


Tecnológico Nacional de México

Instituto Tecnológico de Orizaba

División de Estudios de Posgrado e Investigación

Maestría en Sistemas Computacionales

TESIS

TÍTULO DE LA TESIS:

Estudio de Java 10 y AspectJ para la implementación de aplicaciones modulares

PRESENTADO POR:

I.S.C. Julio Andrés Beverido Castellanos

PARA OBTENER EL GRADO DE:

Maestro en Sistemas Computacionales

DIRECTOR DE TESIS:

Dr. Ulises Juárez Martínez

Orizaba, Ver.

Índice general

Índice de Tablas

Índice de figuras

Índice de Códigos

Resumen

La modularidad es uno de los grandes retos de la Ingeniería de Software, la correcta separación de módulos permite el desarrollo de software de calidad, al cual, se le logra dar un fácil mantenimiento. Los principios de modularidad se derivan de la separación de asuntos, dicho proceso se dificulta por los requerimientos y sus respectivos dominios de aplicación, los cuales, tienden a exceder los límites de los módulos. Si bien existen en la actualidad lenguajes que intentan dar soporte modular como C++, C# y Ptolemy dichos lenguajes carecen de una modularidad a nivel de entorno de ejecución como Java 10.

Considerando el potencial que brinda Java 10, la presente tesis estudia las características que ofrece Java 10 a nivel de código y entorno. Paralelamente estudiando las versiones de AspectJ que estén disponibles para Java 10, realizando pruebas que determinen el impacto positivo que tienen los aspectos en materia de modularidad, de manera que, las aplicaciones desarrolladas bajo dichas tecnologías se encuentren en el esquema modular más cercano posible.

Abstract

Modularity is one of the big challenges in software engineering, separation of concerns allows a high-quality software development, which is maintainable. The principles of modularity derives from the separation of concerns, such process gains difficulty due to the software requirements which exceed module limits. Even though there are languages such as C++, C# and Ptolemy which provide modular development, they lack of an environment-level modularity unlike Java 10.

Considering the high potential of Java 10, this thesis studies the features of Java 10 at a code and environment level. At the same time studying the different available versions of AspectJ for Java 10, applying tests which determine the positive impact that aspects offer in a modular matter, in such a way that, applications developed with these technologies can be in a barely modular scheme.

Introducción

El concepto de modularidad [1, 2] enfatiza la separación de funcionalidad de un programa en módulos independientes e intercambiables, de tal forma que cada uno de los módulos tenga solo lo necesario para la ejecución de un único aspecto de la funcionalidad requerida. Cada módulo exhibe una interfaz donde se especifican los elementos que ofrece y al mismo tiempo los que requiere, de tal forma que otros módulos los detecten. Los paradigmas orientados a objetos, componentes y servicios ofrecen soluciones robustas de modularidad, sin embargo, las particularidades que cada solución ofrece no permiten la definición adecuada de los módulos.

Java 10 brinda soluciones a nivel de código y entorno de ejecución, lo cual, facilita la descomposición modular y permite el desarrollo de aplicaciones que se asemejen a un desarrollo arquitectónico basado en componentes. La Programación Orientada a Aspectos (POA) es una técnica de programación basada en los principios de separación de asuntos, la cual, surge para solventar la interferencia ocasionada por los asuntos de corte bajo un enfoque asimétrico. La encapsulación y reutilización de los asuntos de corte (requerimientos no funcionales y restricciones de diseño) contribuyen a la modularidad.

Considerando lo ya mencionado, la presente tesis propone el estudio de Java 10 y AspectJ para determinar el potencial que tienen dichas tecnologías en conjunto, de tal manera que las aplicaciones desarrolladas bajo las mismas se encuentren en un esquema modular.

La presente tesis está estructurada de la siguiente forma:

En el capítulo 1 presenta los conceptos básicos que abarcan el dominio del problema, así como el planteamiento del mismo, el objetivo general, los objetivos específicos y la justificación que respalda dicha tesis. En el capítulo 2 presenta los artículos más relevantes en el estado del arte relacionados con modularidad, así como una tabla comparativa de los mismos. En el

capítulo 3 se presenta la aplicación de la metodología de desarrollo de software para las pruebas modulares en combinación con AspectJ. En el capítulo 4 se presenta la implementación de las pruebas modulares con AspectJ en el caso de estudio Reactor nuclear de elementos químicos". En el capítulo 5 se presentan las conclusiones obtenidas a lo largo del desarrollo del proyecto de tesis, así como recomendaciones realizadas al mismo.

Capítulo 1

Antecedentes

En este capítulo se explican los conceptos más importantes relacionados con el trabajo presentado. Se da a conocer la problemática a resolver, los objetivos que se llevarán a cabo con la finalidad de atender dicha problemática, y la justificación que respalda la importancia de la realización de dicho trabajo.

1.1. Marco teórico

En esta sección se presentan los conceptos relacionados con el tema de investigación.

1.1.1. Modularización

Booch [3] define la modularización como la división de un programa en módulos que se compilan de forma separada, pero que cuentan con conexiones con otros módulos.

1.1.2. Módulos

De acuerdo con [3] los módulos sirven como contenedores físicos, en los cuales se declaran las clases y objetos del diseño lógico.

1.1.3. Principios de extensibilidad en el desarrollo de software

En esta sección se abordan los principios de extensibilidad que permiten un desarrollo modular adecuado en la ingeniería de software.

Diseño simple: Una arquitectura simple siempre será más fácil que se adapte a nuevos cambios que una compleja [4].

Descentralización: Mientras más autónomos sean los módulos, mayor es la probabilidad de que un simple cambio afecte a solo un módulo, o a un pequeño número de módulos, evitando una reacción en cadena de cambios que afecte a todo el sistema [4].

1.1.4. Interfaz de módulo

Permite a los desarrolladores entender las funciones que provee un módulo sin tener que entender su implementación [4].

1.1.5. Criterios de un sistema modular

Los criterios de un sistema modular según Taylor [4] son:

Descomposición: La capacidad de descomponer un software en un pequeño número de subproblemas menos complejos, conectados por una estructura simple y suficientemente independiente para trabajar con ellos de forma separada.

Composición: La capacidad de producir elementos de software que se combinen libremente para generar nuevos sistemas, posiblemente en nuevos entornos distintos a los iniciales.

Entendimiento: La capacidad de producir software del cual el humano entienda cada módulo sin tener que conocer a los demás, o en el peor escenario, teniendo que examinar pocos módulos.

Continuidad: Una arquitectura de software es continua, si al aplicar un cambio en las especificaciones, cambia un solo módulo, o un pequeño número de módulos.

Protección: Un método satisface la protección modular si produce arquitecturas en las cuales el efecto de una condición anormal a tiempo de ejecución en un módulo permanezca confinada a ese módulo, o en el peor escenario, que se propague a pocos módulos vecinos.

1.1.6. Reglas para conservar la modularidad

Las reglas para conservar la modularidad establecidas por Taylor [4] son:

Mapeo directo: La estructura modular del sistema de software debe permanecer compatible con cualquier otra que surja en el proceso del modelo del dominio del problema.

Pocas interfaces: Cada módulo debe comunicarse con la menor cantidad de módulos posible.

Pequeñas interfaces: Si dos módulos se comunican, deben intercambiar la menor cantidad de información posible.

Interfaces explícitas: Si un módulo A y B se comunican, se debe entender claramente en el texto del módulo A, B o ambos.

Información oculta: El diseñador de cada módulo debe seleccionar un subconjunto de las propiedades del módulo como la información oficial acerca del mismo, para hacerla disponible a los autores de los módulos cliente.

1.1.7. Abstracción

La abstracción se define como la habilidad de distinguir el propósito de cualquier pieza de software, desde los numerosos detalles, de su implementación [5].

1.1.8. Programación Orientada a Aspectos (POA)

Es una técnica de programación que permite la encapsulación y reutilización de requerimientos no funcionales [6].

1.1.9. Punto de unión(*Join Point*)

Un punto de unión es un punto identificable en la ejecución de un programa [6].

1.1.10. Corte en puntos

Un corte en puntos es un constructor de programa que selecciona puntos de unión y recolecta el contexto de dichos puntos [6].

1.1.11. Aviso

Un aviso es el código que se va a ejecutar en un punto de unión seleccionado por un corte en puntos [6].

1.1.12. Enfoque simétrico en la POA

El enfoque simétrico permite el uso de aspectos aplicados en aplicaciones nuevas no legadas. AspectJ está diseñado para un enfoque asimétrico. [7]

1.1.13. Beneficios de los aspectos aplicados en el desarrollo de software de forma asimétrica

La aplicación de aspectos en sistemas legados contribuye modularmente en [7]:

- Mantenimiento de software
- Cambio de requerimientos
- Nuevos requerimientos
- Evolución de software

1.1.14. Aspecto

Unidad de modularización que permite la abstracción y encapsulación de asuntos de corte, los cuales son partes del software que pertenecen lógicamente a un módulo y afectan a todo el sistema. [8].

1.1.15. Java

Es una plataforma formada por un conjunto de bibliotecas de clases, la Máquina Virtual de Java, un cargador de clase, un compilador, un depurador y otras herramientas [9].

1.1.16. Arquitectura

Una arquitectura es el conjunto de las decisiones de diseño realizadas acerca del sistema [10].

1.1.17. Conector

Es un elemento arquitectónico que permite realizar interacciones efectivas y reguladas entre los componentes [10].

1.1.18. Estilo arquitectónico

Es una colección de decisiones de diseño arquitectónicas que son aplicables a un contexto de desarrollo determinado, restringe las decisiones de diseño arquitectónicas que son específicas en un sistema particular con ese contexto, y provee de beneficios a ese sistema resultante [10].

1.1.19. Patron arquitectónico

Es una colección de decisiones de diseño arquitectónicas que son aplicables a un problema de diseño recurrente, parametrizadas acorde a diferentes contextos de desarrollo de software en los cuales dicho problema aparece [10].

1.1.20. Componente

Es una entidad de arquitectura que encapsula un subconjunto de la funcionalidad del sistema o datos, restringe el acceso a ese subconjunto con una interfaz definida explícitamente y tiene dependencias definidas explícitamente sobre su contexto de ejecución requerido [10]. Asimismo, es un elemento de software que se utiliza por muchas aplicaciones diferentes [5].

1.1.21. *Java Runtime Environment* (JRE)

Es un entorno de desarrollo a tiempo de ejecución para desplegar a Java en entornos de servidor y cliente [11].

1.1.22. Inversión de control (IoC, *Inversion of Control*)

Es un patrón de diseño que permite delegar la implementación de una problemática a un módulo de ensamblado aislado (comúnmente *frameworks*), permitiendo que el usuario siga una convención que le permita al módulo inyectar la implementación [12].

1.1.23. Inyección de dependencias (DI, *Dependency Injection*)

Es un patrón de diseño que delega la resolución de dependencias a un inyector de dependencias dedicado que conoce qué objetos inyectar en el código de la aplicación [13].

