementan el "*Template Method*" son refactorizadas junto con las funciones variantes y junto con las clases derivadas que implementan las funciones variantes.

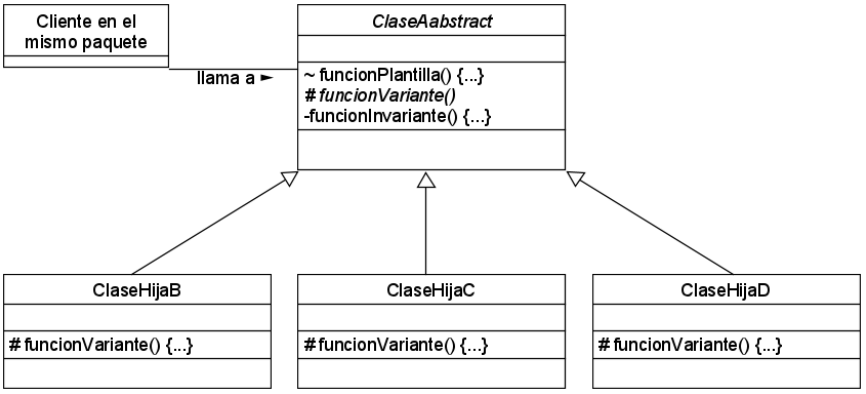


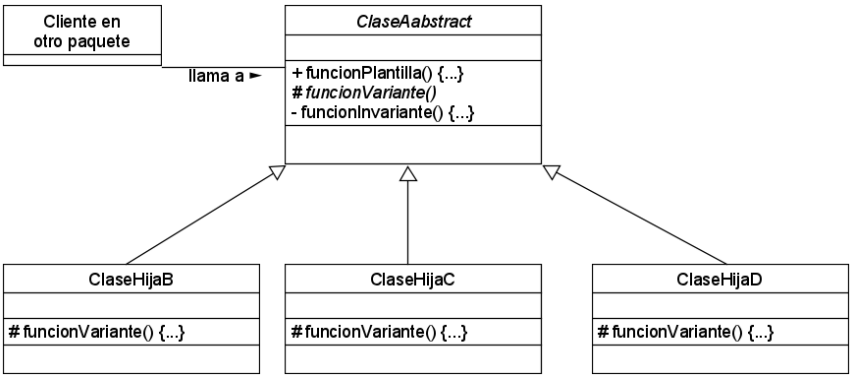
	 5. Se genera el código refactorizado. 6. Se aplican las métricas nuevamente para calcular el grado de protección modular en las clases que implementan el patrón de diseño "*Template Method*". 7. Termina el caso de uso.
Escenario alternativo 5:	1. El código se somete al analizador sintáctico y el analizador sintáctico identifica las clases y métodos al extraer la información. 2. La información obtenida del código analizado se almacena en una estructura contenedora de información (Lista de objetos) y se calculan las métricas. 3. El sistema detecta que el código de entrada cuenta con una arquitectura donde la clase que implementa el patrón de diseño es una clase base o una clase derivada, el cliente se encuentra en el mismo paquete o en otro paquete donde accede a la clase que implementa el patrón de diseño "*Template Method*", pero presenta una mala protección en las funciones variantes y las clases derivadas, esta mala protección se presenta en una o varias clases que implementan el patrón de diseño "*Template Method*".

