SEP
SECRETARÍA DE EDUCACIÓN PÚBLICA

TECNOLÓGICO NACIONAL DE MÉXICO
Secretaría Académica, de Investigación e Innovación
Dirección de Posgrado, Investigación e Innovación

cenidet
Centro Nacional de Investigación y Desarrollo Tecnológico

Centro Nacional de Investigación y Desarrollo Tecnológico

Subdirección Académica

Departamento de Ciencias Computacionales

TESIS DE MAESTRÍA EN CIENCIAS

Generador de Servicios Web desde Marcos Orientados a Objetos con el Enfoque de Clases Internas

presentada por
Ing. Saraí Gallardo Vera

como requisito para la obtención del grado de
Maestra en Ciencias de la Computación

Director de tesis
Dr. René Santaolaya Salgado

Cuernavaca, Morelos, México. Febrero de 2018.

Cuernavaca, Morelos a 06 de febrero del 2018
OFICIO No. DCC/038/2018

Asunto: Aceptación de documento de tesis

DR. GERARDO V. GUERRERO RAMÍREZ
SUBDIRECTOR ACADÉMICO
PRESENTE

Por este conducto, los integrantes de Comité Tutorial de la **Ing. Saraí Gallardo Vera**, con número de control M16CE005, de la Maestría en Ciencias de la Computación, le informamos que hemos revisado el trabajo de tesis profesional titulado "**Generador de servicios web desde marcos orientados a objetos con el enfoque de clases internas"** y hemos encontrado que se han realizado todas las correcciones y observaciones que se le indicaron, por lo que hemos acordado aceptar el documento de tesis y le solicitamos la autorización de impresión definitiva.

DIRECTOR DE TESIS

Dr. René Santaolaya Salgado
Doctor en Ciencias de la Computación
4454821

REVISOR 1

Dra. Olivia Graciela Fragoso Díaz
Doctora en Ciencias en Ciencias de la Computación
7420199

REVISOR 2

M.C. Mario Guillén Rodríguez
Maestro en Ciencias con Especialidad en Sistemas Computacionales
7573768

C.p. M.T.I. María Elena Gómez Torres - Jefa del Departamento de Servicios Escolares.
Estudiante
Expediente

NACS/lmz

SEP
SECRETARÍA DE EDUCACIÓN PÚBLICA

TECNOLÓGICO NACIONAL DE MÉXICO
Centro Nacional de Investigación y Desarrollo Tecnológico

Cuernavaca, Mor., 16 de febrero de 2018
OFICIO No. SAC/121/2018

Asunto: Autorización de impresión de tesis

ING. SARAÍ GALLARDO VERA
CANDIDATA AL GRADO DE MAESTRA 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 **"Generador de servicios web desde marcos orientados a objetos con el enfoque de clases internas"**, 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.

ATENTAMENTE
"CONOCIMIENTO Y TECNOLOGÍA AL SERVICIO DE MÉXICO"

DR. GERARDO VICENTE GUERRERO RAMÍREZ
SUBDIRECTOR ACADÉMICO

SEP TecNM
CENTRO NACIONAL
DE INVESTIGACIÓN
Y DESARROLLO
TECNOLÓGICO
SUBDIRECCIÓN
ACADÉMICA

C.p. M.T.I. María Elena Gómez Torres .- Jefa del Departamento de Servicios Escolares.
Expediente

GVGR/mcr

Dedicatorias

*Con mucho **amor:***

*A **Dios** por su infinita misericordia y por permitirme llegar a disfrutar este día.*

*A mis hijos **Ximena Saraí** y **Jesús Salvador** por ser mis bendiciones y regalarme con su existencia la mayor inspiración.*

*A mi esposo **Jesús Salvador** por brindarme su amor incondiciona y apoyarme en todo momento. Es una bendición compartir y valorar lo más importante de nuestras vidas. Te admiro mucho mi amor.*

*A mis padres **Francisco** y **Yolanda** por guiarme con su sabiduría y siempre estar para mi cuando más lo necesito. Estoy muy orgullosa de ser su hija.*

*A mis hermanos **José Francisco** y **Martín Arturo** por compartirme sus alegrías, por su confianza y cuidarnos creciendo juntos.*

*A mis abuelas **Elvira** y **Ángela** por ser mi ejemplo de fortaleza.*

"Todo parece imposible hasta que se hace"
Nelson Mandela.

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 por el Programa Nacional de Posgrado de Calidad.

Al Instituto Nacional de Electricidad y Energías Limpias por el apoyo otorgado, que a través de la División Tecnologías Habilitadoras y la Gerencia Tecnologías de la Información impulsaron mi crecimiento profesional y laboral al cursar esta maestría. Al Director de División, Dr. Salvador González Castro y al Gerente, Dr. Gustavo Arroyo Figueroa por todo su apoyo y la confianza para la realización del presente trabajo de tesis

A mi director de tesis, Dr. René Santaolaya Salgado, por compartir su conocimiento y dirigirme con su supervisión en el desarrollo de esta investigación, por su paciencia y valiosa confianza, pero sobre todo por su motivación, integridad y el apoyo que me brindo día a día durante está etapa de mi vida profesional y académica

A mis revisores, Dra. Olivia Graciela Fragoso Díaz y MC. Mario Guillen Rodríguez 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 compañeros de generación: Roberto, Carlos, Gibran y Nancy, por compartir este espacio de estudio.

Resumen

Actualmente, los sistemas de software enfrentan múltiples desafíos derivados de factores como la complejidad, el cambio de tecnología y la necesidad de interoperabilidad de datos. Todos estos factores han llevado a las industrias de desarrollo de software a la adaptación de tecnologías para cumplir con las especificaciones de los usuarios finales, buscando estrategias de desarrollo que permitan la adaptación con el menor costo y tiempo.

El reuso de software es una de las estrategias que se considera promisoria para que la industria de desarrollo de software pueda enfrentar el reto de desarrollar productos con niveles de calidad y productividad adecuados en un contexto de negocio altamente complejo, dinámico y con acelerados cambios tecnológicos. En esta tesis se consideran los servicios Web como enfoque estratégico para el reuso de software.

En este trabajo se propone un método para transformar el código de Marcos Orientados a Objetos hacia Marcos de Servicios Web, buscando mejorar la modularidad de los Servicios Web con arquitectura SOA, mediante la composición de clases internas y el agrupamiento racional de métodos que participan en secuencias interactivas de casos de uso, así como los datos que son manipulados por estos métodos. El propósito que se persigue es balancear los niveles de coherencia, cohesión, factor de acoplamiento y el tiempo de respuesta; de la arquitectura interna de servicios; ya sea que estos sean desarrollados como servicios Web o como Microservicios, de tal manera que se logre el equilibrio esperado entre estos atributos de calidad, para obtener un mejor desempeño, así como mejores condiciones de reuso y mantenimiento.

El método desarrollado en esta tesis se implementó en una extensión a la herramienta SR2-Transforming (Sistema de Reingeniería para Transformación) para dar soporte a la Reingeniería de Marcos de Aplicaciones Orientados a Objetos hacia Marcos de Servicios Web equivalentes.

Por último, se establecieron pruebas que comprueban estadísticamente que el procedimiento que se presenta, cumple con las mejoras perseguidas como objetivo.

Palabras clave: marco orientado a objetos, servicios web, reingeniería de software, métricas de calidad de software.

Abstract

Actually, the software systems face a multitude of challenges resulting from different factors such as high complexity, changing technology and the need for interoperability of heterogeneous data. All these factors have led the software development industries to adopt new technologies to meet the specifications of end users, seeking development strategies that allow adaptation at the lowest possible cost and time.

Software reuse is one of the strategies considered promising for the software development industry to face the challenge of developing products with adequate levels of quality and productivity in a highly complex and dynamic business context with accelerated technological changes. In this thesis, Web services are considered as a strategic approach for software reuse.

This paper proposes a method to transform the code from an Object Oriented Frameworks to Web Services Frameworks, seeking to improve the modularity of Web Services with SOA architecture, through the composition of internal classes and the rational grouping of methods that participate in Interactive sequences of use cases, as well as the data that is manipulated by these methods. The purpose is to balance the levels of coherence, cohesion, coupling factor and response time; of the internal architecture of services; whether these are developed as Web services or as Microservices, in such a way that the expected balance between these quality attributes is achieved, to obtain a better performance and better reuse and maintenance conditions.

The method developed in this thesis was implemented in an extension to the tool SR2-Transforming (Reengineering System for Transformation) to support the Reengineering of Object-Oriented Applications Frameworks towards equivalent Web Services Frameworks.

Finally, they established tests that prove statistically that the procedure that is presented, complies with the improvements pursued as objective.

Keywords: object-oriented framework, web services, software reengineering, software quality metrics

Índice

Lista de figuras

Lista de tablas

Tabla de abreviaturas

XML	eXtensible Markup Language (Lenguaje de marcado extensible): es el lenguaje de marcas que se utiliza para describir la información; puede describir datos y documentos.
SOAP	Simple Object Access Protocol (Protocolo simple de acceso a objetos): es un protocolo de mensajería (basado en XML), que indica cómo se deben codificar los mensajes que circularán entre las dos aplicaciones, cliente y proveedor del servicio. Este protocolo permite que se comuniquen programas que corren en diferentes sistemas operativos.
REST	Representational State Transfer (Transferencia de estado representacional): es un protocolo de mensajería que permite que los sistemas accedan y manipulen las representaciones textuales de los recursos web mediante un conjunto uniforme y predefinido de operaciones sin estado.
WSDL	Web Services Description Language (Lenguaje de descripción de servicios Web): lenguaje que define un mecanismo estándar para describir un servicio web. Los documentos de WSDL deben estar disponibles en el servidor web que ofrece los servicios. En realidad, WSDL es un vocabulario XML para describir un servicio web.
CIM	Computational-independent Model (Modelo Computacional Independiente): se centra en los requerimientos y representa el nivel más alto del modelo de negocios.
PIM	Platform-Independent Model (Modelo Independiente de la Plataforma): es un modelo de un sistema de software o un sistema de negocios que es independiente de la plataforma tecnológica específica utilizada para implementarlo.
PSM	Platform-specific Model (Modelo Específico de la Plataforma): representa la proyección de los PIMs en una plataforma específica de desarrollo de software. Un PIM puede generar múltiples PSMs, cada uno para una tecnología distinta.
MOOinMS	Acrónimo del método de transformación de Marcos Orientados a Objetos con arquitectura de clases internas a Marcos de Servicios Web.

Glosario de términos

ACOPLAMIENTO El acoplamiento es una medida del grado de relación de un módulo contra los demás. Se refiere al número de variables que intervienen en la comunicación entre módulos, más que al número de interfaces de comunicación entre módulos. Si dos módulos se comunican, deben de intercambiar tan poca información como sea posible (Brito, Iseg, Goulão, Esteves, & Ist, 1995).

AUTOSUFICIENCIA Los Módulos, Componentes, Clases, Servicios o Microservicios, que son autosuficientes se definen como aquellas unidades de software que contienen toda la información y la funcionalidad, que la manipula, necesarias (ni más-ni menos) para realizar un único objetivo o una única meta; sin necesidad de requerir de otros módulos o unidades de programa para poder llevar a cabo las tarea, para alcanzar el resultado de valor para el cliente (Santaolaya, Fragoso, Rojas, Álvarez, & León, 2016).

CASO DE USO Representa la meta de una interacción entre un actor y el sistema. La meta representa un objetivo medible y significativo para el actor. Capturan y por lo tanto contienen los requerimientos funcionales de los sistemas, en un formato fácil de seguir y fácil de leer (Fowler, 1997).

CLASE Es una entidad de software que participa en conjunto con otras clases en el dominio de un problema de software (Hull, Schwander, Green, & Tung, 2000).

CLASE INTERNA Es una entidad de software que coloca una definición de clase dentro de otra definición de clase, es decir, ubica su definición dentro de una clase envolvente. Permite agrupar clases que lógicamente están relacionadas, además de controlar la visibilidad de una con la otra (CompSci, 2016).

COHERENCIA La coherencia se define como el grado de relación funcional de una responsabilidad en una unidad de programa. La coherencia se basa en el principio de una única responsabilidad Single Responsibility Principle (SRP) el cual indica que: "una clase o módulo debe tener uno y sólo un motivo para cambiar" (R. C. Martin, 2002) .

En términos generales una clase está compuesta por elementos que pueden ser métodos y atributos. En el contexto de la coherencia, una responsabilidad se refiere a una secuencia interactiva de métodos implicados para cumplir o alcanzar una meta u objetivo.

COHESIÓN La cohesión se define como el grado de relación entre los miembros de una clase. La cohesión es el concepto en donde se ve cómo los métodos de una clase están estrechamente relacionados entre sí. Si el módulo es altamente cohesivo la complejidad de las clases se reduce, el módulo es reutilizable y fácil de mantener (Yadav, Sikka, & Shrivastava, 2014).

MARCO DE SERVICIOS WEB Un marco de servicios web se define como un conjunto de servicios web independientes, que contienen la experiencia y el conocimiento para satisfacer las necesidades de aplicaciones de dominios (Santaolaya et al., 2016).

MARCO ORIENTADO A OBJETOS (MOO) Es un conjunto semi-completo de clases en colaboración, que incorpora un diseño genérico el cual puede ser adaptado a una serie de problemas específicos para producir nuevas aplicaciones hechas a la medida (Mattsson, Bosch, & Ronneby, 1997).

META DE VALOR En el contexto de esta investigación una meta u objetivo de valor a nivel de función, como unidad de programa, es realizar una única operación funcional. A nivel de módulo o subsistema, una meta de valor u objetivo es el de realizar un caso de uso o requerimiento especifico. A nivel de sistema una meta de valor u objetivo es resolver una aplicación completa o proceso de negocio planteada por un cliente o usuario.

MÉTODO INICIADOR En el contexto de esta investigación los métodos iniciadores de casos de uso son todos los métodos públicos pertenecientes a clases públicas no abstractas.

MICROLITO Un microlito (Microlith) se define como un servicio de instancia única, aislado, en el que las llamadas a métodos se realizan como llamadas REST síncronas (Bonér, 2017).

MICROSERVICIO Son servicios pequeños y autónomos, cada uno con sus propios datos y aislados de forma independiente, escalable y resistente a fallas (Eisele, 2016).

MODULARIDAD Es una medida de la capacidad de reuso de una unidad de software que puede ser una función, una clase, un paquete, un subsistema o un sistema, y que se distingue principalmente por ser reusable, fácil de mantener y no requiere de otros módulos para obtener un resultado o una meta de valor para el usuario final (Erdemir & Buzluca, 2014).

REFACTORIZACIÓN La refactorización es aquel proceso cuyo objetivo es modificar la arquitectura de clases de un sistema, más no modificar la funcionalidad del mismo. Para asegurarse que esto no ocurra, una refactorización siempre cuenta con precondiciones y pos-condiciones que se deben de cumplir antes y después de aplicar una transformación. Por tanto, la refactorización no ayuda a un sistema actual sino a sistemas futuros que se extiendan del sistema actual (Fowler, 2002).

SECUENCIA INTERACTIVA En el contexto de esta investigación una secuencia interactiva es: "El conjunto de métodos que responden a una secuencia sucesiva de mensajes que son enviados desde los objetos, la secuencia inicia con un evento que activa al método iniciador de un caso de uso a través de una interacción de un actor, y termina con la meta de valor que se espera del caso de uso". Este conjunto de mensajes puede extraerse de un diagrama de secuencias del UML.

SERVICIO WEB Es una unidad lógica para el intercambio de datos con otras unidades lógicas a través de una red usando un grupo de protocolos y estándares tales como XML, WDSL y SOAP (Vinay & Jain, 2012).

TIEMPO DE RESPUESTA Es el tiempo transcurrido desde que se envía una petición de servicio y se recibe su respuesta.

Capítulo 1: Introducción

1.1 Presentación

El desarrollo de software bajo estándares y el uso de las mejores prácticas es un desafío que impacta en la calidad de los productos y la reducción de costos de producción de soluciones de software de las organizaciones. Una manera de ayudar a cumplir con este objetivo es maximizar el reuso de software legado. Se considera que el software legado es el activo de software de las organizaciones que a través del tiempo han sido desarrollados a un alto costo y que implícitamente contienen las reglas del negocio, por lo que no pueden ser simplemente desechados. Un enfoque para facilitar el uso de sistemas legados es organizarlos en Marcos con arquitecturas Orientados a Objetos (MOO's).

Los MOO's representan soluciones en un contexto general y proveen mecanismos eficientes para su adaptación a comportamientos particulares en nuevas aplicaciones. Algunas de las principales desventajas que presentan estos son: su arquitectura está fuertemente integrada, dependen de la plataforma y del lenguaje nativo en que fue desarrollado el MOO.

A finales de la década de los 90's surgió el concepto de servicios web como una combinación de los mejores postulados de la tecnología de componentes y la web. Se entiende que un servicio web es un recurso de información independiente que se puede comunicar con otra aplicación web a través de protocolos estándares de internet. Los servicios web integran los componentes de una Arquitectura Orientada a Servicios (SOA), la cual tiene como objetivo representar una capacidad ofrecida por una organización. Esta tecnología ofreció en su momento una alternativa de construcción de soluciones de software utilizando el software existente de las empresas con la cual se solventaban los problemas mencionados de los MOO's. Algunas desventajas de los servicios web es que demandan de los desarrolladores tener conocimiento y experiencia para diseñarlos de tal forma que tengan un alto grado de modularidad y suficiencia, lo cual no siempre es alcanzable (Nistor, 2015).

Actualmente ha emergido la tecnología de Microservicios. Esta emerge como una nueva tecnología que permite crear sistemas desde una colección de servicios pequeños y autónomos, cada uno con sus propios datos y aislados de forma independiente, escalable y resistente a fallas. Los Microservicios se integran con otros Microservicios con el fin de formar nuevos sistemas (Eisele, 2016).

En el CENIDET se han realizado trabajos de investigación que se encuentran relacionados con el presente tema de tesis. Francisco León en la tesis de maestría: "*Generación de Servicios Web desde Marcos Orientados a Objetos a partir de Clases Colaborativas en Casos de Uso (MOOaSW)*" (León, 2009) plantea una estrategia alternativa para generar servicios web desde marcos de aplicaciones orientadas a objetos. La arquitectura de clases resultante para los servicios web es, únicamente, el conjunto de clases relacionadas y las funciones y datos que son utilizados en la secuencia interactiva del Caso de Uso, respeta las relaciones arquitecturales entre éstas como se definen en el MOO original (su desventaja principal es que no toma en cuenta el rendimiento de los servicios Web generados). El segundo trabajo de maestría: "Transformación de Marcos de Aplicaciones Orientados a Objetos hacia Marcos con Arquitectura Orientada a Servicios" (Legorreta, 2017) construye servicios Web partir de la desintegración del marco orientado a objetos original, en una única clase envolvente, eliminando relaciones de agregación, herencia y dependencia. Cada clase es diseñada en suficiencia y completes con los atributos y las funciones necesarias para atender la secuencia interactiva de un caso de uso, y estas a la vez son convertidas en servicios Web. Para lograr la funcionalidad integrada equivalente al marco original (su desventaja es que no considera balancear los atributos de modularidad con el atributo de rendimiento).

El proyecto de desarrollo tecnológico, que se describe en esta tesis, consiste en mejorar la modularidad de los Servicios Web que conforman Marcos de Servicios Web con arquitectura SOA, mediante la composición de clases internas y el agrupamiento racional de funciones que participan en secuencias interactivas de casos de uso, así como los datos que son manipulados por estas funciones. El propósito que se persigue es balancear los niveles de coherencia, cohesión, factor de acoplamiento y el tiempo de respuesta; de la arquitectura interna de servicios; ya sea que estos sean desarrollados como servicios Web o como Microservicios, de tal manera que se logre el equilibrio esperado entre estos atributos de calidad, para obtener un mejor desempeño y mejores condiciones de reuso y mantenimiento.

El enfoque desarrollado en esta tesis se implementó en una extensión a la herramienta SR2-Transforming para dar soporte a la Reingeniería de Marcos de Aplicaciones Orientados a Objetos hacia Marcos de Servicios equivalentes.

1.2 Organización del documento de tesis

El contenido del presente documento de tesis es el siguiente:

El capítulo dos describe el Estado del Arte del presente trabajo de investigación, seguido por los trabajos relacionados y trabajos realizados por el CENIDET, concluyendo con la descripción de los objetivos, el planteamiento del problema y los alcances y las limitaciones del presente proyecto de tesis.

El capítulo tres incluye el marco teórico o conjunto de ideas, procedimientos y teorías que sirven a la presente investigación.

El cuarto capítulo comprende la descripción del Análisis y Diseño del método de transformación utilizando una arquitectura de clases internas.

El quinto capítulo incluye la Implementación del método de transformación, con elementos definidos de la arquitectura interna de servicios sobre los cuales la herramienta computacional generó cada caso de uso.

En el sexto capítulo se realiza el planteamiento de las hipótesis y el diseño de las pruebas. Este capítulo es de gran importancia debido a que se formulan las hipótesis experimentales, se identifican las variables dependientes e independientes y posteriormente concluye con la metodología para realizar el Plan de Pruebas.

El séptimo y octavo capítulos se incluye el proceso de desarrollo del caso de prueba y la recolección de los resultados y el análisis de los mismos.

El trabajo de tesis concluye en el noveno capítulo donde se discuten los resultados, la experiencia capitalizada del desarrollo, se establecen las conclusiones finales y se proponen los trabajos futuros a desarrollar.

Capítulo 2: Marco de Referencia

2.1 Estado del Arte

En los últimos años el desarrollo de software entró en una fase de transición del software orientado a objetos hacia el software orientado a servicios. Ahora los objetos pueden ser convertidos a servicios los cuales pueden ser accedidos desde cualquier lugar con una arquitectura empresarial. Sin embargo, se desconoce el nivel correcto de granulación de los servicios de software, lo cual dificulta su gestión como unidades de reuso. (Sneed, Verhoef, & Sneed, 2013)

El objetivo final de las arquitecturas orientadas a servicios es tener servicios de propósito general, los cuales puedan ser compartidos por muchos usuarios para múltiples aplicaciones (Allam, 2014).

La arquitectura para servicios web proporciona un marco que puede ser representado con más que potentes ilustraciones y practicas desde el reconocimiento de los enfoques de la ciencia computacional. Un grupo de servicios puede poseer así otros servicios, llamados servicios amalgamados, que representan procesos de negocio. (Vinay & Jain, 2012)

Los Servicios Web, junto con una pila de protocolos, se han convertido en el medio principal de expertos para la generación de aplicaciones de computación orientada a objetos. El *Protocolo Simple de Acceso a Objetos* (SOAP) se utiliza generalmente como protocolo de comunicación (Nistor, 2015). El *Lenguaje de Descripción de Servicios Web* (WSDL) se ha convertido en el estándar de facto para desplegar los límites de los servicios Web. Además, el concepto de *Descripción, Descubrimiento e Integración Universal* (UDDI) está relacionado con la definición de un servicio de directorio para servicios Web. UDDI permite a los clientes de servicios Web localizar los servicios Web candidatos y descubrir sus detalles. Los consumidores de servicio y los proveedores de servicios utilizan estos estándares para realizar los procedimientos básicos de la *Arquitectura Orientada a Servicios* (SOA). (Vinay & Jain, 2012)

Entre las ventajas que proporciona la aplicación de prácticas de la ingeniería de software en los servicios Web están: la encapsulación, la pérdida de acoplamiento y la reusabilidad. (Kwon & Tilevich, 2014)

El ciclo de vida de desarrollo de servicios Web consiste en varias fases. La fase de definición de la interfaz de servicio involucra documentos derivados y especificaciones WSDL. A través del código fuente de los servicios web, los WSDL son automáticamente generados desde la implementación de códigos. (Mateos, Crasso, Zunino, & Coscia, 2015)

La arquitectura de microservicios fue introducida para resolver algunos problemas asociados con las arquitecturas monolíticas. En una arquitectura de microservicios diferentes componentes pueden ser implementados usando diferentes lenguajes de programación.

El panorama actual es que, por lo general, los trabajos de investigación abordan a los atributos QoS como factores independientes útiles para la generación de servicios Web, compuestos o para seleccionar servicios Web, de acuerdo a su funcionalidad. Sin embargo, existe duda acerca del comportamiento conjunto de los atributos de calidad: coherencia, cohesión, acoplamiento y el tiempo de respuesta en arquitecturas de clases internas de servicios Web. Existen trabajos de investigación que están relacionados de cierta forma con los temas estudiados en la presente tesis, los cuales se describen a continuación:

2.2 Trabajos relacionados

Para la evaluación de trabajos relacionados se hizo un estudio de nueve trabajos recientes, principalmente de los últimos cinco años. Para la comparativa, la selección de trabajos es una muestra heterogénea de publicaciones de trabajos que incluyen diferentes valores de las facetas o criterios de evaluación, como se describe a continuación:

- **Objetivo**. Se considera este criterio de comparación para establecer una diferencia en cuanto al objetivo que se persigue con la aportación de cada uno de los trabajos estudiados, en contra con el objetivo perseguido en el trabajo de tesis que se propone en este documento. Los diferentes objetivos de los trabajos estudiados son:

    - SPN: Servicios Reusables de Procesos de Negocio.
    - IS: Identificación de servicios

- **Método**. Analizar el enfoque usado de los trabajos relacionados para alcanzar el objetivo. El enfoque de algunos trabajos estudiados radica en:

- Top-down (TD): realiza un proceso de ingeniería directa para el análisis del negocio o de software, dando como resultado la identificación de casos de uso o servicios.
- Bottom-up (BU): realiza un proceso de ingeniería inversa para analizar el código legado y así derivar casos de uso o servicios.
- Meet-in-the-middle (MM): Realiza una combinación de estrategias Top-down y Bottom-up.

- **Tipo de ejecución**. Es el grado de automatización del enfoque propuesto en los trabajos relacionados:

  - AA: Indica si el enfoque propuesto es automático.
  - SA: Indica si el enfoque propuesto es automático con intervención humana.
  - MA: Indica si el enfoque propuesto es principalmente manual.

- **Tipo de cliente**. Es para conocer si el producto/resultado de los trabajos relacionados es útil para:

  - Actores externos (AE): Son los usuarios que interactúan con el proceso a través de una interfaz.
  - Actores lógicos (AL): Interacción interna entre unidades de programa a través del intercambio de información.
  - (AE+AL) Son ambos tipos de clientes.

- **Entrada**. Para los diferentes trabajos relacionados se indican las entradas requeridas por el enfoque o método estudiado:

  - MS: Modelos de secuencias interactivas
  - CU: Modelos de casos de uso
  - MC: Modelos de clases
  - MM: Meta-Modelos como PIM, CIM, PSM
  - CF: Código fuente

- **Aportación**. Es el criterio usado para establecer el grado de innovación de los diferentes trabajos relacionados, de los cuales se consideran:

  - MO: Un modelo
  - ME: Una metodología

- HS: Una herramienta de software
- FS: Framework de servicios
- AM: Ambiente de modelado

- **Alcance**. Es el criterio usado para describir el nivel obtenido del desarrollo tecnológico:

  - PS: Propuesta de servicios (se propone un modelo o una metodología)
  - IS: Implementación de servicios (del método o modelo)

- **Producto/Resultado**. Este es el producto derivado de la aportación realizada para alcanzar el objetivo del trabajo relacionado, de los cuales se toman en cuenta:

  - SW: Servicios web
  - SS: Servicios de software
  - MS: Microservicios
  - MSOA: Marcos de Servicios Web con arquitectura SOA.

- **Tipo de evaluación.** Se refiere al tipo de evaluación realizada al producto resultado obtenido en los diferentes enfoques de los trabajos que se presentan en este documento. Se muestran en los criterios siguientes:

  - H: Hipotético o teórico
  - EEB: Evaluación empírica de baja escala
  - EEG: Evaluación empírica de gran escala
  - EX: Evaluación experimental

- **Utiliza Métricas.** Se refiere a la integración de indicadores de calidad en el trabajo relacionado que influyen en el reuso del software.

  - MDE: De rendimiento
  - (Tiempo / espacio - recursos de almacenamiento de la maquina)
  - MDM: De modularidad (Coherencia, Cohesión, Acoplamiento)
  - MDX: De flexibilidad (grado de abstracción, valor de NOC)

Tabla 1. Tabla comparativa de trabajos relacionados

Trabajo Relacionado	Objetivo	Método	Tipo de ejecución	Tipo de cliente	Entrada	Aportación	Alcance	Producto/ resultado	Tipo de evaluación	Métrica
"KWATT: a toolkit for automatic web service generation" (Qu, Bollig, & Erlebacher, 2008)	IS	BU	A	AE	CF	HS	ISS	SW	EEG	-
"UML-Based Specification and Generation of Executable Web Services" (Dahman & Grabowski, 2011)	SPN	TD	SA	AE	MC+MM	HS	ISS	SW	H	-
"Reusing Existing Object-oriented Code as Web Services in a SOA" (Sneed et al., 2013)	SPN	BU	A	AE+AL	CF	ME+HS	ISS	SW	EEG	-
"Method for Generating Web Services Frameworks from Object Oriented Frameworks Source Code" (Santaolaya et al., 2016)	SPN+IS	BU	A	AE+AL	CF	ME+HS	ISS	MSOA	EX	MDM
"A learning-based module extraction method for object-oriented systems" (Erdemir & Buzluca, 2014)	IS	TD	SA	AE	MC+CF	FS	ISS	SS	EEB	MDM
"Semi-automatic Generation of Web Services and BPEL Processes - A Model-Driven Approach" (Anzböck & Dustdar, 2005)	SPN	TD	SA	AE	MS	ME+AM	PS	SS	H	-
"Cloud refactoring: automated transitioning to cloud-based services" (Kwon & Tilevich, 2014)	IS	TD	SA	AE	MS	MO	PS	SS	EEB	MDM
"Creating Web Services from Legacy Code" (Vinay & Jain, 2012)	IS	TD	SA	AE	MC+CF	FS	ISS	SW	H	MDE
"A Framework for Goal-Oriented Discovery of Resources in the RESTful Architecture" (Fernández-villamor, Iglesias, & Garijo, 2014)	IS	TD	A	AE	MC+CF	FS	PS	SW	H	-
En esta investigación	SPN+IS	BU	A	AE + AL	CF	ME + HS	IS	MS	EX	MDR

El desarrollo tecnológico que se describe en esta tesis, al cuál de aquí en adelante denominaremos *MOOinMS*, plantea obtener Marcos de servicios Web desde el código de Marcos Orientados a Objetos (MOO), al igual que los desarrollos tecnológicos de antecedente en (León, 2009) y (Legorreta, 2017). La diferencia entre estos dos trabajos consiste en la arquitectura interna de cada servicio Web generado. En la tesis de (León, 2009), la arquitectura interna de cada servicio Web reside en un conjunto de clases relacionadas, con los atributos y funciones necesarias para atender un requerimiento como meta de valor. En la tesis de (Legorreta, 2017), la arquitectura interna de cada servicio Web se basa en reunir en una única clase de objetos, con los atributos y funciones necesarias para atender un requerimiento como meta de valor. En esta tesis, se ha planteado que la arquitectura interna de cada servicio Web consista en una única clase de objetos con composición de clases internas, agrupando las clases, funciones y atributos necesarios para formar una secuencia interactiva y satisfacer un requerimiento como meta de valor, esperando obtener mejor tiempo de respuesta y mejores cualidades de autonomía de los servicios Web obtenidos a partir del MOO original.

Los trabajos estudiados cuyo producto-resultado coincide con el que se propone en esta tesis, en cuanto al método de identificación y la generación de servicios web, son: "*KWATT: a toolkit for automatic web service generation"* (Qu et al., 2008), *"Reusing Existing Object-oriented Code as Web Services in a SOA*" (Sneed et al., 2013) y "*Method for Generating Web Services Frameworks from Object Oriented Frameworks Source Code*" (Santaolaya et al., 2016). La diferencia entre estos trabajos contra nuestra propuesta se detalla en la metodología y la aportación final obtenida.

En el trabajo "*KWATT: a toolkit for automatic web service generation*" propuesto por (Qu et al., 2008), se utiliza un enfoque Bottom-up que parte de análisis de código fuente desde archivos con código *TCL* o *Phyton* por medio del cual se puede transferir código binario de imágenes a través de internet. Su método se caracteriza por utilizar plantillas para distinguir el código estático del dinámico. Mediante un parser ubica los procesos a exportar, a través de la identificación de palabras clave, combina los métodos en una envoltura *C++* para generar un archivo temporal. Utiliza los archivos generados y la lista de código evaluado para obtener todas las salidas de los métodos Web. Esta salida se compila y genera los archivos *WSDL* y enlaces *SOAP* correspondientes a los métodos web en *C++*. Finalmente se despliegan los servicios y se acceden a través de un puerto de salida. La diferencia fundamental de ese trabajo con la investigación en esta tesis es que nuestro desarrollo parte del análisis estático de código de Marcos Orientados a Objetos.

En el trabajo "*Reusing Existing Object-oriented Code as Web Services in a SOA*" propuesto por (Sneed et al., 2013), se presenta la herramienta *SoftReuse*. Esta herramienta genera interfaces para métodos públicos existentes en código *Java* y C#, pero también genera una documentación visual de estas interfaces y códigos de prueba. El enfoque adoptado para envolver el código orientado a objetos es la transformación de la interfaz dentro de una tabla relacional. La siguiente transformación es una definición del archivo WSDL, y es generado desde la tabla relacionada. Por último, realiza la transformación del proceso de negocios en BPEL para probar la definición de la interfaz del servicio WSDL generado.

El trabajo "*Method for Generating Web Services Frameworks from Object Oriented Frameworks Source Code*" (Santaolaya et al., 2016), tiene como objetivo la transformación automática de Marcos Orientados a Objetos a sus equivalentes Marcos de Servicios Web. En contraste el presente trabajo de tesis, es la generación de Marcos de Servicios Web en cuya arquitectura se utiliza el enfoque de clases internas del lenguaje *Java*.