1.2. Planteamiento del problema

Es importante destacar que bajo el enfoque orientado a objetos, la identidad de un objeto no es confiable debido a que únicamente se guarda el estado del mismo y su referencia se pierde, las interfaces carecen de un mecanismo de versiones y una implementación no garantiza la existencia de clases exclusivas. Aunado a ello las referencias a terceros dificultan la reutilización de clases, y en estos casos, patrones como la DI [13] y la IoC [14] ayudan incluso a manejar de forma aceptable, pero limitada, las cosas en tiempo de ejecución. Aunque Java ofrece mecanismos modulares como los paquetes, estos no son suficientes para un adecuado soporte modular. Teniendo en cuenta esta problemática se plantea estudiar el potencial que brinda el sistema de módulos de Java 10 en conjunto con AspectJ, realizando pruebas que permitan el estudio de diferentes puntos de modularidad, de manera que las aplicaciones desarrolladas bajo dicho esquema sean modulares.

1.3. Objetivo general y objetivos específicos

A continuación se presenta el objetivo general y los objetivos específicos.

1.3.1. Objetivo general

Aplicar las capacidades modulares de Java 10 para la implementación y prueba de software utilizando AspectJ como *framework* complementario en aplicaciones de software y su contraparte en el sistema de módulos del JDK y JRE.

1.3.2. Objetivos específicos

1. Revisar las capacidades modulares disponibles en el lenguaje Java 10 para la modularidad en código fuente, entorno de ejecución y evolución de software.

2. Aplicar el lenguaje AspectJ al enfoque modular de Java 10 para la identificación de propiedades modulares adicionales en la implementación de software.

3. Generar una guía de modularidad que permita especificar el uso modular del entorno de ejecución de forma complementaria a las aplicaciones de software.

4. Identificar y desarrollar un caso de estudio que permita mostrar las ventajas del sistema modular de Java 10.

1.4. Justificación

Actualmente, el disponer de un soporte adecuado de modularidad sin las limitaciones mencionadas, ayuda no solo en el código fuente, sino además apoya actividades de administración del entorno a tiempo de ejecución. En ese sentido, la POA [8] ofrece soluciones parciales especialmente en evitar la dispersión de código; no obstante, tampoco se ofrecen soluciones completas debido a la dependencia que existe entre objetos y aspectos.

El lenguaje Java en su versión 9 [15] ofrece un enfoque modular que promete solventar varias de las limitaciones mencionadas [16]. Cabe mencionar que Java migró rápidamente a las versiones 10 y 11 por motivos de soporte pero el sistema de módulos se mantiene transparente en las tres versiones. Esto hace necesario estudiar dicho enfoque incluso en combinación con aspectos para conocer el potencial de aplicaciones modulares disponibles hasta Java 10.

La modularidad es uno de los grandes retos de la Ingeniería de Software. Java 10 brinda capacidades modulares a nivel de código y máquina virtual que proveen un enfoque más cercano al escenario modular ideal en la Ingeniería de Software. Actualmente se reportan pocos trabajos relacionados con Java 10 que estudien a profundidad las capacidades modulares ya mencionadas. Más aún, no hay mucho soporte de AspectJ con Java 10 dada su novedad, por lo que se cuenta con mucho potencial para el desarrollo de aplicaciones modulares robustas que exploten en gran medida la modularidad con el uso de aspectos.

Capítulo 2

Estado de la práctica

En este capítulo se dan a conocer trabajos de diversos campos de investigación que abordan distintos conceptos de modularidad que coadyuvan al entendimiento y desarrollo de la tesis.

2.1. Trabajos relacionados

En [17] se definieron los objetivos de crear DSAL (*Domain-Specific Aspect Language*, Lenguaje de Aspectos de Dominio Específico) de primera clase, es decir, la capacidad de las herramientas de lenguaje para crear DSAL y crear nuevos FCDSAL (*First-Class Domain Specific Aspect Language*, Lenguaje de Aspectos de Primera Clase de Dominio Específico). Se menciona que, en la práctica, la LOM (*Language Oriented Modularity*, Modularidad Orientada al Lenguaje) no es efectiva en costos como la LOP (*Language Oriented Programming*, Programación Orientada al Lenguaje) debido a su falta (o incompatibilidad) de soporte para herramientas de desarrollo. Los DSAL ayudan a resolver los asuntos de corte, promueven la modularidad y corrigen los problemas que los asuntos de corte ocasionan. No se usa la LOM debido a su incapacidad de expresar una transformación de preservación de semánticas de DSAL a DGAL (*Domain-General Aspect Language*, Lenguaje de Aspectos de Dominio General). Se contribuyó con un enfoque basado en AspectJ en el cual los DSAL se implementan como lenguajes específicos gracias a la transformación a DGAL, sin la necesidad de cambiar el compilador cada vez que se presenta un nuevo DSAL. Esto provee una alternativa a los marcos de trabajo de composición de aspectos que requieren que el compilador de escritura codifique una parte de la implementación del DSAL. Con este enfoque, LOM se vuelve práctico para el proceso de desarrollo de software del mundo real, habilitando la creación a demanda y el uso de DSAL para manejar asuntos de corte en puntos, evitando la dispersión de código que prevalece en los proyectos de software moderno.

Erdweg y Ostermann [18] establecieron como objetivo demostrar cómo las características de un modelo se integran en un lenguaje de programación sin romper la modularidad. Los marcos de trabajo dirigidos por modelos existentes no están diseñados de una forma que soporte la modularidad, debido a que muchas dependencias de módulo quedan implícitas. En particular, es difícil razonar las dependencias considerando los artefactos generados. Se planteó que los modelos, metamodelos y transformaciones están acompañados con *scripts* de construcción o archivos de configuración que controlan la aplicación de las transformaciones de modelo, definiendo *pipelines* de transformación y seleccionando los modelos de entrada adecuados. Más aún, existe una gran cantidad de código en el lenguaje base ajeno al código generado por modelos, el cual actúa como código invasivo para el código generado desde los modelos y contiene una funcionalidad algorítmica regular, así como código para dominios que no cuentan con un soporte de modelado dedicado. Con la finalidad de atender dichos problemas, se propuso un sistema de módulos basado en SugarJ para el desarrollo dirigido por modelos que integre modelos, código convencional, metamodelos y transformaciones como módulos en un marco de trabajo para manejar dependencias. El sistema de módulos garantiza la ausencia de dependencias ocultas. El caso de estudio demostró que las propiedades del sistema de módulos no impiden la aplicabilidad en escenarios dirigidos por modelos.

En [19] se propuso como objetivo modularizar los análisis estáticos de programa con la finalidad de obtener reutilización. Se mencionó que un análisis modularizado y bien diseñado siempre se ejecuta desde cero en cada ejecución. Estos pasos conllevan una porción considerable del tiempo de ejecución de un análisis, considerando la investigación de bases de código grandes como la Biblioteca de Clases de Java. Esto resulta problemático durante el desarrollo de análisis estáticos cuando se vuelven a ejecutar los pasos básicos para cada ciclo implementar-probar-depurar, lo cual retrasa el desarrollo. Para lograr dicho objetivo se propuso el desarrollo de un marco de trabajo llamado SootKeeper, el cual utiliza modularidad de OSGi (*Open Services Gateway initiative*, Iniciativa de Puerta de Enlace de Servicios Abiertos), el cual es un consorcio dedicado a la modularidad, con la finalidad de separar análisis en pequeños compartimientos, los cuales se ejecutan de forma individual y en paralelo. Se tomó ventaja de una característica clave de la modularidad de OSGi, la cual hace alusión a mantener activos los módulos en memoria una vez que la ejecución haya finalizado. Esto permite acelerar el ciclo de depuración, ejecutando únicamente los análisis de módulos modificados utilizando los resultados de los análisis, los cuales se mantienen en memoria mientras el marco de trabajo siga activo o hasta que

hayan sido invalidados. Los análisis modularizados ayudan a facilitar un proceso de desarrollo más eficiente y ayudan a racionalizar un *pipeline* de evaluación para esos análisis.

Rentschler et al. [20] definieron el objetivo de controlar las dependencias entre módulos, haciendo hincapié en que estas deben ser explícitas. Mucho del esfuerzo en entender las transformaciones de modelos se produce por la complejidad inducida por el alto nivel de dependencias de control y datos. La complejidad se vincula al tamaño, complejidad estructural y heterogeneidad de los modelos involucrados en una transformación. Actualmente no se reporta en la literatura un concepto para los lenguajes de transformación de modelos que permita a los programadores controlar el ocultamiento de la información y la declaración estricta de dependencias de código y modelo en las interfaces de módulo. Se propuso un sistema de módulos basado en cQVTom (*Core QVT-Operational-Modular*, Núcleo Operacional Modular QVT) que incluya no solo el control de dependencias como parte de sus contratos de interfaz, sino también las dependencias de datos a nivel de clase de los modelos involucrados. Se introdujo un nuevo concepto modular que está enfocado en transformaciones de modelo. El concepto logra que las dependencias de control y datos entre módulos sean explícitas y provee de descripciones de interfaz capaces de ocultar detalles de implementación a los usuarios de los módulos.

En [21] se marcó como objetivo separar la definición de semánticas del lenguaje de la abstracción de la máquina, con la finalidad de obtener una modularidad alta. El enfoque de AAM (*Abstracting Abstract Machine*, Abstracción de Máquina Abstracta) para análisis estático permite a los lenguajes y análisis ser implementados de una forma natural y semántica, tomando un intérprete concreto y sistemáticamente abstrayéndolo en un intérprete abstracto. Muchas formalizaciones de AAM y sus derivados mezclan la semántica del lenguaje con la abstracción de la máquina, conllevando a una pobre modularidad. Se presentó SCALA-AM, el cual es un marco de trabajo para implementar análisis estáticos como máquinas abstractas sistemáticamente abstraídas. Los análisis implementados en SCALA-AM separan la semántica operacional de los asuntos de abstracción de la máquina. Esta modularidad facilita variar el lenguaje analizado y el método de abstracción aplicado en un análisis. SCALA-AM se usa como una base para experimentar y construir con análisis estáticos sin tener que implementar todo desde cero. El marco de trabajo actualmente provee soporte para un gran subconjunto de esquemas, así como de múltiples abstracciones de máquina, también soporta lenguajes de modelado con memoria concurrente compartida.

Schöttle et al. [22] establecieron como objetivo desarrollar un concepto de modularidad que permita separar y empaquetar asuntos en una forma reutilizable, y que permita el uso de mecanismos de composición avanzada para introspección y composición de módulos. MDE (*Model-Driven Engineering*, Ingeniería Dirigida por Modelos) es un marco de trabajo conceptual unificado, en el cual, el desarrollo de software es un proceso de producción de modelos, refinamiento e integración. MDE logra reducir la complejidad accidental y el esfuerzo requerido para transformar el dominio del problema a una solución basada en software. Sin embargo, MDE por sí solo no es capaz de atender la complejidad del desarrollo de software moderno. CORE (*Concern-Oriented Reuse*, Reutilización Orientada a Asuntos) es un nuevo paradigma de desarrollo de software inspirado por las ideas de separación multidimensional de asuntos, las cuales permiten la correcta separación de conceptos de importancia dentro de un dominio. CORE construye sobre las disciplinas de MDE, líneas de productos de software, modelado de objetivos, y técnicas de modularización avanzadas ofrecidas por la orientación a aspectos para definir módulos flexibles de software que permiten la reutilización de software basada en modelos de gran escala. Los modelos de realización con un asunto tienen una visibilidad completa entre ellos, pero las dependencias deben ser declaradas explícitamente. Se utilizó el metamodelo CORE en prácticas para integrar sus capacidades de modularización en dos notaciones de modelado, llamadas Notación de Requerimientos del Usuario y Modelos de Aspectos Reutilizables.