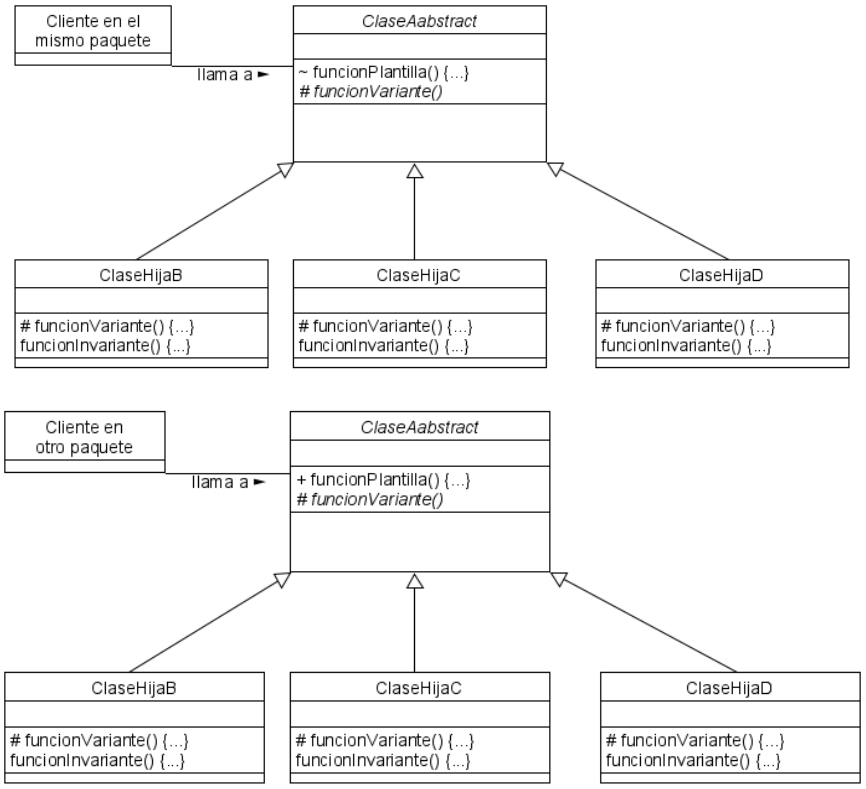




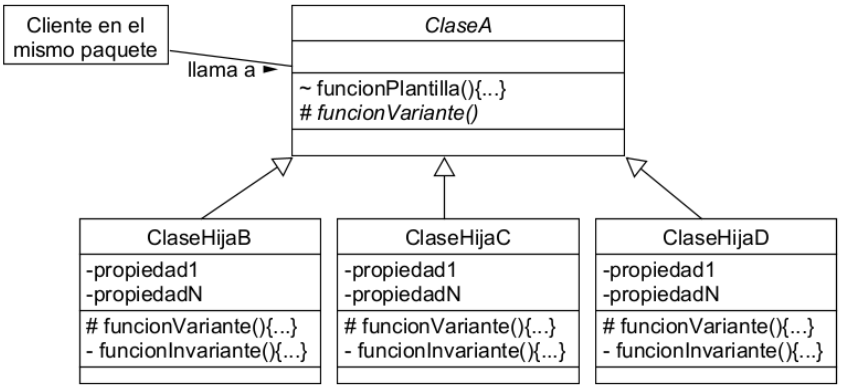
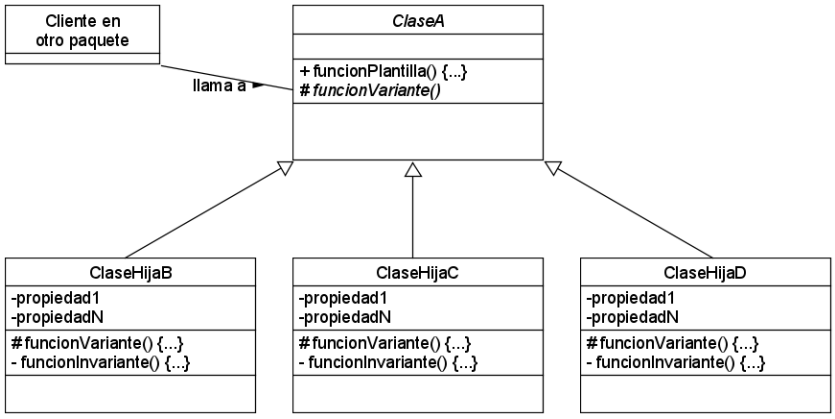
4. Las clases que implementan el "*Template Method*" son refactorizadas junto con las funciones variantes y junto con las clases derivadas que implementan las funciones variantes.

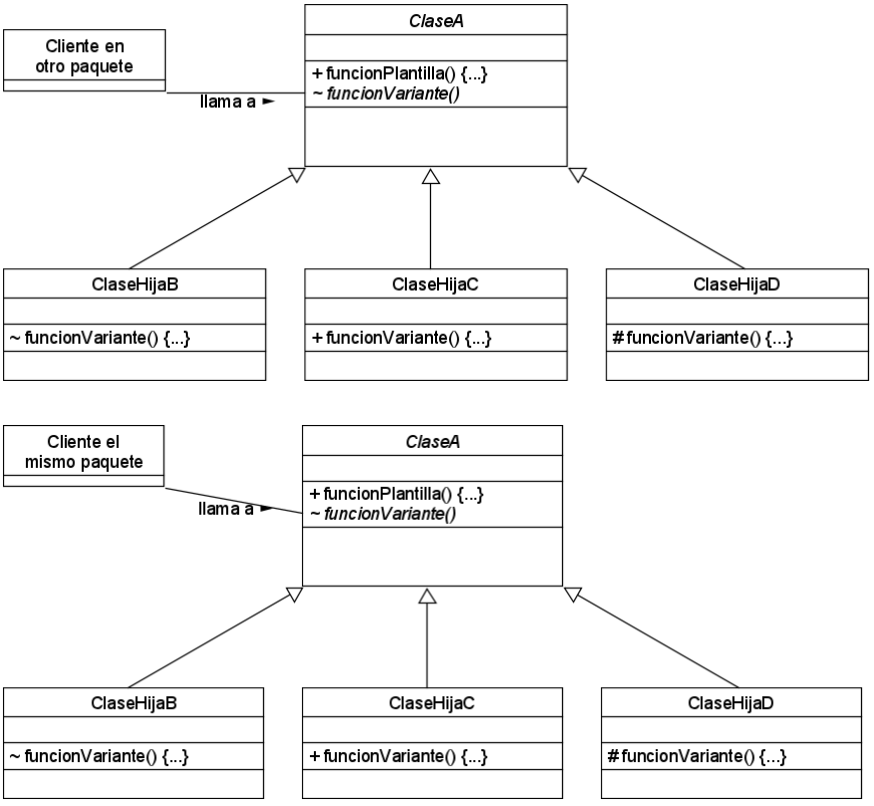


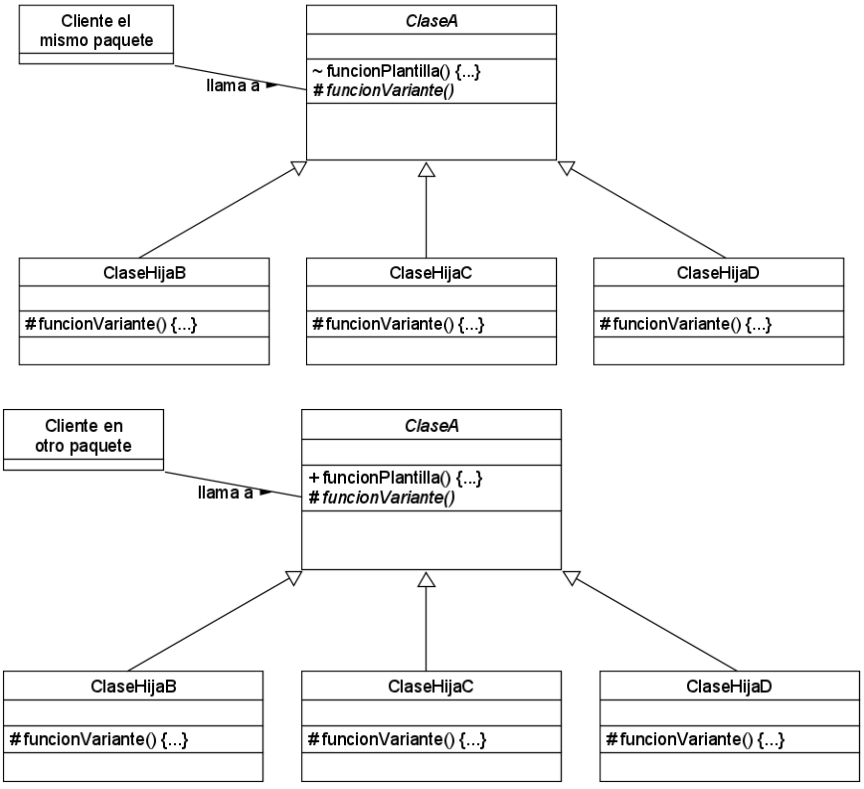
	 5. Se genera el código refactorizado. 6. Se aplican las métricas nuevamente para calcular el grado de protección modular en las clases que implementan el patrón de diseño "*Template Method*". 7. Termina el caso de uso.
Escenario alternativo 6:	1. El código se somete al analizador sintáctico y el analizador sintáctico identifica las clases y métodos al extraer la información. 2. La información obtenida del código analizado se almacena en una estructura contenedora de información (Lista de objetos) y se calculan las métricas. 3. El sistema detecta que el código de entrada cuenta con una arquitectura donde la clase que implementa el patrón de diseño es una clase base o una clase derivada, el cliente se encuentra en el mismo paquete o en otro paquete donde accede a la clase que implementa el patrón de diseño "*Template Method*", pero presenta una mala arquitectura en las funciones invariantes duplicando segmentos de código, esta mala arquitectura se presenta en una o varias clases que implementan el patrón de diseño "*Template Method*".