En los trabajos "*Semi-automatic Generation of Web Services and BPEL Processes - A Model-Driven Approach*" (Anzböck & Dustdar, 2005) , "*Cloud refactoring: automated transitioning to cloud-based services*" (Kwon & Tilevich, 2014) y "*A Framework for Goal-Oriented Discovery of Resources in the RESTful Architecture*" (Fernández-villamor et al., 2014), se proponen servicios de manera automática para actores externos. El enfoque del trabajo propuesto por (Anzböck & Dustdar, 2005) inicia con la representación digital de procesos de negocio en un modelo, para generar de forma manual un diagrama de secuencias. En seguida se extraen los procesos, los mensajes y las propiedades en tiempo de ejecución. De manera automática se genera el diagrama de actividades en base al código extraído en el paso anterior. A continuación, se define una metodología para la orquestación de servicios web desde un diseño dirigido por modelos. Como producto resultado, se obtienen: la definición de procesos BPEL, el método web descrito en un archivo WSDL y los servicios, que son definidos desde el análisis de requerimientos plasmados en el diagrama de secuencias.

En el trabajo "*Cloud refactoring: automated transitioning to cloud-based services*" (Kwon & Tilevich, 2014), se presenta un grupo de técnicas de refactorización que facilitan el proceso de transformación de aplicaciones centralizadas para usar servicios basados en la nube. La refactorización se realiza en tres escenarios: El primer escenario implica trasladar parte de la funcionalidad de una aplicación centralizada a la nube. El segundo escenario implica agregar funcionalidad de tolerancia a fallas al cliente para manejar las fallas planteadas durante la invocación de un servicio basado en la nube. El tercer escenario implica modificar una

aplicación para utilizar servicios alternos desplegados en la nube, expuestos a través de una interfaz de servicio. Si los métodos del servicio adaptado difieren en términos de su número de parámetros o tipo, el programador necesita escribir código para adaptar los parámetros y/o el valor de retorno. Esta parte del enfoque es manual, una adaptación de parámetros es una aplicación altamente específica, de manera que es difícil automatizarla.

En los trabajos "*Reusing Existing Object-oriented Code as Web Services in a SOA*" (Sneed et al., 2013), "*Method for Generating Web Services Frameworks from Object Oriented Frameworks Source Code*" (Santaolaya et al., 2016), "*A learning-based module extraction method for object-oriented systems*" (Erdemir & Buzluca, 2014) y "*A Framework for Goal-Oriented Discovery of Resources in the RESTful Architecture*" (Fernández-villamor et al., 2014), se considera la integración de algunos indicadores de calidad que influyen en el reuso del software. El desarrollo en (Erdemir & Buzluca, 2014), inicia con la identificación de servicios de software. Implementa una herramienta de extracción de módulos para agrupar las clases y sus relaciones usando un árbol de sintaxis abstracta. Obtiene los módulos de servicios a partir de los grafos dirigidos aplicando un algoritmo de agrupamiento jerárquico.

La investigación de la implementación del resultado obtenido de los trabajos "*UML-Based Specification and Generation of Executable Web Services*" (Dahman & Grabowski, 2011), "*Semi-automatic Generation of Web Services and BPEL Processes - A Model-Driven Approach*" (Anzböck & Dustdar, 2005), "*Creating Web Services from Legacy Code*" (Vinay & Jain, 2012) y "*A Framework for Goal-Oriented Discovery of Resources in the RESTful Architecture*" (Fernández-villamor et al., 2014) es teórico, ya que el tipo de evuación que relizan a partir de los datos obtenidos es hipotético. El propósito que se persigue en el trabajo de investigación *"UML-Based Specification and Generation of Executable Web Services"* (Dahman & Grabowski, 2011), es habilitar el desarrollo dirigido por modelos de los servicios Web a través de la generación automática de código desde modelos UML. Para ello se utiliza la herramienta *UP4WS* (*UML Profile for Web Services*), la cual es una implementación orientada a perfiles. UP4WS define extensiones UML que incluyen estereotipos para hacer que los servicios Web sean ejecutables, al permitir la generación de código fuente a partir de los elementos del modelo que aplican esos estereotipos. Para implementar los perfiles usa reglas Xpand con las que habilita la transformación de modelos UML en código fuente.

Las diferencias fundamentales de este trabajo de tesis contra los trabajos relacionados descritos son:

1. El proceso de transformación propuesto es mayormente automático;
2. enfocado hacia la obtención de Marcos de Servicios Web de dominios de aplicaciones de negocios,
3. el método utiliza el enfoque Bottom-Up, iniciando con un proceso de ingeniería inversa del código legado del dominio del negocio y termina con un proceso de ingeniería directa que produce el marco de servicios web del mismo dominio,
4. como punto de partida, utiliza el código fuente del software existente en MOO's del dominio del negocio,
5. brinda soporte tanto a actores externos como a actores lógicos;
6. aplica métricas orientadas a determinar la autonomía de los servicios Web generados para obtener un balance definido por la coherencia, cohesión, acoplamiento y tiempo de respuesta de los servicios Web obtenidos,
7. se realiza una evaluación experimental de los casos de prueba establecidos en la tesis antecedente (Guadarrama Rogel, 2013) y
8. se generan Marcos de servicios web en niveles de granulación de método o función, requerimiento o componente y proceso de negocio.

2.3 Desarrollos tecnológicos realizados en el CENIDET

El presente trabajo de tesis está relacionado directamente con algunos trabajos realizados anteriormente en el CENIDET, como se explica a continuación:

El proyecto "*Generación de Servicios Web desde Marcos Orientados a Objetos a partir de Clases Colaborativas en Casos de Uso (MOOaSW)*" (León, 2009), plantea una estrategia alternativa para generar servicios web desde marcos de aplicaciones orientadas a objetos. La arquitectura de clases resultante, respeta las relaciones entre éstas como se definen en el MOO original. La creación de los servicios web se realiza considerando el conjunto de clases relacionadas y las funciones y datos que son utilizados únicamente en la secuencia interactiva de métodos que caracterizan a cada Caso de Uso.

El proyecto "*Benchmark de pruebas para la evaluación del sistema MOASW*" (Saldivar, 2015) , realizó una evaluación completa del sistema resultante del proyecto MOOaSW. Su objetivo fue determinar las capacidades y las deficiencias de este sistema, para realizar propuestas de mejora. Para este propósito se diseñó un experimento que incluyó un plan de

pruebas, un diseño de pruebas y cinco MOO's como casos de prueba, los cuales conforman un benchmark inicial de pruebas.

El proyecto "*Estudio Experimental para determinar la relación entre la estructura interna de Servicios Web y valores de QoS*" (Guadarrama Rogel, 2013), está dirigido a dar una respuesta relativa a la cuestión del tamaño correcto de granulación de Servicios Web. El resultado obtenido de la investigación concluye que el mejor tiempo de respuesta lo dan arquitecturas de grano grueso, es decir una única entidad de software con toda la funcionalidad requerida para producir un resultado de valor. Sin embargo, también se concluye que, desde el punto de vista de la ingeniería de software, es necesario balancear el tiempo de respuesta contra métricas de calidad para reuso y mantenimiento. Esto puede indicar que el mejor balance lo tienen aquellas arquitecturas de nivel intermedio de granulación.

El proyecto "Refactorización de sistemas legados de software, para equilibrar la coherencia, cohesión y factor de acoplamiento de su estructura interna" (Geovani, 2016), plantea un proceso que incluye métodos para lograr el equilibrio entre los factores de calidad de cohesión, coherencia y factor de acoplamiento de software legado. El resultado obtenido es un proceso de refactorización de factores de calidad de arquitectura de software orientada a objetos que está compuesto por tres métodos de refactorización: 1) Método que logra clases con todos los elementos (métodos y atributos) relacionados, es decir, alta cohesión; 2) Método que logra clases con una única responsabilidad (alta coherencia).

El proyecto "*Transformación de Marcos de Aplicaciones Orientados a Objetos hacia Marcos con Arquitectura Orientada a Servicios*" (Legorreta, 2017), considera que los servicios de grano grueso ofrecen el mejor tiempo de respuesta, según (Guadarrama Rogel, 2013), de esta manera construye servicios Web partir de la desintegración del marco orientado a objetos original, en una única clase envolvente, eliminando relaciones de agregación, herencia y dependencia. Cada clase envolvente es diseñada en suficiencia y completes con los atributos y las funciones necesarias para atender la secuencia interactiva de métodos de cada caso de uso, y éstas a la vez son convertidas en servicios Web. Para lograr la funcionalidad integrada equivalente al marco original, la integración se realiza por mecanismos de orquestación y/o coreografía.

2.4 Objetivos de la tesis

Los objetivos del presente trabajo de tesis son los siguientes:

- Obtener Marcos de Servicios Web a partir de Marcos de Aplicaciones Orientados a Objetos, por medio de la composición de clases internas, para conocimiento del comportamiento de medidas de calidad como: "Rendimiento", "Coherencia", "Cohesión" y "Autosuficiencia" en relación a las metas de valor del usuario.

- Extender la funcionalidad del "SR2 Transforming", para dar soporte a la generación de servicios web desde marcos de aplicaciones orientados a objetos.

2.5 Justificación

Entre las principales ventajas que ofrecen los Marcos de aplicaciones Orientados a Objetos están: la reducción de costos de los procesos de desarrollo de aplicaciones de software para dominios específicos y la mejora de la calidad del producto final.

Sin embargo, no es posible aprovechar los beneficios de los marcos de aplicaciones orientados por varias razones, entre estas están: los servicios están fuertemente integrados de tal forma que no pueden separarse (desfragmentarse) en funciones específicas para alguna aplicación en particular a objetos en ambientes distribuidos por la incompatibilidad de plataformas y lenguajes de desarrollo.

Para tratar esta dificultad se tendría que utilizar alguna de las estrategias que aparecen en los trabajos relacionados que van de métodos manuales, automáticos, estáticos, dinámicos, desde el código fuente o desde el modelado de sistemas, con el objetivo de la identificación y construcción de servicios Web de aplicaciones existentes, pero, a la fecha no se tiene conocimiento de que existan ambientes de reingeniería que den soporte a la transformación de Marcos de Aplicaciones Orientados a Objetos hacia Marcos con arquitectura orientada a servicios Web, que proponga diferentes estrategias de transformación para obtener ya sea servicios auto-suficientes con diferentes niveles de granulación y calidad balanceada entre rendimiento, acoplamiento, coherencia y cohesión.

Realizar una transformación manual, implica que el desarrollador deberá tener la habilidad y conocimiento para realizar la descomposición del Marco original, la selección de clases, funciones y atributos agrupándolos de alguna manera estratégica para brindar las mejores características de calidad, en este proceso existe el riesgo de introducir defectos debido al proceso de conversión manual.

2.6 Planteamiento del problema

Dado que las dimensiones de calidad de coherencia, cohesión y autosuficiencia pueden contraponerse a las dimensiones de acoplamiento y rendimiento, el problema radica en que para seleccionar las clases que integran a un componente, o servicio Web, no se tienen balanceadas estas dimensiones, de tal manera que se obtengan buenos resultados tanto en rendimiento como buenos resultados para efectos de reuso y de mantenimiento.

2.7 Alcances y limitaciones de la investigación

Los alcances obtenidos durante el presente desarrollo de investigación son los siguientes:

- Se utilizó un analizador sintáctico que permitió obtener, desde marcos orientados a objetos, la información del código fuente de un MOO.

- Se separaron, de forma automática, las clases que conformen cada caso de uso en dicho MOO.

- Se llevó a cabo la conversión automática de las clases de cada secuencia interactiva de métodos de casos de uso que corresponden a servicios Web.

- Las características soportadas por el procedimiento estuvieron basadas en los principios de la programación orientada a objetos para el lenguaje *Java*.

- Para el desarrollo de esta tesis se consideraron los enfoques de los trabajos de antecedente de Francisco León (León, 2009), tesis de Nandi Legorreta (Legorreta, 2017) y el conjunto de trabajos relacionados descritos en (§2.2).

- El método desarrollado en la tesis fue implementado en la herramienta SR2-Transforming.

- Se determinó la influencia del tiempo de respuesta en función del *throughput* los casos de uso resultantes con diferentes arquitecturas de clases.

- Los resultados fueron interpretados mediante un análisis estadístico durante el proceso de pruebas experimentales.

Las limitaciones de la investigación de desarrollo tecnológico son los siguientes:

- La herramienta desarrollada no agrega ni modifica la funcionalidad de los marcos de aplicaciones orientadas a objetos de origen.

- La herramienta no verifica los errores lógicos ni sintácticos que pudiera tener el código del marco de aplicaciones orientado a objetos de origen.

- La herramienta sólo realizó transformaciones de marcos de aplicaciones orientadas a objetos desarrollados en el lenguaje de programación *Java* versión 1.5.

- Las pruebas de la valoración de parámetros de calidad para reuso y del tiempo de respuesta, se hicieron únicamente con los casos de pruebas que se realizaron en los trabajos de antecedente (León, 2009) y (Legorreta, 2017).

- El atributo de calidad QoS de Autonomía fue comparado únicamente con los tres enfoques de transformación de servicios Web en un ambiente controlado por el investigador.

Capítulo 3: Marco Teórico

3.1 Clases internas

Una clase interna es una unidad de programa que funciona como un mecanismo de ocultación de código, ubicando la definición de una clase dentro de otra clase (Hull et al., 2000).

Una de las características importantes de las clases internas es que pueden acceder a métodos y campos de la clase contenedora como si les pertenecieran. Es decir, una clase interna tiene acceso automático a los miembros de la clase contenedora. Esto ocurre porque el objeto de una clase interna ordinaria mantiene implícitamente una referencia al objeto de la clase contenedora que lo creó (Gosling, James; Joy, Bill; Steele, Guy; Bracha, Gilad; Buckley, 2014).

La estructura de una clase interna se muestra en la Figura 1.

```
public class ClaseExterna{
    ...
    public class ClaseInterna{
        ...
    } //Fin de la ClaseInterna
} //Fin de la ClaseExterna
```

Figura 1. Estructura de una clase interna

Algunas clases tienen sentido únicamente dentro del contexto de otra clase contenedora. En la Figura 2 se muestra el esquema de la composición de una clase interna

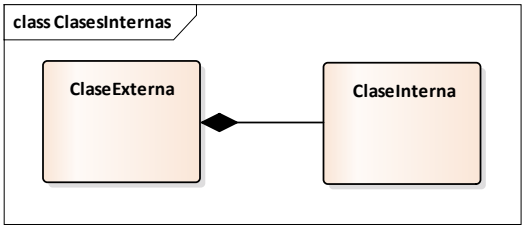


Figura 2. Composición de una clase interna

ClaseInterna tiene acceso completo a los datos y métodos de la clase que la contiene, incluyendo sus declaraciones privadas. Por definición, una clase interna puede ser declarada como estática o no-estática (Java™, 2016).

3.1.1 Clases internas estáticas (Static Nested Class)

Una clase interna se hace **estática** cuando no se necesita una conexión entre el objeto de la clase externa y el objeto de la clase interna (CompSci, 2016).

Que una clase interna sea estática quiere decir que:

1. La clase interna no necesita un objeto de la clase externa para crear un objeto de la clase interna estática.
2. La clase interna puede acceder a variables y métodos privados de la clase contenedora.
3. La clase interna está asociada con la clase contenedora.
4. La clase interna no puede hacer referencia a alguna variable de instancia de la clase contenedora.

En la Figura 3 se muestra un ejemplo de la implementación de una clase interna estática.

```
1  public class ClaseExterna {
2      //atributos de ClaseExterna
3      int x;              //variable de instancia
4      static int y = 100; //variable de clase
5      private static class ClaseInterna{ //Declara clase interna estática
6          //Constructor
7          private ClaseInterna(){ // Inicia ClaseInterna
8              System.out.println("Constructor de la clase interna");
9          }
10         private void metodoDeClaseIntena(){
11             System.out.println("Método de la clase interna");
12             //puede acceder a miembros estáticos de ClaseExterna
13             System.out.println("Acceso y=" + y);
14             //Este método no puede hacer referencia a la variable de instancia 'x'
15             // System.out.println("Acceso x=" + x);   !!Error
16         }
17     } //Fin de ClaseInterna
18     //Método de la ClaseExterna
19     public void metodoExterno(){
20         //Crea un objeto 'n' de ClaseInerna
21         ClaseInterna n = new ClaseInterna();
22         n.metodoDeClaseIntena();    //Aquí la clase contenedora invoca un método
23                                     //privado de ClaseInterna
24         System.out.println("Método externo....");
25     } //Fin del método público de ClaseExterna
26 } //Fin de ClaseExterna
```

Figura 3. Ejemplo de la implementación de una clase interna estática

Una clase interna estática no necesita inicializar una instancia de la clase externa, como los otros miembros estáticos.

Ejemplo: **ClaseExterna.y** // y es una instancia de ClaseInterna

Fuera de la clase externa, se accede a la clase interna estática a través del nombre de su clase externa contenedora.

Ejemplo: ClaseExterna.ClaseInterna n2 = new ClaseExterna.ClaseInterna();
n2.metodoInterno(); // n2 es una instancia de ClaseInterna

Una clase interna estática tiene el mismo comportamiento que algún miembro estático de la clase externa (asociada con la clase contenedora, no necesita ser instanciada).

Una clase interna estática tiene el mismo comportamiento que algún miembro estático de la clase externa (asociada con la clase contenedora, no esta instanciado).

Todas las instancias de las clases internas estáticas están asociadas con la clase contenedora. En la Figura 4 se muestra un ejemplo de asociación de clases internas estáticas.

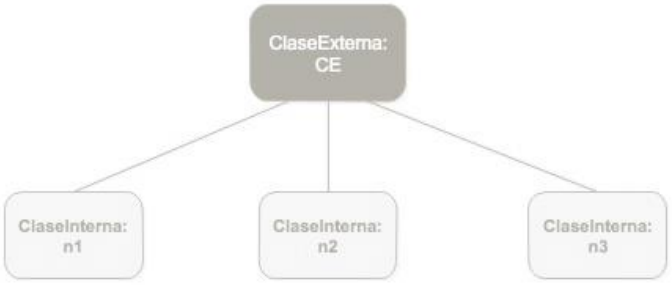


Figura 4. Asociación de la clase externa con las instancias de la clase interna estática n1, n2 y n3

3.1.2 Clases internas no-estáticas (Inner class)

A diferencia de las clases internas estáticas, cada instancia de una clase interna requiere una instancia de su clase externa contenedora, es decir, cada objeto de la clase interna requiere un objeto de la clase externa propiedad del objeto (Sharan, 2014).

Que una clase sea interna quiere decir que:

1. Los miembros de la clase interna están definidos dentro del cuerpo de la clase.
2. Una clase local es una clase definida dentro de un método.
3. Una clase local se declara implícitamente creando una variable de la misma (clase anónima).

Las clases internas no pueden declarar variables estáticas ni métodos estáticos, la clase miembro tiene acceso a todas las variables y objetos de la clase externa o cualquier otra clase interna, incluidos los miembros declarados privados. (CompSci, 2016)

En la Figura 5 se muestra un ejemplo de implementación de las clases internas no estáticas.

```
1  public class A1 { //Clase Externa
2      //variables de la clase externa
3      private int x=1;
4      private static int y=100;
5      //método de la clase externa
6      public void f(){ //inicia método externo
7          // declara una instancia de la clase interna
8          B1 b=new B1();
9          //accede a datos privados de la clase interna
10         System.out.println("b.x="+b.x);
11         //accede a la función de la clase interna
12         b.f();
13     } //termina método externo f()
14
15     //Declaración de la clase interna B1
16     public class B1{
17         private int x=10;
18         public void f(){ //inicia el método interno
19             // se refiere a la 'x' local de la clase interna
20             System.out.println("x="+x);
21             // se refuere a la 'x' de la clase contenedora
22             System.out.println("A1.x=" + A1.this.x + ", y=" + y);
23         } //termina el método interno f()
24     }//termina clase interna
25
26     public static void main(String[] args){
27         //Crea el objeto de la clase externa
28         A1 a1 = new A1();
29         //accede al método de la clase externa
30         a1.f();
31         // Crea un objeto de la clase interna B1
32         A1.B1 b1 = a1.new B1();
33         b1.f();
34     } //termina función main
35 } //termina la clase externa
```

Figura 5. Ejemplo de implementación de una clase interna no estática.

Del ejemplo de la Figura 5, la clase interna B1 hace referencia a:

- variables de la clase externa declarando: **A1.this.nombreVariable,**
- métodos de la clase externa declarando: A1.this.instanciaMetodo() o instanciaMetodo(),
- métodos estáticos declarando: metodoEstatico() o A1.metodoEstatico().

Las clases internas no pueden declarar variables estáticas ni métodos estáticos, la clase miembro tiene acceso a todas las variables y objetos de la clase externa o cualquier otra clase interna, incluidos los miembros declarados privados.

3.1.3 Clases internas locales

Una clase local es una clase definida dentro de un método (CompSci, 2016).
Que una clase interna sea clase local significa que:

- Una clase local existe hasta el final de ese método o bloque de estatutos del método que la declara internamente.
- Una clase local es usada en el caso de necesitar una clase sólo dentro de un método para hacer un trabajo especial, y no necesita ser visible en ningún otro lugar.
- Una clase local no tiene declarado un acceso especifico - el ámbito está siempre restringido al bloque en el que se declaran (no se puede declarar público, protegido, privado o estático), es decir, tiene una relación de dependencia, no de agregación fuerte o composición.
- La clase local no puede incluir variables ni métodos estáticos.
- La clase local puede acceder a los campos de la clase contenedora y a las variables locales del método en el que esta declarado (aun cuando sean declarados como ‘final’).

En la Figura 6 se muestra un ejemplo de implementación de una clase interna local.

```java
public class ClaseExterna {
    //atríbutos de la clase externa
    private final int i=10;
    private String s = "Hola";
    //inicia método publico de la clase contenedora
    public void Prueba(){
        final int j=100;
        int w = 20;
        //se declara la clase local al alcance del método Prueba externo
        class ClaseLocalInterna{
            //inicia el método privado Prueba de la clase ClaseLocalInterna
            private void Prueba(){
                System.out.println("i="+i);
                System.out.println("s="+s);
                System.out.println("j="+j);
                //también puede tener acceso a la variable 'w'
                //por estar en el ámbito del método Prueba externo
                System.out.println("w="+w);
            } //termina métdo local Pueba
        } //Termina la clase interna local
        // declara una instancia de la ClaseLocalInterna
        ClaseLocalInterna c1 = new ClaseLocalInterna();
        c1.Prueba();
    } //termina el método público de la clase externa
    //inicia función main
    public static void main(String[] args){
        ClaseExterna c = new ClaseExterna();
        //accede al método Prueba de la clase contenedora: ClaseExterna
        c.Prueba();
    } //termina función main
} //termina la clase contenedora ClaseExterna
```

Figura 6. Ejemplo de implementación de una clase interna no estática - local

Observaciones de la Figura 6:
La *claseExterna* tiene acceso a todas las instancias.
El método *prueba()* de la *claseExterna* puede tener acceso a variables final.

3.1.4 Clases internas anónimas

Una clase interna anónima es una clase local que no tiene nombre, sino que es declarada de manera explicita al crear una variable con el mismo nombre.

Que una clase interna sea clase anónima significa que:

- La clase anónima se crean mediante una expresión que combina la creación de objetos con la declaración de la clase.
- Las clases anónimas son comúnmente usadas en librerías AWT (Abstract Window Toolkit)

- El súper tipo de una clase anónima puede ser una interface (que la clase anónima implementa) o una clase (que la clase anónima extiende).
- El cuerpo de la clase anónima puede definir métodos pero no puede definir ningún constructor.

En la Figura 7 se muestra un ejemplo de implementación de una clase interna anónima.

```
1  public class claseExterna{
2      //atributos de la clase
3      private Vector items;
4      //método de la clase que retorna una instancia de una
5      //nueva clase que implementa la interface Enumerador
6      public Enumerador enumerador(){
7          return new Enumerador(){    //inicia declaración
8              int aItem = items.size() -1;
9
10             public boolean tieneElementos(){
11                 return (aItem>=0);
12             }
13
14             public Object sigElemento(){
15                 if(tieneElementos())
16                     throw new SinElementos();
17                 else
18                     return items.elementoAt(aItem--)
19             }
20         } //termina declaración de clase anónima
21     };  // ---> Aqui requiere punto y coma " ; "
22
23     //En este caso la clase Enumerador no esta declarada
24     // a este tipo de sentencia se le llama clase anónima
25     class claseA implements Enumerador{
26         int aItem = items.size() -1;
27     } //termina la clase que implementa Enumerador
28 } //termina la clase externa
```

Figura 7. Ejemplo de implementación de una clase interna no estática - anónima

Observaciones de la Figura 7:
La claseA implementa la interface Enumerador, en este momento retorna una instancia de una nueva clase que implementa.

3.2 Marco Orientado a Objetos

3.2.1 Definición de MOO

Un Marco Orientado a Objetos es un conjunto semi-completo de clases en colaboración que incorpora un diseño genérico, el cual puede adaptarse a una variedad de problemas específicos para producir nuevas aplicaciones hechas a la medida (Mattsson et al., 1997).

3.2.2 Estructura de los MOO

La estructura principal de un MOO esta formado por una clase abstracta para cada componente principal. A partir de ella se incluye el resto de las clases, mediante relaciones de asociación, agregación, composición o herencia.

Los patrones de diseño se utilizan como apoyo para estructurar correctamente las clases del marco. Un patrón de diseño es una solución a un problema que se usa repetidamente en contextos similares con algunas variantes en la implementación (Kimmel, 2008).

Los elementos más importantes de un marco orientado a objetos son:

- Reusabilidad
- Dominio de aplicación
- Diseño extensible
- Colaboración entre clases

3.2.2.1 Reusabilidad

Al conjunto de problemas que pueden ser resueltos de manera similar -ya sea por la metodología empleada, los datos requeridos, etc. – se le llama familia de problemas relacionados. Cada MOO se crea para dar solución a una de estas familias.

De esta manera el mismo marco se usa una y otra vez para dar solución a los problemas de la familia, únicamente haciéndole las adaptaciones particulares que para cada problema se requieran.

3.2.2.2 Dominio al que se dirige

La información plasmada en un MOO pertenece a un dominio específico. Por ejemplo, si se requieren resolver problemas de ámbito estadístico, como la media, mediana, varianza,

regresión lineal, etc. se puede crear un MOO que contenga la manera de realizar estos cálculos, y se le puede llamar marco de la estadística.

Sin embargo, un marco puede no considerar a un dominio completo, sino sólo parte de él. Así, se puede contar con varios marcos que cubran a todo el dominio.

3.2.2.3 Diseño extensible

Un MOO debe contar con una estructura de diseño general o abstracta, con el objeto de que al ser utilizado para resolver un problema en particular pueda adaptarse a él. Esta adaptación consiste en asignar un comportamiento específico a un elemento del marco, que originalmente tiene un comportamiento predeterminado o incluso no lo tiene. La herencia es un mecanismo que permite extender el marco.

La herencia consiste en que una clase (subclase) puede ser definida tomando los atributos y el comportamiento de otra (superclase). La extensibilidad consiste en que la superclase permite a la subclase implementar comportamiento adicional al heredado. El principio de herencia es uno de los más importantes en los MOO, pues es así como se permite particularizar la solución a un problema partiendo del diseño abstracto que el marco proporciona. En el proceso de maduración del marco, las subclases se agregan gradualmente; durante la utilización del marco, las superclases siempre están en el marco y las subclases las creas el cliente del mismo.

3.2.2.4 Colaboración entre clases

Un MOO está integrado por varias clases. Cada clase tiene definido cierto comportamiento. Para que el marco realice eficientemente las funciones para las que se pretende crear, las clases que lo componen deben colaborar de la manera más adecuada, utilizando apropiadamente los mecanismos de abstracción y herencia; y tener establecidas correctamente las relaciones de asociación, agregación y composición.

3.3 Marco de Servicios Web

3.3.1 Definición de Marco de Servicios Web

En el contexto de esta investigación, un Marco de Servicios Web se define como un conjunto de servicios web independientes, que contienen la experiencia y el conocimiento para satisfacer las necesidades de aplicaciones de dominios (Santaolaya et al., 2016).

3.3.2 Arquitectura Orientada a Servicios (SOA)

SOA es un entorno para descubrir y utilizar dinámicamente servicios en una red. Se basa en el principio de separar la interfaz de la implementación del servicio. Los servicios se ven simplemente como puntos finales que tratan las solicitudes de los consumidores que siguen un contrato específico representado por su interfaz (Thomas, 2007). La tecnología de implementación del servicio y la forma en que ejecuta las tareas es completamente transparente para los consumidores. Las características de SOA se basan principalmente en dos principios:

- **Interoperabilidad**: es la capacidad de los sistemas que utilizan diferentes plataformas e idiomas para comunicarse entre sí (Bianchini, Cappiello, De Antonellis, & Pernici, 2014). Cada servicio proporciona una interfaz que se puede invocar a través de un tipo de conector. Un conector interoperable consiste en un protocolo y un formato de datos que entiende cada uno de los clientes potenciales del servicio. La interoperabilidad se logra al admitir el protocolo y los formatos de datos de los clientes del servicio. La especificación de mediación mapea el formato de datos interoperable a los formatos de datos específicos de la plataforma (Zur Muehlen, Nickerson, & Swenson, 2005).

- **Acoplamiento**: SOA promueve un acoplamiento flexible entre los consumidores de servicios y los proveedores de servicios. Un grado de acoplamiento del sistema afecta directamente su modularidad . Esto implica que cuanto más estrechamente acoplado esté un sistema, al momento de realizar un cambio en un servicio, requerirá cambios en los consumidores del servicio. El acoplamiento se incrementa cuando los consumidores del servicio requieren una gran cantidad de información sobre el proveedor del servicio para usarlo. Si el consumidor del servicio no necesita un conocimiento detallado del servicio antes de invocarlo, el consumidor y el proveedor están más relajados. Asi, un consumidor solicita una referencia a un registro de terceros sobre el servicio que desea utilizar (Josuttis., 2007).

Las dependencias de comunicación entre consumidores y servicios deben limitarse conforme el contrato de servicios (interfaz). La separación entre la interfaz del servicio y su implementación permite que los servicios Web interactúen sin necesidad de compartir un entorno común de ejecución. Debido a esta opacidad, los servicios se vuelven autónomos ya que pueden elegir libremente modelos e idiomas, entornos de implementación y sustituir una implementación de servicio por otra (Mcgovern, Tyagi, Stevens, & Mathew, 2003).

En la Figura 8 se muestra un ejemplo de una aplicación de reservación de libros (consumidor) que solicita a un registro un servicio que realiza la reserva de un libro. El registro devuelve todas las entradas que admiten dicho servicio. Las entradas también contienen información sobre el servicio, incluyendo el Autor, Editorial y año de publicación. El consumidor selecciona el servicio (proveedor) de la lista en función del año de publicación. Utilizando la referencia de servicio de la entrada del registro, el consumidor se une al proveedor del servicio de reservación de libro utilizando un proxy.

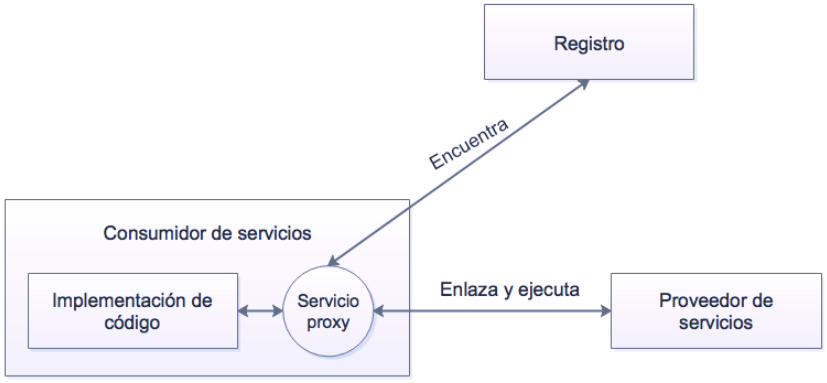


Figura 8. Proxy de servicio (Mcgovern et al., 2003)

El consumidor del servicio ejecuta la solicitud llamando a una función API en el proxy. El proxy es simplemente una referencia local a una implementación de servicio remoto. Si el proxy cambia la interfaz del servicio remoto, técnicamente ya no es un proxy. Un proxy de servicio está escrito en el idioma nativo del consumidor del servicio. Por ejemplo, un proxy para un proveedor de servicios bien definido puede estar en *Java* o *Visual Basic* (Mcgovern et al., 2003).

3.3.2.1 Entidades SOA

La arquitectura SOA esta formada por las siguientes entidades:

- Los proveedores de servicios. Publican la disponibilidad de sus servicios
- Los intermediarios de servicios. Registran y categorizan los servicios publicados y brindan servicios de búsqueda.
- Los solicitantes de servicios. Usan servicios intermediario para localizar un servicio necesario y luego usan ese servicio

En la Figura 9 se representan las tres entidades de la arquitectura SOA.

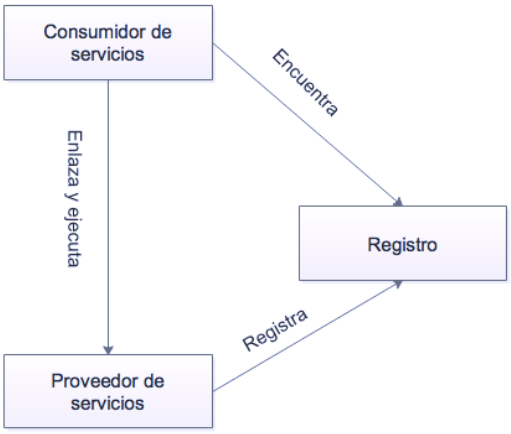


Figura 9. Entidades de la arquitectura SOA

El contrato de servicio (o interfaz) es una especificación de la forma en que un consumidor de un servicio interactúa con el proveedor del servicio. Un contrato de servicio puede requerir un conjunto de condiciones previas y posteriores. El contrato también puede especificar Calidad de Servicio (QoS). Los niveles de QoS son especificaciones para los aspectos no funcionales del servicio. Por ejemplo, un atributo de calidad de servicio es la cantidad de tiempo que lleva ejecutar un método de servicio.