En [23] se definió como objetivo demostrar cómo las funciones de orden superior y la evaluación perezosa contribuyen a la modularidad. La modularidad es un concepto impreciso y confuso. Los lenguajes que tienen como objetivo mejorar la productividad deben dar soporte a la programación modular. La habilidad para descomponer un sistema en partes depende directamente de la capacidad para juntar las soluciones, para dar soporte a la programación modular, un lenguaje debe brindar las herramientas para juntar dichas soluciones. Se demostraron las ventajas de la programación funcional perezosa mediante el desarrollo de varios ejemplos en el lenguaje Miranda (el cual tiene influencia en Haskell): el método Newton-Rhapson para raíces cuadradas, la diferenciación e integración numérica, y la búsqueda alfa-beta de máximos y mínimos. La hipótesis formulada menciona que la comunidad de desarrollo de software en general sobrestima los beneficios de la programación funcional. Sin embargo, se encontró evidencia empírica que demuestra problemas de modularidad en el GHC (*Glasgow Haskell Compiler*, Compilador de Haskell Glasgow), el cual es uno de los proyectos de software más emblemáticos de programación funcional. Se concluyó que existe incertidumbre acerca de los beneficios de la programación funcional en el ámbito modular debido a que no existe suficiente evidencia.

Chiba [24] estableció como objetivo demostrar el alcance de los mecanismos destructivos y no destructivos en donde las extensiones son efectivas y visibles en un programa. La herencia es un mecanismo clásico para extender un módulo existente. Dado que conserva el módulo original, los programadores usan tanto el módulo original como el módulo extendido en el mismo programa, por lo que la herencia se denomina como un mecanismo no destructivo. Por otro lado, existen mecanismos de extensión que modifican directamente un módulo existente, por consiguiente, únicamente el módulo extendido estará disponible en el programa. Estos mecanismos, como los aspectos en AspectJ y los elementos propios del lenguaje GluonJ de Chiba llamados revisores, se catalogan como mecanismos destructivos. Se presentó un tercer enfoque el cual se sitúa en medio de los dos extremos (destructivo y no destructivo). El tercer enfoque permite a los programadores controlar el alcance de las extensiones en un estilo modular. También se presentaron algunos mecanismos de lenguaje (*method shells*) con base en el tercer enfoque y algunos problemas que prevalecen en los contextos de la Programación Orientada a Características. Para lograr una extensión destructiva reutilizable, los usuarios de módulos deben especificar dónde la extensión es efectiva. Un ejemplo basado en el tercer enfoque utiliza el patrón de diseño *Abstract Factory* emulando a la POA. Sin embargo, el método *main* debe ser modificado manualmente para el intercambio de clases (o utilizar un aspecto cayendo en la destructividad).

Cazzola y Shaqiri [25] establecieron como objetivo demostrar cómo es posible expresar la optimización a nivel de módulo dado que la optimización requiere contexto (a veces global), el cual, es información que normalmente no se encuentra disponible cuando se desarrolla. La complejidad de la implementación de un lenguaje de programación se resuelve con modularización, la cual, favorece la separación de asuntos, desarrollo independiente, mantenibilidad y reutilización. Sin embargo, la modularidad interfiere con la optimización del lenguaje debido a sus requerimientos contextuales que rebasan los límites del módulo a optimizar e involucra a terceros. Se propuso como solución un modelo implementado en el marco de trabajo Neverlang para el desarrollo de lenguaje modular con un intermediario de acciones semánticas múltiples basado en las condiciones evaluadas a tiempo de ejecución. La solución tiene como contexto los interpretadores de lenguajes de programación basados en sintaxis de árboles que son construidos con un marco de trabajo para el desarrollo del lenguaje modular. Los beneficios del modelo se demostraron al optimizar Nevelang.JS, una implementación de Neverlang del lenguaje de programación JavaScript. La optimización desarrollada está definida a nivel de módulo y mo-

dernizada en los componentes existentes.

Ubayashi y Kamei [26] propusieron como objetivo lograr una nueva visión de modularidad para integrar el modelado de diseño con la programación. Los módulos de programa son artefactos que no se consideran como módulos de software sino como documentos complementarios. Esto ocasiona que sea difícil mantener la trazabilidad entre un programa y sus artefactos, debido a que el desarrollador debe revisar la consistencia manualmente. Especialmente, esto causa un problema en las fases de diseño e implementación, debido a que el modelo de diseño y el programa representan e implementan la arquitectura de software. Se planteó un enfoque que trata al modelo de diseño como un módulo de software de primera clase. Un modelo de diseño como un diagrama UML se considera como un módulo de diseño. Para realizar el diseño de módulos, se presentó Archface, la cual es una interfaz arquitectónica. El compilador de Archface está basado en AspectJ, dicho compilador genera código en AspectJ de las descripciones de Archface y código Java. Archface expuso una serie de puntos arquitectónicos que involucran al diseño y al código. La noción del diseño orientado a interfaces y la programación propuesta ayuda al desarrollador a capturar una estructura abstracta de modelo adecuada. Archface es un mecanismo que ayuda a diseñar no solo módulos sino abstracción.

En [27] se definió como objetivo presentar un primer enfoque que resuelva la modularización de modelos de una forma genérica y reutilizable. El modelado se considera como la técnica para afrontar la complejidad de los sistemas. Los conceptos de modularización se introducen en muchos lenguajes de modelado para atender los modelos del mundo real que se vuelven rápidamente artefactos monolíticos, especialmente cuando se deben modelar sistemas complejos. Los modelos heredados carecen de una estructura adecuada dado que los conceptos de modularización no están disponibles. Se presentó una transformación de modularización, la cual, es reutilizable para varios lenguajes de modelado transformando los conceptos concretos en genéricos ofrecidos por el metamodelo de modularización. Dicha transformación es suficiente para reutilizar diferentes estrategias de modularización, la mayoría basadas en métricas de calidad, provistas por las transformaciones de modelo basadas en búsqueda. Se demostró la aplicabilidad del enfoque modular en los modelos Ecore, los cuales son la base del EMF (*Eclipse Modeling Framework*, Marco de Trabajo de Modelado de Eclipse). Se alcanzó el objetivo combinando varios enfoques de transformación: genérico, de consulta, estructurado y transformaciones basadas en búsqueda.

Golra et al. [28] marcaron como objetivo lograr la homogeneidad de los modelos usados

durante la fase de diseño de sistemas de software complejos. La heterogeneidad de los modelos en el espacio de modelado conduce a múltiples problemas en la sincronización de modelos, trazabilidad y manejo de una consistencia global del sistema. Los sistemas de software complejos usualmente necesitan integrar puntos de vista de varios stakeholders y expertos de sistema. Dichos puntos de vista tienden a estar relacionados con múltiples modelos, los cuales se encargan de diferentes asuntos. Añadir los asuntos de corte a la ecuación hace que el modelado de dichos sistemas se vuelva más complicado. Una forma de manejar la complejidad de los sistemas es mejorando la modularidad de los modelos heterogéneos. Se propuso un enfoque que busca crear puentes entre los distintos paradigmas para lograr la sincronización de los mismos, consistencia global, trazabilidad bidireccional y desarrollo de modelos de asuntos cruzados. A pesar de que la perspectiva de modularidad presentada parece ser una antítesis de la separación de asuntos, encargarse de los asuntos cruzados en múltiples modelos asegura una mejor modularización. Sin embargo, el enfoque presenta una limitante en la disponibilidad de los diferentes conectores tecnológicos.

Şutîi et al. [29] establecieron como objetivo emplear la modularidad para mejorar el entendimiento de los modelos y su reutilización (lo cual beneficia a los modelos complejos). El incremento de las funcionalidades en los productos aumenta la complejidad del software. El uso de MDE y lenguajes de dominio específico ayudan en la construcción de software complejo. La complejidad y el tamaño de los modelos hacen difícil la comprensión de los mismos e incrementa la necesidad de usar mecanismos de reutilización. La contribución principal fue establecer un mecanismo de modularidad basado en grupos (similar a estructuras de bloque), fragmento de abstracciones y aplicaciones (similar a las subrutinas). La novedad yace en las semánticas asociadas a los grupos y el uso de un mecanismo de substitución basado en cálculo de lambas no tipificado combinado con los elementos del modelo para obtener fragmentos de abstracciones y fragmentos de aplicaciones. Se desarrolló Metamod, un lenguaje multinivel de metamodelado, en el cual se emplean mecanismos de modularidad. El aspecto de modularidad se construyó alrededor de un núcleo formado de conceptos y relaciones. La simpleza del núcleo permite obtener una mejor visión del enfoque modular. Todos los niveles de modelado en MetaMod están unificados y un elemento de modelo funge con el rol de tipo y valor, creando un ambiente multinivel de metamodelado.

Melicher et al. [30] definieron como objetivo desarrollar un sistema de módulos que permita a los desarrolladores limitar y controlar la autoridad otorgada a cada módulo en un sistema de

software. El principio de la autoridad mínima es una técnica fundamental para diseñar sistemas de software seguros. Dicha técnica establece que cada componente de un sistema debe acceder únicamente a la información y a los recursos necesarios para operar. Sin embargo, los lenguajes de programación actuales no proveen de un control adecuado para la autoridad de módulos no confiables, y los enfoques no lingüísticos no logran un control seguro de autoridad. Se presentó un sistema de módulos que ayuda a los desarrolladores de software a controlar la autoridad de código tratando los módulos como módulos de primera clase, permitiendo el acceso a un recurso y protegiendo el acceso a los módulos relacionados con la seguridad y la privacidad, siguiendo el estilo del lenguaje de programación E. Si un módulo A quiere acceder a un módulo B, A accederá a B únicamente si posee los permisos requeridos. El control de las capacidades permite dar soporte a los módulos de primera clase proveyendo un modelo fuerte para los aspectos de seguridad y aislamiento de módulos. El sistema presentado soporta módulos de primera clase y usa capacidades para proteger el acceso a los módulos de recursos relacionados con seguridad y privacidad. Además, simplifica el razonamiento para determinar la autoridad de un módulo examinando la interfaz del mismo, sus importaciones y las interfaces de los módulos que importa, logrando que la revisión de seguridad sea más práctica.

Ritschel y Erdweg [31] marcan como objetivo resolver el problema de evitar la captura de variables cuando se aplican transformaciones a los programas. Las transformaciones de programa y programación generativa tienen una amplia aplicación en el desarrollo de software. El reto que se presenta para el desarrollador es cerciorarse que el código transformado y el código sin alterar sigan interactuando correctamente. Cuando una transformación mueve, agrega o altera referencias nombradas o declaraciones en un programa, siempre existe la posibilidad de que una declaración modificada capture una referencia no modificada, o que una referencia modificada captura una declaración sin modificar. La captura de variables confunde a los programadores, rompe la intención del código o la transformación, y ocasiona *bugs* que son difíciles de detectar y eliminar. Se desarrolló un algoritmo llamado *name-fix* el cual detecta y elimina la captura de variables modularmente. Se extendió una versión previa de este algoritmo minimizando el renombrado de declaraciones exportadas, propagando los renombramientos necesarios de las declaraciones exportadas a los clientes, y evitando el renombramiento de las declaraciones importadas. Para demostrar la aplicabilidad de dicho algoritmo se utilizaron recomposiciones en *Lightweight* Java. El algoritmo *name-fix* provee una solución modular genérica para eliminar la captura de variables resultantes de las transformaciones de programas. Dicho algoritmo logra eliminar el proceso manual de verificar la consistencia de código proveyendo un proceso total-

mente automatizado.