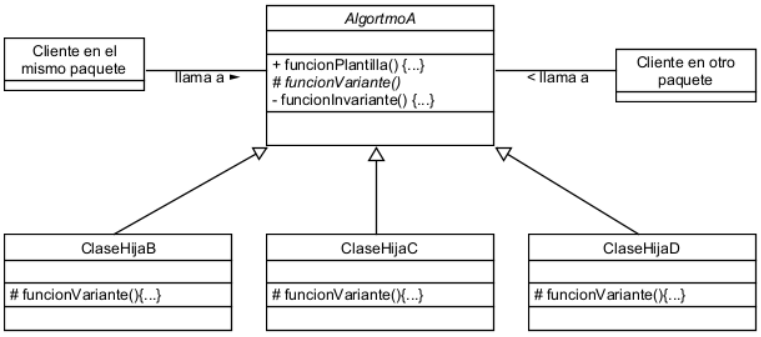
	 4. La arquitectura de las clases que implementan el "*Template Method*" en esta situación, no es posible refactorizar, ya que **la identificación de código duplicado** requiere un estudio más profundo, por lo que este escenario queda propuesto para trabajos futuros. 5. Se genera el código refactorizado de otros escenarios. 6. Se aplican las métricas nuevamente para calcular el grado de protección modular en las clases que implementan el patrón de diseño "*Template Method*". 7. Termina el caso de uso.
Escenario alternativo 7:	1. El código se somete al analizador sintáctico y el analizador sintáctico identifica las clases y métodos al extraer la información.

	2. La información obtenida del código analizado se almacena en una estructura contenedora de información (Lista de objetos) y se calculan las métricas. 3. El sistema detecta que el código de entrada cuenta con una arquitectura donde la clase que implementa el patrón de diseño es una clase base o una clase derivada, el cliente se encuentra en el mismo paquete o en otro paquete donde accede a la clase que implementa el patrón de diseño "*Template Method*", pero presenta una mala arquitectura en las funciones invariantes duplicando segmentos de código en las clases hijas, las propiedades de la clase son usadas en esas funciones, esta mala arquitectura se presenta en una o varias clases que implementan el patrón de diseño "*Template Method*".

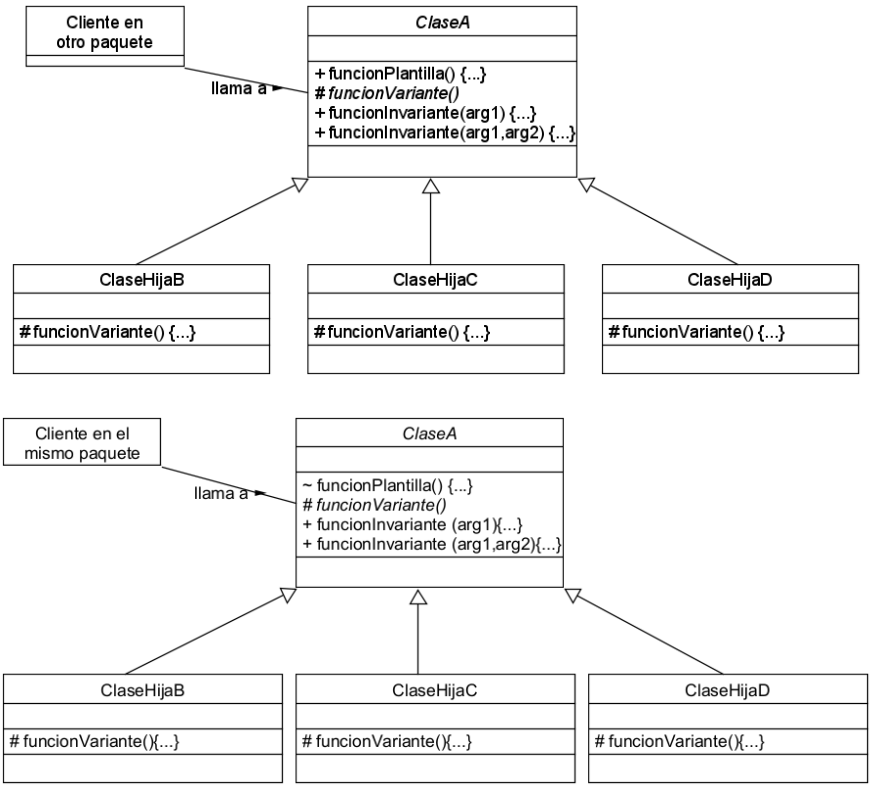
	  4. La arquitectura de las clases que implementan el "*Template Method*" en esta situación, no es posible refactorizar, ya que **la identificación de código duplicado** requiere un estudio más profundo, por lo que este escenario queda propuesto para trabajos futuros. 5. Se genera el código refactorizado de otros escenarios. 6. Se aplican las métricas nuevamente para calcular el grado de protección modular en las clases que implementan el patrón de diseño "*Template Method*". 7. Termina el caso de uso.
Escenario alternativo 8:	1. El código se somete al analizador sintáctico y el analizador sintáctico identifica las clases y métodos al extraer la información.

	2. La información obtenida del código analizado se almacena en una estructura contenedora de información (Lista de objetos) y se calculan las métricas. 3. El sistema detecta que el código de entrada cuenta con una arquitectura donde la clase que implementa el patrón de diseño es una clase base o una clase derivada, el cliente se encuentra en el mismo paquete o en otro paquete donde accede a la clase que implementa el patrón de diseño "*Template Method*", la clase solamente cuenta con funciones variantes y la función plantilla pero presenta una mala protección en las funciones variantes y en las funciones variantes derivadas, esta mala arquitectura se presenta en una o varias clases que implementan el patrón de diseño "*Template Method*".  4. Las clases que implementan el "*Template Method*" son refactorizadas incluidas las funciones variantes junto

	con las clases derivadas que implementan las funciones variantes.  5. Se genera el código refactorizado. 6. Se aplican las métricas nuevamente para calcular el grado de protección modular en las clases que implementan el patrón de diseño "*Template Method*". 7. Termina el caso de uso.
Escenario alternativo 9:	1. El código se somete al analizador sintáctico y el analizador sintáctico identifica las clases y métodos al extraer la información. 2. La información obtenida del código analizado se almacena en una estructura contenedora de información (Lista de objetos) y se calculan las métricas.