3.3.3 Servicios Web

Un servicio web es una unidad lógica para el intercambio de datos con otras unidades lógicas a través de una red usando un grupo de protocolos y estándares tales como XML, WDSL y SOAP (Vinay & Jain, 2012).

Los sistemas que tienen que comunicarse con otros sistemas usan protocolos de comunicación y formatos de datos que ambos sistemas entienden. Algunas aplicaciones basadas en servicios se pueden utilizar protocolos de comunicación como : SOAP y RESTful.

3.4 Patrones de diseño

Los patrones de diseño pueden utilizarse para lograr un buen diseño del marco. Los patrones de diseño son "descripciones para comunicar objetos y clases que son adaptados para resolver un problema de diseño general en contextos particulares" (Erich, Helm, Johnson, & Vlissides, 2002). Esta comunicación entre objetos y clases se da a través de las relaciones de asociación, agregación y composición.

Actualmente existen catálogos con una gran cantidad de patrones, dirigidos a la solución de problemas diversos. El catálogo de Gamma (Erich et al., 2002), divide a los patrones en tres categorías:

- **De creación**. Hacen abstracto el proceso de instanciación, de forma que el sistema cliente se independiente de cómo se crean, componen y representan sus objetos.
- **Estructurales**. Se utilizan para definir cómo se componen las clases y objetos para formar estructuras más grandes.
- **De comportamiento**. Se utilizan para definir algoritmos y la asignación de responsabilidades entre los objetos. No sólo definen patrones de objetos o clases, también definen patrones de comunicación entre ellos.

En la Tabla 2 se muestra el catálogo de patrones de diseño de Gamma clasificados en categorías.

Tabla 2. Patrones de diseño del catálogo de Gamma (Erich et al., 2002)

		De creación	Estructurales	De comportamiento
Ambito	Clase	- Factory Method	- Adapter	- Interpreter - Template method
	Objeto	- Abstract factory - Builder - Prototype - Singleton	- Adapter - Bridge - Composite - Decorator - Façade - Flyweight - Proxy	- Chain of responsability - Command - Iterator - Mediator - Memento - Observer - State - Strategy - **Visitor**

3.4.1 Patron de diseño Visitor

Representa una operación para ser ejecutada sobre una estructura. El patron de diseño visitor deja que se defina una nueva operación sin cambiar las clases de los elementos sobre los que opera (Gamma, Helm, Johnson, & Vlissides, 2002).

En la Figura 10 se muestra el ejemplo de un diagrama de clases del patron visitor.

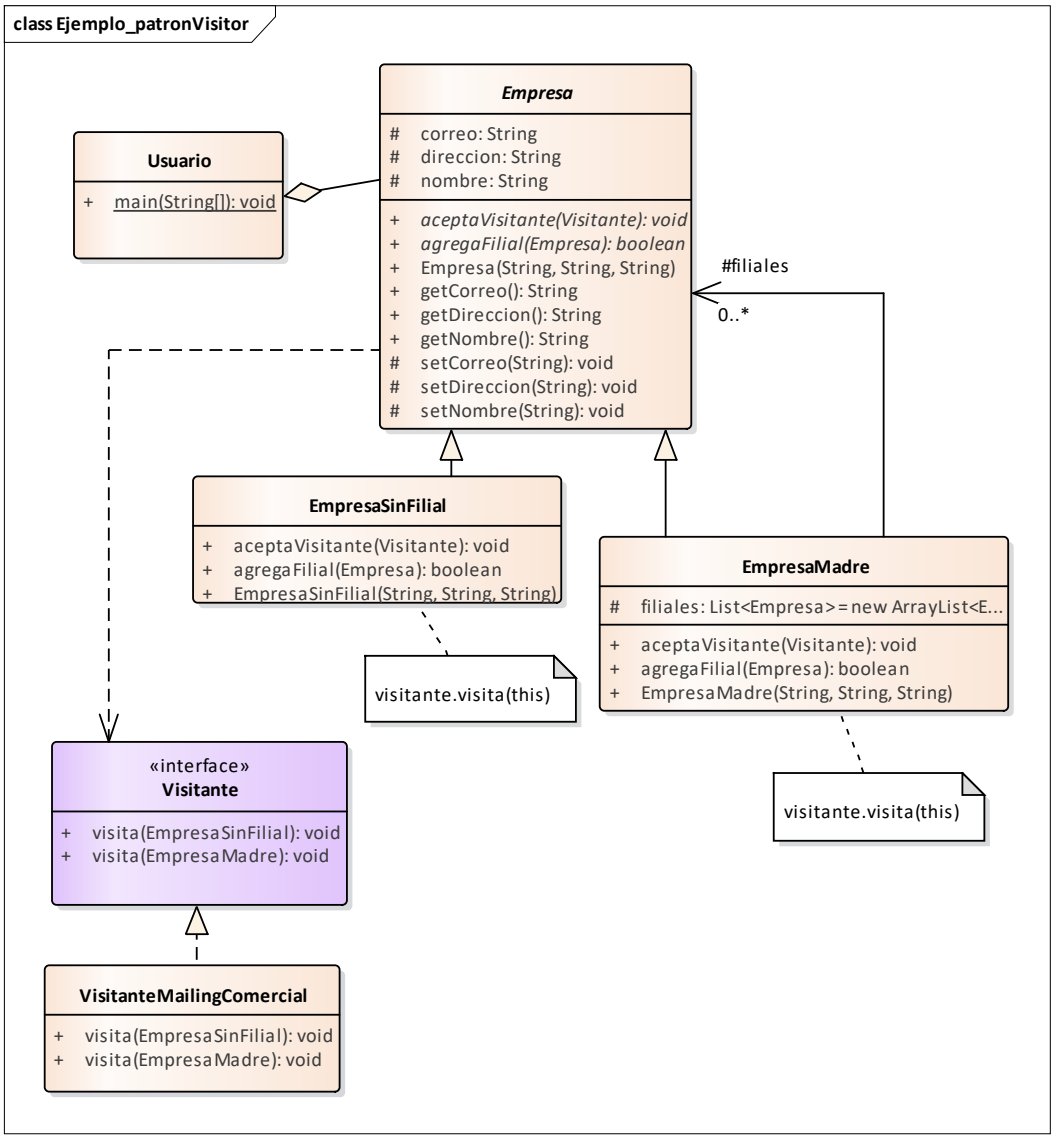


Figura 10. Diagrama de clases ejemplo del patrón Visitor

Implementación.
Este patrón debe utilizarse cuando (Laurent, 2005):

- Una estructura de objetos contenga muchas clases de objetos con distintas interfaces y se desea llevar a cabo operaciones sobre estos objetos que son distintas en cada clase concreta.
- Se quieren llevar a cabo muchas operaciones dispares sobre objetos de una estructura de objetos sin tener que incluir dichas operaciones en las clases.
- Las clases que definen la estructura de objetos no cambian, pero las operaciones que se llevan a cabo sobre ellas si.

Dado que patrón Visitor separa un algoritmo de la estructura de un objeto, es ampliamente utilizado en intérpretes, compiladores y procesadores de lenguajes, en general.

Las Figuras 11, 12, 13 y 14 muestra un ejemplo de la implementación patrón de diseño Visitor.

```
1  public abstract class Empresa {
2      protected String nombre, correo, direccion;
3      //Constructor de la clase Empresa
4      public Empresa(String nombre, String correo, String direccion){
5          this.setNombre(nombre);
6          this.setCorreo(correo);
7          this.setDireccion(direccion);
8      }
9      //Métodos get y set
10     public String getNombre(){
11         return nombre;
12     }
13     protected void setNombre(String nombre){
14         this.nombre=nombre;
15     }
16     public String getCorreo(){
17         return correo;
18     }
19     protected void setCorreo(String correo)
20     {
21         this.correo = correo;
22     }
23     public String getDireccion(){
24         return direccion;
25     }
26     protected void setDireccion(String direccion){
27         this.direccion = direccion;
28     }
29     //Métodos abstractos
30     public abstract boolean agregaFilial(Empresa filial);
31     public abstract void aceptaVisitante(Visitante visitante);
32 }
```

Figura 11. Elemento Abstracto de la implementación del patrón de diseño Visitor

```java
public class EmpresaSinFilial extends Empresa {
    //Constructor de la clase EmpresaSinFilial
    public EmpresaSinFilial(String nombre, String correo, String direccion){
        super(nombre, correo, direccion);
    }
    public void aceptaVisitante(Visitante visitante){
        visitante.visita(this);
    }
    public boolean agregaFilial(Empresa filial){
        return false;
    }
}
```

```java
public class EmpresaMadre extends Empresa {
    protected List<Empresa> filiales = new ArrayList<Empresa>();
    //constructor de la clase
    public EmpresaMadre(String nombre, String correo, String direccion){
        super(nombre, correo, direccion);
    }
    //método aceptaVisitante
    public void aceptaVisitante(Visitante visitante){
        visitante.visita(this);
        for(Empresa filial: filiales)
            filial.aceptaVisitante(visitante);
    }
    //otro método de la clase
    public boolean agregaFilial(Empresa filial){
        return filiales.add(filial);
    }
}
```

Figura 12. Elementos concretos de la implementación del patrón de diseño Visitor

```java
//Interfaz del Visitante
public interface Visitante {
    //métodos a visitar
    void visita(EmpresaSinFilial empresa);
    void visita(EmpresaMadre empresa);
}
//Elementos concretos del Visitante
public class VisitanteMailingComercial implements Visitante{
    //implementa al Visitante: visita para la empresa sin Filial
    public void visita(EmpresaSinFilial empresa){
        System.out.println("Envía un correo a " + empresa.getNombre() + ", dirección: "
            + empresa.getCorreo() + ", propuesta comercial para su empresa...");
    }
    //implementa al Visitante: visita para la empresa Madre
    public void visita(EmpresaMadre empresa){
        System.out.println("Envía un email a " + empresa.getNombre() + ", dirección: "
            + empresa.getDireccion() + ", Propuesta comercial para su grupo...");
    }
}
```

Figura 13. Interface del visitante concreto de la implementación del patrón de diseño Visitor

```
81  //El cliente
82  public class Usuario {
83      public static void main(String[] args) {
84          //Agrega empresas sin filial
85          Empresa oEmpresa1 = new EmpresaSinFilial
86              ("Empresa #1", "infoSomosNo1@empresaNo1.com","Calle de la empresa no.1");
87          Empresa oEmpresa2 = new EmpresaSinFilial
88              ("La empresa X", "infoSomosX@empresaX.com","Calle de la empresa X");
89          Empresa oEmpresa3 = new EmpresaSinFilial
90              ("La empresa YZ", "infoSomosX@empresaYZ.com","Calle de la empresa YZ");
91          //agrega grupos de otras empresas
92          Empresa grupo1 = new EmpresaMadre
93              ("grupo1","infoGrupo@grupo1.com","calle del grupoNo.1");
94          Empresa grupo2 = new EmpresaMadre
95              ("grupo2","infoGrupo@grupo2.com","calle del grupoNo.2");
96          //agregaFilial grupo1
97          grupo1.agregaFilial(oEmpresa1);
98          grupo1.agregaFilial(oEmpresa2);
99          //agregaFilial grupo2
100         grupo2.agregaFilial(grupo1);
101         grupo2.agregaFilial(oEmpresa3);
102         //********* Acepta visitante
103         grupo2.aceptaVisitante(new VisitanteMailingComercial());
104     }
105 }
```

Figura 14. Cliente de la implementación del patrón de diseño Visitor

Observaciones de las Figuras 11, 12, 13 y 14:

- **Visitor** (*public interface Visitante*): declara una operación de "visita" para cada uno de los elementos concretos de la estructura de objetos. Esto es, el método visita().

- **VisitorConcreto** (*public class VisitanteMailingComercial*): implementa cada una de las operaciones declaradas por Visitor.

- **Element** (*public abstract class Empresa*): define la operación que le permite aceptar la visita de un Visitor.

- **ConcreteElement** (*public class EmpresaSinFilial* y *public class EmpresaMadre*): implementa el método accept() que se limita a invocar su correspondiente método del Visitor. El ConcreteElement ejecuta el método de visitar y se pasa a sí mismo como parámetro.

- **ObjectStructure** (*public class Usuario*): gestiona la estructura de objetos y puede ofrecer una interfaz de alto nivel para permitir a los Visitor visitar a sus elementos.

Capítulo 4: Análisis y Diseño del Método de Transformación MOOinMS

4.1 Introducción a MOOaSW

Un Marco Orientado a Objetos es un conjunto semi-completo de clases en colaboración que incorpora un diseño genérico, el cual puede adaptarse a una variedad de problemas específicos para producir nuevas aplicaciones hechas a la medida (Mattsson et al., 1997).

MOOaSW es una herramienta de reingeniería de software que transforma, de manera semi-automática, un Marco Orientado a Objetos a Servicios Web (León, 2009).

MOOaSW implementa un procedimiento para transformar el código legado en formato *Java*. El elemento en el que se basa la generación de los servicios Web es el caso de uso. Cada servicio Web debe contener el código necesario para satisfacer un caso de uso identificado en el marco. La identificación de los casos de uso dentro del código del marco se realiza mediante un análisis estático al código.

El proceso se compone de tres pasos principales:

1. **Analizar el marco orientado a objetos.** Se obtiene la información de los elementos del código del marco.

2. **Definir el código necesario para satisfacer cada caso de uso**. Se selecciona el código del marco para cada caso de uso.

3. **Generar los servicios Web**. Se genera el código de cada servicio Web con base en el código seleccionado en el paso anterior.

El diagrama de procesos de MOOaSW se muestra en la Figura 15:

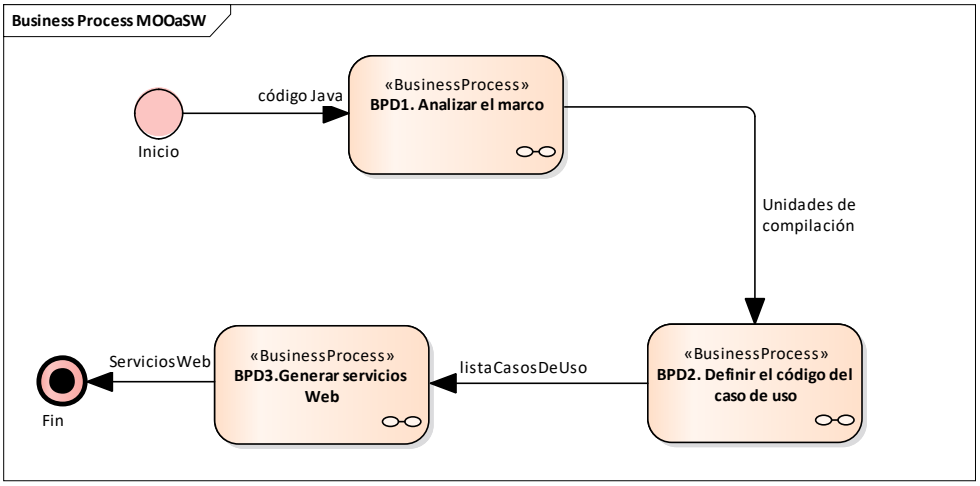


Figura 15. Diagrama de procesos de MOOaSW

El diagrama de subprocesos de BPD1. Analizar el marco se muestra en la Figura 16.

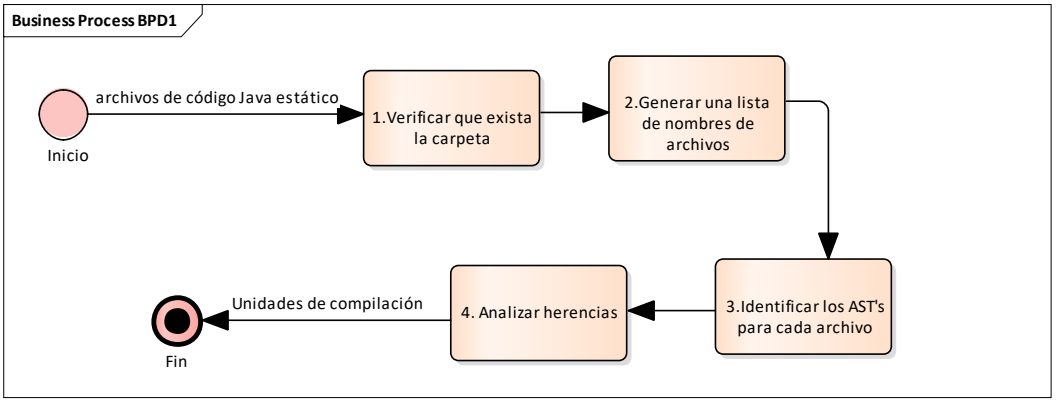


Figura 16. Diagrama de subprocesos del proceso BDP1

El diagrama de subprocesos de BPD2. Definir el código del CU se muestra en la Figura 17.

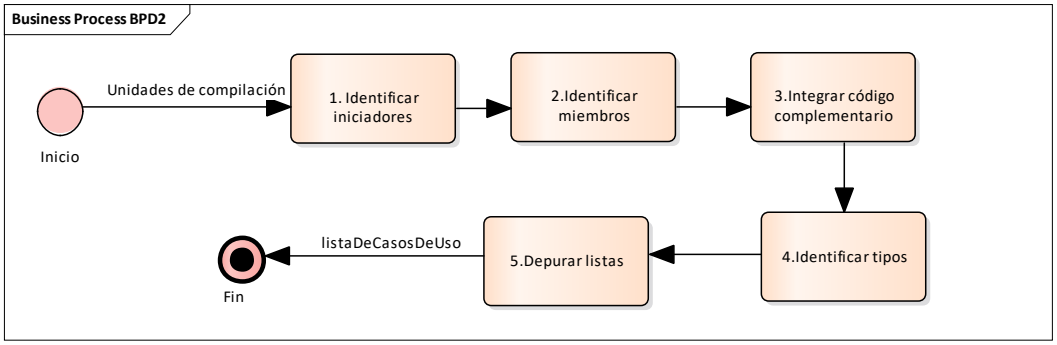


Figura 17. Diagrama de subprocesos del proceso BPD2

El diagrama de subprocesos de BPD3. Generar servicios Web se muestra en la Figura 18.

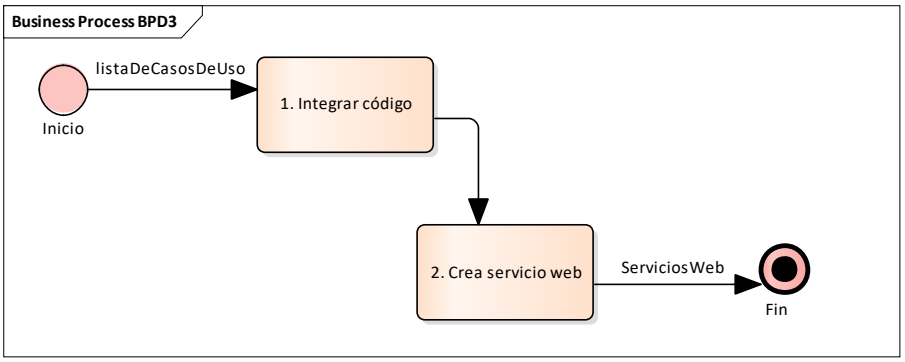


Figura 18. Diagrama de subprocesos del proceso BPD3

A partir del diagrama de procesos principal de la herramienta MOOaSW, se diseñó el diagrama de casos de uso. En la Figura 19 se muestra el diagrama de uso de la herramienta.

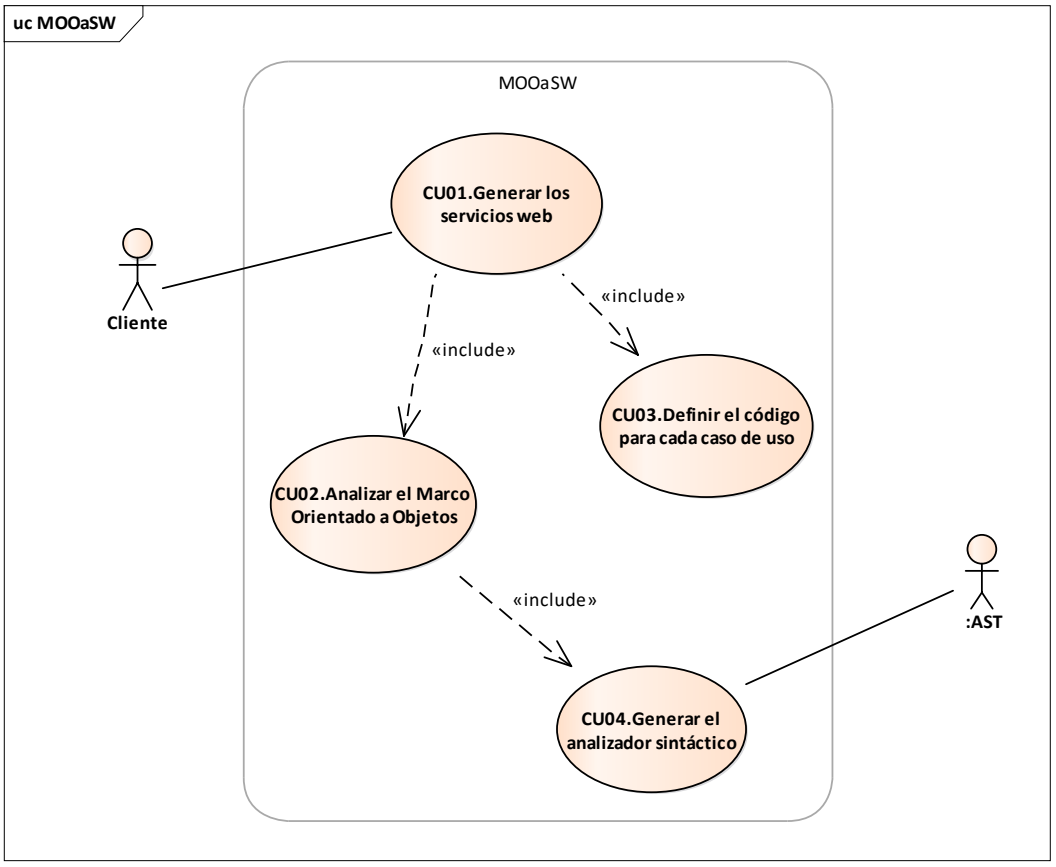


Figura 19. Diagrama de casos de uso de la herramienta MOOaSW

4.1.1 Analizar el marco orientado a objetos

Este proceso consiste en ejecutar un analizador sintáctico sobre el código del MOO y obtener un repositorio con la información de cada elemento de clases definidas en el marco. A este repositorio se le denomina Diccionario de datos, propuesto en (León, 2009).

El MOO deberá estar desarrollado en el lenguaje *Java* y el código deberá estar libre de errores sintácticos y un funcionamiento correcto. Este proceso requiere de las siguientes acciones:

a) Selección del MOO que debe analizarse desde la ruta de directorio en el que se crearán los servicios Web. El usuario deberá proporcionar la estructura de directorios donde se encuentra el código del MOO.

b) Análisis sintáctico al código del MOO de entrada. Cada archivo *.java* del marco se procesará con un analizador sintáctico de lenguaje *Java*. La información que deberá almacenarse consistirá en:

    1) Información básica de las clases: nombre calificado de la clase, modificador de acceso, nombre del archivo en el que se encuentra definida, nombre de su clase base e interfaces cuando aplique e información de sus constructores, atributos, métodos y clases internas.
    2) Información básica para las interfaces: nombre calificado de la interfaz, nombre del archivo en el que se encuentra definida, nombre de las interfaces que implementa (cuando aplique) e información de sus atributos.
    3) Información de constructores de una clase: nombre del constructor, firma del constructor y clase origen.
    4) Información de atributos de clases e interfaces: modificador de acceso, nombre del atributo, clase origen y tipo de atributo.
    5) Información de métodos de una clase: modificador de acceso, nombre del método, firma del método, tipo de retorno y clase origen.
    6) Información de clases internas: será la misma que en el inciso 1) y además la clase base.
    7) Información de interfaces internas es la misma que el inciso 2) y además la clase origen.
    8) Información de inicializadores sintácticos: es solo la línea y columna en donde se encuentra.
    9) Información de enumeraciones: son los números de línea y columna en donde se encuentra, y la cantidad de valores que posea.

4.1.2 Definir el código para cada caso de uso

Consiste en seleccionar las clases (y sus elementos) del marco orientado a objetos que son necesarias para satisfacer cada caso de uso encontrado en el mismo. Los elementos de las clases son: constructores, métodos inicializadores, enumeraciones y miembros de clase (atributos, métodos, clases internas e interfaces internas).

Se debe contar con el diccionario de datos resultante de analizar el marco orientado a objetos. Al finalizar este paso se tendrán dos listas para cada caso de uso:

1. Una lista que contiene las clases e interfaces que cada caso de uso requiere, llamada *lista de clases*.

2. Para cada clase de la lista anterior, una lista de los elementos utilizados en el caso de uso. Se llamarán *listas de elementos de clase*. No incluirán los elementos que, aunque pertenezcan a la clase, no sean útiles al caso de uso.

Para definir el código para cada caso de uso se deberán integrar correctamente las listas mencionadas con la información de las clases y de sus elementos. Para ello, se efectuarán las siguientes acciones:

a) Identificar los métodos iniciadores de cada caso de uso.
b) Identificar precondición del caso de uso.
c) Identificar los métodos de caso de uso y miembros de clase utilizados.
d) Identificar las relaciones de agregación, composición y dependencia.
e) Identificar cláusulas de manejo de excepciones utilizados.
f) Depurar el contenido de las listas.
g) Agrupar el código del caso de uso a una única entidad.

La realización de las primeras cuatro acciones da como resultado clases y sus elementos que se agregarán a las correspondientes listas de cada caso de uso. El inciso *(c)* tiene el objetivo de garantizar que el código seleccionado para satisfacer cada caso de uso funcione correctamente en el servicio Web, al agregar las declaraciones que sean indispensables. El inciso *(d)* tiene a cargo identificar el anidamiento de clases internas que componen a la clase contenedora a través de la lista de relaciones identificadas en los atributos de clase y declaración de variables

de los métodos participantes en la secuencia interactiva del caso de uso. La ultima acción *(g)* tiene como objetivo agrupar toda la funcionalidad del caso de uso en una única clase contenedora, partiendo de esta manera el método de transformación presentado en este trabajo con el nombre *MOOinMS*.

4.1.3 Generar servicios Web

A partir de la incorporación de una única clase de objetos (Clase Iniciadora de caso de uso) de todos los elementos y miembros que lo conforman, obtenidos en el paso anterior, para cada caso de uso se generará un servicio Web cuyo nombre se obtendrá con base en el nombre del método iniciador y el nombre calificado de su clase. Para cada caso de uso se crea un directorio y se coloca en él a los archivos que contengan el código seleccionado para satisfacer el caso de uso. Como se describe en la tesis de antecedente "*Generación de Servicios Web desde Marcos Orientados a Objetos a partir de Clases Colaborativas en Casos de Uso (MOOaSW)"* (León, 2009).

4.2 Integración del AST (Árbol de Sintaxis Abstracto)

La herramienta MOOaSW integra la estructura de un **AST** (Árbol de Sintaxis Abstracto) que genera el *parser* para obtener información específica del código del marco. Los AST que se generan contemplan el uso de aproximadamente 100 clases/interfaces que representan los posibles tipos de los nodos que los integran (León, 2009).

El AST desintegra el código fuente *Java* en forma de árbol. Este árbol es más conveniente y fiable para analizar y modificar programáticamente que sólo archivos de texto plano (Kuhn & Thomann, 2010).

La clase base de todos los nodos del AST es la clase *Node*. Todos los AST tienen como raíz a un nodo de tipo *CompilationUnit*, que representa la unidad de compilación en *Java*, en este caso, un archivo *.java*. Este nodo raíz tiene como hojas a:

- ***PackageDeclaration***. Representa la declaración del paquete al que pertenece la unidad de compilación.
- ***ImportDeclaration***. Representan las declaraciones *import* de la unidad de compilación.
- ***TypeDeclaration*** (clase abstracta) que representa las declaraciones de tipos de la unidad de compilación.

La Figura 20 presenta un AST de la estructura básica de un archivo de *Java*:

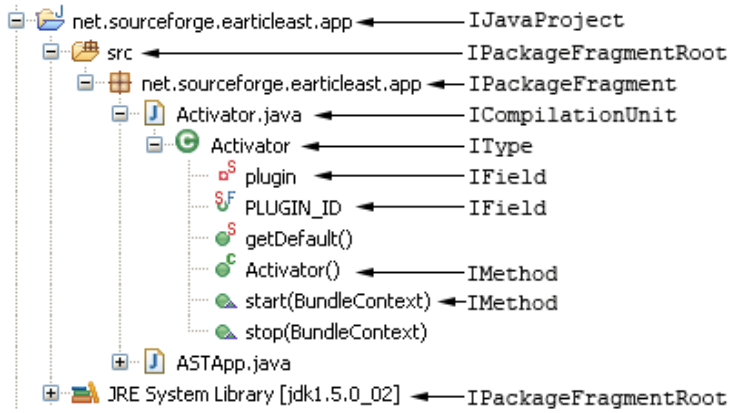


Figura 20. Estructura básica del AST de un archivo java

El diseño arquitectural del AST se realizó utilizando el patrón de diseño *Visitor*, por lo cual todas poseen sus correspondientes métodos *accept*. La implementación de estos métodos, de acuerdo al patrón *Visitor*, se realiza en clases separadas, las cuales son los *visitantes*. El código obtenido en Internet proporciona una interfaz llamada *VoidVisitor* que declara todos los métodos que deben implementarse en sus clases derivadas. Todos estos métodos son sobrecargados y su nombre es *visit*. La diferencia es que en su firma cada método recibe un tipo diferente que deriva de la clase *Node*, es decir, recibe un nodo de un tipo específico. De esta manera, en la implementación cada método *visit* realizar acciones a partir del nodo que se reciba como parámetro (León, 2009).

4.3 Criterio de identificación de casos de uso

Se considera que la invocación de un método público de una clase pública del MOO representa el inicio de una solicitud de un requerimiento por el cliente. Entonces, un caso de uso representa la meta de una interacción entre un actor y el resultado que espera del sistema.

Para identificar un caso de uso (CU) se emplean los siguientes criterios:

a) Si una clase es pública significa que está disponible a clientes del MOO, otorgando posibilidad de acceder a ésta y a sus elementos.

b) Todos los métodos públicos de las clases públicas no abstractas del MOO son iniciadores de casos de uso.

c) Cuando un método iniciador invoca a otro método se inicia una secuencia de interacción, la cual puede continuar hasta que el último método invocado de la secuencia ya no invoca a ningún otro método.

d) Se considera que el extremo final de un caso de uso sucede cuando un método iniciador termina satisfactoriamente consiguiendo la meta del Caso de Uso.

En la Figura 21 se representa mediante un grafo dirigido a un ejemplo de secuencias de interacción entre métodos.

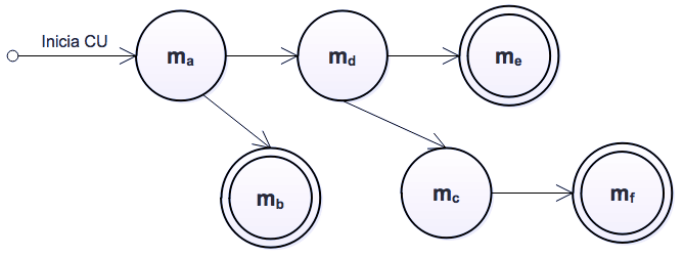


Figura 21. Ejemplo de grafo difigido de una secuencia de interacción entre métodos

En el grafo de la Figura 4, se considera que el método m_a es público y que pertenece al comportamiento de una clase pública no-abstracta, por lo tanto, m_a es una entrada para este caso de uso, en el cual existen dos secuencias de interacción. En este ejemplo, cada secuencia de interacción representa un escenario alternativo que lleva a la meta de valor del caso de uso, partiendo del método iniciador m_a. Para este ejemplo, cada una de las secuencias termina en m_b, m_e y m_f.

A los métodos que integran las secuencias de interacción encontradas se les llamará *métodos del caso de uso*.

Los métodos del caso de uso, además de los métodos que invocan, utilizan a otros miembros de clase durante su ejecución. Estos miembros son: atributos, clases internas e interfaces internas, que también son necesarios para la satisfacción del caso de uso. Los constructores se consideran también métodos del caso de uso y forman nuevas secuencias de interacción.

Con base en lo anterior, el código necesario para satisfacer el caso de uso iniciado por un método *m* perteneciente a una clase es el correspondiente a:

1. El método *m*.
2. Los métodos del caso de uso. Éstos son los métodos integrantes de todas las secuencias de interacción $S_1, S_2,... ,S_n$ a partir de *m*.
3. Los miembros de clase *x1, x2,... xn* que los métodos del caso de uso utilicen.
4. Las declaraciones de las clases a las que pertenecen los métodos del caso de uso (punto 2) y los miembros de clase utilizados (punto 3).
5. La declaración *package* y las declaraciones *import* utilizadas (en caso de existir).

4.4 Criterio de identificación de las relaciones de agregación, composición y dependencia

Una clase describe un conjunto de objetos similares que comparten una estructura común y un comportamiento común.

Un objeto es una unidad atómica que encapsula el estado y el comportamiento identificados en la clase. Los objetos son los elementos que representan la solución en un espacio del problema. (Hull et al., 2000)

Dado que una clase describe a un conjunto de objetos con características (datos) y comportamientos (funcionalidad) idénticos, una clase es realmente un tipo de datos de manera similar al tipo *float*, por ejemplo, que también tiene un conjunto de características y comportamientos comunes para las variables de este tipo.