2.2. Análisis comparativos

El análisis comparativo para esta tesis se presenta en el apartado 2.2.1 con los trabajos mencionados anteriormente y en el apartado 2.2.2 se presenta un análisis entre lenguajes de programación que dan soporte modular respecto a Java.

2.2.1. Análisis comparativo acerca de conceptos de modularidad

La tabla 2.1 muestra la relevancia de los trabajos más importantes en materia de modularidad.

Tabla 2.1: Análisis comparativo de los artículos

ARTÍCULO	OBJETIVO	PROBLEMA	TECNOLOGÍAS	RESULTADO	ESTADO
Hadas y Lorenz [17]	Crear DSAL de primera clase y nuevos FCDSAL.	No se usa la LOM debido a su incapacidad de expresar una transformación de preservación de semánticas de DSAL a DGAL.	AspectJ Java	LOM se vuelve práctico para el proceso de desarrollo de software del mundo real.	Finalizado
Erdweg y Ostermann [18]	Demostrar cómo las características del modelo son integradas en un lenguaje de programación sin romper la modularidad.	Existe una gran cantidad de código escrito a mano en el lenguaje base que actúa como código pegamento para el código generado desde los modelos.	SugarJ (basado en Java)	El sistema de módulos garantiza la ausencia de dependencias ocultas.	Finalizado
Kübler et al. [19]	Lograr modularizar los análisis estáticos de programa con la finalidad de obtener reutilización.	Un análisis modularizado y bien diseñado siempre se ejecuta desde cero en cada ejecución. Estos pasos conllevan una porción considerable del tiempo de ejecución de un análisis.	Modularidad de OSGi	Los análisis modularizados ayudan a facilitar un proceso de desarrollo más eficiente.	Con seguimiento

2.1 Análisis comparativo de los artículos (Cont.)

ARTÍCULO	OBJETIVO	PROBLEMA	TECNOLOGÍAS	RESULTADO	ESTADO
Rentschler et al. [20]	Controlar las dependencias entre módulos, haciendo hincapié en que estas deben ser explícitas.	Mucho del esfuerzo en entender las transformaciones de modelos se produce por la complejidad inducida por el alto nivel de dependencias de datos y control.	cQVTom	Se introdujo un nuevo concepto modular que está enfocado en transformaciones de modelo. El concepto logra que las dependencias de control y datos entre módulos sean explícitas.	Finalizado
Stiévenart et al. [21]	Separar la definición de semánticas del lenguaje de la abstracción de la máquina, con la finalidad de obtener una modularidad alta.	Muchas formalizaciones de AAM y sus derivados mezclan la semántica del lenguaje con la abstracción de la máquina, conllevando a una pobre modularidad.	SCALA-AM	El marco de trabajo actualmente provee soporte para un gran subconjunto de esquemas, así como de múltiples abstracciones de máquina, también soporta lenguajes de modelado con memoria concurrente compartida.	Con seguimiento

2.1 Análisis comparativo de los artículos (Cont.)

ARTÍCULO	OBJETIVO	PROBLEMA	TECNOLOGÍAS	RESULTADO	ESTADO
Schöttle et al. [22]	Desarrollar un concepto de modularidad que permita separar y empaquetar asuntos en una forma reutilizable, y que permita el uso de mecanismos de composición avanzada para introspección y composición de módulos.	MDE por sí solo no es capaz de atender la complejidad del desarrollo de software moderno.	CORE (basado en MDE)	Se realizaron prácticas para integrar las capacidades de modularización de CORE en dos notaciones de modelado, llamadas Notación de Requerimientos del Usuario y Modelos de Aspectos Reutilizables.	Finalizado
Figueroa y Robbes [23]	Demostrar cómo las funciones de orden superior y la evaluación perezosa contribuyen a la modularidad.	La modularidad es un concepto impreciso y confuso. Los lenguajes que tienen como meta mejorar la productividad deben dar soporte a la programación modular.	Miranda	Existe incertidumbre acerca de los beneficios de la programación funcional en el ámbito modular debido a que no se reporta suficiente evidencia.	Finalizado

2.1 Análisis comparativo de los artículos (Cont.)

ARTÍCULO	OBJETIVO	PROBLEMA	TECNOLOGÍAS	RESULTADO	ESTADO
Chiba [24]	Demostrar el alcance de los mecanismos destructivos y no destructivos en donde las extensiones son efectivas y visibles en un programa.	No se reporta una noción precisa que determine cuándo es más conveniente usar mecanismos destructivos o no destructivos.	Java AspectJ GluonJ	Para lograr una extensión destructiva reutilizable, los usuarios de módulos deben especificar dónde la extensión es efectiva.	Finalizado
Cazzola y Shaqiri [25]	Demostrar cómo se logra expresar la optimización a nivel de módulo dado que la optimización requiere contexto.	La modularidad interfiere con la optimización del lenguaje debido a sus requerimientos contextuales que rebasan los límites del módulo a optimizar e involucra a terceros.	Neverlang	Los beneficios del modelo se demostraron al optimizar Nevelang.JS, una implementación de Neverlang del lenguaje de programación JavaScript.	Con seguimiento

2.1 Análisis comparativo de los artículos (Cont.)

ARTÍCULO	OBJETIVO	PROBLEMA	TECNOLOGÍAS	RESULTADO	ESTADO
Ubayashi y Kamei [26]	Lograr una nueva visión de modularidad para integrar el diseño de modelado con la programación.	Los módulos de programa se consideran documentos complementarios. Esto ocasiona que sea difícil mantener la trazabilidad entre un programa y sus artefactos, debido a que el desarrollador debe revisar la consistencia manualmente.	Archface (basado en AspectJ)	La noción del diseño orientado a interfaces y la programación propuesta ayuda al desarrollador a capturar una estructura abstracta de modelo adecuada.	Con seguimiento
Fleck et al. [27]	Presentar un primer enfoque que lidie con la modularización de modelos de una forma genérica y reutilizable.	Los modelos heredados carecen de una estructura adecuada dado que los conceptos de modularización no están disponibles.	No presenta	Se alcanzó el objetivo combinando varios enfoques de transformación: genérico, de consulta, estructurado y transformaciones basadas en búsqueda.	Con seguimiento

2.1 Análisis comparativo de los artículos (Cont.)

ARTÍCULO	OBJETIVO	PROBLEMA	TECNOLOGÍAS	RESULTADO	ESTADO
Golra et al. [28]	Lograr la homogeneidad de los modelos usados durante la fase de diseño de sistemas de software complejos.	La heterogeneidad de los modelos en el espacio de modelado conduce a múltiples problemas en la sincronización de modelos, trazabilidad y manejo de una consistencia global del sistema.	No presenta	Presenta una limitante en la disponibilidad de los diferentes conectores tecnológicos.	Con seguimiento
Şutîiet al. [29]	Emplear la modularidad para mejorar el entendimiento de los modelos y su reutilización.	La complejidad y el tamaño de los modelos hacen difícil la comprensión de los mismos e incrementa la necesidad de usar mecanismos de reutilización.	No presenta	Todos los niveles de modelado en MetaMod están unificados y un elemento de modelo funge con el rol de tipo y valor, creando un ambiente multinivel de metamodelado.	Finalizado

2.1 Análisis comparativo de los artículos (Cont.)

ARTÍCULO	OBJETIVO	PROBLEMA	TECNOLOGÍAS	RESULTADO	ESTADO
Melicher et al. [30]	Desarrollar un sistema de módulos que permita a los desarrolladores limitar y controlar la autoridad otorgada a cada módulo en un sistema de software.	Los lenguajes de programación actuales no proveen de un control adecuado para la autoridad de módulos no confiables, y los enfoques no lingüísticos no logran un control seguro de autoridad.	Sistema de módulos basado en E	El sistema presentado soporta módulos de primera clase y usa capacidades para proteger el acceso a los módulos de recursos relacionados con la seguridad y la privacidad.	Finalizado
Ritschel y Erdweg [31]	Resolver el problema de evitar la captura de variables cuando se aplican transformaciones a los programas.	La captura de variables confunde a los programadores, rompe la intención del código o la transformación, y ocasiona bugs que son difíciles de detectar y eliminar.	Java	El algoritmo *name-fix* eliminar el proceso manual de verificar la consistencia de código proveyendo un proceso totalmente automatizado.	Con seguimiento

Tras analizar los artículos ya mencionados, se concluye que la modularidad existe en diversas áreas de la Ingeniería de Software, entre las cuales, se destaca la modularidad a nivel de diseño y a nivel de código. Los conceptos de modularidad que abordan distintos autores se centran principalmente en la reutilización y el control de las dependencias entre módulos. El sistema de módulos de Java en las versiones 9, 10 y 11 tiene gran potencial para la composición de sistemas con aplicaciones modulares, las cuales, satisfacen los conceptos de modularidad que son muy deseados en la Ingeniería de Software, contemplando la capacidad de añadir funcionalidad a los módulos con el uso de aspectos utilizando la biblioteca de AspectJ, permitiendo así, el desarrollo de aplicaciones bajo un esquema modular.

2.2.2. Análisis comparativo de los lenguajes de programación que dan soporte modular

La tabla 2.2 muestra un análisis entre los lenguajes que dan soporte modular respecto a Java.

Tabla 2.2: Análisis comparativo realizado entre los lenguajes de programación OO compatibles con aspectos

Nombre	Soporte modular
Java	Sí, su versión 9 soporta modularidad a nivel de código mediante descriptores de módulo y de entorno mediante el grafo de dependencias provisto por el JDK.
C++	Sí, a nivel de código mediante agrupación por *namespaces*.
C#	Sí, a nivel de código usando tipos de módulo mediante *Reflection*.

Con el análisis previo se puede concluir que Java es el lenguaje que actualmente posee el enfoque modular más robusto y cuyo potencial modular requiere ser analizado.

2.3. Solución propuesta

Esta sección presenta la elección de la propuesta de solución que ofrece mayores ventajas para dar correcta solución a la problemática que plantea la tesis.

2.3.1. Justificación de la solución seleccionada

La elección de dicha solución se basa en el desempeño y compatibilidad de las tecnologías, así como las cualidades ofrecidas por la metodología de desarrollo de software descritas a continuación:

Lenguaje de programación: Java en su versión 10 ofrece un soporte modular a nivel de código y entorno. El hecho de que Java ofrezca un soporte a nivel de entorno implica un avance en el desarrollo modular a nivel arquitectónico. Java 10 en conjunto con AspectJ brindan un desarrollo modular con una correcta separación de asuntos.