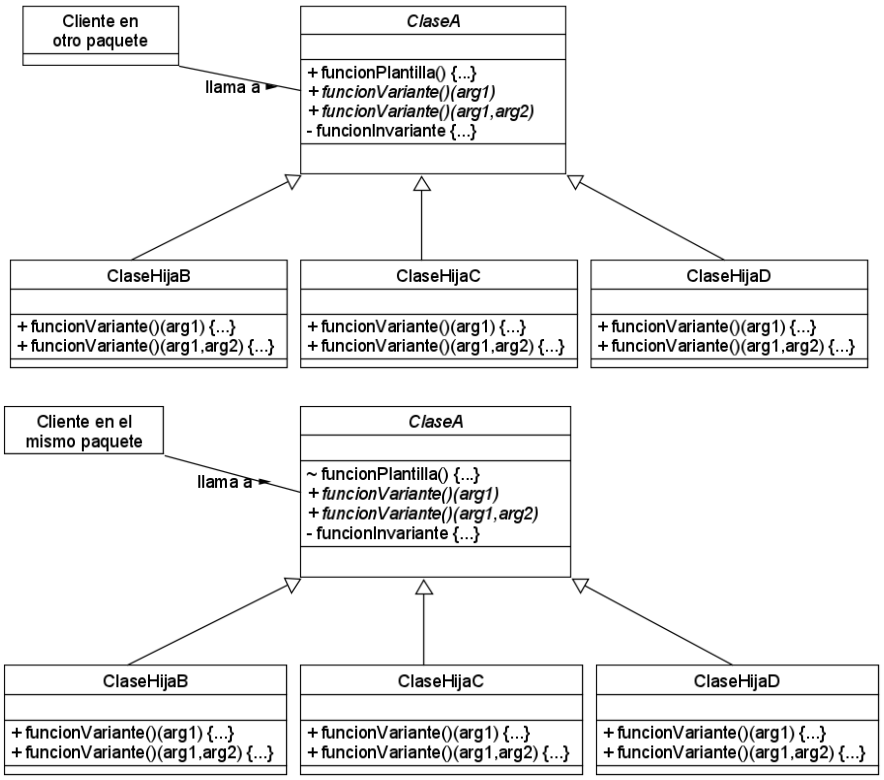
	3. El sistema detecta que el código de entrada cuenta con una arquitectura donde la clase que implementa el patrón de diseño es una clase base o una clase derivada, dos o más clientes que se encuentran en el mismo paquete y en paquetes externos acceden a la clase que implementa el patrón de diseño "*Template Method*", la clase implementa el patrón de diseño "*Template Method*" utiliza un calificador de alcance "public" en la función plantilla.  4. Las clases que implementan el "*Template Method*" no son refactorizadas ya que es necesario el acceso public para las clases que acceden desde otro paquete. 5. Se genera el código refactorizado. 6. Se aplican las métricas nuevamente para calcular el grado de protección modular en las clases que implementan el patrón de diseño "*Template Method*". 7. Termina el caso de uso.
Escenario alternativo 10:	1. El código se somete al analizador sintáctico y el analizador sintáctico identifica las clases y métodos al extraer la información. 2. La información obtenida del código analizado se almacena en una estructura contenedora de información (Lista de objetos) y se calculan las métricas.

3. El sistema detecta que el código de entrada cuenta con una arquitectura donde la clase que implementa el patrón de diseño es una clase base o una clase derivada, el cliente se encuentra en el mismo paquete o en otro paquete donde accede a la clase que implementa el patrón de diseño "*Template Method*", la clase cuenta con funciones variantes, funciones invariantes y método plantilla, pero las funciones invariantes se encuentran sobrecargadas. Se presenta una mala protección en las funciones invariantes sobrecargadas, esta mala arquitectura se presenta en una o varias clases que implementan el patrón de diseño "*Template Method*".

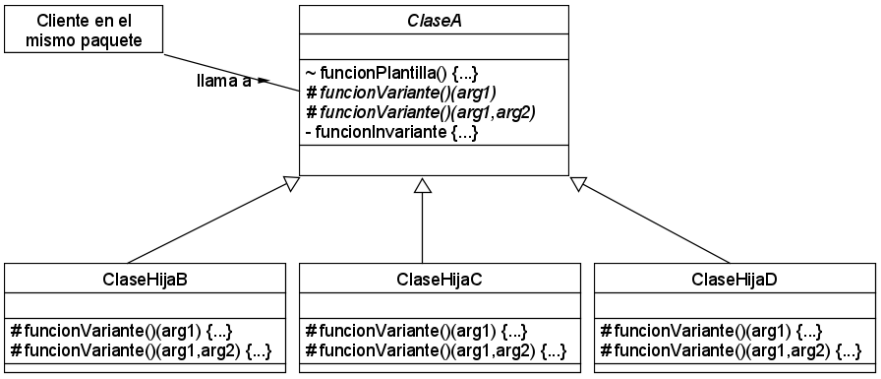


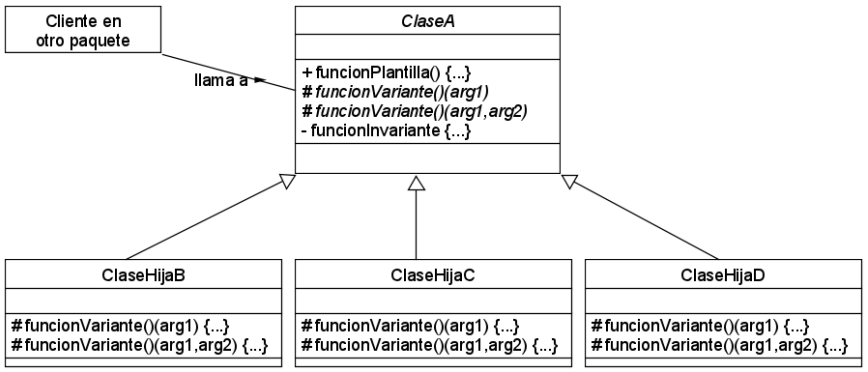
4. Las clases que implementan el "*Template Method*" son refactorizadas sin contar a las funciones invariantes sobrecargadas, esto sucede porque el analizador sintáctico no permite dar seguimiento a la información