Las clases encapsulan atributos y métodos de tal modo que sólo se hace visible una parte de esos atributos y métodos, los estrictamente necesarios para que podamos trabajar con las instancias de esa clase.

Para un dominio de problema específico las clases pueden estar relacionadas formando una estructura del diseño de las clases. (Sierra & Casado, 2011)

Existen tres tipos básicos de relaciones entre clases:

1. **Asociación**. Denota alguna dependencia semántica entre clases.
2. **Agregación y Composición**. Denota una relación <<es parte de>>
3. **Generalización y Especialización**. Denota una relación << es un >>

4.4.1 Asociación

La **asociación** describe una relación entre dos clases. Sólo denota una dependencia semántica y no establece la dirección de esta dependencia (a menos que se diga lo contrario, una asociación implica relación bidireccional) ni establece la forma exacta en que una clase se relaciona con otra (sólo puede denotarse esta semántica nombrando el papel que desempeña cada clase en relación con la otra).

Existen tipos de cardinalidad en una asociación:

- Uno a uno
- Uno a muchos
- Muchos a muchos

Las relaciones de asociación deben usarse cuando se prevé una temporalidad alta en la relación de dos objetos. De conformidad con (Hull et al., 2000), las asociaciones tienen las siguientes consideraciones:

1. Una asociación no pertenece a ninguna de las clases involucradas, y
2. la dirección de la asociación se da en los dos sentidos.

Las implementaciones de las asociaciones no pueden considerar estos puntos debido a la propia naturaleza de los lenguajes de programación. Para realizar la implementación se considera la navegabilidad y la multiplicidad de la asociación.

En el caso de una asociación bidireccional, la navegabilidad se representa mediante un atributo correspondiente a cada clase de la asociación. Para una asociación unidireccional, el atributo se coloca sólo en la clase de donde "parte" la relación. En la Figura 22 se muestra un ejemplo de asociación entre dos clases.

Figura 22. Relación de asociación entre dos clases

En un ámbito local, las relaciones de asociación tienen la siguiente sintaxis:

```
public class E211 extends E21 {
  private void m3(){
         E3 obj_E3 = new E3();              //relación de asociación
  }
}
```

Se debe verificar que los métodos contengan declaraciones de variables, ya sea que genere una instancia con la instrucción *New()* o solo se declare el objeto. En caso de existir declaraciones de tipo dinámico se debe identificar la clase y registrar en la *lista de relaciones* con su tipo dinámico.

El siguiente ejemplo, de la Figura 23, ilustra un diagrama de clases con relaciones de dependencia.

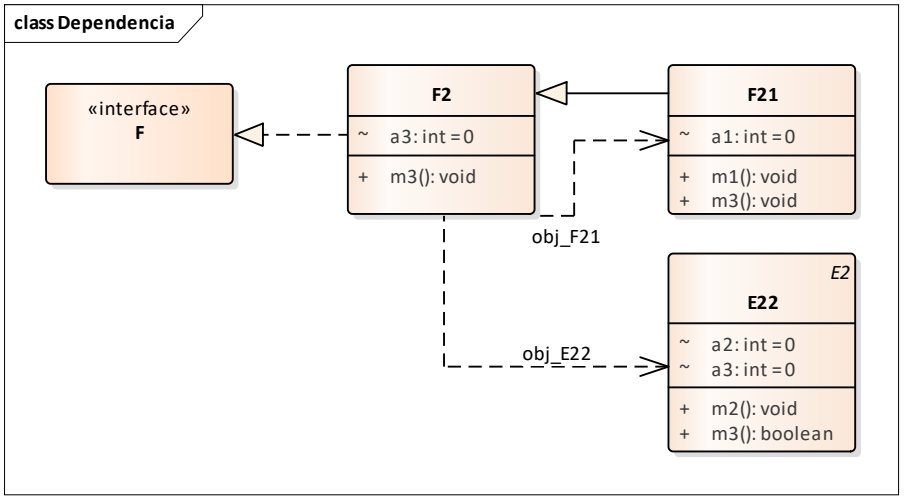


Figura 23. Diagrama de clases para relaciones de dependencia.

Suponer que el método *m3()* es el método iniciador del caso de uso *F2.m3()*. En el método *m3()* se identifican 2 relaciones de dependencia que se declaran en el ámbito local del mismo. La clase *F2* depende de la clase *E22* debido a que en su método *m3()* se invoca al método *m3()* de la clase *E22* a través de una variable local de tipo *obj_E22*. También depende de la clase *F21* debido a que en su método *m3()* se invoca al método *m1()* de la clase *F21* a través de una variable local de tipo *obj_F21*.

1. Para remover las relaciones de dependencia, se considera anidar el código de las clases *F21* y *E22*, ya que al hacer el código propietario de la clase *F2* se podrá acceder a todos los miembros de la clase contenedora, aunque sean privados o protegidos. La declaración del método ***m3()*** se muestra en código de la Figura 24:

```
1  package dummy2;
2
3  public class F2 implements F {
4
5     int a3 = 0;
6
7     public void m3() {
8          F21 obj_F21 = new F21();
9          obj_F21.m1();
10         E22 obj_E22 = new E22();
11         obj_E22.m3();
12         a3++;
13     }
14 }
```

Figura 24. Declaración del método ***m3***() de la clase *F2*

2. Se eliminan las declaraciones de los objetos: *obj_F21* y *obj_E22*, como se muestra a continuación en el código de la Figura 25:

```
1  package dummy2;
2
3  public class F2 implements F {
4
5     int a3 = 0;
6
7     public void m3() {
8          obj_F21.m1();
9          obj_E22.m3();
10         a3++;
11     }
12 }
```

Figura 25. Declaración del método ***m3***() de la clase *F2* sin los objetos *obj_F21* y *obj_E22*

3. Se declaran las instancias de objetos, como atributos de clase, en la clase contenedora.

   Al eliminar las relaciones de dependencia del método *F2.m3()* el siguiente paso es declarar las instancias de los objetos de las clases *F21* y *E22* dentro de la clase contenedora *F2*. En la Figura 26, se muestra el código modificado para la clase contenedora *F2*:

```
1  package dummy2;
2
3  public class F2 implements F {
4
5     int a3 = 0;
6
7     //declara las instancias de los objetos obj_F21 y obj_E22
8      F21 obj_F21 = this.new F21();   //utiliza la propiedad this de la clase F2
9      E22 obj_E22 = this.new E22();   //utiliza la propiedad this de la clase F2
10
11    public void m3() {
12         obj_F21.m1();
13         obj_E22.m3();
14         a3++;
15     }
16 }
```

Figura 26. Código de la clase *F2* integrando los objetos *obj_F21* y *obj_E22*

Una vez identificadas todas las declaraciones de variables del método iniciador, se continua la búsqueda en los métodos identificados para la secuencia interactiva del caso de uso.

Es posible que las declaraciones de variables sean recursivas en la lista de métodos participantes de la secuencia interactiva, así las relaciones de dependencia pueden tener un nivel de profundidad.

4. Identificar el nivel de profundidad de las clases internas (clases internas anidadas).

   Considerar el ejemplo de la clase interna *F21*, del caso de uso *F2.m3()* del ejemplo anterior, donde en el método *m1()* se declara una relación de dependencia con la clase *H3*. El código de la clase *F21* se muestra en la Figura 27:

```
1  package dummy2;
2
3  public class F21 extends F2 {   // inicia clase interna F21
4
5      int a1 = 0;
6
7      private void m1() {         // inicia método m1()
8          a1++;
9          H3 obj_H3 = new H3();   // Relación de dependencia con la clase H3
10         obj_H3.m1();
11     } //termina método m1()
12
13 }//termina clase interna F21
```

Figura 27. Declaración del método ***m1***() de la clase *F21*

5. Para anidar la funcionalidad de la clase *H3* en la clase *F21*, se elimina la declaración de dependencia y se declara un objeto de la clase interna *H3* en la clase de nivel superior. En la Figura 28, se muestra el código resultante para la clase interna *F21*:

```
1  package dummy2;
2
3  private class F21 {              // inicia clase interna F21
4      private int a1 = 0;
5
6      //declara la instancia del objeto obj_H3
7      H3 obj_H3 = this.new H3();  //utiliza la propiedad this de la clase F21
8
9      private void m1() {          // inicia método m1()
10                 a1++;
11         obj_H3.m1();
12     } //termina método m1()
13
14     private class H3{            // Inicia la clase interna H3 – nivel 2
15         private int a4=0;
16         private void m1(){       // inicia método m1()
17             a4++;
18         } //termina método m1()
19     }//termina clase interna H3
20 } //termina clase interna F21
```

Figura 28. Código de la clase interna F21 integrando al objeto *obj_H3*

Como observar que el nivel de profundidad de las clases internas es recursivo y está definido por las relaciones de dependencia de los métodos participantes en la secuencia interactiva del caso de uso

En la Figura 29 se muestra el código final para este caso especial con 2 niveles de anidamiento en clases internas:

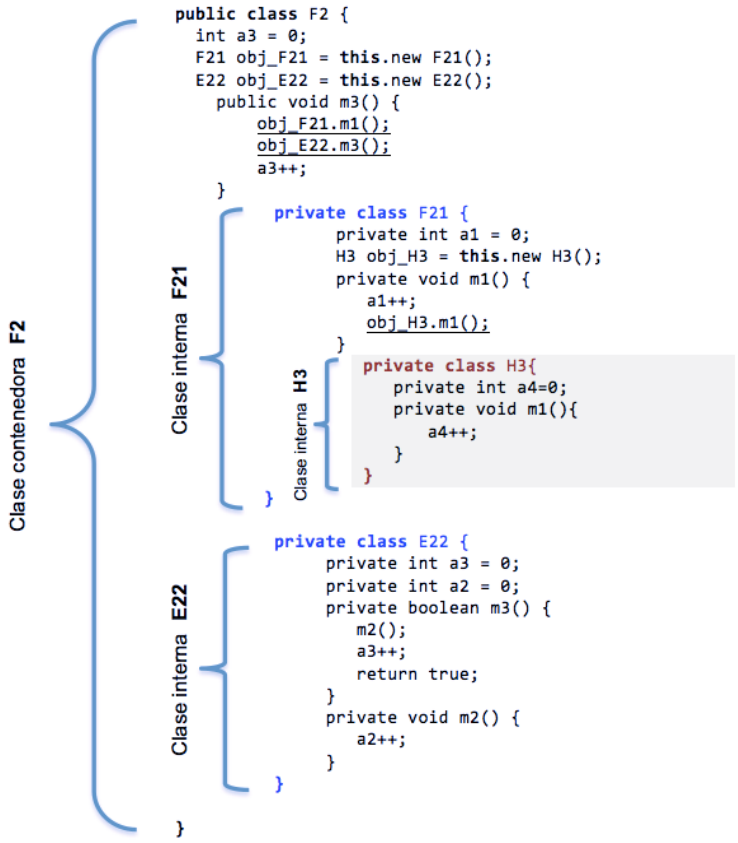


Figura 29. Clases internas con dos niveles de anidamiento

4.4.2 Agregación y composición

Las relaciones de **Agregación** son las asociaciones en las que una parte contiene a elementos de otra parte. Normalmente responden a la pregunta "*tiene un*" (Sierra & Casado, 2011). El diagrama de la Figura 30 presenta una relación de agregación en la que un objeto de la clase "*Empresa"* puede contener otros de la clase "*Empleado"*. En UML la relación de agregación solo añade información semántica, aunque suele significar que el objeto contenedor tiene una referencia al objeto parte. Por ejemplo, la relación una "Empresa" tiene 4 "Empleados", es una relación de agregación.

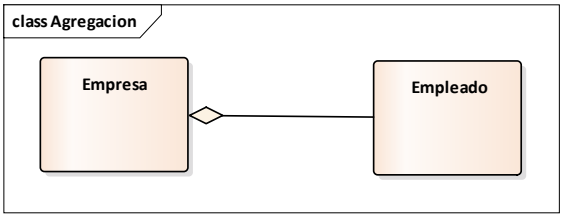


Figura 30. Relación de agregación.

Las relaciones de **Composición**, son relaciones de agregación con una relación de pertenencia fuerte. Esta pertenencia suele implicar que el objeto contenido se implementa como una propiedad del objeto contenedor. Como consecuencia, tanto contenedor como contenido, una vez creados permanecen juntos hasta su destrucción.

En la Figura 31 se muestra un ejemplo de una relación de composición: si se tiene la clase "*Empresa"* y la clase "*Departamento"*, la relación -una empresa "tiene varios" departamentos- es una relación de composición. Claramente la eliminación de un objeto de tipo "*Empresa"* implica eliminar los objetos de la clase "*Departamento"*.

En UML la composición se representa gráficamente de la misma manera que la agregación, sólo que el diamante se rellena de color negro.

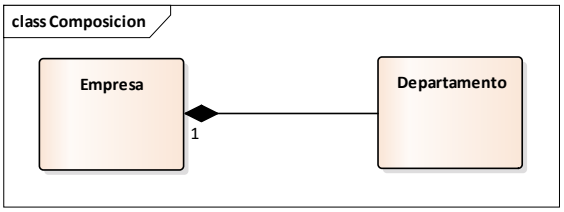


Figura 31. Relación de composición.

Una relación de agregación fuerte se da entre una clase cuya existencia depende de la existencia de la otra.

Una relación de agregación débil se da en aquellas clases cuya existencia no depende de la existencia de la otra.

En esta clasificación, la agregación fuerte corresponde al concepto de composición, y la agregación débil corresponde al de agregación.

Implementación

Para el caso de la composición, del diagrama de la Figura 31, la implementación se realiza haciendo que una clase "*Empresa"* (ensamblado) haga la creación de un objeto de clase "*Departamento"* (parte). Por definición:

a) El objeto de clase "*Departamento"* no puede ser utilizado por otro ensamblado,

b) si el objeto de clase "*Empresa"* desaparece, desaparece también el objeto de clase "*Departamento"*.

Para cumplir con a) basta con crear en la clase "*Empresa"* un objeto **privado** de clase "*Departamento"*. En Java, la clase "*Departamento"* no se incluye en el espacio de memoria asignado al objeto de clase "*Empresa"*, por lo que estrictamente no cumple con b). Sin embargo, el recolector de basura de Java emula este comportamiento, pues al destruirse el objeto de clase "*Empresa"*, su "creación", el objeto "*Departamento"*, será "recogido" y "tirado" posteriormente por el recolector. La Figura 32 se muestra un ejemplo del código de la implementación de la composición.

```
1  class Empresa{
2      //la Empresa se compone de un departamento exclusivo
3      private Departamento depto1;
4  }
5
6  class Departamento{
7
8  }
```

Figura 32. Implementación de la composición

La implementación de la composición es similar a la relación de asociación, sólo que en la composición la dirección es muy importante y siempre sólo de un lado al otro (la clase componente no puede ser clase compuesta en la misma relación).

La agregación conceptualmente difiere en que:

a) el objeto de clase "*Empleado"* definido en la clase "*Empresa"* puede ser compartido por otros ensamblados, y
b) si la clase *Empresa* desaparece el objeto de clase *Empleado* no desaparecerá.

Para cumplir con a) bastaría con que la clase "*Empresa"* contara con un atributo **público** de la clase "*Empleado"*, sin embargo, esto no cumpliría con b). Para cumplir también con b) no debe dejarse a la clase *Empresa* crear al objeto de clase *Empleado*. Para lograr la relación, el objeto de la clase *Empleado* debe estar contenido en la clase *Empresa* de alguna otra forma. Esta forma puede ser que el objeto de clase *Empleado* se pase como parámetro en algún método (pudiendo ser el constructor) a la clase *Empresa*. De esta forma, cuando el objeto de la clase *Empresa* desaparezca, el objeto de la clase *Empleado* seguirá existiendo. La figura 33 muestra esta implementación de la relación de agregación.

```
1  public class Empresa {
2      //la clase Empresa se compone de varios empleados
3      Empleado empleados[] = new Empleado[10];
4      int contador = 0;
5
6      //método que recibe al objeto Empleado
7      boolean AgregaEmpleado(Empleado oEmp){
8          if(contador<10){
9              this.empleados[contador] = oEmp;
10             contador++;
11             return true;
12         }
13         else
14             return true;
15     }
16 }
17 public class Empleado {
18     //definición de Empleado
19 }
20 ...
21 //en algún método de asignación
22 public class AsignaEmpleadoaEmpresa {
23     Empleado oEmp = new Empleado();
24     Empresa oEmpresa1 = new Empresa();
25     Empresa oEmpresa2 = new Empresa();
26
27     //un mismo empleado puede pertenecer a 2 empresas
28     oEmpresa1.AgregaEmpleado(oEmp);
29     oEmpresa2.AgregaEmpleado(oEmp);
30 }
```

Figura 33. Implementación de una relación de agregación

Como variable de clase, las relaciones de agregación o composición tienen la siguiente sintaxis:

```
public class E211 extends E21 {
   E3 obj_E3 = new E3();                    //relación de agregación
   //Más código . . .
}

public class E211 extends E21 {
   E3 obj_E3, obj_2E3, obj_3E3;             //relaciones de agregación
   //Más código . . .
}
```

```
public class E211 extends E21 {
   private static E3 obj_E3 = new E3();       //relación de composición
   //Más código . . .
}

public class E211 extends E21 {
   private static E3 obj_E3 = new E3();       //relación de composición
   //Más código . . .
}
```

4.5 Criterio de integración con la arquitectura de clases internas

Una clase interna es una unidad de programa que funciona como un mecanismo de ocultación de código, ubicando la definición de una clase dentro de otra clase (Java™, 2016).

Una de las características importantes de las clases internas es que pueden acceder a métodos y campos de la clase contenedora como si les pertenecieran. Es decir, una clase interna tiene acceso automático a los miembros de la clase contenedora. Esto ocurre porque el objeto de una clase interna ordinaria mantiene implícitamente una referencia al objeto de la clase contenedora que lo creó.

En la Figura 34 se muestra el esquema de la composición de una clase interna

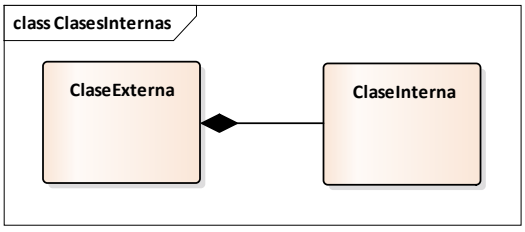


Figura 34. Composición de una clase interna.

La clase interna tiene acceso completo a los datos y métodos de la clase que la contiene, incluyendo sus declaraciones privadas. Una clase interna puede ser declarada como estática o no-estática (Java™, 2016).

4.5.1 Clases internas estáticas (Static Nested Class)

Una clase interna se hace estática cuando no se necesita una conexión entre el objeto de la clase externa y el objeto de la clase interna. (Sharan, 2014)
Que una clase interna sea estática quiere decir que:

1. La clase interna no necesita un objeto de la clase externa para crear un objeto de la clase interna estática.
2. La clase interna puede acceder a variables y métodos privados de la clase contenedora.
3. La clase interna está asociada con la clase contenedora.
4. La clase interna no puede hacer referencia a alguna variable de instancia de la clase contenedora.

Una clase interna estática no necesita inicializar una instancia de la clase externa, como los otros miembros estáticos.

Ejemplo: **ClaseExterna.y** // y es una instancia de ClaseInterna

Fuera de la clase externa, se accede a la clase interna estática a través del nombre de su clase externa contenedora.

Ejemplo: ClaseExterna.ClaseInterna n2 = new ClaseExterna.ClaseInterna();
n2.metodoInterno(); // n2 es una instancia de ClaseInterna

Una clase interna estática tiene el mismo comportamiento que algún miembro estático de la clase externa (asociada con la clase contenedora, no necesita ser instanciada).

Todas las instancias de objetos de las clases internas estáticas están asociadas con la clase contenedora. En la Figura 35 se muestra una representación del conjunto de instancias de objetos de clase interna que comparten sus elementos con la clase contenedora.

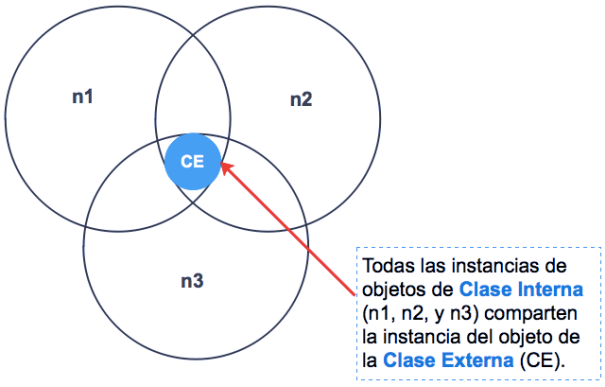


Figura 35. Clase contenedora CE en conjunto con las clases internas estáticas n1, n2 y n3.

En la Figura 36 se representan las instancias de las clases internas que apuntan a una única unidad de memoria de la clase externa.

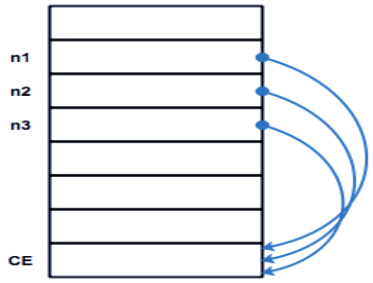


Figura 36. Referencia de las clases internas n1, n2 y n3 a la misma clase contenedora CE.

La Figura 36 muestra una representación de que las referencias clase externa son compartidas por todos los objetos instanciados de clase interna.

4.5.2 Clases internas no-estáticas (Inner class)

A diferencia de las clases internas estáticas, cada instancia de una clase interna requiere una instancia de su clase externa contenedora, es decir, cada objeto de la clase interna requiere un objeto de la clase externa propiedad del objeto (Sharan, 2014).

Que una clase sea interna quiere decir que:

1. Los miembros de la clase interna están definidos dentro del cuerpo de la clase.
2. Una clase local es una clase definida dentro de un método.
3. Una clase local se declara implícitamente creando una variable de la misma (clase anónima).

Las clases internas no pueden declarar variables estáticas ni métodos estáticos, la clase miembro tiene acceso a todas las variables y objetos de la clase externa o cualquier otra clase interna, incluidos los miembros declarados privados. (CompSci, 2016)

En la Figura 37 se muestra la representación de las instancias de objeto de la clase interna y las instancias de objetos de la clase contenedora.

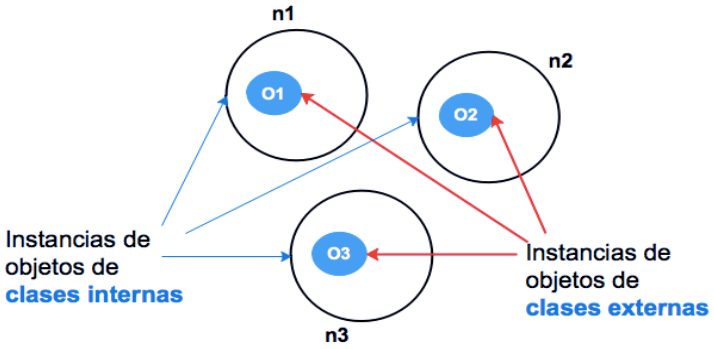


Figura 37. Instancias de objetos de clase interna: n1, n2 y n3.

La clase interna n1 hace referencia a:

- variables de la clase externa declarando: **O1.this.nombreVariable,**
- métodos de la clase externa declarando: **O1.this.instanciaMetodo()** o **instanciaMetodo(),**
- métodos estáticos declarando: metodoEstatico() o O1.metodoEstatico().

La clase n1 tiene su propio objeto de la clase O1, al cual hace referencia mediante la propiedad **this**. En la Figura 38 se muestra un esquema para el objeto de la clase n1.

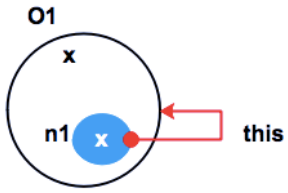


Figura 38. Objetos de una clase interna n1

En la Figura 39 se representa las instancias de las clases internas que apuntan a referencias en las unidades de memoria.

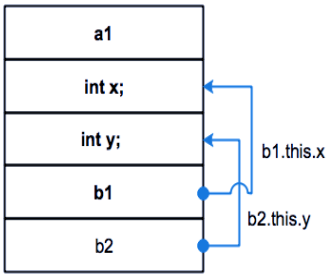


Figura 39. Objetos de clase interna con referencia a entidades de objetos de clase externa.

A partir de los pasos de Análisis estático del Marco Orientado a Objetos y definir el código necesario para cada caso de uso, se integra el código para cada caso de uso en una clase contenedora con arquitectura de clases internas no estáticas.

El subproceso se compone de 3 actividades principales:

1. Remover relaciones de dependencia, agregación y composición. Se deben remover las variables de ámbito local dentro de la lista de métodos de clases y variables de clase.

2. Remover relaciones de herencia de las clases internas. Se deben remover abstracciones (clases abstractas e interfaces) dentro de la lista de clases internas e integrar sus elementos hacia la clase de nivel superior que le sea de utilidad para su funcionamiento.

3. Incorporar elementos y miembros de caso de uso hacia la clase iniciadora. Consiste en eliminar cualquier ambigüedad mediante un análisis de los nombres de enumeradores, atributos y métodos, e integrar los elementos de la clase concreta hacia la clase contenedora para anidar su funcionamiento en clases internas.

En la Figura 40 se muestra el subproceso: Integrar el código de cada caso de uso con clases internas.

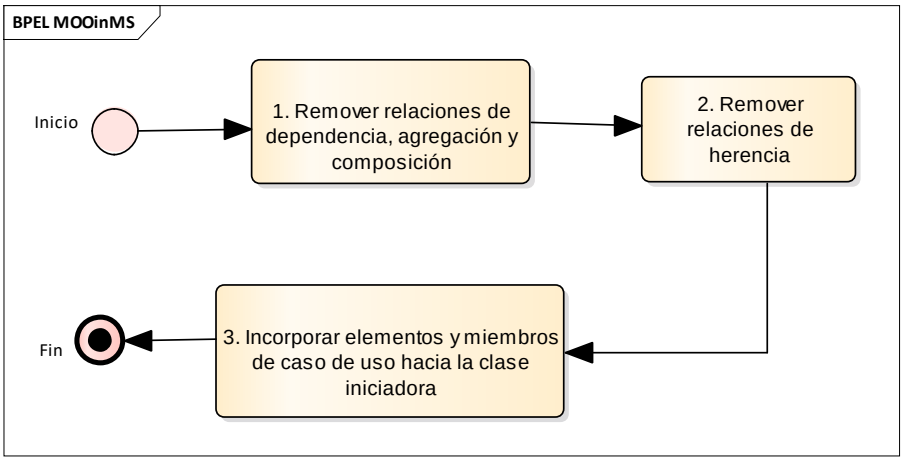


Figura 40. Diagrama del subproceso: integrar el código de cada caso de uso con clases internas.

En la Figura 41 se muestran los casos de uso identificados para generar servicios web con arquitectura de clases internas.

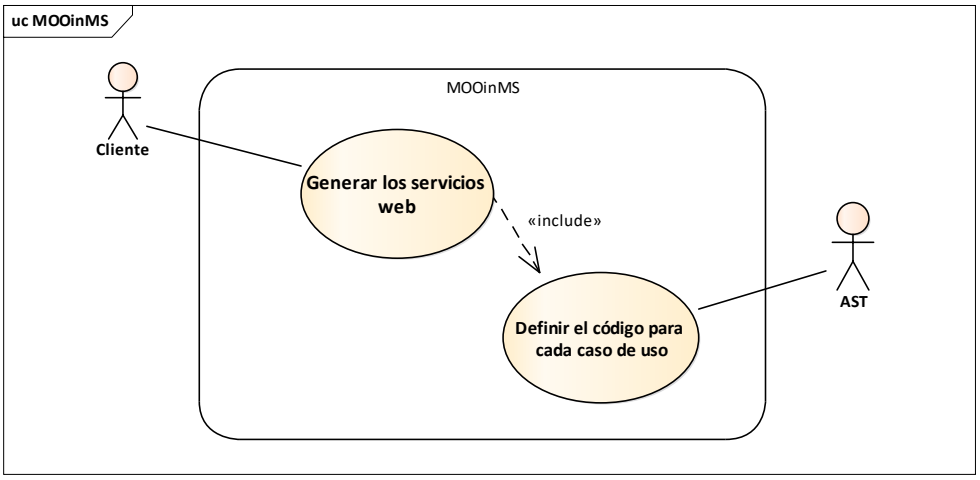


Figura 41. Diagrama de casos de uso del método MOOinMS

Descripción de los casos de uso:

Tabla 3. Descripción del caso de uso Definir código para cada caso de uso

CU_01	**Definir el código de cada caso de uso**
Descripción:	Definir los elementos del código de algún marco orientado a objetos, los cuales son necesarios para satisfacer cada caso de uso encontrado en el marco. Estos elementos son: definiciones de clase, constructores, inicializadores y miembros de cada clase (métodos, atributos y constantes).
Prioridad:	Alta
Actores:	Cliente
Casos de uso relacionados:	CU_02
Precondición:	Contar con las tablas de información de cada clase resultante del análisis aplicado al marco orientado a objetos.
Secuencia normal:	1. Determinar los métodos iniciadores de cada caso de uso. 2. Identificar la secuencia interactiva de cada caso de uso y miembros de clase utilizados. 3. Identificar las relaciones de dependencia, agregación y composición. 4. Integrar el código complementario. 5. Identificar los tipos utilizados. 6. Depurar las listas de los casos de uso. 7. Integrar el código que satisface a un caso de uso en una sola clase de objetos.
Alternativas:	No existen métodos iniciadores de casos de uso: se informa al usuario y no se generará ningún Servicio Web.
Pos-condición:	Se tendrá identificada la información de los elementos de código necesarios para satisfacer cada caso de uso encontrado en el marco.
Observaciones:	Para proceder este caso de uso, el Marco Orientado a Objetos debe estar escrito en lenguaje Java. La herramienta no corrige errores de diseño dentro del Marco.

Tabla 4. Descripción del caso de uso Generar servicios Web

CU_02	**Generar servicios web**
Descripción:	Genera los servicios Web que satisfacen cada caso de uso encontrado en el marco orientado a objetos de entrada.
Prioridad:	Alta
Actores:	Cliente
Casos de uso relacionados:	CU_01
Precondición:	Contar con una estructura de información de los elementos del código del marco que integrará cada servicio Web.
Secuencia normal:	1. Generar el WebMethod con el que se pueda consumir cada servicio Web para cada caso de uso. 2. Envolver el código de cada caso de uso como servicio Web.
Escenario de fracaso:	1. No es posible generar los WebMethods: se informa al cliente y se detiene el proceso. 2. No es posible envolver el código de cada caso de uso como servicio Web: se informa al cliente y se detiene el proceso.
Pos-condición:	Se obtiene un servicio web para cada caso de uso identificado en el Marco OO original
Observaciones:	Para proceder este caso de uso, el Marco Orientado a Objetos debe estar escrito en lenguaje Java. La herramienta no corrige errores de diseño dentro del Marco.

4.6 Diseño conceptual del método de transformación MOOinMS

El diseño arquitectural de la herramienta de transformación se integra de 2 grupos de clases:

a) **Clases del sistema**. Son las que contienen la lógica o comportamiento para realizar los tres métodos de transformación del sistema: MOOaSWCU, InlineClass y MOOinMS.

b) **Clases Visitor**. Son las que implementan la lógica para obtener la información de los AST generados por el analizador sintáctico. Instancian al patrón de diseño *visitor* e implementan la interfaz *Visitor* (que proporcionó el analizador sintáctico). La función de estas clases es ayudar a las clases del sistema a desarrollar las tareas encomendadas. Para cada clase se tendrán los mismos métodos, sin embargo, se implementan con distinto comportamiento. El visitante que reproduce el código original (*VisLectorCodigo*) es el único que tendrá código en todos los métodos.

En la Figura 42 se muestra un diagrama general de la herramienta para representar las dependencias entre los paquetes que componen el modelo dentro del sistema.

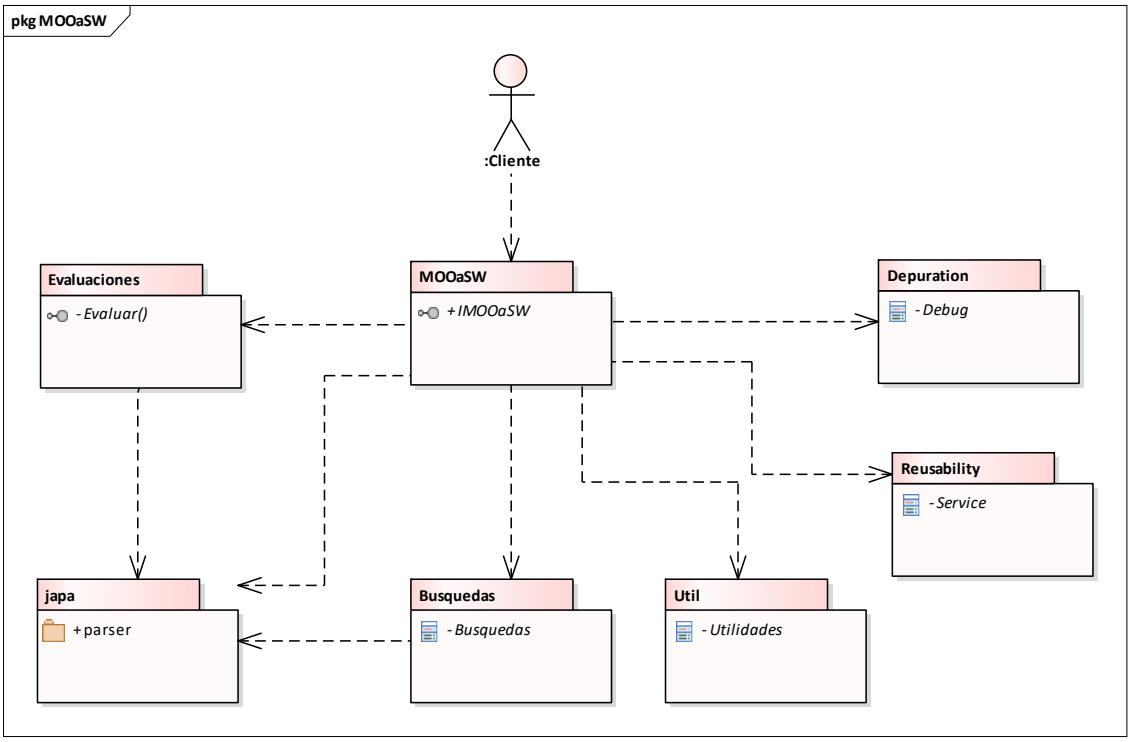


Figura 42. Diagrama de paquetes de la herramienta que implementa al método MOOinMS

Capítulo 5: Implementación de la Herramienta SR2-Transforming

El sistema “SR2-Transforming” (Sistema de Reingeniería para Transformación) da soporte a tres métodos de transformación, que permiten organizar el código de MOO’s en MSOA, con algoritmos estratégicos para agrupar el código en diferentes niveles de granulación, los cuales son: *MOOaSWCU* (León, 2009), *Inlineclass* (Legorreta, 2017) y *MOOinMS* (método de reingeniería desarrollado en esta tesis).