IDE: Eclipse es un IDE que se caracteriza por su modelo de *plug-ins*¸ esto lo convierte en un IDE ligero pero potente. Asimismo, soporta las últimas actualizaciones de AspectJ de manera nativa, por lo que se convierte en el IDE idóneo para realizar un estudio entre Java 10 y AspectJ.

***Plug-in* de POA**: AspectJ es el *plug-in* de vanguardia en la actualidad porque soporta 17 primitivas de corte, así como su alto desempeño. Los avances que se tienen de manera fuerte en la parte de aspectos radican en AspectJ.

Metodología de desarrollo de software: SCRUM es una metodología de desarrollo ágil que permite la documentación únicamente de los artefactos requeridos por los stakeholder. Asimismo, ofrece el desarrollo iterativo e incremental de software con *sprints*, los cuales se acoplan al cronograma que tendrá el proyecto de tesis. De igual manera, se seleccionó XP debido al potencial que brinda para realizar pruebas claras y eficientes. Se optará por una adaptación de las mismas, dado que no existe un equipo de desarrollo.

2.3.2. Metodología para el desarrollo de la tesis

En esta sección se presentan las actividades para el desarrollo de la tesis, las cuales garantizan el desarollo satisfactorio de la tesis en tiempo y forma. A continuación se listan las actividades a realizar:

1. Análisis del estado del arte de los trabajos relacionados con modularidad.
2. Revisión sobre qué se tiene reportado en el OPENJDK respecto al sistema de módulos de Java 10.
3. Revisión sobre la documentación oficial de AspectJ respecto a nuevas características que permitan mejorar la modularidad de Java 10.
4. Revisión sobre las características modulares provistas por Java 10 a nivel de código fuente, entorno de ejecución y evolución de software.
5. Realización de pruebas utilizando los archivos JAR de AspectJ como módulos para la correcta integración con el sistema de módulos de Java.
6. Análisis arquitectónico para realizar una transformación modular en sistemas legados desarrollados con Java 8.
7. Estudio y realización de pruebas de las primitivas de corte que ofrece AspectJ en conjunto con Java 10 con la finalidad de determinar posibles mejoras modulares.
8. Selección de un caso de estudio que permita mostrar las ventajas del sistema modular de Java 10.
9. Escritura de la guía de modularidad que permita especificar el uso modular del entorno de ejecución de forma complementaria a las aplicaciones de software.
10. Realización de las pruebas con el caso de estudio seleccionado.

2.3.3. Justificación de la metodología para el desarrollo de la tesis

La metodología para el desarrollo de la tesis está basada en un plan de trabajo que consiste en una revisión constante de mejoras provistas por el equipo de trabajo de AspectJ así como el equipo de trabajo de Oracle para Java publicadas en sus respectivos foros oficiales, combinadas con pruebas experimentales que permitan determinar el potencial modular disponible de las

versiones estables más recientes. Dicha metodología plantea las actividades más importantes necesarias para lograr un estudio adecuado entre el sistema de módulos de Java y AspectJ.

Capítulo 3

Aplicación de la metodología

En este capítulo se presenta la aplicación de la metodología de la tesis tomando como caso de estudio académico el proyecto nombrado "Tabla de elementos químicos", en el cual se pretende realizar una transformación modular al sistema de módulos de Java y combinar los módulos del caso de estudio con aspectos mejorando la modularidad del mismo.

Como parte del desarrollo de la transformación modular y aplicación de aspectos en el caso de estudio se realizaron pruebas de concepto que permitieran establecer el alcance que ofrece AspectJ con el sistema de módulos de Java. De igual forma, al ser factible la modularidad a nivel de entorno de ejecución se realizaron análisis y pruebas a nivel arquitectónico para determinar un diseño modular adecuado.

3.1. Revisión sobre las características modulares provistas por Java 10 a nivel de código fuente

El sistema de módulos de Java provee de nuevas características que permiten dividir un programa en distintos módulos, en el cual cada módulo contiene únicamente lo que necesita para su correcto funcionamiento.

Para que Java reconozca a un conjunto de clases como módulo, dichas clases se contienen en un paquete que debe estar declarado en el descriptor de módulo "module-info.java". Un módulo tiene la capacidad estar conformado de varios submódulos (paquetes), sin embargo, solo se requiere de un descriptor de módulo.

3.1.1. Elementos del descriptor de módulo module-info.java

El descriptor de módulo es una característica clave en el sistema de módulos de Java. Toda interfaz expuesta, requerida y manejo de servicios debe estar declarada apropiadamente en el archivo *module-info.java*. Entre las propiedades que permite el descriptor de módulo para describir apropiadamente las características del módulo en cuestión son:

1. *Requires*: La palabra reservada *requires* establece las dependencias modulares requeridas por el módulo para garantizar su correcto funcionamiento.

2. *Exports*: La palabra reservada *exports* establece una interfaz expuesta del módulo para que otros módulos pueden consumirla.

3. *Export...to*: Las palabras reservadas *Export...to* permiten la exportación selectiva del paquete de un módulo a un módulo en específico.

4. *Provides...with*: La palabras reservadas *provides...with* establecen la implementación de la interfaz de un servicio por medio de un módulo que se consume por un tercero.

5. *Uses*: La palabra reservada *uses* indica que el módulo hará uso de un módulo de servicio, el cual provee una interfaz.

6. *Open*: Palabra reservada que antecede a la palabra reservada *module* y que permite el uso de mecanismos de reflexión sobre el módulo.

7. *Opens*: La palabra reservada *opens* permite la utilización de mecanismos de reflexión de manera selectiva sobre un paquete en específico del módulo.

3.1.2. Prueba *Hola mundo modular*

A continuación, se muestra una prueba "Hola mundo modular", en la cual se explican las características básicas de los módulos en Java.

La prueba consta de un módulo `org.holamundomod.main` y un módulo `org.holamundomod.helloworld` en la cual el módulo `org.holamundomod.main` consume al módulo `org.holamundomod.helloworld`, dicho ejemplo representa un ejemplo básico de interfaces expuestas y requeridas.

1. Módulo `org.holamundomod.main`: El módulo `org.holamundomod.main` contiene el método main que consume al método `org.holamundomod.helloworld`.

Código 3.1: Módulo `org.holamundomod.main/Main.java`:

```
1 package org.holamundomod.main;
2 import org.holamundomod.helloworld.*;
3 public class Main {
4     public static void main(String ... args) {
5         System.out.println(new HelloWorld().saludar());
6     }
7 }
```

Código 3.2: Módulo `org.holamundomod.main/module-info.java`:

```
1 module org.holamundomod.main {
2     requires org.holamundomod.helloworld;
3 }
```

2. Módulo `org.holamundomod.helloworld`: El módulo `org.holamundomod.helloworld` contiene el método saludar() que será expuesto al módulo `org.holamundomod.main` mediante su descriptor de módulo.

Código 3.3: `Módulo org.holamundomod.helloworld/HelloWorld.java`:

```
1 package org.holamundomod.helloworld;
2 public class HelloWorld {
3     public String saludar(){
4         return "Hola mundo modular";
5     }
6 }
```

Código 3.4: `Módulo org.holamundomod.helloworld/module-info.java`:

```
1 module org.holamundomod.helloworld {
2     exports org.holamundomod.helloworld;
3 }
```

Compilación de Prueba *Hola mundo modular* con el sistema de módulos

Todo módulo expone y/o requiere interfaces de otros módulos, por lo que la compilación en Java utilizando el sistema de módulos considera el manejo de dependencias modulares al momento de la compilación y ejecucción del programa.

Los módulos al ser alojados en paquetes y contar con un descriptor de módulo en la raíz del paquete resultan en una gran cantidad de rutas de archivos que terminan en compilaciones altamente tediosas. En la Figura 3.1.2 se muestra la estructura del directorio de un módulo:

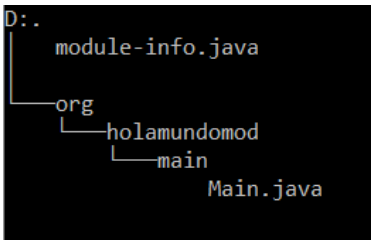

Figura 3.1: Estructura del directorio del módulo org.holamundomod.main

Por lo cual, una solución eficiente que reduce en gran medida el tiempo de compilación es con el manejo de archivos de texto, en los cuales se alojan todas las rutas de archivos de los módulos. Para lo cual se utiliza el siguiente comando de generación de archivos de texto:

Para el módulo `org.holamundomod.main`:

```
dir /B /S org.holamundomod.main\*.java > src1.txt
```

Para el módulo `org.holamundomod.helloworld`:

```
dir /B /S org.holamundomod.helloworld\*.java > src2.txt
```

Si se desea comprobar las rutas que se guardaron en los archivos de texto, se ingresa el siguiente comando:

Comprobación del archivo del módulo -:

```
type src1.txt
```

Comprobación del archivo del módulo -:

```
type src2.txt
```

Una vez que se tengan los archivos de texto listos como se muestra en la Figura 3.2, se procede a la compilación, el sistema de módulos de Java considera dos tipos de compilación con base en el perfil del módulo, existe una compilación para los módulos que no cuentan con dependencias hacia otros módulos y otra compilación que sí considera este elemento clave. Los comandos para dichos escenarios son los siguientes:

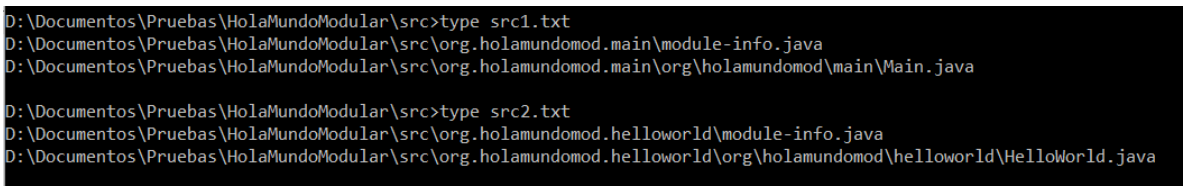

Figura 3.2: Comprobación de archivos de Prueba *Hola mundo modular*

Compilación sin dependencias modulares:

```
javac -d classes/classes.helloworld @src2.txt
```

Compilación con dependencias modulares:

```
javac --module-path classes/classes.helloworld -d classes/classes.main @src1.txt
```

Cuando se termine la compilación de todos los módulos requeridos por el programa, se procede a la ejecucción como se muestra en la Figura 3.1.2, considerando la ruta en la cual los módulos están alojados, con la finalidad de que Java resuelva las dependencias modulares al momento de la ejecucción.

Ejecucción del programa:

```
java --module-path classes -m org.holamundomod.main/org.holamundomod.main.Main
```

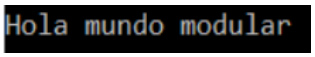

Figura 3.3: Ejecucción de Prueba *Hola mundo modular*

3.2. Revisión sobre las características modulares provistas por Java 10 a nivel de entorno de ejecución

Entre las características más importantes que se introdujeron en el sistema de módulos de Java es el manejo de máquinas virtuales personalizadas mediante la herramienta JLINK. Dichas máquinas virtuales contienen únicamente los módulos necesarios para la ejecucción del sistema.