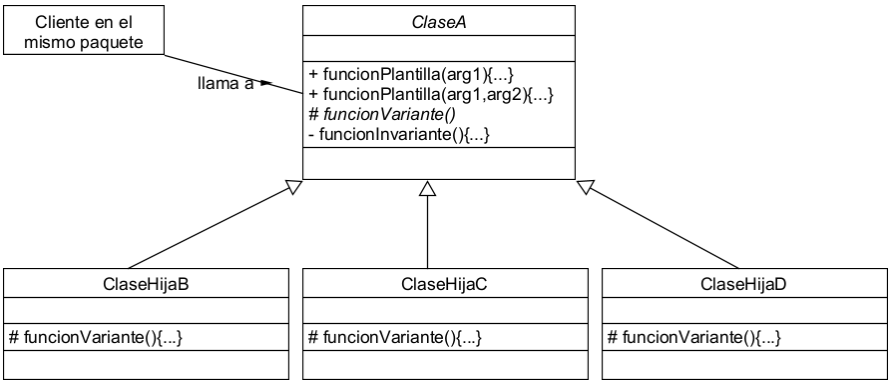
	necesaria de los parámetros de las funciones sobrecargadas. 5. Se genera el código refactorizado. 6. Se aplican las métricas nuevamente para calcular el grado de protección modular en las clases que implementan el patrón de diseño "*Template Method*". 7. Termina el caso de uso.
Escenario alternativo 11:	1. El código se somete al analizador sintáctico y el analizador sintáctico identifica las clases y métodos al extraer la información. 2. La información obtenida del código analizado se almacena en una estructura contenedora de información (Lista de objetos) y se calculan las métricas. 3. El sistema detecta que el código de entrada cuenta con una arquitectura donde la clase que implementa el patrón de diseño es una clase base o una clase derivada, el cliente se encuentra en el mismo paquete o en otro paquete donde accede a la clase que implementa el patrón de diseño "*Template Method*", la clase cuenta con funciones variantes, funciones invariantes y método plantilla, pero las funciones variantes se encuentran sobrecargadas. Se presenta una mala protección en las funciones variantes sobrecargadas, esta mala arquitectura se presenta en una o varias clases que implementan el patrón de diseño "*Template Method*".



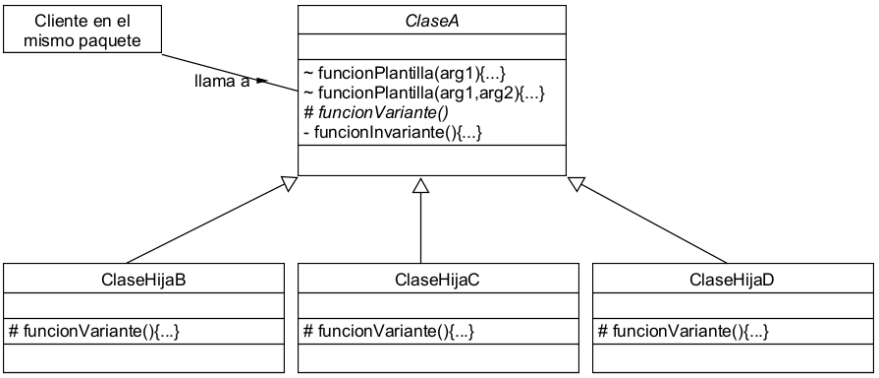
4. Las clases que implementan el "*Template Method*" son refactorizadas junto con las funciones variantes con sobrecarga.



	 5. Se genera el código refactorizado. 6. Se aplican las métricas nuevamente para calcular el grado de protección modular en las clases que implementan el patrón de diseño "*Template Method*". 7. Termina el caso de uso.
Escenario alternativo 12	1. El código se somete al analizador sintáctico y el analizador sintáctico identifica las clases y métodos al extraer la información. 2. La información obtenida del código analizado se almacena en una estructura contenedora de información (Lista de objetos) y se calculan las métricas. 3. El sistema detecta que el código de entrada cuenta con una arquitectura donde la clase que implementa el patrón de diseño es una clase base o una clase derivada, el cliente se encuentra en el mismo paquete donde accede a la clase que implementa el patrón de diseño "*Template Method*", la clase cuenta con funciones variantes, funciones invariantes y método plantilla, pero la función plantilla se encuentran sobrecargado. Se presenta una mala protección en la función plantilla sobrecargado, esta mala arquitectura se presenta en una o varias clases que implementan el patrón de diseño "*Template Method*".

<table>
<tr>
<td></td>
<td>


4. Las clases que implementan el “Template Method” son refactorizadas junto con las funciones variantes con sobrecarga.

5. Se genera el código refactorizado.

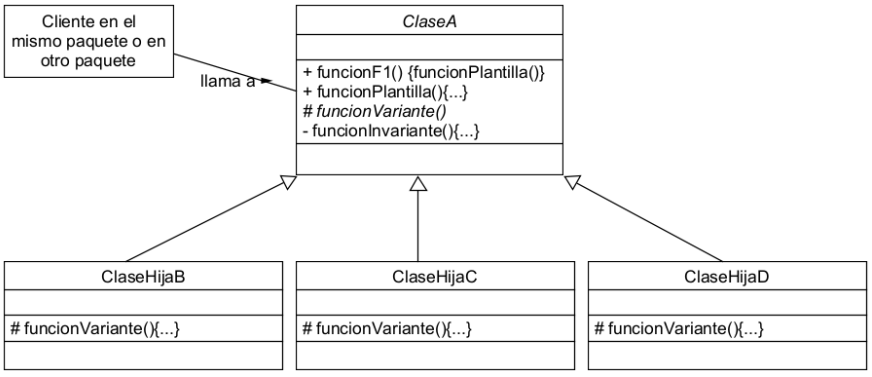
6. Se aplican las métricas nuevamente para calcular el grado de protección modular en las clases que implementan el patrón de diseño “Template Method”.

7. Termina el caso de uso.</td>
</tr>
<tr>
<td>Escenario alternativo 13</td>
<td>1. El código se somete al analizador sintáctico y el analizador sintáctico identifica las clases y métodos al extraer la información.