El objetivo del sistema es alcanzar un balance entre las métricas de rendimiento, calidad para reuso y mantenimiento y control de la evolución de los servicios web.

Para automatizar la transformación de Marcos de Aplicaciones Orientados a Objetos (MOO) hacia Marcos de Arquitectura Orientada a Servicios, mediante la detección de Casos de uso dentro del código del MOO, se debe de contar con una herramienta que facilite al cliente incorporar el código del MOO que se desea transformar, de tal manera se realice la transformación mediante el enfoque de agrupamiento de casos de uso que el cliente considere viable para conseguir su meta de valor.

5.1 Diagrama de secuencia

Con base a la especificación de los casos de uso descritos en la sección anterior, se crearon los diagramas de secuencia para cada caso de uso del sistema de reingeniería de esta tesis. De esta manera se identifican los objetos y clases que conforman la arquitectura de clases. En la Figura 43 se muestra el diagrama de secuencias.

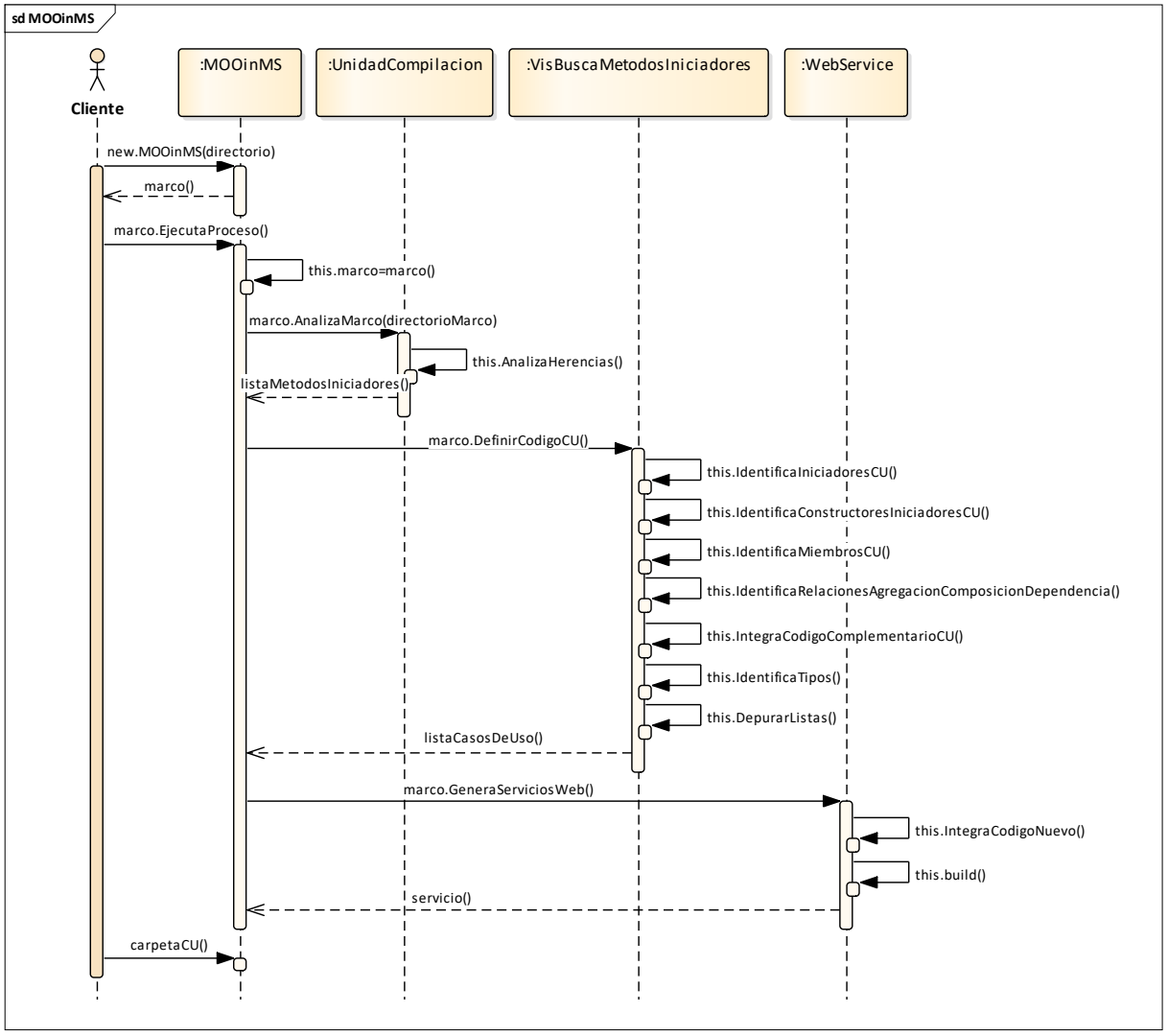


Figura 43. Diagrama de secuencia del método MOOinMS.

5.2 Definir código para cada caso de uso

En la Figura 44 se muestra el diagrama correspondiente al caso de uso: Definir código para cada caso de uso.

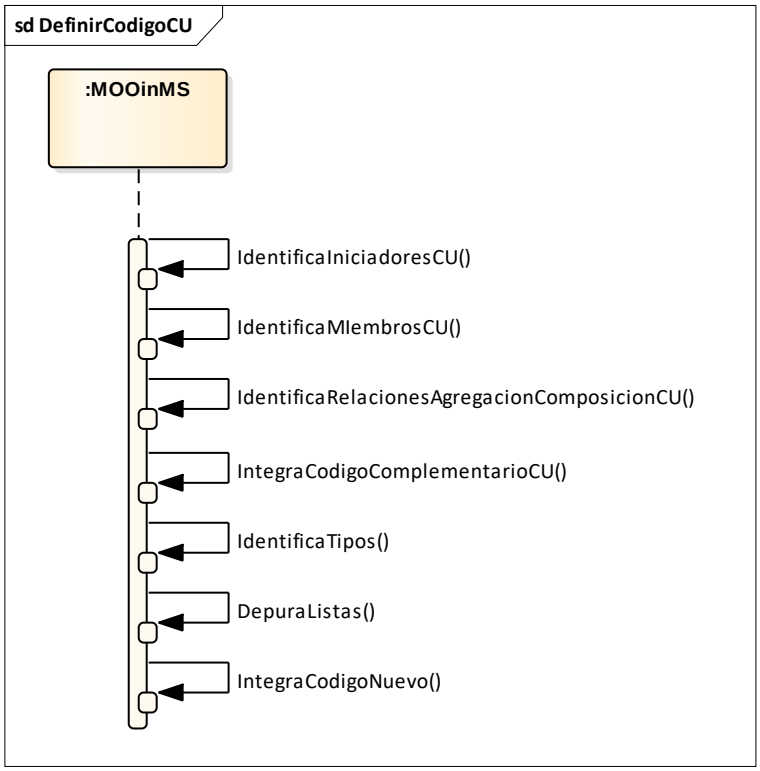


Figura 44. Diagrama de secuencias para el caso de uso “Definir código para cada caso de uso”.

Para cada una de las funciones del sistema se realizó un diagrama de secuencias como se detalla a continuación:

5.2.1 Identificar iniciadores de casos de uso

Para la identificación de los métodos iniciadores de caso de uso, un objeto de tipo MOOinMS utiliza a un objeto del tipo *VisBuscaMetodosIniciadores* (que implementa a la interfaz Visitor). Se usa para recorrer los nodos del AST, analizar las definiciones de clase y declaraciones de métodos e identificar los métodos iniciadores de caso de uso. Para cada unidad de compilación del marco, se envía al constructor de dicho objeto la referencia al nodo inicial del AST (la propia unidad de compilación) y luego un mensaje para iniciar la búsqueda. Se recibe como respuesta una lista de objetos de tipo MethodDeclaration. El diagrama de secuencias se muestra en la Figura 45.

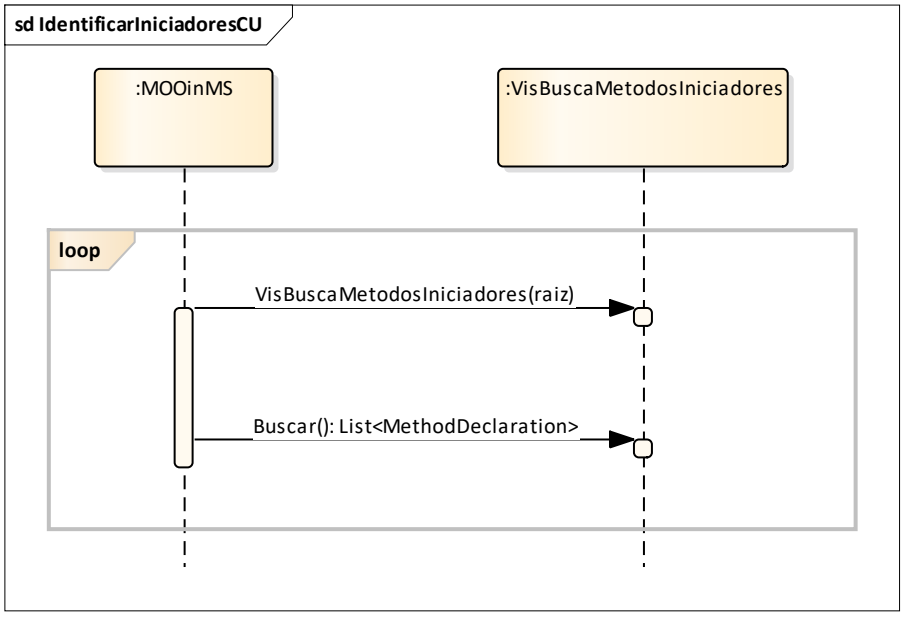


Figura 45. Diagrama de secuencias para el proceso Identificar iniciadores de CU

5.2.2 Identificar métodos y miembros de casos de uso

Para la identificación de los métodos y miembros utilizados por el caso de uso, se realiza un ciclo en el que se envían mensajes a objetos de tipos derivados de la interfaz *VoidVisitor*. Los mensajes se envían en pareja, uno al constructor de la clase y otro solicitando la ejecución del método *Evaluar()*. La llamada al constructor tiene como finalidad enviar al objeto aquellos parámetros necesarios para realizar el proceso: el marco donde se encuentra el código, el caso de uso que se está analizando y el método del caso de uso que se está analizando. En la Figura 46 se muestra la secuencia para identificar métodos y miembros de casos de uso.

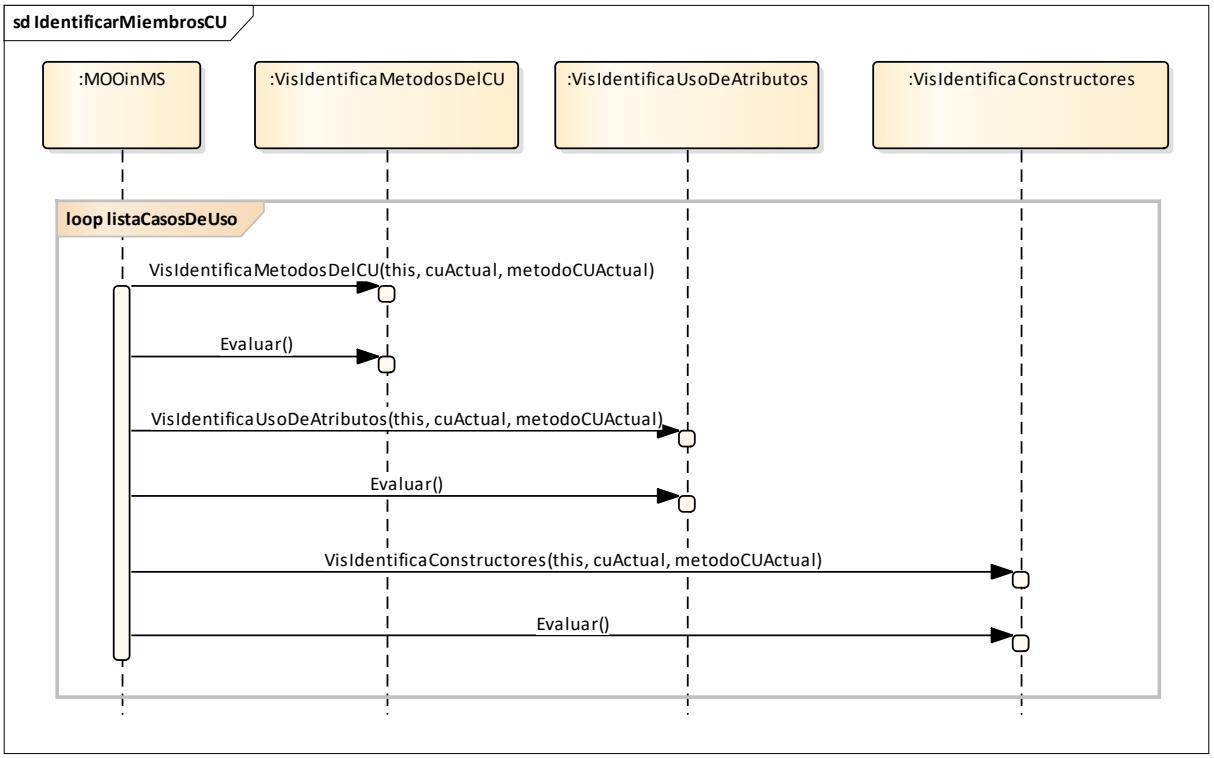


Figura 46. Diagrama de secuencias para el proceso Identificar miembros de CU

5.2.3 Identificar relaciones de dependencia, agregación y composición de casos de uso

Para la identificación de las relaciones de dependencia en métodos y relaciones de agregación y composición de las clases utilizadas por el caso de uso, se realiza un ciclo en el que se envían mensajes a objetos de tipos derivados de la interfaz *VoidVisitor*. Los mensajes se envían en pareja, uno al constructor de la clase y otro solicitando la ejecución del método *Evaluar()*, el cual retorna una lista de objetos *<RelationEval>* con el detalle de las relaciones identificadas y el nivel de profundidad en la secuencia de interacción del CU. En la Figura 47 se detalla el diagrama de secuencias para identificar las relaciones de dependencia, agregación y composición.

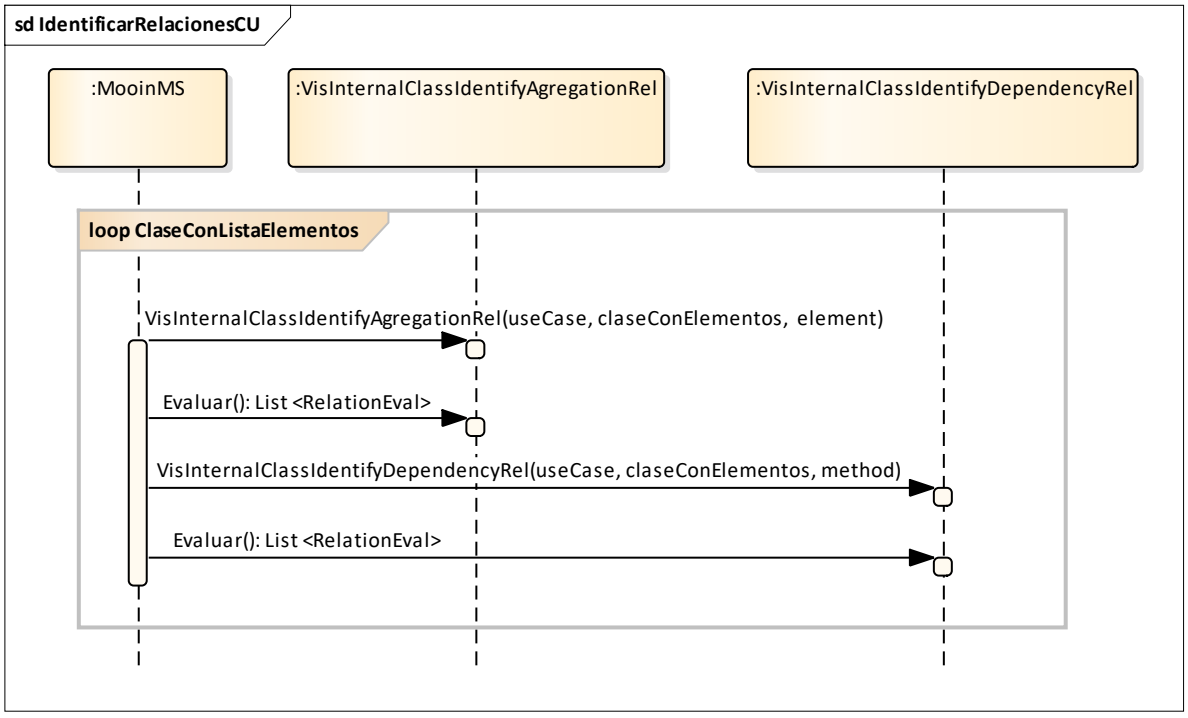


Figura 47. Diagrama de secuencias para identificar relaciones de agregación y composición de CU

5.2.4 Integrar código complementario

En la Figura 48 se muestra la secuencia para la identificación de código complementario (inicializadores, enumeraciones) en los métodos del caso de uso se realiza un ciclo en el que se envía un par de mensajes a un objeto de tipo *VisintegraCodigoComplementario*.

El primer mensaje es la llamada al constructor de la clase, al que se envía como argumentos: el caso de uso que se esté evaluando y la clase o interfaz en donde se debe buscar el código complementario. El segundo mensaje es la invocación al método Evaluar(), que identifica en el código de la clase recibida a los elementos de código complementario y los agrega al caso de uso.

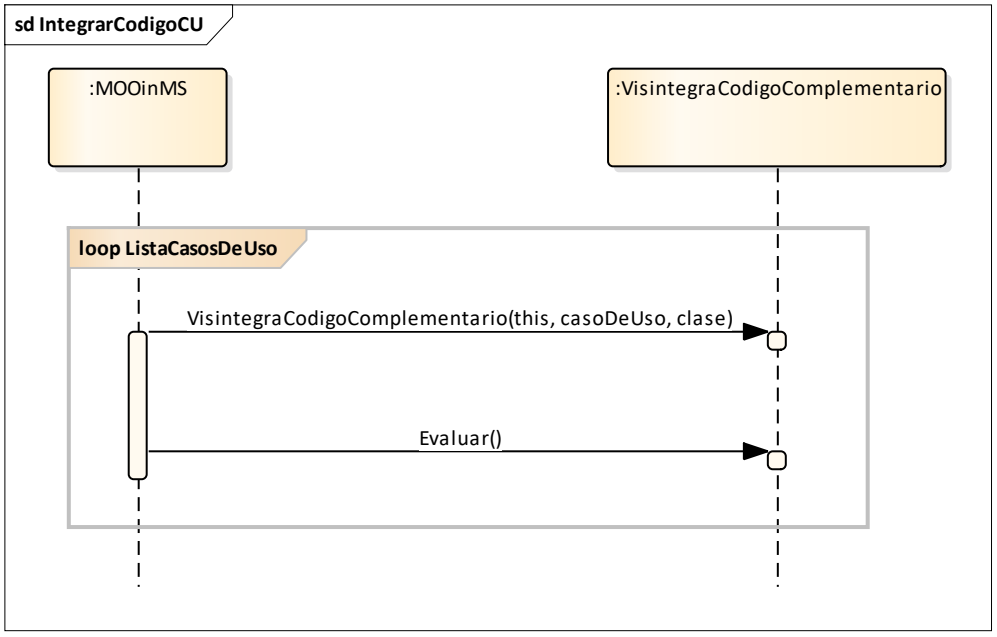


Figura 48. Diagrama de secuencia para el proceso Integrar código complementario del caso de uso

5.2.5 Identificar tipos utilizados

Para la identificación de tipos aún no encontrados en el caso de uso se realiza un ciclo en el que se envía un par de mensajes a un objeto de tipo *VisIdidentificaTipos*.

El primer mensaje es la llamada al constructor de la clase, al que se envían como argumentos: el caso de uso y el elemento del caso de uso en el que se deseen buscar los tipos utilizados (atributo, método o constructor) y un apuntador ***this***. El segundo mensaje es la invocación al método *Evaluar()*, que busca en el código del elemento recibido cada tipo utilizado, localiza la clase al alcance que lo define, si existe, y la agrega al caso de uso. En la Figura 49 se muestra el diagrama de secuencia para identificar tipos utilizados por el caso de uso.

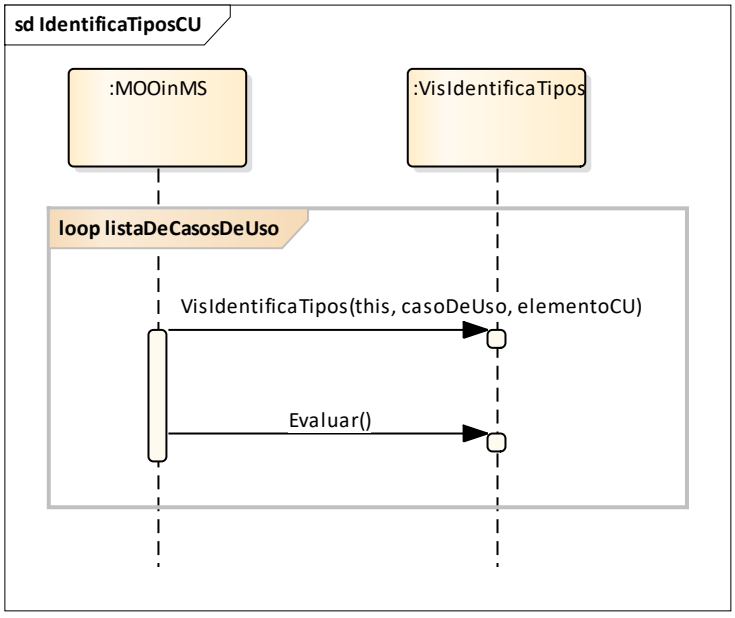


Figura 49. Diagrama de secuencia para el proceso Identificar tipos utilizados del caso de uso

5.2.6 Depurar listas

Para la depuración de las listas de los casos de uso se envían mensajes a la propia clase *MOOinMS*. Los mensajes tienen la intención de agregar en las listas de los casos de uso las declaraciones que sean indispensables y aún no se encuentren en ellas. En la Figura 50 se muestra la secuencia de interacción correspondiente.

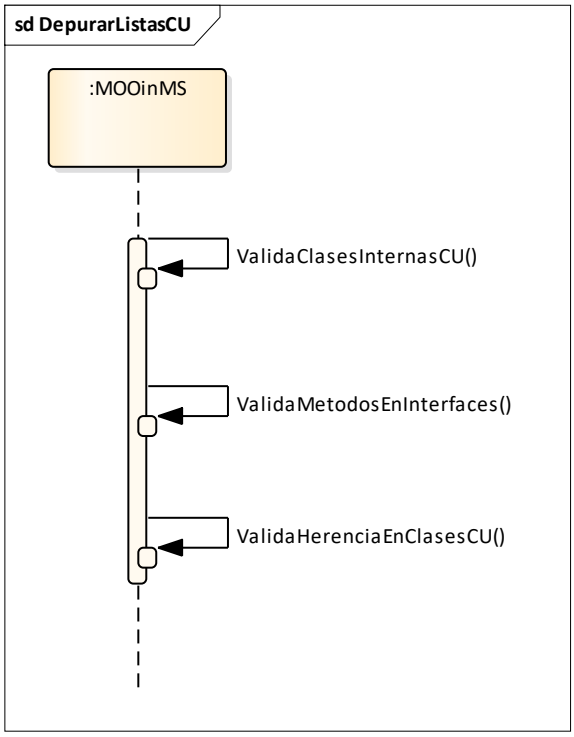


Figura 50. Diagrama de secuencia para el proceso Depurar listas del caso de uso

5.2.7 Integrar código nuevo

Para integrar el código correspondiente para cada caso de uso, debe de pasar por un proceso de depuración y agrupar las sentencias en una única entidad de objetos (una clase). Los métodos invocados se obtienen con el método *getClaseIniciador()*, donde se busca la clase que contiene el método iniciador correspondiente de cada caso de uso y *removeHeritance()* donde se elimina la herencia lo cual implica remover todo tipo de variables de referencia, ámbito de una variable o método invocado y las palabras reservadas *extends* e *implements*. En el diagrama de la Figura 51 se muestra la secuencia correspondiente.

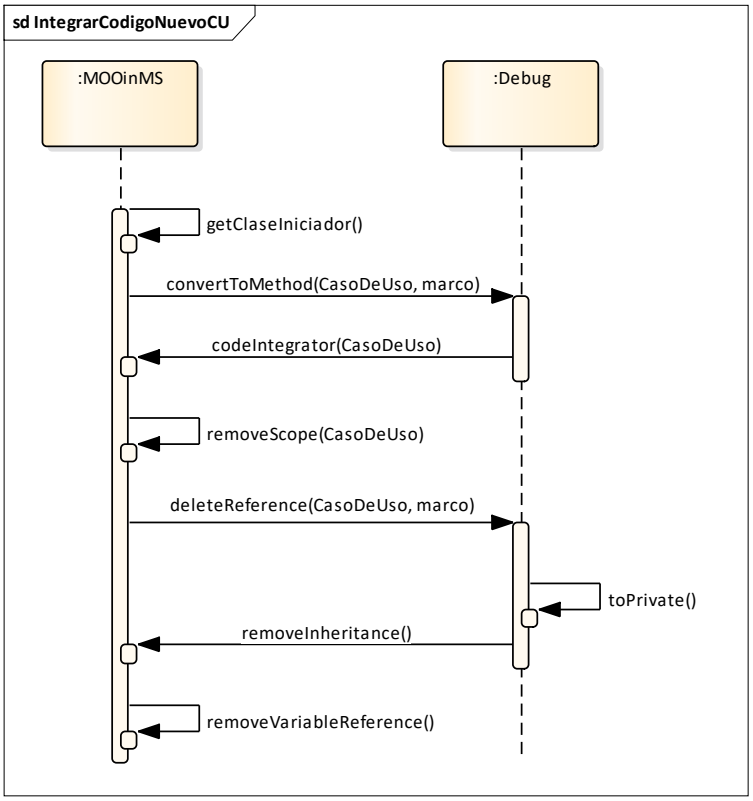


Figura 51. Diagrama de secuencia para el proceso de integrar código nuevo para cada caso de uso

5.3 Generar los servicios Web

La Figura 52 muestra el diagrama de secuencias para el caso de uso *Generar los servicios Web*. Se requieren dos acciones básicas para cada caso de uso generado por la herramienta: la primera es integrar el código de las listas de las clases seleccionadas en nuevos archivos de código *Java* y almacenarlos en directorios específicos del caso de uso; la segunda es generar el archivo del servicio Web, incluyendo los *WebMethods* que se determine debe contener.

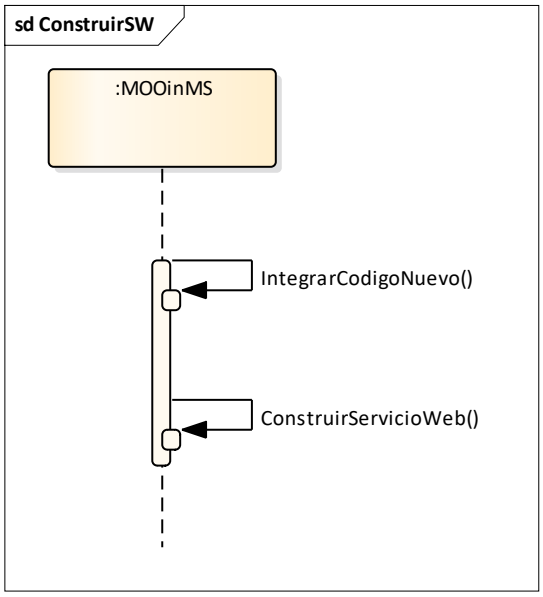


Figura 52. Diagrama de secuencia para el caso de uso Generar los servicios Web

Capítulo 6: Diseño Experimental y Plan de Pruebas

6.1 Diseño experimental

Para efectos del presente experimento, fueron establecidas las siguientes hipótesis:

6.1.1 Hipótesis general

La organización del diseño estructural de la arquitectura interna de servicios, utilizando el concepto de clases internas, cohesivas y coherentes, mejora el rendimiento en términos del tiempo de respuesta y mejora la independencia de los servicios.

6.1.2 Hipótesis específica

Los canales de comunicación entre unidades de programa durante la ejecución de la aplicación de software, sobrecargan el sistema operativo debido a que éste almacena y recupera el bloque de contexto del procesador por cada petición funcional en una secuencia interactiva, por lo tanto, esta sobrecarga influye en el tiempo de respuesta del servicio. Así mismo, el número de canales de comunicación entre las unidades de programa influye directamente en el grado de independencia de estas unidades de programa.

6.1.3 Hipótesis estadísticas

Se plantea el análisis estadístico de los resultados experimentales ya que es de gran importancia para aceptar o rechazar la hipótesis general y la hipótesis específica, por ello es que se han definido las siguientes hipótesis estadísticas de los datos:

6.1.4 Hipótesis nula

- **H0**: Los tiempos de respuesta entre la arquitectura del servicio Web generada por el método *MOOinMS* (M1) y los métodos *InlineClass* (M2) y *MOOaSWCU* (M3) son estadísticamente iguales. M1 = M2 ^ M3.

- **H1**: La arquitectura de clases de *MOOinMS* (M1) presenta las mismas cualidades de autonomía con respecto a las arquitecturas de los métodos *InlineClass* (M2) y *MOOaSWCU* (M3). M1 = M2 ^ M3.

6.1.5 Hipótesis alternativa

- **H2**: El tiempo de respuesta del método *MOOinMS* (*M1*) tiene, estadisticamente, un mejor tiempo de respuesta sobre el método *InlineClass* (*M2*) y el método *MOOaSWCU* (*M3*). *M1 < M2 ^ M3*.

- **H3**: La arquitectura de clases de *MOOinMS* (M1) presenta mejores cualidades de autonomía con respecto a las arquitecturas de los métodos *InlineClass* (M2) y *MOOaSWCU* (M3). M1 > M2 ^ M3.

6.1.6 Formulación de un modelo general en términos de importancia

Una variable experimental es un atributo que al ser medido en diferentes situaciones es susceptible a adoptar distintos valores. Para el presente estudio experimental, se han identificado 3 tipos de variables: las variables dependientes, las variables independientes y las variables intervinientes o factores de ruido.

6.1.7 Variables dependientes

Son las variables de respuesta que se observan en el estudio experimental y que podrían estar influenciadas por los valores de las variables de entrada. Para el presente estudio experimental se definieron las siguientes variables dependientes:

- **Tiempo de respuesta**: Es el tiempo transcurrido desde que se envía una petición de servicio y se recibe su respuesta. El objetivo principal a alcanzar es determinar si el método de transformación MOOinMS mejora de alguna manera al tiempo de respuesta de los Servicios Web generados, con respecto al método de transformación.

- **Autonomía (coherencia, cohesión y acoplamiento)**: En una arquitectura de clases, la autonomía engloba en una única métrica la coherencia, cohesión y acoplamiento. En este sentido la autonomía de una entidad de software para un resultado de valor está en función de la carencia de coherencia, carencia de cohesión y el bajo acoplamiento.

6.1.8 Variables independientes

Son las variables que pueden cambiar libremente su valor y que no dependen de otra variable. Para el presente estudio experimental se definieron las siguientes variables independientes:

- *Método de transformación*: Los casos de prueba fueron diseñados en función a los Servicios Web generados por los métodos *MOOinMS* (M1), *InlineClass* (M2) y *MOOaSWCU* (M3), cuyas estructuras internas, basadas en un diseño orientado a objetos, tienen un nivel de granulación distinta.

- *throughput*: Es una variable de entrada utilizada para estresar cada una de las pruebas experimentales y no depende ni directa ni indirectamente de otra variable externa.

6.1.9 Variables intervinientes o factores de ruido

- *Factores controlables*. Entre los factores controlables se encuentran las especificaciones técnicas de los equipos de cómputo utilizados, la administración de procesos en segundo plano, la configuración del servicio de internet, el acceso entre los equipos que acceden a la red local y la configuración de la misma.

- *Factores no controlables*. Algunos de los factores no controlables son la administración de memoria, el funcionamiento interno de la máquina virtual de Java, la forma en que el servidor internamente administra las peticiones, entre otros.

6.2 Diseño del plan de pruebas

El Plan de Pruebas para el estudio experimental se describe a continuación:

1. Se diseñaron los casos de prueba a partir de una muestra aleatoria simple de los Marcos Orientados a Objetos utilizados en el Plan de Pruebas en (Saldivar, 2015).