La máquina virtual de Java en versiones del JDK 8 y anteriores presentaba una estructura

altamente monolítica, lo cual impedía la escalabilidad y modularidad en los sistemas de software, así como un diseño de sistemas basado en componentes. Para atender esta problemática, el equipo de desarrollo de Oracle dividió el JRE en un grafo de dependencias modulares teniendo como módulo raíz *java.base*.

Retomando la prueba *Hola mundo modular*, se generó su respectiva máquina virtual personalizada conteniendo los módulos: `org.holamundomod.main`, `org.holamundomod.helloworld` y, por defecto, `java.base` para su ejecucción. Para su generación, se utilizaron los siguientes comandos:

1. Para la generación de la máquina virtual personalizada:

   ```
   jlink -p classes --add-modules org.holamundomod.main,org.holamundomod.helloworld
    --output JRE
   ```

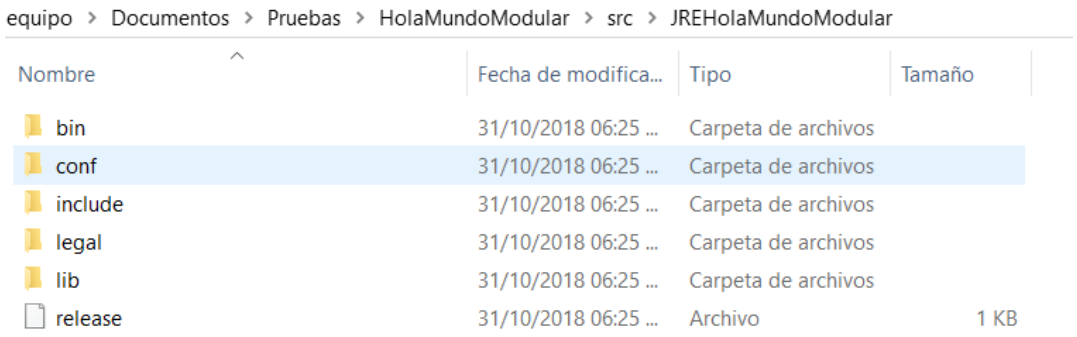

Figura 3.4: Generación de *JRE* personalizado de Prueba *Hola mundo modular*

   Como se aprecia en la Figura 1, la estructura de una máquina virtual personalizada ya no cuenta con el directorio del grafo de dependencias de Java sino únicamente con lo necesario para trabajar.

2. La ejecucción del programa (ubicándose dentro del directorio *bin* de la máquina virtual generada) se presenta a continuación en la Figura 3.2:

   ```
   java -m org.holamundomod.main/org.holamundomod.main.Main
   ```

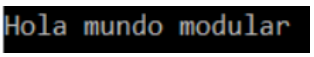

Figura 3.5: Ejecucción del programa utilizando *JRE* personalizado

3.3. Revisión sobre las características modulares provistas por Java 10 a nivel de evolución de software

La evolución del software surge a raíz de la mejora continua de los sistemas, con la finalidad de garantizar sistemas estables, mantenibles y extensibles.

El sistema de módulos se cimentó como una característica robusta de Java que se mantendrá transparente en Java 10, incluso en Java 11. Al permitir el manejo de múltiples módulos los sistemas desarrollados bajo este enfoque serán extensibles y mantenibles con mayor facilidad, aislando las responsabilidades en módulos. Un sistema basado en módulos es menos propenso a ser obsoleto que un sistema robusto.

3.4. Realización de pruebas utilizando los archivos JAR de AspectJ como módulos para la correcta integración con el sistema de módulos de Java

AspectJ en respuesta al sistema de módulos de Java, permitió la conversión de sus bibliotecas JAR en módulos que el sistema de módulos integre y reconozca al momento de la compilación de módulos. Los pasos a seguir para la correcta configuración, integración y compilación de AspectJ son los siguientes (se ejemplifican los pasos en las Figuras 3.6, 3.7 y 3.8):

1. Integración de la biblioteca de *AspectJ Runtime* como módulo:
   ```
   java --module-path <pathto>/lib/aspectjrt.jar --list-module
   ```

2. Integración de la biblioteca del entrelazador de AspectJ como módulo:
   ```
   java --module-path <pathto>/lib/aspectjweaver.jar
   --describe-module org.aspectj.weaver
   ```

```
jdk.scripting.nashorn@10.0.1
jdk.scripting.nashorn.shell@10.0.1
jdk.sctp@10.0.1
jdk.security.auth@10.0.1
jdk.security.jgss@10.0.1
jdk.snmp@10.0.1
jdk.unsupported@10.0.1
jdk.xml.bind@10.0.1
jdk.xml.dom@10.0.1
jdk.xml.ws@10.0.1
jdk.zipfs@10.0.1
oracle.desktop@10.0.1
oracle.net@10.0.1
org.aspectj.runtime file:///C:/aspectj1.9/lib/aspectjrt.jar automatic
```

Figura 3.6: Integración de la biblioteca de *AspectJ Runtime*

```
org.aspectj.weaver file:///C:/aspectj1.9/lib/aspectjweaver.jar automatic
requires java.base mandated
contains aj.org.objectweb.asm
contains aj.org.objectweb.asm.signature
contains org.aspectj.apache.bcel
contains org.aspectj.apache.bcel.classfile
contains org.aspectj.apache.bcel.classfile.annotation
contains org.aspectj.apache.bcel.generic
contains org.aspectj.apache.bcel.util
contains org.aspectj.asm
contains org.aspectj.asm.internal
contains org.aspectj.bridge
contains org.aspectj.bridge.context
contains org.aspectj.internal.lang.annotation
contains org.aspectj.internal.lang.reflect
```

Figura 3.7: Integración de la biblioteca del entrelazador de AspectJ

3. Integración de la biblioteca de herramientas de AspectJ como módulo:

```
org.aspectj.tools file:///C:/aspectj1.9/lib/aspectjtools.jar automatic
requires java.base mandated
contains aj.org.objectweb.asm
contains aj.org.objectweb.asm.signature
contains org.aspectj.ajde
contains org.aspectj.ajde.core
contains org.aspectj.ajde.core.internal
contains org.aspectj.ajde.internal
contains org.aspectj.ajde.ui
contains org.aspectj.ajde.ui.internal
contains org.aspectj.ajde.ui.javaoptions
```

Figura 3.8: Integración de la biblioteca de herramientas de AspectJ

```
java --module-path <pathto>/lib/aspectjtools.jar
--describe-module org.aspectj.tools
```

4. Incorporación en el descriptor de módulo requerido la dependencia modular a la biblio-

teca de *AspectJ Runtime*:

```
requires org.aspectj.runtime;
```

5. Utilización el compilador de AspectJ para compilar los módulos que dependan de AspectJ utilizando la biblioteca de *AspectJ Runtime*:

```
ajc -1.9 --module-path "<pathto>/lib/lib/aspectjrt.jar;classes"
@org.holamundomodaj.main.txt
@main.aspects.txt -d classes/main.classes
```

6. Ejecucción del programa indicando la ubicación de la biblioteca de *AspectJ Runtime*:

```
java --module-path "<pathto>/lib/aspectjrt.jar;classes" -m
org.holamundomodaj.main/org.holamundomodaj.main.Main
```

```
Desde AspectJ
Hola mundo modular
```

Figura 3.9: Ejecucción del programa utilizando aspectos y módulos

3.4.1. Prueba *Hola mundo modular* con AspectJ

1. Módulo `org.holamundomodaj.main`: El módulo `org.holamundomodaj.main` contiene el método main que consume al método `org.holamundomodaj.helloworld`. A su vez, el módulo
`org.holamundomodaj.main` contiene un aspecto llamado `AspectTest`, el cual modifica la salida del método *main* (ejemplificado en los códigos 3.5, 3.6 y 3.7).

Código 3.5: Módulo org.holamundomodaj.main/Main.java

```
1 package org.holamundomodaj.main;
2 import org.holamundomodaj.helloworld.HelloWorld;
3 public class Main {
4     public static void main(String ... args) {
5         System.out.println(new HelloWorld().saludar());
6     }
7 }
```

En el Código 3.5 se presenta la estructura de la clase `Main`, la cual contiene el método *main* en la línea 4.

Código 3.6: Módulo org.holamundomodaj.main/AspectTest.aj

```
1 package org.holamundomodaj.main;
2 public aspect AspectTest {
3   before(): execution(void main(..)) {
4     System.out.println("Desde AspectJ");
5   }
6 }
```

En el Código 3.6 se presenta la estructura de la clase `AspectTest`, en la cual se presenta un corte del método *main* en la línea 3. Dicho corte inserta antes de la ejecución del método *main* el mensaje `Desde AspectJ`.

Código 3.7: Módulo org.holamundomodaj.main/module-info.java

```
1 module org.holamundomodaj.main {
2     requires org.holamundomodaj.helloworld;
3     requires org.aspectj.runtime;
4 }
```

En el Código 3.7 se presenta la declaración del descriptor del módulo `org.holamundomodaj.main`. Como dato importante a resaltar es la utilización de la biblioteca de *AspectJ Runtime* en la línea 3.

2. Módulo `org.holamundomodaj.helloworld`: El módulo `org.holamundomodaj.helloworld` contiene el método saludar() que será expuesto al módulo `org.holamundomodaj.main` mediante su descriptor de módulo.

Código 3.8: Módulo org.holamundomodaj.helloworld/HelloWorld.java

```
1 package org.holamundomodaj.helloworld;
2 public class HelloWorld {
3     public String saludar(){
4         return "Hola mundo modular";
5     }
6 }
```

En el Código 3.8 se presenta la estructura de la clase `HelloWorld`, dicha clase contiene un método *saludar* ubicado en la línea 3.

Código 3.9: Módulo org.holamundomodaj.helloworld/module-info.java

```
1 module org.holamundomodaj.helloworld {
2     exports org.holamundomodaj.helloworld;
3 }
```

En el Código 3.9 se presenta la estructura del descriptor de módulo de `org.holamundomodaj.helloworld`, dicha estructura contempla la exportación del paquete del módulo.

3.5. Generación de una máquina virtual personalizada incorporando la utilización de aspectos

La herramienta JLINK de Java tiene la capacidad de alojar módulos tanto de su grafo de dependencias como módulos de usuarios en la máquina virtual generada. Si bien esto permite grandes capacidades modulares debido a su enfoque de componentes, al momento de la experimentación con un *framework* como AspectJ presentó ciertas limitaciones, entre ellas:

1. JLINK no tiene la capacidad de tratar de manera natural con los *framework*, debido a que estos se consideran módulos automáticos de Java (módulos sin descriptor de módulo, es decir, *JARs* comúnes) y estan vinculados al *CLASSPATH*.

   Uno de los objetivos principales que busca Java mediante su herramienta JLINK es el aumentar la escalabilidad y portabilidad de sus sistemas, por lo que las rutas absolutas de las cuales dependen los módulos automáticos no están permitidas debido a que no pueden asegurar que la configuración prevalezca en todos los sistemas (errores de tipo *NotDefinedClass*, entre otros).

2. Si se desea trabajar con módulos automáticos como los *frameworks*, estos deben ser transformados manualmente a módulos nombrados (módulos con descriptor de módulo), de manera que JLINK pueda alojarlos en la máquina virtual personalizada.