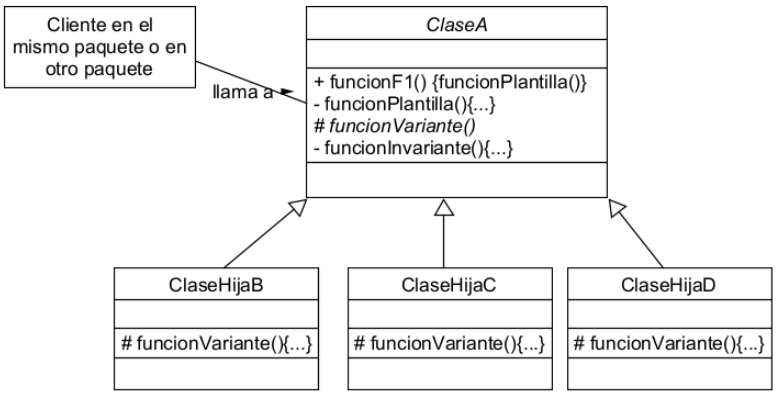
2. La información obtenida del código analizado se almacena en una estructura contenedora de</td>
</tr>
</table>

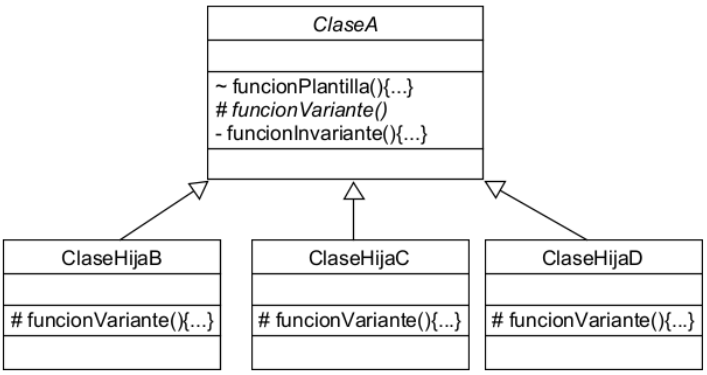
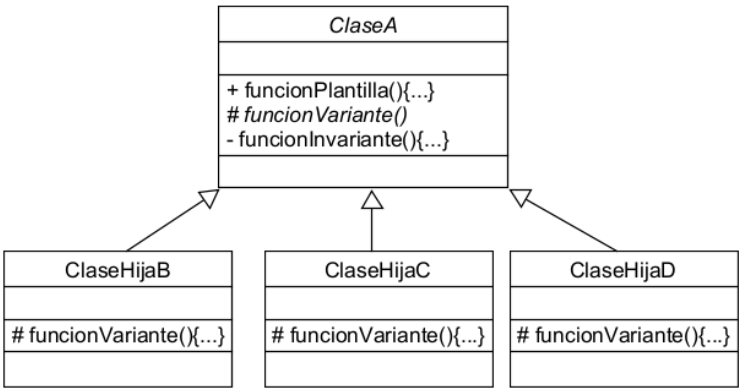
información (Lista de objetos) y se calculan las métricas.

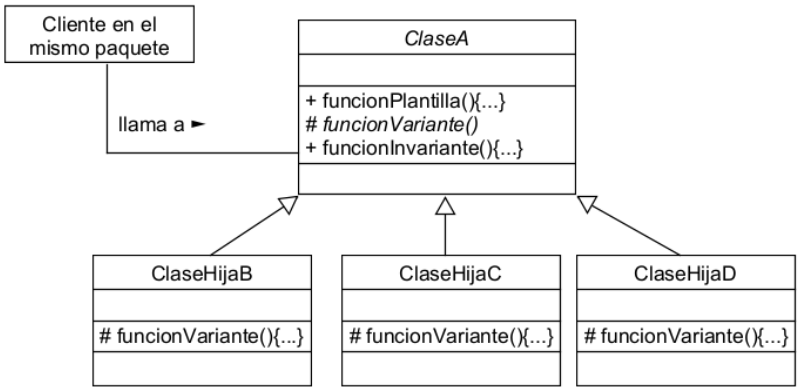
3. El sistema detecta que el código de entrada cuenta con una arquitectura donde la clase que implementa el patrón de diseño es una clase base o una clase derivada, el cliente se encuentra en el mismo paquete o en otro externo, donde accede a la clase que implementa el patrón de diseño "*Template Method*", la clase cuenta con funciones variantes, funciones invariantes y método plantilla, pero la función plantilla se encuentran dentro de otra función la cual es accedida por el cliente. Se presenta una mala protección en la función plantilla, esta mala arquitectura se presenta en una o varias clases que implementan el patrón de diseño "*Template Method*".

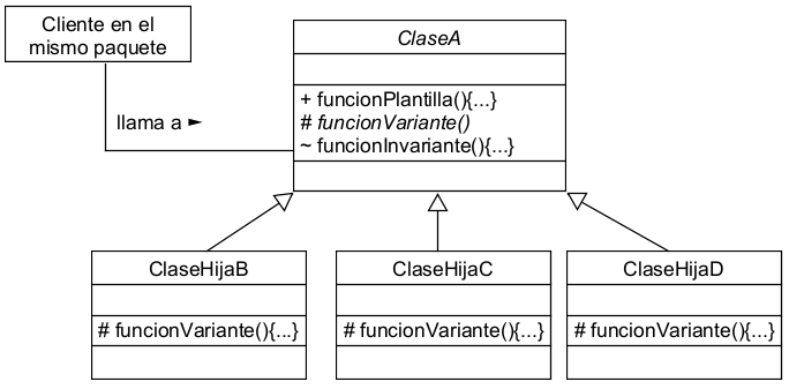



```
objA.funcionF1() → {
    funcionPlantilla() → {
        funcionVariante();
        funcionInvariante()
    }
}
```

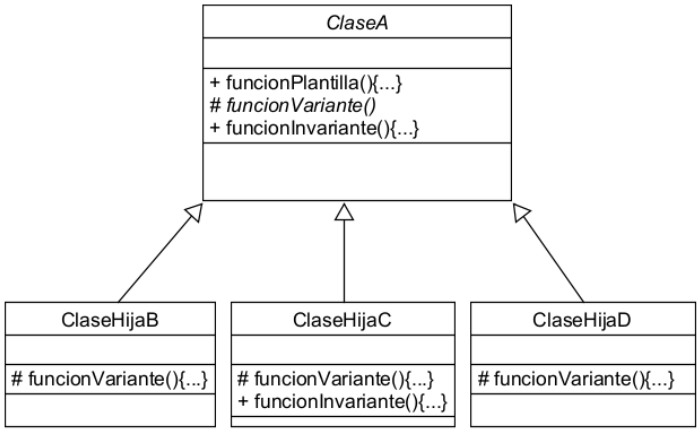
	4. Las clases que implementan el "*Template Method*" son refactorizadas junto con las funciones plantilla almacenados dentro otra función.  5. Se genera el código refactorizado. 6. Se aplican las métricas nuevamente para calcular el grado de protección modular en las clases que implementan el patrón de diseño "*Template Method*". 7. Termina el caso de uso.
Escenario alternativo 14	1. El código se somete al analizador sintáctico y el analizador sintáctico identifica las clases y métodos al extraer la información. 2. La información obtenida del código analizado se almacena en una estructura contenedora de información (Lista de objetos) y se calculan las métricas. 3. El sistema detecta que el código de entrada cuenta con una arquitectura que implementa el patrón de diseño "*Template Method*", la clase cuenta con funciones variantes, funciones invariantes y método plantilla, en donde la función plantilla tiene el

	calificador de alcance friendly, pero la función plantilla no es accedido por ningún cliente.  4. Las clases que implementan el "*Template Method*" son refactorizadas junto con las funciones plantilla que no son accedidos por ningún cliente colocando el calificador de alcance correcto.  5. Se genera el código refactorizado. 6. Se aplican las métricas nuevamente para calcular el grado de protección modular en las clases que implementan el patrón de diseño "*Template Method*". Termina el caso de uso.