2. Se establecen las entradas y el valor del *throughput* que recibirán cada uno los casos de prueba durante las pruebas experimentales.

3. Se ejecutan las pruebas experimentales mediante una herramienta computacional y se recolectan los resultados obtenidos.

4. Se realizan los análisis estadísticos inferenciales para aceptar o rechazar las hipótesis experimentales y realizar conclusiones sobre el fenómeno de estudio.

6.3 Especificación EDP-MOOinMS-01

6.3.1 Características que se probarán

Se probará la identificación de las relaciones de dependencia, agregación y composición de las clases participantes de casos de uso.

6.3.2 Refinamiento del enfoque

Una asociación describe una relación entre dos clases. La asociación sólo denota una dependencia semántica. Las relaciones de dependencia se describen como una asociación entre dos clases que son de ámbito local. Las relaciones de Agregación son las asociaciones en las que una parte contiene a elementos de otra parte. Normalmente responden a la pregunta “tiene un” (Hull et al., 2000). Las relaciones de Composición, son relaciones de agregación con una relación de pertenencia fuerte. Esta pertenencia suele implicar que el objeto contenido se implementa como una propiedad del objeto contenedor. Como consecuencia, tanto contenedor como contenido, una vez creados permanecen juntos hasta su destrucción.

En las Figuras 53, 54 y 55 se muestra la arquitectura de clases de la misma prueba CP01 (correspondientes a la identificación de las relaciones de agregación, composición y dependencia de las clases participantes de casos de uso) obtenida por el método *MOOinMS* de esta tesis.

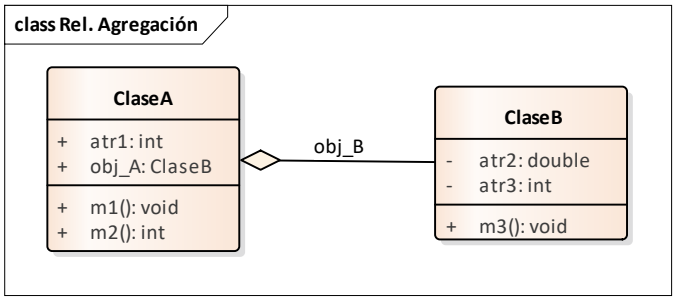


Figura 53. Diagrama de clases del caso de prueba CP01: Agregación

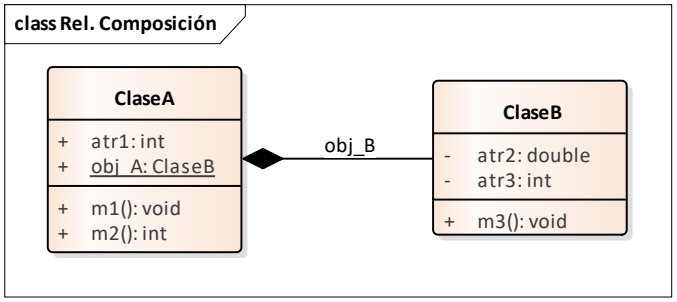


Figura 54. Diagrama de clases del caso de prueba CP01: Composición

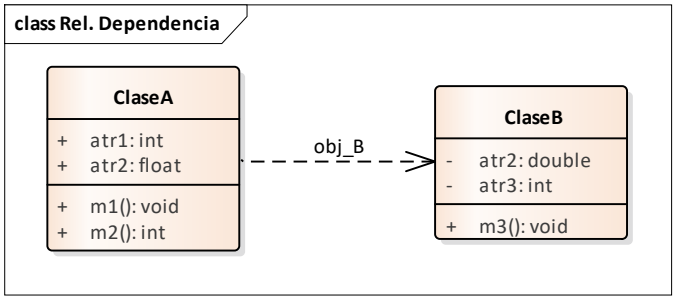


Figura 55. Diagrama de clases del caso de prueba CP01: Dependencia

Para efectos comparativos de esta prueba es necesario que se haya creado de forma manual una lista de las relaciones de agregación, composición y dependencia de la lista de clases participantes de casos de uso a partir del marco orientado a objetos original sujeto a la prueba.

6.3.3 Criterio Pasa/falla

Se comparará la lista de las relaciones de asociación identificadas en las clases participantes de casos de uso creada manualmente contra la lista obtenida automáticamente por el sistema que implementa el método de esta tesis. Si son equivalentes, el sistema pasa la prueba, de lo contrario el sistema no pasa la prueba.

6.4 Especificación EDP-MOOinMS-02

6.4.1 Características que se probarán

Se probará la identificación de las clases internas anidadas de casos de uso.

6.4.2 Refinamiento del enfoque

Las invocaciones de variables de clase y variables de ámbito local identificadas en el método iniciador son el primer nivel de anidamiento de las clases internas. Enseguida se identifican,

de manera recursiva, las relaciones de dependencia, agregación y/o composición en los métodos participantes de las secuencias interactivas del caso de uso. De esta manera se construye una lista de clases internas anidadas.

En la Figura 56 se muestra el código ejemplo de clases internas anidadas para el caso de uso F2.m3().

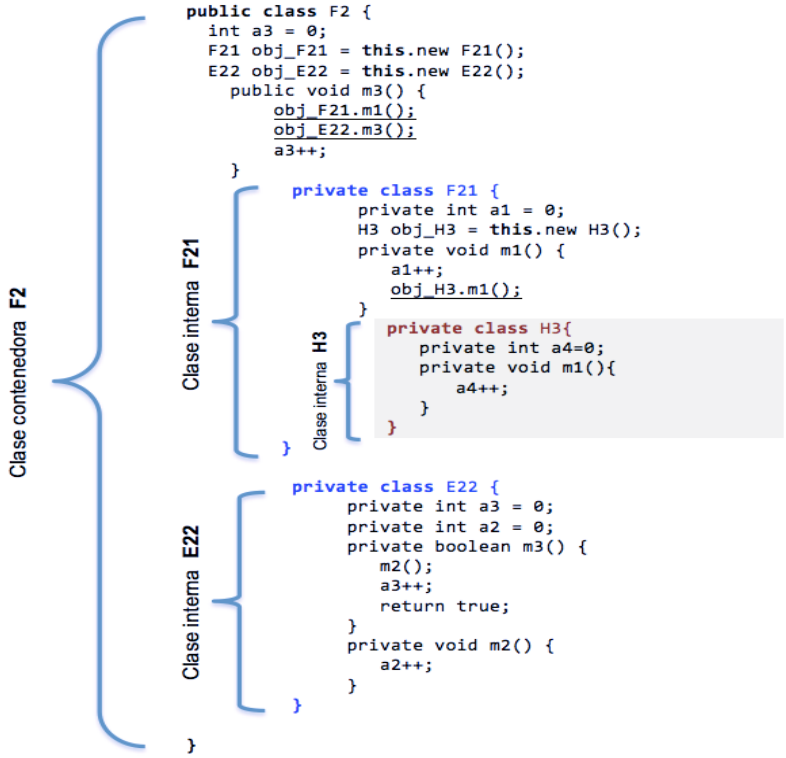


Figura 56. Código de ejemplo de clases internas anidadas

Caso de Uso: **F2.m3()**

Lista de clases internas anidadas:

Método de invocación	**Clase interna**	**Nivel de Profundidad**
F2.m3()	F21	1
F21.m1()	H3	2
F2.m3()	E22	1

De la misma manera, para efectos comparativos de esta prueba es necesario que se haya creado de forma manual una lista de las relaciones de métodos de invocación con la clase interna dependiente y el nivel de profundidad de estas clases participantes de casos de uso a partir del marco orientado a objetos original sujeto a la prueba

6.4.3 Criterio Pasa/No Pasa

Se comparará la lista de las clases internas anidadas de casos de uso creada manualmente contra la lista obtenida del sistema. Si son equivalentes, el sistema pasa la prueba, de lo contrario el sistema no pasa la prueba.

6.5 Especificación EDP-MOOinMS-03

6.5.1 Características que se probarán

Se probará la creación del nuevo código para cada caso de uso.

6.5.2 Refinamiento del enfoque

La herramienta creará una carpeta para cada caso de uso encontrado en el marco orientado a objetos probado. En cada carpeta se generarán los archivos *.java* que contendrán el código necesario para satisfacer cada caso de uso por composición de clases internas.

Con base en lo anterior, el código necesario para satisfacer un caso de uso iniciado por un método 'm' es el correspondiente a:

1. El método '*m'*.
2. Los métodos del caso de uso. Éstos son los métodos integrantes de todas las secuencias de interacción '*S1'*, '*S2'*,..., '*Sn'* a partir de '*m'*.
3. Los miembros de clase '*x1'*, '*x2'*,... '*xn'* que utilizan los métodos del caso de uso.
4. Las declaraciones de las clases a las que pertenecen los métodos del caso de uso (punto 2) y los miembros de clase utilizados (punto 3).
5. La declaración *package* y las declaraciones import utilizadas (en caso de existir).

El código determinado en los puntos 2, 3, 4 y 5 corresponde a elementos (o entidades de software) de los que depende el método iniciador del caso de uso para realizar cualquier ejecución posible.

Para realizar esta prueba es necesario que para cada caso de uso se haya definido de forma manual el código necesario para satisfacer cada caso de uso y se creen los servicios Web, aplicando los métodos *MOOinMS,* desarrollado e implementado en esta tesis, y los métodos *MOOaSWCU* e *InlineClass,* a partir del marco orientado a objetos original sujeto a la prueba.

Por ejemplo:
La Figura 57 muestra la arquitectura de clases del Caso de Uso obtenido de la prueba CP08 mediante el método *MOOaSWCU*. Esta arquitectura consiste de un conjunto de clases en colaboración las cuales únicamente contienen los atributos y los métodos que participan en la secuencia interactiva para satisfacer el caso de uso.

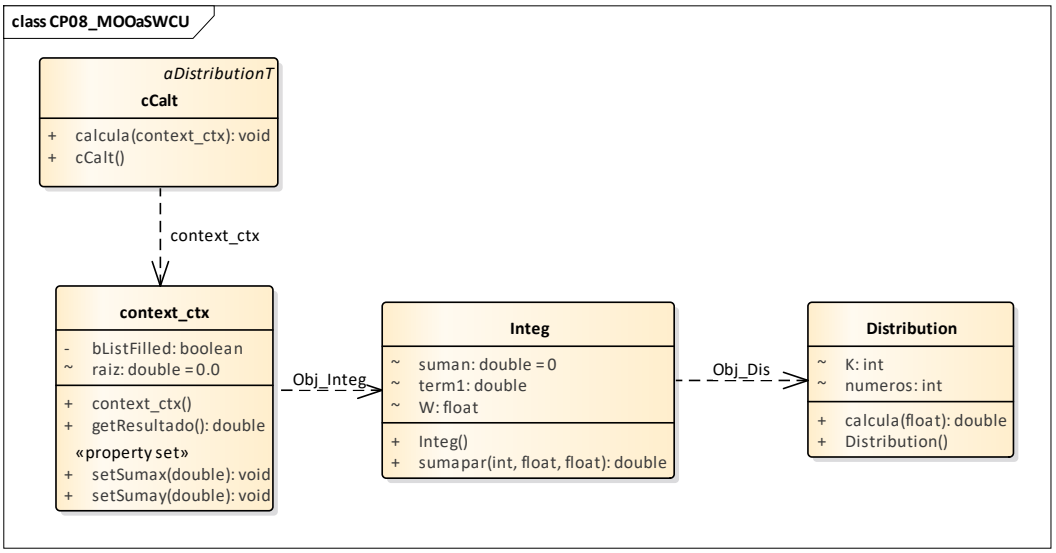


Figura 57. Diagrama de clases del Caso de Prueba CP08 mediante el método MOOaSWCU

La Figura 58 muestra la arquitectura de clases de la misma prueba CP08 obtenida por el método *Inlineclass.* Esta arquitectura consiste de una única clase, la cual contiene únicamente los atributos y los métodos de la secuencia interactiva que satisface el mismo caso de uso.

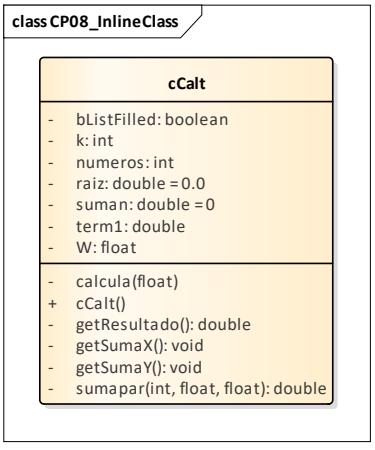


Figura 58. Diagrama de clases del Caso de Prueba CP08 mediante el método InlineClass

La figura 59 muestra la arquitectura de clases de la misma prueba CP08 obtenida por el método *MOOinMS*, de esta tesis. Esta arquitectura consiste de una única clase, la cual contiene el anidamiento de clases internas que componen a la clase contenedora a través de la lista de relaciones identificadas en los atributos y los métodos de la secuencia interactiva que satisface el mismo caso de uso.

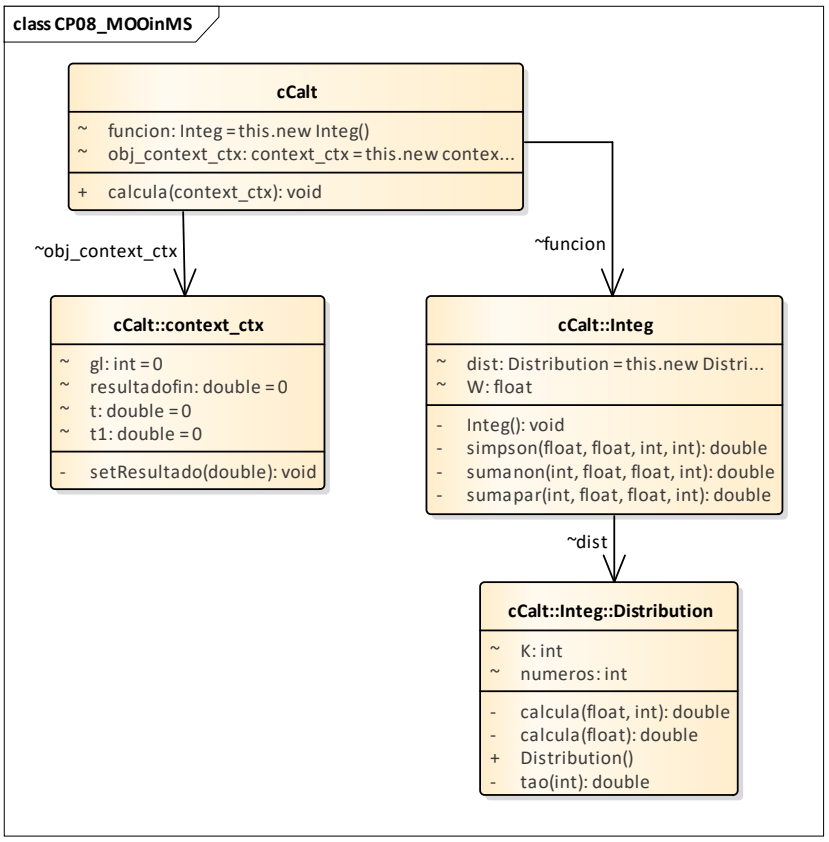


Figura 59. Diagrama de clases del Caso de Prueba CP08 mediante el método MOOinMS

El diseño de las tres arquitecturas implementa el mismo caso de uso para satisfacer al mismo requerimiento, sólo difieren en el nivel de granulación del servicio Web obtenido apropiadamente a partir de su propia arquitectura de clases.

6.5.3 Criterio Pasa/No Pasa

Se comparará el código definido manualmente contra el código generado por el sistema. Si son equivalentes, el sistema pasa la prueba, de lo contrario el sistema no pasa la prueba.

6.6 Especificación EDP-MOOinMS-04

6.6.1 Características que se probarán

Se probará la generación del código de los servicios web.

6.6.2 Refinamiento del enfoque

A partir de las listas de clases y de sus elementos conformadas en el diseño de la prueba EDP-MOOinMS-01, para cada caso de uso identificado por el método MOOinMS, se generará un servicio Web cuyo nombre se obtendrá con base en el nombre del método iniciador y el nombre calificado de su clase. Para cada caso de uso se creará un directorio y se colocarán en él los archivos que contengan el código seleccionado para satisfacer el caso de uso. Este código de envoltura se guardará en la misma carpeta donde estén los archivos *.java* del caso de uso.

Para realizar esta prueba es necesario que para cada caso de uso se haya definido manualmente el código para envolver al código del caso de uso como servicio Web.

6.6.3 Criterio Pasa/No Pasa

Se comparará el código definido manualmente contra el código generado por el sistema. Si son equivalentes, el sistema pasa la prueba, de lo contrario el sistema no pasa la prueba.

6.7 Especificación EDP-MOOinMS-05

6.7.1 Características que se probarán

Se probará el rendimiento en los Servicios Web generados de los métodos *MOOaSWCU*, *InlineClass* y *MOOinMS*.

6.7.2 Refinamiento del enfoque

Para efectos de esta investigación de tesis, el rendimiento es el tiempo transcurrido desde que se envía una petición de servicio y se recibe la respuesta (tiempo de respuesta). Se espera que los servicios Web creados con el código que genera el sistema *MOOinMS* proporcionen menor tiempo de respuesta que los servicios Web creados con el código de los métodos *MOOaSWCU* e *InlineClass*.

Así mismo, las pruebas consisten en obtener el tiempo de respuesta de un conjunto de Servicios Web seleccionados, a partir de una muestra aleatoria simple para una población finita, con un nivel de confianza del 90% y un error permitido del 10%; donde la población considerada son 13 Servicios Web, omitiendo aquellos Servicios Web que no estén integrados por una arquitectura de clases internas o el Servicio no arroja resultados de valor para su evaluación. La muestra obtenida para los casos de prueba fue de 56 iteraciones por ejecución de Servicio Web. Partiendo de las conclusiones de (Guadarrama Rogel, 2013), cuyo estudio arrojó que el mejor tiempo de respuesta lo dan aquellas arquitecturas de grano grueso, es decir, toda la funcionalidad requerida para un Caso de Uso, integrada en una única clase. Se realizó la comparativa de ambos resultados para determinar cuál de los tres métodos arrojó el mejor tiempo de respuesta.

6.7.3 Criterio Pasa/No Pasa

Se comparará el tiempo de respuesta del método *MOOinMS* contra el tiempo de respuesta de los métodos *MOOaSWCU* e *InlineClass*. Si el tiempo de respuesta es menor, el sistema pasa la prueba y confirma lo esperado, de lo contrario el sistema no pasa la prueba.

6.8 Especificación EDP-MOOinMS-06

6.8.1 Características que se probarán

Se probará la autonomía de los casos de uso en el Marco de Servicios Web para los enfoques *MOOaSWCU*, *InlineClass* y *MOOinMS*

6.8.2 Refinamiento del enfoque

La Autonomía es una métrica compuesta por 3 atributos de calidad: coherencia, cohesión y acoplamiento.

Para realizar esta prueba, se deben crear los servicios Web para los casos de uso identificados en el marco y utilizar en ellos el código generado por la herramienta. La prueba consiste en medir la coherencia, cohesión y el acoplamiento del código generado para los servicios web obtenidos de los tres métodos, para al final, comparar sus resultados.

6.8.2.1 Coherencia

La coherencia se define como el grado de relación funcional de una responsabilidad en una unidad de programa. La coherencia se basa en el principio de una única responsabilidad *Single Responsibility Principle (SRP)* el cual indica que: "una clase o módulo debe tener uno y sólo un motivo para cambiar" (R. C. Martin, 2002)

En términos generales, una clase está compuesta por elementos que pueden ser métodos y atributos. En el contexto de este trabajo de tesis, la coherencia mide la cantidad de funciones que interactúan para satisfacer una responsabilidad en relación al número total de funciones en la entidad de software que los contiene y se refiere a una secuencia interactiva de métodos implicados para cumplir un caso de uso.

Para calcular la métrica de coherencia de un caso de uso se propone la siguiente formula:

coherenciaCU = Σ (msi)/tm

Dónde:
coherenciaCU = Coherencia del caso de uso.
msi = secuencia interactiva de métodos para satisfacer una única responsabilidad.
tm = total de métodos participantes.
Restricciones: tm > 0

La métrica **coherenciaCU** necesita saber el total de métodos (tm) participantes en la entidad de software y la cantidad de métodos de la secuencia interactiva (msi) que participan para resolver una única responsabilidad. Se divide (msi) entre (tm) para calcular el grado de coherencia del caso de uso. Los valores van de 0 a 1, donde 0 significa incoherencia total y 1 significa alta coherencia.

En la Figura 60 se muestra el diagrama de clases de un caso de uso.

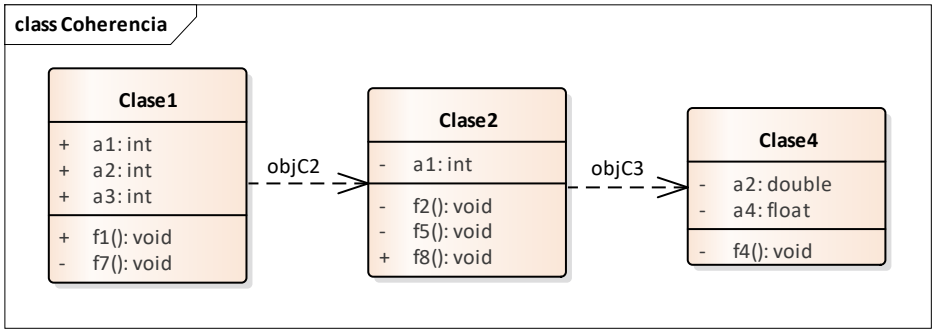


Figura 60. Diagrama de clases de un caso de uso.

En el diagrama de clases de la Figura 58 se puede observar la siguiente secuencia interactiva: Suponer que el método iniciador es f1() de la Clase1, la secuencia interactiva del caso de uso involucra los siguientes métodos:

f1 → f2 → f4

Para calcular la coherencia de la entidad de software se obtienen los valores msi y tm.

msi = 3 y tm = 6

Aplicando la fórmula de coherencia se obtiene el siguiente resultado:

coherenciaCU = 3/6 = 0.5 ≈ 50% de coherencia

El resultado indica que el caso de uso atiende a más de una responsabilidad, ya que el valor ideal es **coherenciaCU = 1**.

6.8.2.2 Cohesión

La cohesión se define como el grado de relación entre los miembros de una clase (Chidamber & Kemerer, 1994). La cohesión es el concepto en donde se ve cómo los métodos de una clase están estrechamente relacionados entre sí. Si el módulo es altamente cohesivo la complejidad de las clases se reduce, el módulo es reutilizable y fácil de mantener (Yadav et al., 2014).
A nivel de caso de uso la cohesión es una medida de la relación entre los miembros de las clases en un conjunto de clases de una secuencia interactiva (Bieman, Member, & Kang, 1998).

En la Figura 61 se muestra un diagrama de clases con el diseño de la cohesión de un caso de uso.

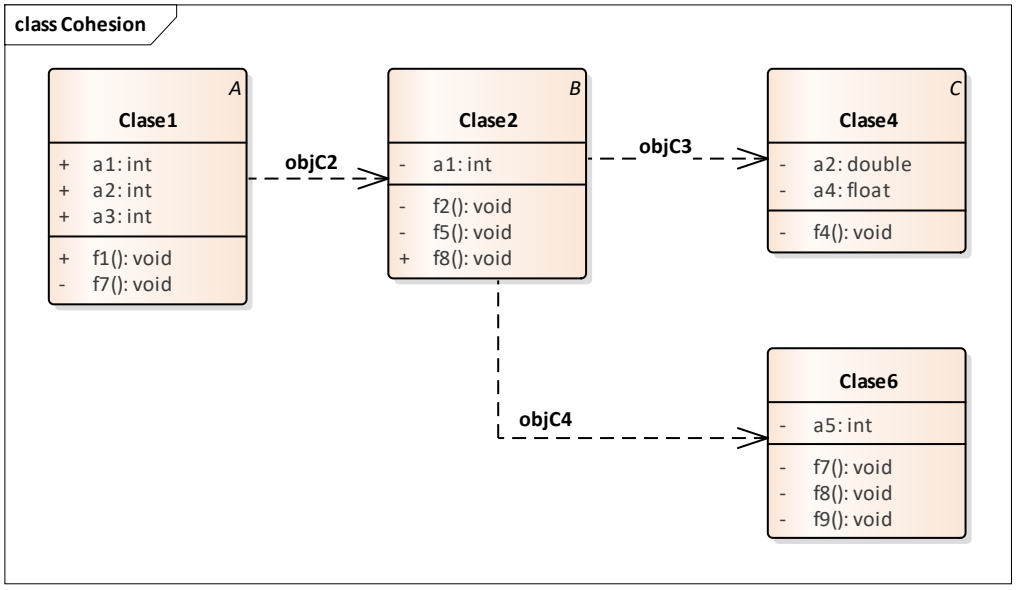


Figura 61. Diagrama de clases de cohesión de un caso de uso.

Para calcular la métrica de cohesión se toma la métrica Lack of Cohesion of Methods (LCOM4). Esta métrica es la apropiada para medir la cohesión de un caso de uso y se deriva del trabajo antecedente (Geovani, 2016).

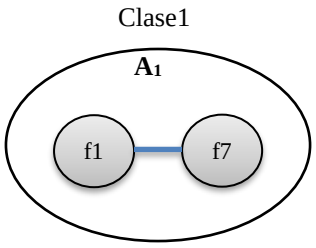
La métrica LCOM4 considera un grafo no dirigido G, donde los vértices son los métodos de una clase y existe una arista entre dos vértices si los métodos *Mi y Mj correspondientes comparten al menos un atributo o si Mi invoca a Mj o viceversa*. El número de conjuntos que se formen en el grafo indica el valor de cohesión en la clase (Hitz & Montazeri, 1996).

El método para calcular la cohesión en base al valor LCOM4 en relación al número de conjuntos identificados en la clase es:

Si LCOM4 = 1 indica una clase con alta cohesión.
Si LCOM4 >= 2 indica un problema (baja cohesión).
Si LCOM4 = 0 no hay conjuntos de métodos en esa clase.

Por ejemplo:

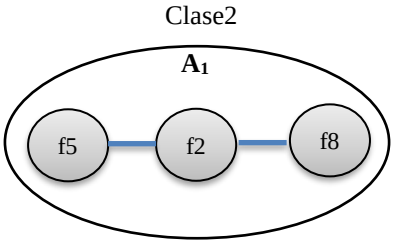
Para el diagrama de clases de la Figura 61 se calculan los siguientes valores de cohesión:



Conjuntos de métodos de Clase1:
A_1={f1, f7}

Relaciones de métodos:
f1() invoca a f7(), por lo tanto:
$f1 \cap f7 \neq \varnothing$

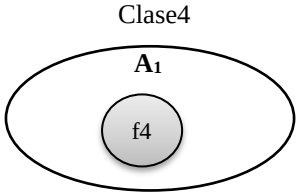
Número de relaciones identificadas = 1
LCOM4=1



Conjuntos de métodos de Clase2:
A_1={f5, f2, f8}

Relaciones de métodos:
F5() invoca a f2(), y f2() invoca a f8(), por lo tanto:
$f5 \cap f2 \cap f8 \neq \varnothing$

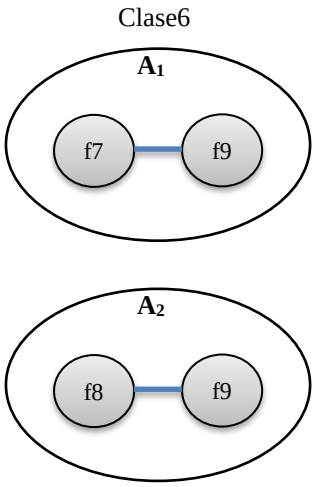
Número de relaciones identificadas = 1
LCOM4=1



Conjuntos de métodos de Clase4:
A_1={f4}

Relaciones de métodos:
f4() no invoca a otros métodos, por lo tanto:
$f4 = \varnothing$

Número de relaciones identificadas = 0
LCOM4=0



Conjuntos de métodos de Clase6:
A_1={f7, f9}
A_2={f8, f9}

Relaciones de métodos:
f7() invoca a f9() y f8() invoca a f9(), por lo tanto:
$f7 \cap f9 \neq \varnothing$
$f8 \cap f9 \neq \varnothing$

Número de relaciones identificadas = 2
LCOM4=2

Para calcular la cohesión a nivel de un conjunto de clases se utiliza la formula **cohesionCU**, que es la relación de la suma de la métrica Lack of Cohesion of Methods (LCOM4) de cada clase entre el número total de clases del caso de uso (N).

$$\mathbf{cohesionCU} = \mathbf{\Sigma_i^N\ LCOM4_i/N}$$

Donde:

cohesionCU = Cohesión del caso de uso.
Σ LCOM4 = La suma de la métrica LCOM4 (Lack of Cohesion of Methods) de cada clase participante en el caso de uso.
N = total de clases.

Restricciones: N > 0

La cohesión a nivel de caso de uso para el diagrama de la Figura 37 es:

cohesionCU = (1 + 1 + 0 + 2) / 4

cohesionCU = 1 → El caso de uso tiene alta cohesión.

6.8.2.3 Acoplamiento

El acoplamiento es una medida del grado de relación de un módulo contra los demás. Se refiere al número de variables que intervienen en la comunicación entre módulos, más que al número de interfaces de comunicación entre módulos. Si dos módulos se comunican, deben de intercambiar tan poca información como sea posible (Geovani, 2016).

Para calcular el acoplamiento se considera la métrica **COF** (factor de acoplamiento) que se refiere específicamente al tipo de acoplamiento por paso de mensajes (llamadas a métodos), ya que únicamente se miden los canales de comunicación entre clases. Es decir, no se toman en cuenta otras relaciones, por ejemplo: declaración de objetos (Brito et al., 1995).

COF sirve para calcular el factor de acoplamiento de un sistema debido a múltiples relaciones de dependencia entre clases incluyendo las llamadas a funciones, la creación de objetos y destrucción de objetos. En esta prueba sólo se consideran las llamadas a métodos o funciones que ocurren en primera instancia, es decir, llamadas a funciones de otras clases de manera directa o a través de clases abstractas, excepto mensajes entre clases hermanas o entre clases base con sus clases derivadas.

En la métrica no se toma en cuenta la cantidad de métodos de una clase que son invocados por otra clase, sino que todos cuentan como una sola dependencia, lo único que importa es cuántas clases tienen dependencia con otra clase en particular. En la métrica se considera la sumatoria de estas dependencias. La fórmula **COF** es la siguiente:

$$COF = \frac{\sum D}{TC^2 - TC}$$ (Brito et al., 1995)

Dónde: Σ D es la sumatoria de las llamadas de dependencia.
TC es el total de clases del caso de uso.

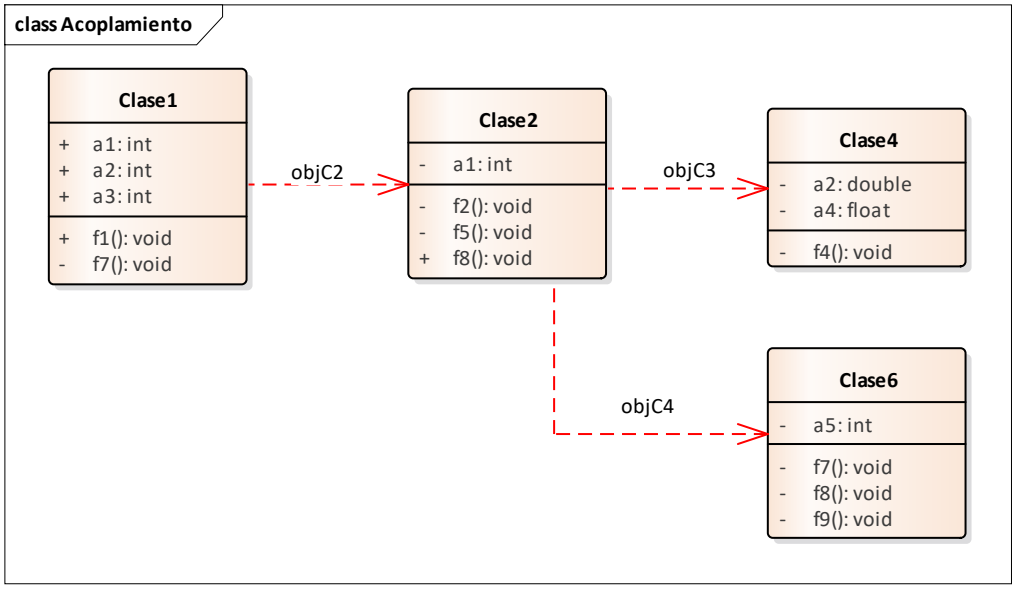


Figura 62. Diagrama de clases del diseño para el acoplamiento de un caso de uso.

COF puede ser una medida indirecta de los atributos con los cuales está relacionado: complejidad, falta de encapsulamiento, carencia de reuso, facilidad de comprensión y poca facilidad de mantenimiento (Brito et al., 1995).

En el marco de la Figura 62 se muestra las relaciones de dependencia que existen entre clases. Se puede observar que el total de clases TC = 4, y el total de la sumatoria de relaciones de dependencia es ΣD=3, por lo tanto,

COF = 0.25

Valores muy altos del factor de acoplamiento deben ser evitados y es deseable que este factor se encuentre en los límites más bajos. Como resultado de experimentación en (Brito et al., 1995) se sugiere que el factor de acoplamiento (COF) en un sistema sea lo más cercano a cero.

La **autonomía** para un caso de uso se define como:

autonomiaCU = [(coherenciaCU + cohesionCU)/2] + COF

Donde:
autonomiaCU = Autonomía del caso de uso.
coherenciaCU = Coherencia del caso de uso.

cohesiónCU = Cohesión del caso de uso.
COF = Factor de acoplamiento del caso de uso

Restricciones:
El valor ideal de autonomía es lo más cercano a 1.
Si autonomiaCU ≠ 1 indica que no hay un balance entre coherencia, cohesión y acoplamiento.