3. La transformación de módulos automáticos implica desempaquetar (en caso de tratarse de un *JAR*), generar el descriptor de módulo, recompilar y si es necesario, reempaquetar, por lo que se requieren de los archivos fuente (pueden no estar disponibles para el desarrollador) para dicho proceso.

En la Figura 3.10 se muestra el tratamiento que brinda el sistema de módulos de Java a la biblioteca de *AspectJ Runtime*:

```
jdk.scripting.nashorn@10.0.1
jdk.scripting.nashorn.shell@10.0.1
jdk.sctp@10.0.1
jdk.security.auth@10.0.1
jdk.security.jgss@10.0.1
jdk.snmp@10.0.1
jdk.unsupported@10.0.1
jdk.xml.bind@10.0.1
jdk.xml.dom@10.0.1
jdk.xml.ws@10.0.1
jdk.zipfs@10.0.1
oracle.desktop@10.0.1
oracle.net@10.0.1
org.aspectj.runtime file:///C:/aspectj1.9/lib/aspectjrt.jar automatic
```

Figura 3.10: Comprobación del módulo automático de AspectJ

3.5.1. Transformación de módulos automáticos a módulos nombrados

Para que un módulo sea considerado nombrado requiere de un descriptor de módulo. Java provee de una herramienta llamada JDEPS, la cual funge como un analizador de código estático capaz de generar un prototipo de descriptor de módulo lo más aproximado posible a la realidad. Para este caso se debe generar el descriptor de módulo del módulo automático de AspectJ, lo cual se presenta en la Figura 3.11:

```
D:\Documentos\Pruebas\HolaMundoModularAspectos\src>jdeps --generate-module-info test C:/aspectj1.9/lib/aspectjrt.jar
writing to test\org.aspectj.runtime\module-info.java

D:\Documentos\Pruebas\HolaMundoModularAspectos\src>
```

Figura 3.11: Generación del descriptor de módulo de la biblioteca de *AspectJ Runtime*

La estructura del descriptor generado por el comando de la Figura 3.11 se muestra en la Figura 3.12:

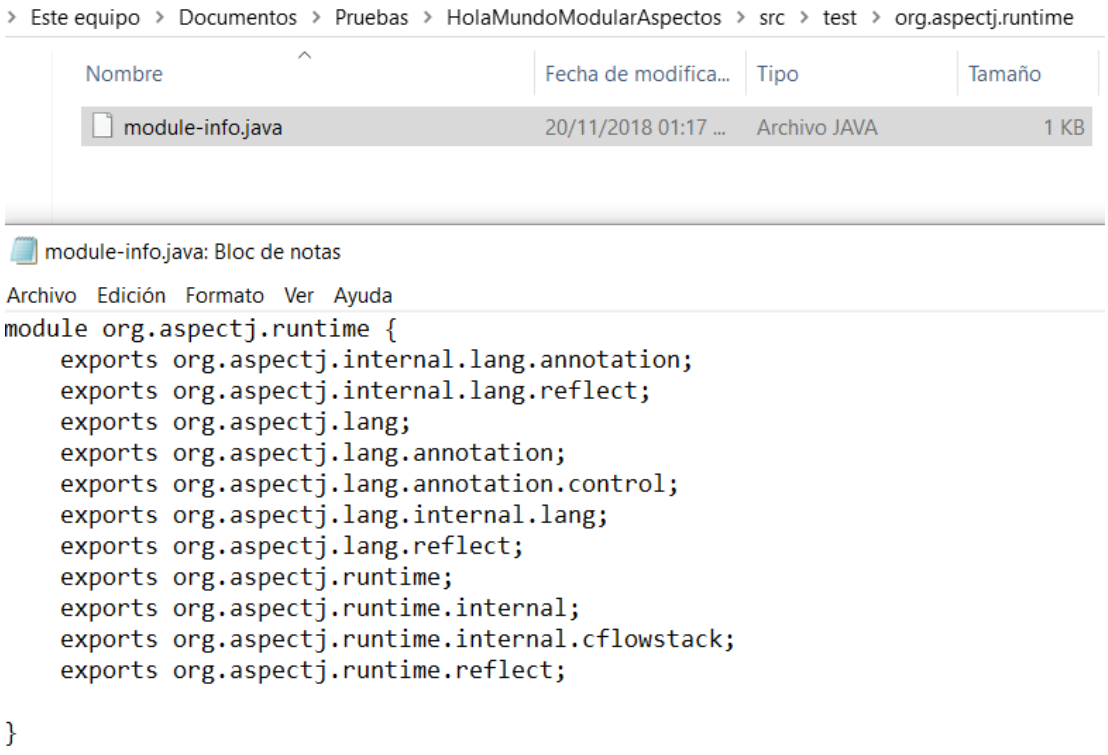

Figura 3.12: Estructura del descriptor de módulo de la biblioteca de *AspectJ Runtime*

Una vez generado el descriptor de módulo utilizando el analizador de código estático JDEPS, es posible realizar un compilado modular normal de la biblioteca de *AspectJ Runtime* para su conversión a módulo nombrado.

Una vez completados estos pasos, la generación de la máquina virtual es idéntica a la mencionada en la Sección 3.2, la cual se muestra en la Figura 3.13:

```
D:\Documentos\Pruebas\HolaMundoModularAspectos\src>jlink --module-path "out;out/classes" --add-modules org.aspectj.runti
me,org.holamundomodaj.main,org.holamundomodaj.helloworld --output custom_jre

D:\Documentos\Pruebas\HolaMundoModularAspectos\src>
```

Figura 3.13: Máquina virtual personalizada de la *Prueba Hola mundo modular con AspectJ*

3.6. Análisis arquitectónico para realizar una transformación modular en sistemas legados desarrollados con Java 8

Los sistemas legados desarrollados con versiones previas al sistema de módulos no presentan de manera natural un diseño modular que permita aislar responsabilidades, dado que los paquetes de Java *per se* no dan un soporte modular adecuado y, considerando que las versiones previas a Java 9 presentaban una estructura robusta debido a la máquina virtual, es importante considerar un análisis arquitectónico que permita realizar una transformación modular en los sistemas de Java 8.

3.6.1. Análisis y detección de módulos en el sistema

Si bien los paquetes tienen un soporte modular limitado sin un descriptor de módulo que los defina como tal, los paquetes en sistemas no modulares brindan indicios de cuáles son los posibles módulos en dicho sistema. Los pasos a seguir para establecer un correcto análisis de dependencias modulares son los siguientes:

1. Revisar los paquetes y/o conjuntos de clases que sean identificables como módulos.
2. Revisar la integridad de los módulos candidatos y determinar si cumplen con la responsabilidad acorde a su módulo. En caso contar con responsabilidades de módulos ajenos:
   2.1) Subdividir los módulos en nuevos módulos candidatos.
3. Diagramar los módulos candidatos utilizando una herramienta CASE.
4. Identificar y diagramar las dependencias modulares entre los módulos del sistema. Las dependencias modulares en sistemas de Java 8 son identificables debido a la instaciación de otras clases (de otro paquete y/o conjunto de clases).

3.6.2. Problema de dependencia cíclica

El diseño modular preeliminar de sistemas legados de Java 8 tiende a presentar muchas dependencias entre módulos volviendo un diagrama de componentes difícil de leer. El problema potencial que surge en la mayoría de los escenarios es el problema de dependencia cíclica, dicho

problema dificulta la transformación modular de los sistemas legados debido a la incapacidad del sistema de módulos de Java de resolverlo por sí solo.

El problema de dependencia cíclica surge cuando un módulo `org.mod.a` requiere acceder a los recursos de un módulo `org.mod.b`, el módulo `org.mod.c` de igual manera requiere el acceso al módulo `org.mod.c`, así como el módulo `org.mod.c` requiere los recursos de `org.mod.a`, creando un ciclo de dependencias. Antes de empezar a trabajar con un determinado módulo a nivel de código, Java resuelve todas las dependencias con las que dicho módulo requiere trabajar (*module-path*), sin embargo, cuando se sucita un problema de dependencia cíclica Java desconoce el orden correcto para resolver ese cíclo de dependencias, ocasionando el error de dependencia cíclica. Dicho error solo es atendido si se rediseñan esas dependencias modulares. Un ejemplo que muestra el error de dependencia cíclica es el siguiente:

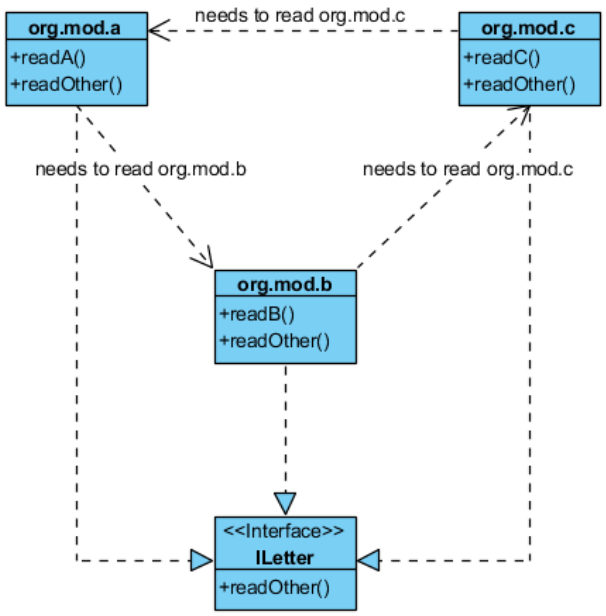

Figura 3.14: Problema de dependencia cíclica

El ejemplo de la Figura 3.14 plantea la problemática de los módulos `org.mod.a`, `org.mod.b` y `org.mod.c`, los cuales proveen de un método de lectura del módulo, de igual forma ellos tienen un método que les permite acceder a los recursos de otro módulo, siendo así que `org.mod.a` lee a `org.mod.b`, `org.mod.b` a `org.mod.c` y `org.mod.c` a `org.mod.a`, ocasionando un problema de dependencia cíclica.

3.6.3. Propuesta de solución al problema de dependencia cíclica

Una propuesta de solución que permite un correcto manejo de dependencias modulares es la utilización del principio de separación de asuntos, el cual es pilar fundamental de la modularidad. Una vez detectado el problema de dependencia cíclica, es conveniente detectar si dicho problema se presenta en otros módulos, para de esta forma, brindar una solución más completa, debido a que puede existir una dependencia modular que ocasione la generación de más de un problema de dependencia cíclica a la vez. Los pasos a seguir para dar solución al problema de dependencia cíclica son los siguientes:

1. Identificar todos los problemas de dependencia cíclica en el sistema.
2. Identificar los módulos comúnes causantes de dicho problema (ej. los que generan muchas dependencias en el sistema).
3. Identificar la causa del problema en los módulos (ej. una instancia que requieren muchos módulos entre sí).
4. Aplicar el principio de separación de asuntos aislando dicho problema en un nuevo módulo.