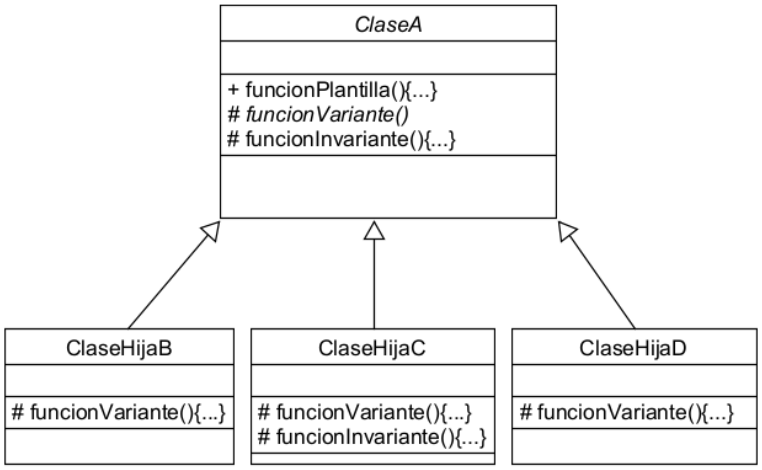
Escenario alternativo 15	1. El código se somete al analizador sintáctico y el analizador sintáctico identifica las clases y métodos al extraer la información. 2. La información obtenida del código analizado se almacena en una estructura contenedora de información (Lista de objetos) y se calculan las métricas. 3. El sistema detecta que el código de entrada cuenta con una arquitectura que implementa el patrón de diseño "*Template Method*", la clase cuenta con funciones variantes, funciones invariantes y método plantilla, en donde la función invariante tiene el calificador de alcance public, pero la función invariante es accedida por otro cliente en el mismo paquete violando el principio de "*única responsabilidad*".  4. A petición del Usuario se recomienda continuar con la refactorización o parar la refactorización y ejecutar el método de refactorización que cubra el principio de "*única responsabilidad*". Si se continúa, las clases que implementan el "*Template Method*" son refactorizadas junto con las funciones invariantes colocando el calificador de alcance correcto dado el alcance del cliente.

	 5. En este escenario no se verá reflejada la protección en la función invariante ya que el patrón de diseño está mal implementado por lo que como hallazgo se recomienda un método de refactorización en un enfoque funcional, este método debe duplicar el paquete y la clase donde se viola el principio de única responsabilidad la cual contendrá al método invariante llamado por otro cliente. 6. Se genera el código refactorizado. 7. Se aplican las métricas nuevamente para calcular el grado de protección modular en las clases que implementan el patrón de diseño "*Template Method*". Termina el caso de uso.
Escenario alternativo 16	1. El código se somete al analizador sintáctico y el analizador sintáctico identifica las clases y métodos al extraer la información. 2. La información obtenida del código analizado se almacena en una estructura contenedora de información (Lista de objetos) y se calculan las métricas. 3. El sistema detecta que el código de entrada cuenta con una arquitectura que implementa el patrón de

diseño "*Template Method*", la clase cuenta con funciones variantes, funciones invariantes y método plantilla, en donde la función invariante, tiene el calificador de alcance public y se vuelve virtual al ser sobrescrita en una o más clases hijas.



4. Las clases que implementan el "*Template Method*" son refactorizadas junto con las funciones invariantes virtuales, las cuales se les dará el tratamiento de una función variante al ser sobrescritas por clases derivadas.



5. Se genera el código refactorizado.

6. Se aplican las métricas nuevamente para calcular el grado de protección modular en las clases que implementan el patrón de diseño "*Template Method*".

	Termina el caso de uso.
Postcondiciones:	• Al final del proceso los archivos del código generado y refactorizado se guardarán en una carpeta (véase el CU.2). • Pueden existir varios de estos escenarios en un sistema a refactorizar, e incluso presentarse escenarios combinatorios, para desarrollo del producto de esta tesis estos son los escenarios que se usarán.
Escenario de fracaso 1:	1. El código se somete al analizador sintáctico y el analizador sintáctico identifica las clases y métodos al extraer la información. 2. La información obtenida del código analizado se almacena en una estructura contenedora de información (Lista de objetos) y se calculan las métricas. 3. El sistema detecta algún escenario no tratable, al no abarcar ese escenario no se refactoriza y continúa buscando otros escenarios.

Nombre de caso de Uso:	CU.2 Generar código
Descripción:	El código refactorizado en una lista generada en el CU.1 pasa a la plantilla “StringTemplate” (librería de ANTLR) para generar el código refactorizado, que es equivalente a la aplicación original.
Actor primario:	Estructura contenedora de información.
Precondiciones:	1) Contar con la lista de objetos refactorizados generada en el CU.1.
Escenario de éxito principal:	1. Se recorre la estructura contenedora de información (Lista de objetos) y se plasma la información de las clases refactorizadas en la plantilla StringTemplate.

	2. A partir de la plantilla StringTemplate se genera el código final refactorizado equivalente a la aplicación original. 3. Los archivos refactorizados se almacenan en una carpeta. 4. Termina el caso de uso.
Postcondiciones:	El código generado es equivalente en funcionalidad al código original
Escenario de fracaso 1:	1. Se recorre la estructura contenedora de información (Lista de objetos) y se plasma la información de las clases refactorizadas en la plantilla StringTemplate. 2. Cuando el usuario selecciona la opción “no refactorizar”, la plantilla no genera el código y termina el proceso de generación de código.

Nombre de caso de Uso:	CU.3 Métrica PMMP
Descripción:	El analizador sintáctico realiza un análisis sobre el código legado proveniente del CU.1, con el objetivo de obtener la información necesaria para medir la protección modular acerca de las funciones que implementa el patrón de diseño *“Template Method”*.
Actor primario:	Estructura contenedora de información.
Precondiciones:	1) Contar con la lista de objetos generado por el analizador sintáctico.
Escenario de éxito principal:	1. Se identifican las funciones plantillas junto con su respectivo calificador de alcance. 2. Se calcula la métrica PMMP.