6.8.3 Criterio Pasa/No Pasa

Para cada caso de uso, se comparará de forma manual el código de los enfoques *MOOaSWCU* e *InlineClass* contra el código *MOOinMS* resultantes de la ejecución de la herramienta. Si el código *MOOinMS* tiene mejores características de autonomía, el sistema pasa la prueba, de lo contrario el sistema no pasa la prueba.

Capítulo 7: Ejecución del Experimento y Recolección de Resultados

7.1 Especificación de los casos de prueba

7.1.1 Elemento de prueba: Tiempo de respuesta entre los métodos *MOOaSWCU, Inlineclass* y *MOOinMS*

Especificaciones de entrada:
Las entradas para las pruebas serán las siguientes:

1. La ruta del directorio en donde se encuentren los archivos *.java* que contienen el código de las muestras tomadas del Plan de Pruebas propuesto en (Saldivar, 2015).

2. Los archivos *.java* que contienen el código de cada muestra seleccionada. Estos archivos deberán estar en una carpeta con el nombre del Marco original. La muestra obtenida de 13 Servicios Web considerada como se establece en el diseño de los casos de prueba, se muestra en la Tabla 5.

Tabla 5. Tabla con los casos de uso obtenidos del Maro Orientado a Objetos: Statistic

Caso de prueba	Nombre del caso de uso
CP01	ordena_cBubleXY_statistic
CP02	ordena_cBuble_statistic
CP03	calcula_cDistributionX2_statistic
CP04	calcula_cDistributionT_statistic
CP05	calcula_cDistribution_statistic
CP06	calcula_cDistItems_statistic
CP07	calcula_cCorrelation_statistic
CP08	calcula_cCalt_statistic
CP09	ordenacion1_List_statistic
CP10	ordenacion_List_statistic
CP11	CalculaCorrelacion_context_ctx_statistic
CP12	calcula_cSumXY_statistic
CP13	calcula_cRange_statistic

Especificación de salida: La salida correspondiente para cada caso de prueba, será un archivo con extensión *.txt* con los tiempos arrojados por los métodos de transformación :*MOOaSWCU,*

Inlineclass y MOOinMS, obtenidos desde la herramienta "*JMeter*" (Apache Software, 2017) la cuál fue utilizada como instrumento de medición para el tiempo de respuesta. De tal manera que posteriormente se puedan interpretar estadísticamente para hacer una inferencia hacia todos los Casos de Uso en el Plan de Pruebas de (Saldivar, 2015). Los tiempos obtenidos están registrados en milisegundos (*ms*).

7.1.2 Elemento de prueba: Autonomía de los servicios Web generados por los métodos *MOOaSWCU, Inlineclass* y *MOOinMS*

Especificaciones de entrada: Las entradas para las pruebas serán las siguientes:

1. La ruta del directorio en donde se encuentren los servicios Web, resultado del proceso de transformación de cada uno de los tres métodos de transformación: MOOaSWCU, InlineClass y MOOinMS.

2. La muestra obtenida de 13 Servicios Web considerada como se establece en el diseño de los casos de prueba, se muestra en la Tabla 6.

Tabla 6. Tabla con los Servicios Web obtenidos del Maro Orientado a Objetos: Statistic

Caso de prueba	Nombre del servicio Web
CP01	ordena_cBubleXY_statistic
CP02	ordena_cBuble_statistic
CP03	calcula_cDistributionX2_statistic
CP04	calcula_cDistributionT_statistic
CP05	calcula_cDistribution_statistic
CP06	calcula_cDistItems_statistic
CP07	calcula_cCorrelation_statistic
CP08	calcula_cCalt_statistic
CP09	ordenacion1_List_statistic
CP10	ordenacion_List_statistic
CP11	CalculaCorrelacion_context_ctx_statistic
CP12	calcula_cSumXY_statistic
CP13	calcula_cRange_statistic

Especificación de salida: Para la autonomía se calculan, de forma manual, las métricas que involucra la formula (coherencia, cohesión y factor de acoplamiento) para cada caso de prueba y los datos serán almacenados en un archivo con extensión .xsl.

7.2 Instrumentos de medición aplicados

7.2.1 Instrumento de medición para el tiempo de respuesta

Para obtener resultados significativos acerca del fenómeno de estudio es necesario usar una herramienta computacional que sirva como instrumento de pruebas para recopilar los resultados de las pruebas experimentales con respecto al tiempo de respuesta. Por lo que se decidió utilizar la herramienta computacional de código abierto con el nombre *JMeter* (Apache Software, 2017).

JMeter es un proyecto de Apache que puede ser utilizado como una herramienta de pruebas para analizar y medir el desempeño de una variedad de servicios, con énfasis en aplicaciones Web. Donde se pueden establecer pruebas de estrés para Servicios Web con capa SOAP y medir el tiempo de respuesta según el valor de *throughput* establecido. La herramienta fue obtenida del sitio de Internet http://jmeter.apache.org/ en la versión 3.2 (Apache Software, 2017).

La herramienta computacional requiere de los siguientes elementos descritos en las Tablas 7 y 8.

Tabla 7. Especificaciones técnicas del cliente y el servidor

	Nombre	Procesador	Memoria
Equipo Cliente/Servidor	MacBook Air macOS Sierra v 10.12	1.6 GHz Intel Core i5	8GB 1600 MHz DDR3

Tabla 8. Datos de red

Nombre:	Router TP-LINK
Ancho de banda:	Descarga: 9 Mbps, Subida: 1.00 Mbps
Medio de comunicación:	Conexión Ethernet mediante cables UTP cat 6
Topología de conexión:	Punto a Punto (Cliente-Servidor)

7.2.2 Instrumento para el análisis estadístico.

Los resultados de las pruebas por sí solos no serían capaces de proporcionar información suficiente para identificar patrones de comportamiento o realizar conclusiones mediante una inferencia del fenómeno de estudio. Es por ello que se optó por utilizar una herramienta estadística para analizar los resultados y aceptar o rechazar las hipótesis establecidas en el presente trabajo.

La herramienta computacional utilizada, de la compañía IBM, con el nombre *SPSS Estadistic* (IBM-Analitycs, 2017), es un programa estadístico que servirá para realizar pruebas de normalidad y, en su caso, análisis estadístico paramétrico o no parámetro de los resultados obtenidos. La herramienta fue obtenida del sitio de Internet: http://www.ibm.com/analytics/us/en/technology/spss/

7.2.3 Eliminación del ruido experimental

El ruido experimental son factores externos que pueden afectar los resultados de las pruebas experimentales. Antes de iniciar las pruebas experimentales se redujo el ruido experimental mediante el proceso propuesto en (Guadarrama Rogel, 2013).

Capítulo 8: Análisis Estadístico e Interpretación de Resultados

8.1 Análisis estadístico

El análisis de datos es el proceso de inspeccionar, limpiar y transformar datos con el objetivo de resaltar información útil y realizar conclusiones. Una vez finalizadas las pruebas, el análisis de los resultados es de suma importancia para aceptar o rechazar las hipótesis establecidas (Guadarrama Rogel, 2013).

8.1.1 Pruebas de normalidad

Los datos obtenidos por sí solos no proporcionan la suficiente información para realizar conclusiones inferenciales, por ello es necesario realizar el análisis estadístico.

Conocer la distribución de densidad de los resultados experimentales es el primer paso para cumplir con el propósito anterior. Se dice que los resultados experimentales *son normales* cuando su función de densidad tiene una forma acampanada y es simétrica respecto de un determinado parámetro estadístico (Walpole, 1999) .

Para verificar la normalidad de cada una de las poblaciones en cada ciclo de pruebas acorde a los resultados obtenidos, se utilizó la prueba de normalidad Kolmogorov-Smirnov (para muestras grandes) con un intervalo de confianza del 95% mediante la herramienta *SPSS Estadistic*. Como resultado de la prueba de normalidad se permite concluir que ninguno de los resultados obtenidos en los casos de prueba proviene de una distribución normal, es decir, son datos *no paramétricos*.

Una vez que se determinó que los resultados de las poblaciones son datos *no paramétricos*, las pruebas no paramétricas a realizar y que resultan adecuadas para un experimento con $k > 2$ poblaciones independientes son la prueba de Kruskal-Wallis , como alternativa no paramétrica de la prueba *t Student* para muestras independientes

8.1.2 Prueba no paramétrica de Kruskal-Wallis

Se realizó la prueba no paramétrica de Kruskal-Wallis (Walpole, 1999) la cual compara las medias para más de dos poblaciones independientes. La prueba de Kruskal-Wallis se basa en calcular los rangos de la muestra de cada población y comparar los promedios de los rangos.

Los resultados se muestran a continuación, indicando con una "x" los casos que difieren estadísticamente y donde $\mathcal{R}$ es igual al Rango promedio de los resultados obtenidos de cada metido de transformación correspondiente al caso de prueba, el cual servirá para evaluar el mejor tiempo de respuesta, en el caso de ser aceptada la hipótesis alternativa (H_2):

Tabla 9. Resultados de la prueba no paramétrica: Kruskal-Wallis

Prueba no paramétrica de Kruskal-Wallis					
Caso de prueba	P-Valor	Difieren estadisticamente	$\mathcal{R}$ MOOaSWCU	$\mathcal{R}$ InlineClass	$\mathcal{R}$ MOOinMS
CP01	0.00000	x	6114.50	4953.00	3128.50
CP02	0.25304	-	5182.50	4404.50	4609.00
CP03	0.00000	x	6626.00	4640.00	2930.00
CP04	0.00000	x	6562.50	5200.50	2433.00
CP05	0.00000	x	6548.50	4276.50	3371.00
CP06	0.00000	x	7361.00	3437.00	3398.00
CP07	0.00000	x	6091.00	5314.00	2791.00
CP08	0.00000	x	7237.50	3370.00	3588.50
CP09	0.00000	x	6556.00	4025.50	3614.50
CP10	0.00000	x	6698.00	3938.00	3560.00
CP11	0.00058	x	4960.50	3643.50	5592.00
CP12	0.00013	x	5868.00	3694.00	4634.00
CP13	0.00000	x	6857.00	4118.00	3221.00

Los resultados de la prueba Kruskal-Wallis, presentados en la Tabla 9, muestran que en el 92.3% de los casos difieren estadísticamente. De esta forma, se rechaza la hipótesis general $\boldsymbol{H_0}$ *"Los tiempos de respuesta entre la arquitectura del servicio Web generada por el método MOOinMS (M1) y los métodos InlineClass (M2) y MOOaSWCU (M3) son estadísticamente iguales. M1 = M2 ^ M3."* y se acepta la hipótesis alternativa $\boldsymbol{H_2}$ *"El tiempo de respuesta del método MOOinMS (M1) tiene, estadisticamente, un mejor tiempo de respuesta sobre el método InlineClass (M2) y el método MOOaSWCU (M3). M1 < M2 ^ M3."*.

8.1.3 Cálculo de la métrica de autonomía

Para verificar cuál de las tres arquitecturas tiene mejores cualidades de autonomía se aplicó la fórmula de autonomíaCU a cada método de transformación. En la Tabla 10 se muestra el resultado de la prueba.

Tabla 10. Resultados del cálculo de autonomiaCU

Caso de prueba	MOOaSWCU	InlineClass	MOOinMS
CP01	1.550	3.000	1.100
CP02	2.083	2.500	1.083
CP03	1.542	3.000	1.208
CP04	1.542	3.000	1.208
CP05	1.542	3.000	1.208
CP06	1.375	3.000	1.208
CP07	1.095	3.500	1.048
CP08	2.000	1.500	1.500
CP09	1.150	3.000	1.050
CP10	1.150	3.000	1.050
CP11	1.082	3.000	0.982
CP12	1.167	2.500	1.083
CP13	1.078	4.000	1.033

El valor promedio de la autonomia para cada arquitectura se muestra en la Tabla 11.

Tabla 11.Valor promedio de la autonomía para cada arquitectura del MOOaSW

	Promedio	Diferencia	% de autonomía
MOOaSWCU	1.412	0.276	24
InlineClass	2.923	1.787	64
MOOinMS	**1.136**	-	100

Las arquitecturas de los servicios web generados con el método MOOinMS tienen mejores cualidades de autonomía, ya que el promedio de los valores de autonomía de los casos de uso del marco de servicios web tiene un valor cercano a 1. De esta forma, se rechaza la hipótesis general ***H_1*** *"La arquitectura de clases de MOOinMS (M1) presenta las mismas cualidades de autonomía con respecto a las arquitecturas de los métodos InlineClass (M2) y MOOaSWCU (M3). M1 = M2 ^ M3"* y se acepta la hipótesis alternativa ***H_3*** *"La arquitectura de clases de MOOinMS (M1) presenta mejores cualidades de autonomía con respecto a las arquitecturas de los métodos InlineClass (M2) y MOOaSWCU (M3). M1 > M2 ^ M3"*.

8.2 Comparación entre arquitecturas

La tabla 12, muestra la comparativa entre resultados para cada caso de prueba, a partir de los tiempos de respuesta promedio obtenidos durante las pruebas experimentales, considerando la arquitectura más eficiente como la arquitectura con menor tiempo de respuesta y la arquitectura menos eficiente como la arquitectura con mayor tiempo de respuesta, donde:

M_1 = MOOaSWCU
M_2 = InlineClass
M_3 = MOOinMS

Tabla 12. Comparativa entre los tiempos de respuesta

Caso de prueba	M_1	M_2	M_3	Arquitectura más eficiente
CP01	2.804	2.375	1.804	M_3
CP02	2.214	2.036	1.857	M_3
CP03	3.964	2.214	1.482	M_3
CP04	2.964	2.250	1.232	M_3
CP05	2.643	1.821	1.571	M_3
CP06	3.232	1.536	1.696	M_2
CP07	2.554	4.089	1.411	M_3
CP08	3.464	1.679	1.750	M_2
CP09	3.643	3.321	1.821	M_3
CP10	2.339	1.482	1.375	M_2
CP11	2.929	2.482	3.804	M_2
CP12	2.179	1.554	1.911	M_2
CP13	4.839	1.589	1.339	M_3

En 8 de 13 casos resultó que las arquitecturas formadas por clases internas, método M_1, de los Servicios Web cuentan con el menor tiempo de respuesta. En 5 de 13 casos el método M_2 obtiene el mejor tiempo de respuesta en función de su arquitectura.

De estos resultados se deduce que el 62% de los Servicios Web obtenidos con el método M_1 obtiene el mejor Tiempo de respuesta.

Para la métrica de autonomía, en 12 de 13 casos resultó que las arquitecturas formadas por clases internas tienen un balance entre las métricas de coherencia, cohesión y acoplamiento. Esto corresponde a un 92.3% de autonomía del método M_3.

Capítulo 9: Conclusiones

9.1 Conclusiones generales

A través del diseño experimental y el diseño de las pruebas se demuestrá que el objetivo de la tesis fue alcanzado. Se diseñaron y ejecutaron casos de prueba para servicios web. Todos los casos resultaron exitosos bajo los criterios de resultado esperado - resultado obtenido. De la misma manera se llevaron a cabo pruebas de estrés con la herramienta *JMeter* de *java*, siendo todos los casos de igual forma, exitosos.

El análisis de resultados permitió comparar los tiempos de respuesta de las pruebas realizadas a los servicios Web obtenidos con tres métodos de reingeniería incluyendo al desarrollado en está tesis. Los resultados muestran que las arquitecturas más eficientes son las que tienen una arquitectura con composición de clases internas. Esto puede justificarse debido a que el código se concentra en una única clase, eliminando las llamadas innecesarias de memoria y la sobrecarga del procesador, mejorando así el tiempo de respuesta entre la interacción de clases.

En un 62% de los casos, el método MOOinMS fue más eficiente y en un 38% de los casos el método MOOaSWCU. Esto podría deberse a la complejidad algorítmica de la arquitectura de los servicios web. Esta hipótesis deberá ser planteada y probada en trabajos futuros para comprobarla.

El concepto de autonomía, puede ser interpretado desde la perspectiva de la calidad de software cuando una unidad de código es reusable y mantenible, es decir, cuando existe una única responsabilidad entre las clases, funciones y los módulos que lo componen.

Cada uno de los servicios web han sido probados para conocer sus cualidades de autonomía. El análisis de resultados ayudó a determinar que la mejor arquitectura de los servicios web está relacionada con el diseño y el anidamiento interno de las clases que los componen. Debido a la arquitectura de clases internas se obtiene alta cohesión, bajo acoplamiento y mejora la independencia a nivel de requerimiento, lo que beneficia su reuso y mantenimiento. Esto se puede atribuir a que cumplen con el principio de una única responsabilidad de (R. C. Martin, 2002), por el cuál un servicio web está formado sólo por el código que se necesita para cumplir una meta de valor.

La arquitectura con mejores cualidades de autonomía la tuvieron los servicios Web generados con el método MOOinMS con un valor promedio de 1.136, de un valor ideal de 1. La diferencia con la arquitectura con menos autonomía es de 1.787 que equivale a un 64% menos reusable y mantenible. En el caso del método MOOaSWCU el código empleado realiza llamadas innecesarias de memoria y sobrecarga del procesador, haciendo mayor el tiempo de respuesta entre la interacción de clases. Para el caso del método InlineClass puede atribuirse a tener más de un conjunto disjunto de atributos y métodos innecesarios ya que los declara en una única clase contenedora.

Se concluye que la arquitectura MOOinMS puede proporcionar resultados favorables en cuanto a rendimiento, reuso y mantenimiento siempre y cuando todos sus módulos cumplan con las propiedades de aislamiento, independencia y encapsulamiento.

9.2 Experiencias aprendidas

Debido a la complejidad y tiempo de ejecución en sistemas grandes puede aumentar, en esta tesis, se estableció identificar relaciones de depedencia, agregación y composición hasta encontrar paquetes de código java.

Para la realización de este trabajo de investigación fue necesario proponer una métrica para conocer la autonomía de los servicios Web. En este sentido, se usó el resultado de los atributos de calidad: coherencia, cohesión y factor de acoplamiento a nivel del conjunto de clases que integran cada caso de uso.

Una restricción a considerar en el código de entrada es definir sólo un método iniciador por cada caso de uso y el resto de los métodos deben ser privados o protegidos. Esto ayuda a distinguir aquellos métodos únicos que permiten la interacción del usuario con el sistema para pedir un servicio o atender un requerimiento e inician una secuencia interactiva de métodos que dan como resultado una meta de valor u objetivo para el usuario final.

Cabe señalar que con este desarrollo se obtienen servicios Web que por sus cualidades de calidad pueden ser comparados con los *microlitos* como se definen en (Bonér, 2017). De conformidad con (Bonér, 2017) los *microlitos* tienen características de unidades autónomas, resistentes a fallas, independientes y escalables al igual que los Servicios Web obtenidos en esta tesis. La diferencia de estos últimos es que están organizados en Marcos de Servicios Web.

Los *microlitos*, así como los Servicios Web obtenidos en esta tesis, representan sólo un paso atrás de la actual tecnología de servicios reusables en la nube computacional denominados *microservicios* (F. Martin, 2017)

9.3 Trabajo futuro

Como trabajo futuro se considera generar un banco de pruebas y realizar experimentos a marcos orientados a objetos de dominios de aplicaciones reales, con el propósito de generalizar el método propuesto a una mayoría de código legado de MOOS, por las diferentes capacidades del lenguaje *Java*, e identificar diferentes medidas de coherencia, cohesión, factor de acoplamiento y rendimiento de los servicios Web formados con la arquitectura de clases internas. Con este conocimiento se puede obtener otra forma de calcular la autonomía y rendimiento propuestos en esta tesis.

Para atender la restricción del código del Marco Orientado a Objetos de entrada, que considera que un Caso de Uso se distingue por una secuencia interactiva de métodos, iniciando desde los métodos públicos de clases públicas; se propone refactorizar un pre-procesamiento para cambiar el modificador de alcance de aquellos métodos públicos que no son iniciadores de casos de uso a métodos privados o protegidos. Esto es para obtener únicamente servicios Web que representan las capacidades del dominio de aplicaciones y evitar construir servicios Web que no tengan el nivel correcto de granulacióon.

Debido a que en este trabajo de tesis sólo se implementó un método de transformación de servicios web para conocer el balance de los atributos de calidad de autonomía y rendimiento, se propone generar servicios Web que consideren lo siguiente:

- Aislamiento: Separar del almacenamiento de datos la información que es requerida por el servicio Web.

- Independencia: Construir servicios Web de negocio complejos mediante la coordinación de servicios individuales.

- Encapsulamiento: Mostrar sólo los atributos y métodos necesarios para iniciar una secuencia interactiva de métodos.

Referencias Bibliográficas

Allam, D. (2014). *Loose coupling and substitution principle in objet-oriented frameworks for web services*. l'Université de Nantes Angers Le Mans.

Anzböck, R., & Dustdar, S. (2005). Semi-automatic Generation of Web Services and BPEL Processes – A Model-Driven Approach. *Business Process Management*, 64–79. https://doi.org/10.1007/11538394_5

Apache Software, F. (2017). Apache JMeter™. Retrieved November 12, 2017, from http://jmeter.apache.org/

Bianchini, D., Cappiello, C., De Antonellis, V., & Pernici, B. (2014). Service identification in interorganizational process design. *IEEE Transactions on Services Computing*, *7*(2), 265–278. https://doi.org/10.1109/TSC.2013.26

Bieman, J. M., Member, S., & Kang, B. (1998). Measuring Design-Level Cohesion, *24*(2), 111–124.

Bonér, J. (2017). *Reactive Microsystems*. (O'REILLY, Ed.) (2017th-8th–7th ed.). United States of America: Lightbend. Retrieved from https://www.lightbend.com/lagom-framework

Brito, F., Iseg, I., Goulão, M., Esteves, R., & Ist, I. (1995). Toward the Design Quality Evaluation of Object-Oriented Software Systems. *5th International Conference on Software Quality*, (October).

Chidamber, S. R., & Kemerer, C. F. (1994). A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, *20*(6), 476–493. https://doi.org/10.1109/32.295895

CompSci, S. C. (2016). *Nested Classes*. New Zeland. Retrieved from https://www.cs.auckland.ac.nz/courses/compsci230s1c/lectures/

Dahman, W., & Grabowski, J. (2011). UML-based specification and generation of executable web services. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *6598 LNCS*, 91–107. https://doi.org/10.1007/978-3-642-21652-7_6

Eisele, M. (2016). *Developing Reactive Microservices*. https://doi.org/978-1-491-96236-7

Erdemir, U., & Buzluca, F. (2014). A learning-based module extraction method for object-oriented systems. *Journal of Systems and Software*, *97*, 156–177. https://doi.org/10.1016/j.jss.2014.07.038

Erich, G., Helm, R., Johnson, R., & Vlissides, J. (2002). *Patrones de diseño*. Addison Wesley.

Fernández-villamor, J. I., Iglesias, C. A., & Garijo, M. (2014). A Framework for Goal-Oriented Discovery of Resources in the RESTful Architecture, *44*(6), 796–803.

Fowler, M. (1997). A Survey of Object Oriented Analysis and Design Methods. *Software Engineering, 1997., Proceedings of the 1997 International Conference on*. https://doi.org/10.1145/253228.253783

Fowler, M. (2002). Refactoring. *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002, IEEE Confe*, 13472. https://doi.org/10.1145/581441.581453

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (2002). *Design Patterns – Elements of Reusable Object-Oriented Software*. *A New Perspective on Object-Oriented Design*. https://doi.org/10.1093/carcin/bgs084

Geovani, V. D. S. (2016). *Refactorización de sistemas legados de software, para equilibrar la coherencia, cohesión y factor de acoplamiento de su estructura interna*. CENIDET.

Gosling, James; Joy, Bill; Steele, Guy; Bracha, Gilad; Buckley, A. (2014). The Java® Language Specification - java SE 8. *Addison-Wesley*, 688. Retrieved from https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf

Guadarrama Rogel, L. C. (2013). *Estudio Experimental para determinar la relación entre la estructura interna de Servicios Web y valores de QoS*. CENIDET.

Hitz, M., & Montazeri, B. (1996). Chidamber and Kemerer's Metrics Suite: A Measurement Theory Perspective, *22*(4), 267–271.

Hull, R., Schwander, P., Green, M. L., & Tung, R. T. (2000). Piensa en Java.

IBM-Analitycs. (2017). IBM SPSS Software. Retrieved December 7, 2017, from https://www.ibm.com/analytics/data-science/predictive-analytics/spss-statistical-software

Java™, T. (2016). *A Static Nested Class : Accessing Methods and Fields A Static Nested Class*.

Josuttis., N. M. (2007). *SOA in Practice, The Art of Distributed System Design*. O'Reilly Media.

Kimmel, P. (2008). *UML Demystified*. McGraw-Hill.

Kuhn, T., & Thomann, O. (2010). The Abstract Syntax Tree (AST). *Syntax*, pp. 1–11.

Kwon, Y. W., & Tilevich, E. (2014). Cloud refactoring: Automated transitioning to cloud-based services. *Automated Software Engineering*, *21*(3), 345–372. https://doi.org/10.1007/s10515-013-0136-9

Laurent, D. (2005). *Patrones de diseño en Java*.

Legorreta, R. N. E. (2017). *Transformación de Marcos de Aplicaciones Orientados a Objetos hacia Marcos con Arquitectura Orientada a Servicios*. Tesis de maestría - CENIDET.

León, P. F. (2009). *Generación de Servicios Web desde Marcos Orientados a Objetos a partir de Clases Colaborativas en Casos de Uso (MOOaSW)*. Tesis de maestría - CENIDET.

Martin, F. (2017). Micro-services. Retrieved November 12, 2017, from https://martinfowler.com/articles/microservices.html

Martin, R. C. (2002). *AGILE SOFTWARE DEVELOPMENT: PRINCIPLES, PATTERNS, AND PRACTICES*. *Nursing management*. https://doi.org/10.1002/pfi

Mateos, C., Crasso, M., Zunino, A., & Coscia, J. L. O. (2015). A Stitch in Time Saves Nine: Early Improving Code-First Web Services Discoverability. *Int. J. Cooperative Inf. Syst.*, *24*(2), 1–39. https://doi.org/10.1142/S0218843015500045

Mattsson, M., Bosch, J., & Ronneby, S.-. (1997). Framework Composition : Problems , Causes and Solutions Department of Computer Science and Business Administration. *Technology of Object-Oriented Languages and Systems, 1997. TOOLS 23. Proceedings*. https://doi.org/10.1109/TOOLS.1997.654724

Mcgovern, J., Tyagi, S., Stevens, M. E., & Mathew, S. (2003). *Java Web Services Architecture*. USA: Elsevier Science.

Nistor, A. (2015). Software Architectures – Present and Visions, *19*(4), 13–28. https://doi.org/10.12948/issn14531305/19.4.2015.02

Qu, Y., Bollig, E. F., & Erlebacher, G. (2008). KWATT: A toolkit for automatic web service generation. *Visual Geosciences*, *13*(1), 59–69. https://doi.org/10.1007/s10069-008-0009-8

Saldivar, M. G. (2015). *Plan de Pruebas para la Evalucación Experimental del Sistema*. CENIDET.

Santaolaya, R., Fragoso, O. G., Rojas, J. C., Álvarez, F. J., & León, F. (2016). *Method for Generating Web Services Frameworks from Object Oriented Frameworks Source Code*. *hoja de trabajo*.

Sharan, K. (2014). *Beginning Java 8 Language Features*. (Apress, Ed.) (3th ed.).

Sierra, A., & Casado, A. (2011). *Técnicas Avanzadas de Diseño de Software :* Universidad Rey Juan Carlos.

Sneed, H. M., Verhoef, C., & Sneed, S. H. (2013). Reusing existing object-oriented code as web services in a SOA. *c2013 IEEE 7th International Symposium on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems, MESOCA 2013*, 31–39. https://doi.org/10.1109/MESOCA.2013.6632732

Thomas, E. (2007). *SOA : Principles of Service Design*. (P. P. Hall, Ed.) (first edit). NJ, USA: Upper Saddle River.

Vinay, G., & Jain, A. (2012). Creating Web Services from Legacy Code. *International Journal of Computer Applications*, *1*(1), 21–23. Retrieved from http://www.researchgate.net/profile/Amit_Jain16/publication/256499136_Creating_Web_Services_from_Legacy_Code/links/00b4952329cf41896b000000.pdf

Walpole, R. E. (1999). *Probabilidad y Estadística para Ingenieros*. (6a, Ed.). México: Prentice-Hall Hispanoamericana.

Yadav, S., Sikka, S., & Shrivastava, U. (2014). A Review of Object-Oriented Coupling and Cohesion Metrics, *2*(5), 101–108.

Zur Muehlen, M., Nickerson, J. V., & Swenson, K. D. (2005). Developing web services choreography standards - The case of REST vs. SOAP. *Decision Support Systems*, *40*(1 SPEC. ISS.), 9–29. https://doi.org/10.1016/j.dss.2004.04.008

Anexos

Anexo A: Pruebas de rendimiento

A continuación, se presentan las pruebas de rendimiento para el marco statistic de los métodos MOOaSWCU, InlineClass y MOOinMS.

En la Tabla 13 se muestran los 13 Servicios Web del Marco Orientado a Objetos *Statistic*, como se establece en el diseño de los casos de prueba.

Tabla 13. Tabla con los casos de uso obtenidos del Maro Orientado a Objetos: *Statistic*

Caso de prueba	Nombre del caso de uso
CP01	ordena_cBubleXY_statistic
CP02	ordena_cBuble_statistic
CP03	calcula_cDistributionX2_statistic
CP04	calcula_cDistributionT_statistic
CP05	calcula_cDistribution_statistic
CP06	calcula_cDistItems_statistic
CP07	calcula_cCorrelation_statistic
CP08	calcula_cCalt_statistic
CP09	ordenacion1_List_statistic
CP10	ordenacion_List_statistic
CP11	CalculaCorrelacion_context_ctx_statistic
CP12	calcula_cSumXY_statistic
CP13	calcula_cRange_statistic

La salida correspondiente para la prueba ECP-MOOinMS-05, se registró en un archivo *.txt* con los tiempos arrojados por cada método (MOOaSWCU, Inlineclass y MOOinMS) en la herramienta *JMeter*. Se consideró seleccionar servicios web conformados por clases internas, de tal manera que posteriormente se puedan interpretar estadísticamente para hacer una inferencia hacia todos los Casos de Uso. Los tiempos obtenidos están registrados en milisegundos (ms).

Las pruebas de rendimiento para el marco statistic de los métodos MOOaSWCU, InlineClass y MOOinMS se muestran a continuación en las Tablas: 14, 15 y 16 respectivamente:

Tabla 14. Tiempo de respuesta* de los CU del marco statistic con el método MOOaSWCU

MOOASWCU													
No.	CP01	CP02	CP03	CP04	CP05	CP06	CP07	CP08	CP09	CP10	CP11	CP12	CP13
1	3	2	8	2	3	3	3	4	3	2	2	2	3
2	3	1	4	3	3	3	2	4	4	1	2	2	2
3	3	2	4	2	6	4	2	2	3	2	4	2	2
4	4	2	2	3	2	3	2	3	2	1	2	3	2
5	3	1	3	3	3	2	4	4	3	3	2	2	3
6	3	1	5	3	3	3	4	3	3	3	3	2	3
7	3	3	3	3	2	5	3	3	3	2	2	2	2
8	2	3	3	3	3	4	2	4	4	2	2	2	2
9	3	3	2	2	3	2	3	2	6	1	2	2	7
10	3	1	2	4	2	5	3	3	6	2	2	2	13
11	2	2	2	1	2	5	3	4	8	2	2	3	19
12	2	3	3	3	4	3	3	4	7	2	4	1	12
13	4	2	3	3	4	2	2	2	4	3	2	2	7
14	3	3	3	4	4	6	2	3	2	2	3	2	6
15	2	2	8	6	3	6	3	4	3	2	2	2	1
16	4	3	3	3	3	4	2	3	3	2	3	1	2
17	2	3	3	3	5	3	2	1	1	3	3	3	2
18	4	2	4	1	4	5	3	3	3	2	2	2	1
19	2	1	2	2	3	3	2	3	2	1	2	2	1
20	2	2	3	2	3	4	2	4	3	3	2	3	2
21	3	1	2	3	4	2	3	2	3	4	2	3	2
22	3	2	2	1	4	4	2	3	2	2	2	1	2
23	3	2	2	2	3	4	2	4	1	2	3	1	2
24	3	1	3	2	3	4	2	2	2	3	2	3	3
25	4	1	4	3	3	3	3	3	2	3	3	3	2
26	4	1	1	2	2	3	3	3	3	2	6	3	2
27	4	1	5	2	2	3	3	2	3	3	10	2	2
28	2	1	6	1	2	3	3	3	2	3	5	1	3
29	2	4	10	2	2	2	3	2	2	3	7	2	3
30	2	1	8	2	2	3	3	4	2	3	4	3	4
31	3	2	11	1	2	2	2	4	2	3	4	3	3
32	3	2	9	2	2	3	3	4	3	3	3	2	3
33	4	1	9	4	3	3	3	3	2	3	2	1	4
34	1	1	5	3	3	3	3	3	4	2	2	2	2
35	3	3	4	2	3	3	1	3	4	2	2	3	3
36	2	2	17	4	1	3	1	3	11	3	3	2	2
37	3	2	11	3	2	4	2	4	1	2	5	3	2
38	3	4	2	4	3	3	2	3	17	3	3	2	2
39	2	3	2	5	2	3	3	6	5	1	3	3	3
40	4	3	2	5	3	3	3	5	4	2	3	3	2
41	2	4	2	2	2	2	2	4	5	2	2	2	6
42	3	9	2	3	2	2	2	3	2	3	2	3	2
43	1	4	3	3	2	2	3	14	3	4	2	2	2
44	3	4	3	3	1	2	3	3	3	2	1	2	2
45	3	2	3	3	2	3	3	4	3	3	3	2	2
46	2	2	2	5	3	3	2	6	3	2	3	2	3
47	2	3	2	5	3	3	2	6	6	3	4	3	2
48	4	1	2	4	2	2	3	4	4	2	2	2	2
49	2	3	2	3	1	3	3	2	2	3	3	2	2
50	3	2	2	6	2	2	2	3	4	2	2	1	2
51	2	2	2	4	2	3	2	2	2	2	2	2	3
52	2	1	4	4	2	4	3	2	2	2	2	2	2
53	3	1	2	3	2	4	3	3	4	3	4	2	38
54	4	1	2	3	2	2	3	3	4	1	3	2	25
55	4	3	2	3	2	4	3	3	3	2	4	2	19
56	2	2	2	3	2	4	2	3	6	2	3	3	16

*****Tiempo de respuesta en milisegundos (ms) aplicando a 306 pruebas por caso de uso y tomando una muestra heterogenea de 56 elementos.

Tabla 15. Tiempo de respuesta* de los CU del marco statistic con el método InlineClass

InlineClass													
No.	CP01	CP02	CP03	CP04	CP05	CP06	CP07	CP08	CP09	CP10	CP11	CP12	CP13
1	1	2	2	3	1	2	13	1	2	2	2	2	1
2	3	2	2	1	1	1	11	1	2	1	1	1	2
3	3	2	2	1	2	2	5	1	1	1	1	1	2
4	2	2	2	2	2	1	6	3	2	1	2	1	2
5	2	3	0	2	2	2	12	3	2	1	2	2	2
6	2	1	1	2	2	2	13	2	2	2	1	1	2
7	3	2	1	2	2	1	12	2	2	2	1	2	2
8	3	1	2	1	2	2	11	2	1	1	3	2	2
9	2	2	2	1	2	1	10	2	2	1	1	1	2
10	1	2	1	3	2	1	7	2	2	2	2	2	2
11	2	2	1	4	1	2	6	1	1	2	3	2	2
12	1	2	1	2	1	1	5	2	1	1	2	2	2
13	2	2	1	3	1	2	9	2	1	3	2	2	1
14	2	2	3	3	3	1	12	2	1	2	1	2	2
15	3	1	2	3	2	1	13	1	2	2	2	1	2
16	3	2	1	3	1	1	1	1	2	2	2	2	1
17	2	1	2	2	2	2	1	1	1	2	2	1	1
18	4	2	2	3	2	2	8	1	1	1	1	0	1
19	2	2	2	2	2	2	8	1	1	0	1	2	2
20	2	1	1	3	1	3	4	2	2	1	2	1	2
21	2	4	2	1	2	2	2	2	1	2	3	2	1
22	3	1	1	2	2	1	1	2	2	2	3	2	3
23	3	3	2	2	2	1	2	1	2	1	3	1	2
24	3	3	2	2	2	1	1	2	2	1	2	1	2
25	3	1	2	2	2	2	2	2	2	1	2	1	2
26	3	8	2	3	2	1	2	1	3	1	2	1	2
27	3	1	2	2	2	1	1	2	2	2	2	1	2
28	3	1	1	2	2	1	1	2	2	2	2	3	1
29	3	9	1	1	2	2	2	1	2	2	2	1	1
30	2	5	8	2	1	1	1	1	5	1	4	1	1
31	3	3	7	2	1	2	2	2	18	2	3	2	1
32	2	2	2	2	2	2	2	1	14	2	1	2	1
33	2	2	3	2	3	1	2	2	7	1	2	1	1
34	2	1	3	2	2	2	1	2	25	2	6	1	2
35	1	2	3	1	2	1	2	2	31	2	4	1	2
36	2	2	1	3	2	2	2	1	2	2	3	3	2
37	2	1	4	2	2	2	2	1	2	1	2	2	1
38	2	2	2	3	2	2	2	2	1	1	3	2	2
39	1	1	2	3	3	1	2	3	3	1	2	2	2
40	2	1	2	2	1	1	2	2	2	1	4	2	1
41	3	1	2	1	2	1	2	1	2	1	1	3	2
42	2	1	2	2	1	1	2	2	2	1	1	2	2
43	3	1	2	2	2	1	2	1	2	2	2	1	2
44	3	1	3	2	2	2	2	1	2	1	2	2	1
45	2	2	4	2	1	1	2	1	2	1	1	2	1
46	3	1	3	1	2	2	1	2	1	1	2	1	1
47	1	1	2	2	2	2	2	2	2	2	2	1	1
48	2	1	2	2	2	2	2	2	2	1	3	2	1
49	3	2	3	2	2	2	1	2	2	2	2	1	1
50	3	2	2	3	1	2	1	2	1	1	2	2	2
51	4	1	2	3	3	1	1	2	1	1	2	2	1
52	1	1	3	3	3	1	2	2	2	1	2	1	1
53	3	1	3	3	2	2	2	1	1	3	11	1	1
54	3	4	2	3	1	1	1	1	2	1	9	1	2
55	2	3	3	4	1	2	3	3	2	2	6	2	1
56	3	2	3	4	2	2	2	2	1	2	2	1	2

*****Tiempo de respuesta en milisegundos (ms) aplicando a 306 pruebas por caso de uso y tomando una muestra heterogénea de 56 elementos.

Tabla 16. Tiempo de respuesta* de los CU del marco statistic con el método MOOinMS

MOOinMS													
No.	CP01	CP02	CP03	CP04	CP05	CP06	CP07	CP08	CP09	CP10	CP11	CP12	CP13
1	1	2	0	1	1	1	2	2	1	2	3	4	1
2	2	2	1	1	3	2	1	2	2	1	6	6	1
3	2	2	1	1	2	1	2	0	2	2	14	4	2
4	3	2	2	2	2	2	2	2	1	1	10	2	1
5	1	2	2	2	2	1	2	1	2	1	15	2	2
6	2	1	2	3	2	2	2	1	2	1	12	3	2
7	2	2	1	1	1	2	2	1	2	2	7	3	1
8	1	2	2	1	2	2	3	2	1	2	3	1	1
9	1	2	1	1	2	2	2	1	2	1	3	2	1
10	2	2	0	1	2	1	1	1	1	2	4	3	2
11	2	2	2	1	2	2	1	2	1	2	4	3	1
12	2	2	2	1	2	1	2	2	2	1	3	2	3
13	2	2	1	1	2	2	2	1	1	2	2	2	1
14	2	1	1	1	2	1	2	1	1	2	3	2	1
15	2	1	2	2	3	1	1	1	1	1	8	2	1
16	3	1	2	1	2	2	1	2	2	1	4	1	1
17	2	1	1	1	1	1	1	1	3	1	2	2	1
18	2	1	1	2	1	1	2	1	2	1	3	3	1
19	2	1	1	1	1	2	1	2	2	1	4	2	1
20	1	2	2	2	1	1	3	2	2	2	5	2	2
21	2	2	1	1	1	1	2	4	2	1	3	2	1
22	2	2	1	2	1	1	1	3	2	2	4	2	2
23	2	2	2	1	1	1	2	3	1	1	4	1	2
24	2	2	2	2	2	1	1	2	1	2	2	1	2
25	1	1	2	0	2	2	2	2	1	1	3	1	2
26	2	1	1	1	1	2	1	2	2	2	3	1	2
27	3	1	2	1	1	2	1	2	1	1	3	1	1
28	1	1	2	2	1	1	1	2	1	1	3	2	1
29	1	1	1	1	2	1	2	2	2	2	1	2	0
30	1	2	1	1	2	1	1	2	2	1	3	1	1
31	1	2	2	3	2	1	2	2	2	1	10	2	2
32	2	2	2	1	1	2	1	2	2	1	3	2	2
33	3	2	2	1	2	1	1	2	1	1	2	2	1
34	2	2	1	1	1	1	0	3	2	2	3	1	2
35	1	2	1	2	2	1	0	1	1	0	3	2	1
36	2	2	2	1	2	1	1	2	1	1	1	2	2
37	2	2	1	1	2	8	1	1	1	1	1	1	1
38	1	2	2	2	1	8	2	2	1	2	2	1	1
39	2	2	2	1	1	3	1	2	1	1	3	2	1
40	1	2	1	1	1	1	1	1	2	1	4	1	2
41	2	3	1	0	1	1	1	2	2	2	3	1	2
42	1	3	2	1	1	2	1	2	3	1	3	2	1
43	2	4	2	1	2	2	2	1	1	1	3	3	2
44	2	3	1	1	1	1	2	2	4	2	2	2	2
45	2	2	2	1	1	1	1	2	3	2	2	1	1
46	2	2	2	1	2	2	1	3	2	2	2	1	1
47	3	4	1	1	2	2	1	1	2	1	2	1	1
48	3	1	1	1	2	3	1	1	3	1	1	1	2
49	1	1	2	1	1	1	1	2	4	2	4	2	1
50	1	3	2	1	2	2	1	2	3	2	4	2	1
51	1	2	2	2	1	1	2	1	3	2	2	1	1
52	1	2	1	1	1	1	1	2	2	1	2	3	1
53	2	2	1	1	2	1	1	1	1	1	2	2	1
54	3	2	1	0	1	1	1	2	2	1	2	2	1
55	2	1	2	1	1	2	2	3	3	1	2	1	0
56	2	1	2	2	2	2	1	1	2	1	1	1	1

*****Tiempo de respuesta en milisegundos (ms) aplicando a 306 pruebas por caso de uso y tomando una muestra heterogénea de 56 elementos.

Una vez que se determinó que los resultados de las poblaciones son datos no paramétricos, las pruebas no paramétricas a realizar y que resultan adecuadas para un experimento con k > 2 para poblaciones independientes es la prueba ANOVA como alternativa no paramétrica de la t de Student para muestras independientes. En la Tabla 17 se muestran los resultados obtenidos.

Tabla 17. Resultados de la prueba no paramétrica ANOVA para los métodos: MOOaSWCU, InlineClass y MOOinMS

ServicioWeb	MOOaSWCU	InlineClass	MOOinMS	Arquitectura más eficiente
1. cBubleX	2.804	2.375	**1.804**	MOOinMS
2. cBuble	2.214	2.036	**1.857**	MOOinMS
3. cDistributionX2	3.964	2.214	**1.482**	MOOinMS
4. cDistributionT	2.964	2.250	**1.232**	MOOinMS
5. cDistribution	2.643	1.821	**1.571**	MOOinMS
6. cDistItems	3.232	**1.536**	1.696	InlineClass
7. cCorrelation	2.554	4.089	**1.411**	MOOinMS
8. cCalt	3.464	**1.679**	1.750	InlineClass
9. List	3.643	3.321	**1.821**	MOOinMS
10. List	2.339	**1.482**	1.375	InlineClass
11. context	2.929	**2.482**	3.804	InlineClass
12. cSumXY	2.179	**1.554**	1.911	InlineClass
13. cRange	4.839	1.589	**1.339**	MOOinMS

En la Figura 63 se muestra la gráfica correspondiente a la Tabla 17.

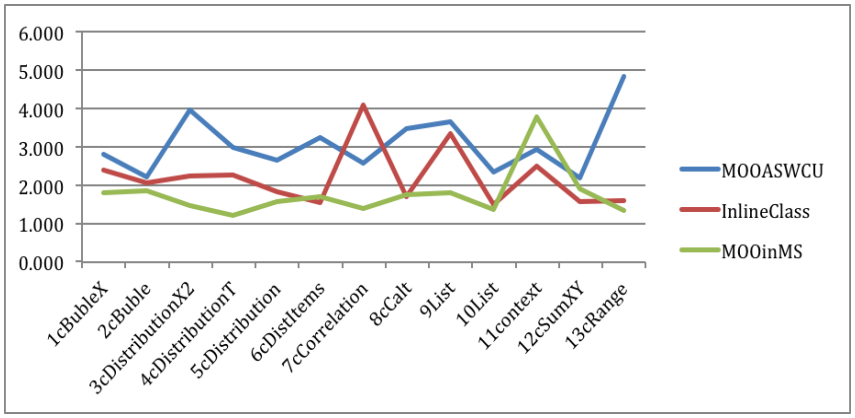


Figura 63. Tiempo de respuesta en microsegundos para los servicios web por métodos de transformación.

En la Figura 64 se muestra la arquitectura más eficiente como resultado de la prueba no paramétrica ANOVA.

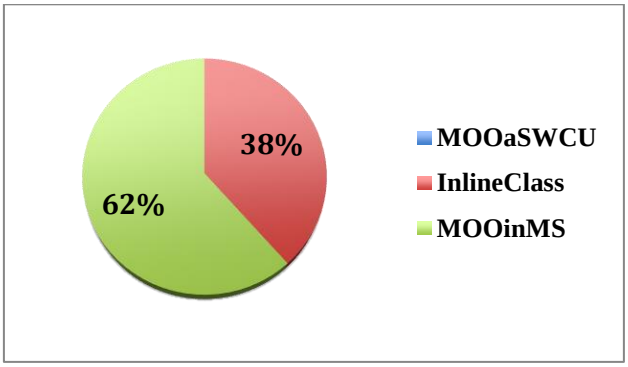


Figura 64. Porcentaje de eficiencia del tiempo de respuesta de los servicios web del MOO statistic

Anexo B: Pruebas de Autonomía (coherencia, cohesión y acoplamiento)

A continuación, se presentan las pruebas de autonomía de los servicios web con los enfoques MOOaSWCU, InlineClass y MOOinMS.

En la Tabla 18 se muestran los 13 Servicios Web del Marco Orientado a Objetos *Statistic*, como se establece en el diseño de los casos de prueba

Tabla 18. Tabla con los casos de uso obtenidos del Maro Orientado a Objetos: *Statistic*

Caso de prueba	Nombre del caso de uso
CP01	ordena_cBubleXY_statistic
CP02	ordena_cBuble_statistic
CP03	calcula_cDistributionX2_statistic
CP04	calcula_cDistributionT_statistic
CP05	calcula_cDistribution_statistic
CP06	calcula_cDistItems_statistic
CP07	calcula_cCorrelation_statistic
CP08	calcula_cCalt_statistic
CP09	ordenacion1_List_statistic
CP10	ordenacion_List_statistic
CP11	CalculaCorrelacion_context_ctx_statistic
CP12	calcula_cSumXY_statistic
CP13	calcula_cRange_statistic

El cálculo correspondiente para la prueba ECP-MOOinMS-06, de coherencia, cohesión y acoplamiento, para cada uno de los casos de uso del marco statistic se muestran a continuación:

a. Coherencia

En la tabla 19 se muestra la coherencia para el marco statistic.

Tabla 19. Coherencia de los casos de uso de los métodos MOOaSWCU, InlineClass y MOOinMS

No.	CU	MOOaSWCU			InlineClass			MOOinMS		
		tm	msi	Chr	tm	msi	Chr	tm	msi	Chr
1	ordena_cBubleXY_statistic	6	6	1	6	6	1	6	6	1
2	ordena_cBuble_statistic	9	9	1	9	9	1	9	9	1
3	calcula_cDistributionX2_statistic	12	12	1	12	12	1	12	12	1
4	calcula_cDistributionT_statistic	8	8	1	8	8	1	8	8	1
5	calcula_cDistribution_statistic	10	10	1	10	10	1	10	10	1
6	calcula_cDistItems_statistic	10	10	1	10	10	1	10	10	1
7	calcula_cCorrelation_statistic	20	20	1	20	20	1	20	20	1
8	calcula_cCalt_statistic	10	10	1	10	10	1	10	10	1
9	ordenacion1_List_statistic	12	12	1	12	12	1	12	12	1
10	ordenacion_List_statistic	10	10	1	10	10	1	10	10	1
11	CalculaCorrelacion_context_ctx_statistic	17	17	1	17	17	1	17	17	1
12	calcula_cSumXY_statistic	7	7	1	7	7	1	7	7	1
13	calcula_cRange_statistic	15	15	1	15	15	1	15	15	1

tm = total de métodos del CU, donde tm > 0
msi = métodos de la secuencia intractiva del CU

b. Cohesión

A continuación, para cada servicio Web se realiza el cálculo de la cohesión.

1. CU: ordena_cBubleXY_statistic

No.	Clase	LCOM4		
		MOOaSWCU	**InlineClass**	**MOOinMS**
1	cBubleXY	1	5	1
2	context_ctx	0	-	0
3	cElement	2	-	2
4	List	1	-	1
5	cBuble	1	-	1

La cohesión para el caso de uso: **ordena_cBubleXY_statistic**
MOOaSWCU_CUCoh = (1 + 0 + 2 + 1 + 1) / 5
MOOaSWCU_CUCoh = 1

InlineClass_CUCoh = 5 / 1
InlineClass_CUCoh = 5

MOOinMS_CUCoh = (1 + 0 + 2 + 1 + 1) / 5
MOOinMS_CUCoh = 1

2. CU: ordena_cBuble_statistic

No.	Clase	LCOM4		
		MOOaSWCU	**InlineClass**	**MOOinMS**
1	cBuble	1	4	1
2	context_ctx	0	-	0
3	cElement	2	-	2
4	List	1	-	1

La cohesión para el caso de uso: **ordena_cBuble_statistic**
MOOaSWCU_CUCoh = (1 + 0 + 2 + 1) / 4
MOOaSWCU_CUCoh = 1

InlineClass_CUCoh = 4 / 1
InlineClass_CUCoh = 4

MOOinMS_CUCoh = (1 + 0 + 2 + 1) / 4
MOOinMS_CUCoh = 1

3. CU: calcula_cDistributionX2_statistic

No.	Clase	LCOM4		
		MOOaSWCU	InlineClass	MOOinMS
1	cDistributionX2	1	5	1
2	Integ	1	-	2
3	Distrtibution	2	-	2
4	context_ctx	1	-	1

La cohesión para el caso de uso: **calcula_cDistributionX2_statistic**
MOOaSWCU_CUCoh = (1 + 1 + 2 + 1) / 4
MOOaSWCU_CUCoh = 1.25

InlineClass_CUCoh = 5 / 1
InlineClass_CUCoh = 5

MOOinMS_CUCoh = (1 + 1+ 2 + 1) / 4
MOOinMS_CUCoh = 1.25

4. CU: calcula_cDistributionT_statistic

No.	Clase	LCOM4		
		MOOaSWCU	InlineClass	MOOinMS
1	cDistributionT	1	5	1
2	Integ	1	-	1
3	Distrtibution	2	-	2
4	context_ctx	1	-	1

La cohesión para el caso de uso: **calcula_cDistributionT_statistic**
MOOaSWCU_CUCoh = (1 + 1 + 2 + 1) / 4
MOOaSWCU_CUCoh = 1.25

InlineClass_CUCoh = 5 / 1
InlineClass_CUCoh = 5

MOOinMS_CUCoh = (1 + 1 + 2 + 1) / 4
MOOinMS_CUCoh = 1.25

5. CU: calcula_cDistribution_statistic

No.	Clase	LCOM4		
		MOOaSWCU	InlineClass	MOOinMS
1	cDistribution	1	5	1
2	Integ	1	-	1
3	Distrtibution	2	-	2
4	context_ctx	1	-	1

La cohesión para el caso de uso: **calcula_cDistribution_statistic**
MOOaSWCU_CUCoh = (1 + 1 + 2 + 1) / 4
MOOaSWCU_CUCoh = 1.25

InlineClass_CUCoh = 5 / 1
InlineClass_CUCoh = 5

MOOinMS_CUCoh = (1 + 1 + 2 + 1) / 4
MOOinMS_CUCoh = 1.25

6. CU: calcula_cDistItems_statistic

No.	Clase	LCOM4		
		MOOaSWCU	InlineClass	MOOinMS
1	cDistItems	1	5	1
2	Integ	1	-	1
3	Distrtibution	2	-	2
4	context_ctx	1	-	1

La cohesión para el caso de uso: **calcula_cDistItems_statistic**
MOOaSWCU_CUCoh = (1 + 1 + 2 + 1) / 4
MOOaSWCU_CUCoh = 1.25

InlineClass_CUCoh = 5 / 1
InlineClass_CUCoh = 5

MOOinMS_CUCoh = (1 + 1 + 2 + 1) / 4
MOOinMS_CUCoh = 1.25

7. CU: calcula_cCorrelation_statistic

No.	Clase	LCOM4		
		MOOaSWCU	InlineClass	MOOinMS
1	cCorrelation	1	6	1
2	cSumXY	0	-	0
3	cSum	0	-	0
4	cSquareSum	1	-	1
5	context_ctx	1	-	1
6	List	2	-	2
7	cElement	2	-	2

La cohesión para el caso de uso: **calcula_cCorrelation_statistic**
MOOaSWCU_CUCoh = (1 + 0 + 0 + 1 + 1 + 2 + 2) / 7
MOOaSWCU_CUCoh = 1

InlineClass_CUCoh = 6 / 1
InlineClass_CUCoh = 6

MOOinMS_CUCoh = (1 + 0 + 0 + 1 + 1 + 2 + 2) / 7
MOOinMS_CUCoh = 1

8. CU: calcula_cCalt_statistic

No.	Clase	LCOM4		
		MOOaSWCU	InlineClass	MOOinMS
1	cCalt	1	2	1
2	context_ctx	1	-	1

La cohesión para el caso de uso: **calcula_cCalt_statistic**
MOOaSWCU_CUCoh = (1 + 1) / 2
MOOaSWCU_CUCoh = 1

InlineClass_CUCoh = 2 / 1
InlineClass_CUCoh = 2

MOOinMS_CUCoh = (1 + 1) / 2
MOOinMS_CUCoh = 1

9. CU: ordenacion1_List_statistic

No.	Clase	LCOM4		
		MOOaSWCU	InlineClass	MOOinMS
1	List	1	5	1
2	context_ctx	0	-	0
3	cBubleXY	1	-	1
4	cBuble	1	-	1
5	cElement	2	-	2

La cohesión para el caso de uso: **ordenacion1_List_statistic**
MOOaSWCU_CUCoh = (1 + 0 + 1 + 1 + 2) / 5
MOOaSWCU_CUCoh = 1

InlineClass_CUCoh = 5 / 1
InlineClass_CUCoh = 5

MOOinMS_CUCoh = (1 + 0 + 1 + 1 + 2) / 5
MOOinMS_CUCoh = 1

10. CU: ordenacion_List_statistic

No.	Clase	LCOM4		
		MOOaSWCU	InlineClass	MOOinMS
1	List	1	5	1
2	context_ctx	0	-	0
3	cBubleXY	1	-	1
4	cBuble	1	-	1
5	cElement	2	-	2

La cohesión para el caso de uso: **ordenacion_List_statistic**
MOOaSWCU_CUCoh = (1 + 0 + 1 + 1 + 2) / 5
MOOaSWCU_CUCoh = 1

InlineClass_CUCoh = 5 / 1
InlineClass_CUCoh = 5

MOOinMS_CUCoh = (1 + 0 + 1 + 1 + 2) / 5
MOOinMS_CUCoh = 1

11. CU: CalculaCorrelacion_context_ctx_statistic

No.	Clase	LCOM4		
		MOOaSWCU	InlineClass	MOOinMS
1	context_ctx	1	5	1
2	List	0	-	0
3	cCorrelation	1	-	1
4	cSumXY	1	-	1
5	cSum	1	-	1
6	cSquareSum	1		1

La cohesión para el caso de uso: **CalculaCorrelacion_context_ctx_statistic**

MOOaSWCU_CUCoh = (1 + 0 + 1 + 1 + 1 + 1) / 6
MOOaSWCU_CUCoh = 0.83

InlineClass_CUCoh = 5 / 1
InlineClass_CUCoh = 5

MOOinMS_CUCoh = (1 + 0 + 1 + 1 + 1 + 1) / 6
MOOinMS_CUCoh = 0.83

12. CU: calcula_cSumXY_statistic

No.	Clase	LCOM4		
		MOOaSWCU	InlineClass	MOOinMS
1	cSumXY	1	4	1
2	cElement	1	-	1
3	context_ctx	1	-	1
4	List	1	-	1

La cohesión para el caso de uso: **calcula_cSumXY_statistic**
MOOaSWCU_CUCoh = (1 + 1 + 1 + 1) / 4
MOOaSWCU_CUCoh = 1

InlineClass_CUCoh = 4 / 1
InlineClass_CUCoh = 4

MOOinMS_CUCoh = (1 + 1 + 1 + 1) / 4
MOOinMS_CUCoh = 1

13. CU: calcula_cRange_statistic

No.	Clase	LCOM4		
		MOOaSWCU	InlineClass	MOOinMS
1	cRange	1	7	1
2	cDevY	1	-	1
3	cSumY	0	-	0
4	cDistributionT	2	-	2
5	aStatistic	0	-	0
6	context_ctx	1	-	1
7	Integ	1	-	1
8	Distribution	2	-	2
9	List	1	-	1
10	cElement	1	-	1

La cohesión para el caso de uso: **ocalcula_cRange_statistic**
MOOaSWCU_CUCoh = (1 + 1 + 0 + 2 + 0 + 1 + 1 + 2 + 1 + 1) / 10
MOOaSWCU_CUCoh = 1

InlineClass_CUCoh = 7 / 1
InlineClass_CUCoh = 7

MOOinMS_CUCoh = (1 + 1 + 0 + 2 + 0 + 1 + 1 + 2 + 1 + 1) / 10
MOOinMS_CUCoh = 1

En la Tabla 20 se muestra la cohesión de los casos de uso del MOO statistic para cada uno de los métodos de transformación.

Tabla 20. Cohesión de los casos de uso de los métodos MOOaSWCU, InlineClass y MOOinMS

No.	CU	MOOaSWCU	InlineClass	MOOinMS
		CUCoh	CUCoh	CUCoh
1	ordena_cBubleXY_statistic	1	5	1
2	ordena_cBuble_statistic	1	4	1
3	calcula_cDistributionX2_statistic	1.25	5	1.25
4	calcula_cDistributionT_statistic	1.25	5	1.25
5	calcula_cDistribution_statistic	1.25	5	1.25
6	calcula_cDistItems_statistic	1.25	5	1.25
7	calcula_cCorrelation_statistic	1	6	1
8	calcula_cCalt_statistic	1	2	1
9	ordenacion1_List_statistic	1	5	1
10	ordenacion_List_statistic	1	5	1
11	CalculaCorrelacion_context_ctx_statistic	0.83	5	0.83
12	calcula_cSumXY_statistic	1	4	1
13	calcula_cRange_statistic	1	7	1

c. Acoplamiento

En la Tabla 21 se muestra el acoplamiento para el marco statistic.

Tabla 21. Factor de Acoplamiento de los métodos MOOaSWCU, InlineClass y MOOinMS

No.	CU	MOOaSWCU			InlineClass			MOOinMS		
		Σ_D	TC	COF	Σ_D	TC	COF	Σ_D	TC	COF
1	ordena_cBubleXY_statistic	11	5	0.550	11	1	0	2	5	0.100
2	ordena_cBuble_statistic	13	4	1.083	13	1	0	1	4	0.083
3	calcula_cDistributionX2_statistic	5	4	0.417	5	1	0	1	4	0.083
4	calcula_cDistributionT_statistic	5	4	0.417	5	1	0	1	4	0.083
5	calcula_cDistribution_statistic	5	4	0.417	5	1	0	1	4	0.083
6	calcula_cDistItems_statistic	3	4	0.250	3	1	0	1	4	0.083
7	calcula_cCorrelation_statistic	4	7	0.095	4	1	0	2	7	0.048
8	calcula_cCalt_statistic	2	2	1.000	2	1	0	1	2	0.500
9	ordenacion1_List_statistic	3	5	0.150	3	1	0	1	5	0.050
10	ordenacion_List_statistic	3	5	0.150	3	1	0	1	5	0.050
11	CalculaCorrelacion_context_ctx_statistic	5	6	0.167	5	1	0	2	6	0.067
12	calcula_cSumXY_statistic	2	4	0.167	2	1	0	1	4	0.083
13	calcula_cRange_statistic	7	10	0.078	7	1	0	3	10	0.033

d. Autonomía

En la Tabla 22 se muestran los resultados para la métrica de autonomía para los casos de uso generados con el método MOOaSWCU.

Tabla 22. Métrica de autonomía del método MOOaSWCU para el marco statistic

No.	CU	MOOaSWCU			
		Chr	CUCoh	COF	**CUaut**
1	ordena_cBubleXY_statistic	1	1	0.550	1.550
2	ordena_cBuble_statistic	1	1	1.083	2.083
3	calcula_cDistributionX2_statistic	1	1.25	0.417	1.542
4	calcula_cDistributionT_statistic	1	1.25	0.417	1.542
5	calcula_cDistribution_statistic	1	1.25	0.417	1.542
6	calcula_cDistItems_statistic	1	1.25	0.250	1.375
7	calcula_cCorrelation_statistic	1	1	0.095	1.095
8	calcula_cCalt_statistic	1	1	1.000	2.000
9	ordenacion1_List_statistic	1	1	0.150	1.150
10	ordenacion_List_statistic	1	1	0.150	1.150
11	CalculaCorrelacion_context_ctx_statistic	1	0.83	0.167	1.082
12	calcula_cSumXY_statistic	1	1	0.167	1.167
13	calcula_cRange_statistic	1	1	0.078	1.078

Promedio de Cuaut MOOaSWCU = **1.412**

En la Tabla 23 se muestran los resultados para la métrica de autonomía para los casos de uso generados con el método InlineClass.

Tabla 23. Métrica de autonomía del método InlineClass para el marco statistic

No.	CU	InlineClass			
		Chr	CUCoh	COF	**CUaut**
1	ordena_cBubleXY_statistic	1	5	0.000	3.000
2	ordena_cBuble_statistic	1	4	0.000	2.500
3	calcula_cDistributionX2_statistic	1	5	0.000	3.000
4	calcula_cDistributionT_statistic	1	5	0.000	3.000
5	calcula_cDistribution_statistic	1	5	0.000	3.000
6	calcula_cDistItems_statistic	1	5	0.000	3.000
7	calcula_cCorrelation_statistic	1	6	0.000	3.500
8	calcula_cCalt_statistic	1	2	0.000	1.500
9	ordenacion1_List_statistic	1	5	0.000	3.000
10	ordenacion_List_statistic	1	5	0.000	3.000
11	CalculaCorrelacion_context_ctx_statistic	1	5	0.000	3.000
12	calcula_cSumXY_statistic	1	4	0.000	2.500
13	calcula_cRange_statistic	1	7	0.000	4.000

Promedio de Cuaut InlineClass = **2.923**

En la Tabla 24 se muestran los resultados para la métrica de autonomía para los casos de uso generados con el método MOOinMS.

Tabla 24. Métrica de autonomía del método MOOinMS para el marco statistic

No.	CU	MOOinMS			
		Chr	CUCoh	COF	CUaut
1	ordena_cBubleXY_statistic	1	1	0.100	1.100
2	ordena_cBuble_statistic	1	1	0.083	1.083
3	calcula_cDistributionX2_statistic	1	1.25	0.083	1.208
4	calcula_cDistributionT_statistic	1	1.25	0.083	1.208
5	calcula_cDistribution_statistic	1	1.25	0.083	1.208
6	calcula_cDistItems_statistic	1	1.25	0.083	1.208
7	calcula_cCorrelation_statistic	1	1	0.048	1.048
8	calcula_cCalt_statistic	1	1	0.500	1.500
9	ordenacion1_List_statistic	1	1	0.050	1.050
10	ordenacion_List_statistic	1	1	0.050	1.050
11	CalculaCorrelacion_context_ctx_statistic	1	0.83	0.067	0.982
12	calcula_cSumXY_statistic	1	1	0.083	1.083
13	calcula_cRange_statistic	1	1	0.033	1.033

Promedio de Cuaut MOOinMS = **1.136**

Para verificar cuál de las tres arquitecturas tiene mejores cualidades de autonomía se generó la gráfica de la Figura 65 que muestra que el método MOOinMS conserva un balance entre sus atributos de calidad: Coherencia, Cohesión y Acoplamiento.

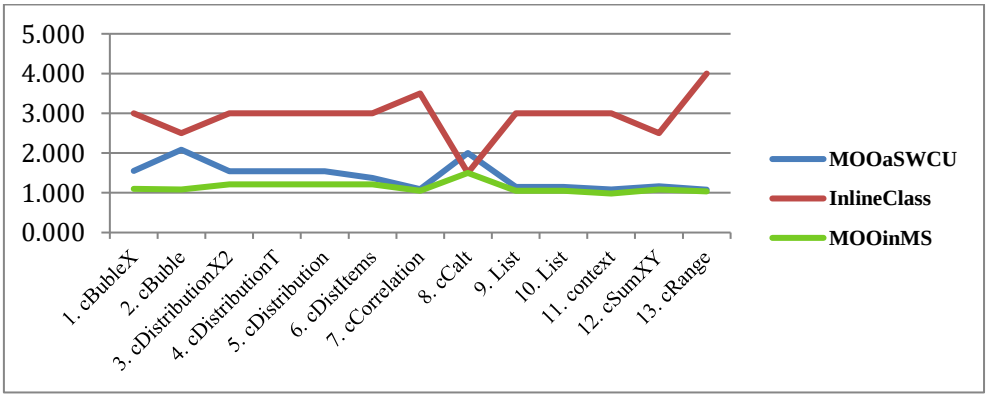


Figura 65. Autonomía de los servicios web del marco statistic.

La diferencia de los promedios de autonomía de los servicios web generados con el método MOOinMS y los métodos MOOaSWCU e InlineClass se muestra en la Tabla 25.

Tabla 25. Diferencia de promedios de autonomía entre las arquitecturas de los métodos: MOOaSWCU, InlineClass y MOOinMS

	Promedio	Diferencia	% de autonomía
MOOaSWCU	1.412	0.276	24
InlineClass	2.923	1.787	64
MOOinMS	**1.136**	-	100

La arquitectura con menores cualidades de autonomía la tienen los servicios web generados con el método InlineClass.