	3. Termina el caso de uso.
Postcondiciones:	1) El resultado obtenido es el resultado de la métrica PMMP. 2) Este resultado será usado para calcular la métrica PTTM. (Véase CU.6).
Escenario de fracaso 1:	La arquitectura a medir no presenta el patrón de diseño "*Template Method*"

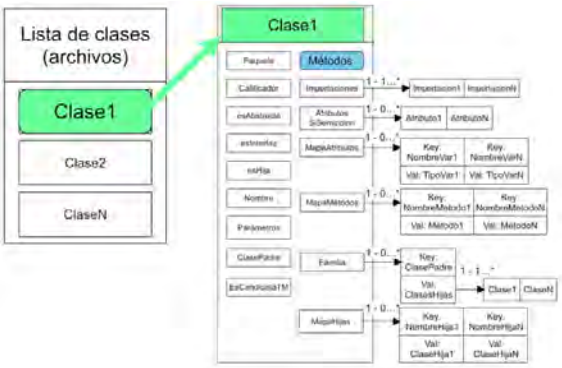
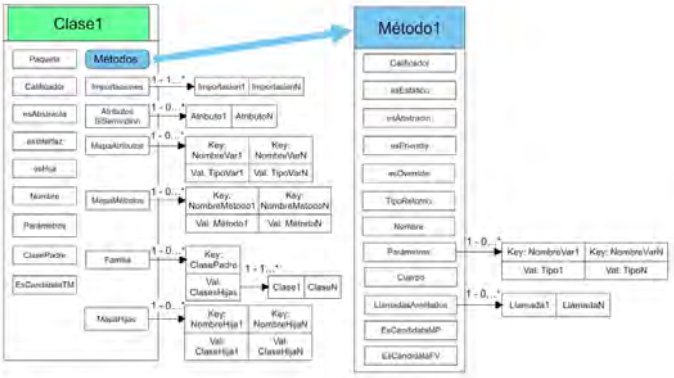
Nombre de caso de Uso:	CU.4 Métrica PMFV
Descripción:	El analizador sintáctico realiza un análisis sobre el código legado proveniente del CU.1 con el objetivo de obtener la información necesaria para medir la protección modular acerca de las funciones que implementa el patrón de diseño *"Template Method"*.
Actor primario:	Estructura contenedora de información.
Precondiciones:	1) Contar con la lista de objetos generado por el analizador sintáctico.
Escenario de éxito principal:	1. Se identifican las funciones variantes de la clase plantilla con su respectivo calificador de alcance. 2. Se calcula la métrica PMFV. 3. Termina el caso de uso.
Postcondiciones:	1) El resultado obtenido es el resultado de la métrica PMFV. 2) Este resultado será usado para calcular la métrica PTTM. (Véase CU.6).
Escenario de fracaso 1:	La arquitectura a medir no presenta el patrón de diseño "*Template Method*"

Nombre de caso de Uso:	CU.5 Métrica PMFI
Descripción:	El analizador sintáctico realiza un análisis sobre el código legado proveniente del CU.1 con el objetivo de obtener la información necesaria para medir la protección modular acerca de las funciones que implementa el patrón de diseño *"Template Method"*.
Actor primario:	Estructura contenedora de información.
Precondiciones:	1) Contar con la lista de objetos generado por el analizador sintáctico.
Escenario de éxito principal:	1. Se identifican las funciones invariantes de la clase plantilla con su respectivo calificador de alcance. 2. Se calcula la métrica PMFI. 3. Termina el caso de uso.
Postcondiciones:	1) El resultado obtenido es el resultado de la métrica PMFI. 2) El resultado será usado para calcular la métrica PTTM. (Véase CU.6).
Escenario de fracaso 1:	La arquitectura a medir no presenta el patrón de diseño "*Template Method*"

Nombre de caso de Uso:	CU.6 Métrica PTTP
Descripción:	El analizador sintáctico realiza un análisis sobre el código legado proveniente del CU.1 con el objetivo de obtener la información necesaria para medir la protección modular

	acerca de las funciones que implementa el patrón de diseño *"Template Method"*.
Actor primario:	Estructura contenedora de información.
Escenario de éxito principal:	1. Se realizan los cálculos de las métricas PMMP, PMFV, PMFI. (Véase CU.3, CU.4, CU.5). 2. Se calcula la métrica PTTP. 3. Termina el caso de uso.
Postcondiciones:	• El resultado obtenido es el resultado de la métrica PTTP
Escenario de fracaso 1:	La arquitectura a medir no presenta el patrón de diseño "*Template Method*"

Nombre de caso de Uso:	CU.7 Identificar información de funciones plantilla de clases
Descripción:	Cada archivo que contiene código legado a refactorizar será sometido a un proceso de análisis sintáctico y semántico haciendo uso de la gramática del lenguaje de programación Java en su versión 8 para obtener la información de metadatos requerida para el cálculo de métricas de protección modular del patrón de diseño "*Template Method*", así como para el proceso de refactorización.
Actor primario:	Analizador sintáctico.
Precondiciones:	1) Contar con la lista de archivos que contienen el código legado a refactorizar.
Escenario de éxito principal:	1. Se genera la estructura de datos lista para contener la información necesaria de las clases del código legado. 2. Se analiza cada una de las clases del sistema bajo refactorización y una vez identificadas las clases se extrae y almacena la información en variables de

	objeto de tipo “Clase”, a su vez cada objeto es almacenado en una lista compleja de objetos de tipo “Clase”, y cada nodo de esta lista tiene una referencia a una lista de métodos de cada objeto “Clase”.  3. Se analiza cada método de cada clase, una vez que son identificados se almacena la información en un objeto de tipo “Método”, cada objeto de este tipo es almacenado en la lista que contiene las funciones de los nodos de objeto “Clase”.  4. Termina el caso de uso.
Postcondiciones:	El resultado obtenido es la lista de objetos de tipo “Clase” con la información requerida para el cálculo de las métricas de protección modular de métodos asociados al patrón de

	diseño *“Template Method”* y la información requerida para la refactorización.
Escenario de fracaso 1:	1. Se genera una lista para contener la información necesaria de las clases del código legado. 2. Se almacena la información en listas de objetos de tipo “Clase” y en listas de objetos de tipo “Método”. 3. El intérprete falla en el análisis sintáctico. 4. El intérprete no genera el código analizado y envía un mensaje de error. 5. Termina el caso de uso.