Retomando el problema de la Figura 3.14, aplicando la solución previamente mencionada quedaría de la siguiente forma en la Figura 3.6.3:

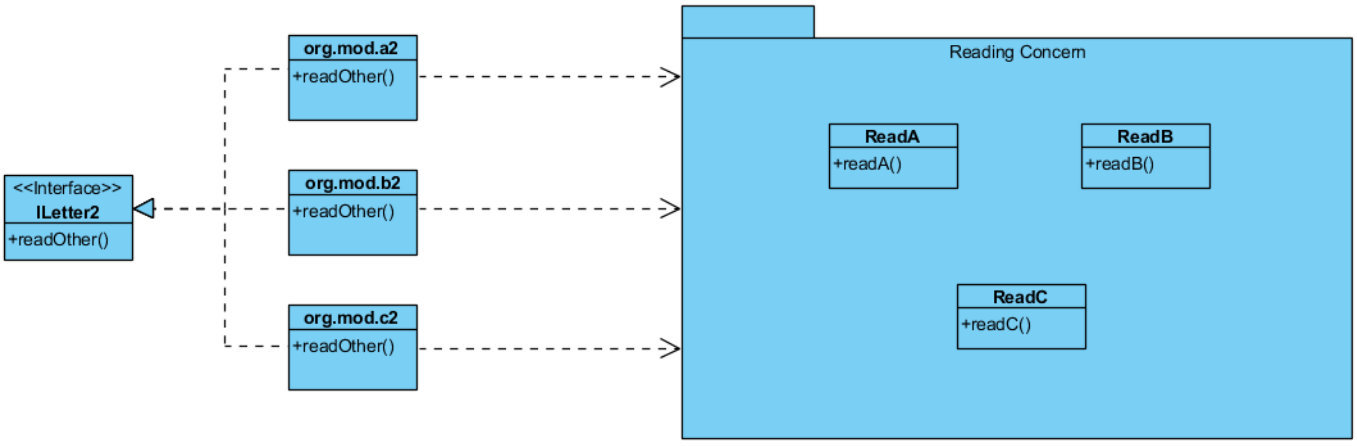

Figura 3.15: Solución arquitectónica propuesta

Al ser `Lectura` el asunto causante del problema de dependencia cíclica, el cual se presenta en los módulos `org.mod.a`, `org.mod.b` y `org.mod.c`, se aisla en un nuevo módulo *ReadingConcern*, en el cual se alojan las responsabilidades para consumir un servicio de lectura del módulo

que se requiera, centralizando de esta forma las dependencias modulares, dado que `org.mod.a` ahora depende de *ReadingConcern* al igual que `org.mod.b` y `org.mod.c`.

3.7. Estudio y realización de pruebas de las primitivas de corte que ofrece AspectJ en conjunto con Java 10 con la finalidad de determinar posibles mejoras modulares

El caso de estudio académico "Tabla periódica de elementos químicos"presenta la necesidad de enseñar a los alumnos de programación a realizar una correcta jerarquización y tipificación de los datos de los elementos químicos pertenecientes a la tabla periódica.

En esta sección se presenta la aplicación de algunos aspectos que se abordaran más adelante en el Capítulo 4, de manera que se comprenda el alcance modular que ofrece AspectJ en el sistema de módulos de Java.

El primer aspecto aborda una prueba sencilla, la cual busca demostrar la funcionalidad de AspectJ ante aplicaciones más robustas, dicho aspecto se muestra en el Código 3.10 y en la Figura 3.7.

Código 3.10: Aspecto *Prueba de compatibilidad*/Test.aj

```
1 package org.main.principal;
2 public aspect Test{
3   before(): execution(void org.main.principal.Laboratorio.main(..)){
4     System.out.println("Modularizando con Aspectos - Prueba 1");
5   }
6 }
```

```
D:\Documentos\Pruebas\TablaDeElementosQuimicos\out>java --module-path "C:/aspectj1.9/lib/aspectjrt.jar;classes" -m org.m
ain.principal/org.main.principal.Laboratorio
Modularizando con Aspectos - Prueba 1
```

Figura 3.16: Aspecto *Prueba de compatibilidad* aplicado al Caso de estudio

Como segundo aspecto, se tiene la problemática presentada en la Figura 3.7, la cual muestra

que la sección de moléculas OTS presenta una serie de errores que dificulta la utilización de la aplicación, con la finalidad de mejorar la modularidad del módulo `org.view.vista` se propone un aspecto que elimine esa sección defectosa, mejorando así la capacidad de utilización de la aplicación.

Figura 3.17: Problemática del segundo caso

A continuación se presenta la implementación y prueba de dicho aspecto en el Código 3.11 y en la Figura 3.8.

Código 3.11: Aspecto *Eliminación de moléculas OTS*/MoleculaOTS.aj

```
1  package org.view.vista;
2  import javax.swing.JPanel;
3  public aspect MoleculaOTS{
4    JPanel jpanel3;
5    JPanel jpanel8;
6
7    pointcut asignarComponentes():
8      execution(void org.view.vista.VentanaPrincipal.agregarComponentes());
9
10   pointcut leerCampoJP3():
```

```
11      get(JPanel org.view.vista.VentanaPrincipal.jp3);
12
13    pointcut leerCampoJP8():
14      get(JPanel org.view.vista.VentanaPrincipal.jp8);
15
16    JPanel around(): leerCampoJP3(){
17      jpanel3 = proceed();
18      return proceed();
19    }
20
21    JPanel around(): leerCampoJP8(){
22      jpanel8 = proceed();
23      return proceed();
24    }
25
26    after(): asignarComponentes(){
27      jpanel3.remove(jpanel8);
28    }
29  }
```

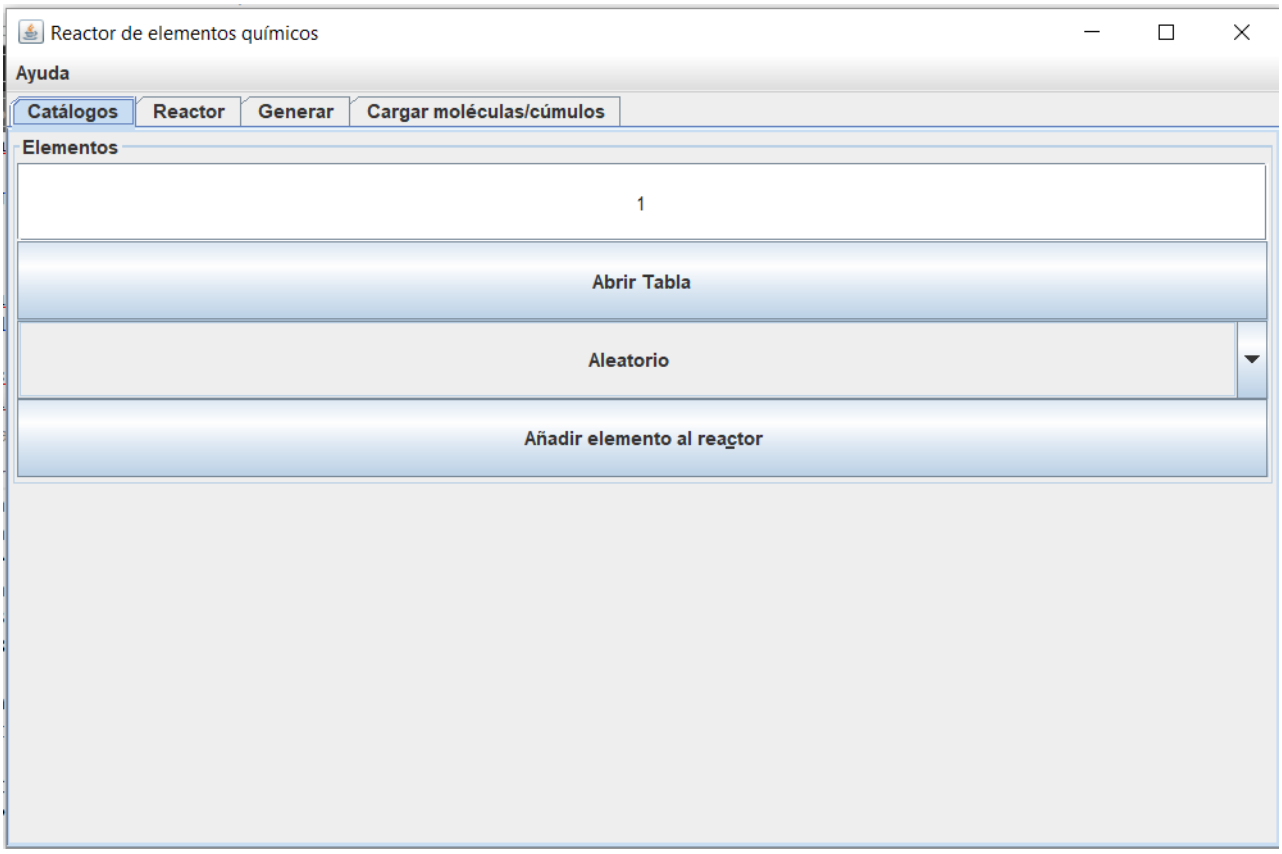

Figura 3.18: Aspecto *Eliminación de moléculas OTS* aplicado al Caso de estudio

3.8. Limitantes de AspectJ respecto al sistema de módulos de Java

AspectJ es una herramienta que permite mejorar la modularidad de los sistemas al eliminar el código disperso, invasivo y enmarañado. Cabe destacar que AspectJ al ser una herramienta sujeta en su totalidad a los avances de Java queda limitada en características ante las nuevas versiones de Java.

Andrew Clement, líder del equipo de desarrollo de AspectJ, señala que Java al introducir el sistema de módulos liberó Java 9, 10 y 11 con gran rapidez, por lo que AspectJ ante esa premura se procuró garantizar la compatibilidad de sus bibliotecas con el sistema de módulos de Java incorporandolas como módulos automáticos [32]. Java al proveer un esquema modular tan prometedor en conjunto con AspectJ tiene gran potencial modular, sin embargo, existen elementos como trabajo a futuro que, al momento de la escritura de esta tesis no están disponibles, tales como: los cortes intermodulares. La comunicación entre módulos está sujeta al descriptor de módulo, por lo que, si se necesitara un corte de AspectJ intermodular se tendría que actualizar el descriptor de módulo y por el momento, no está disponible dicha característica.

Bibliografía

[1] J. Rusnak A. MacCormack and C. Baldwin. The impact of component modularity on design evolution. *Harvard Business School*, 2007.

[2] F. Oquendo. Software architecture. *Springer-Verlag Berlin Heidelberg*, 2007.

[3] G. Booch. *Object-oriented Analysis and Design With Applications*. Addison-Wesley Professional, third edition, 2007.

[4] B. Meyer. *Object-oriented software construction*. Upper Saddle River, NJ: Prentice Hall, second edition, 1997.

[5] B. Meyer. *Touch of Class*. Springer-Verlag Berlin An, first edition, 2016.

[6] R. Laddad. *AspectJ in action*. Greenwich, CT: Manning, first edition, 2004.

[7] S. Clarke and E. Baniassad. *Aspect-oriented analysis and design*. Upper Saddle River, NJ: Addison-Wesley, 2005.

[8] Hilsdale M. Kersten G. Kiczales, J. Hugunin E. and J. Palm. *Aspect Oriented Programming*. Ft. Belvoir: Defense Technical Information Center, 2003.

[9] Oracle.com. Glossary. "http://www.oracle.com/technetwork/java/glossary-135216.html", 2018. Accessed: 18 Feb. 2018.

[10] N. Medvidovic R. Taylor and E. Dashofy. *Software architecture*. Hoboken, N.J: Wiley, first edition, 2010.

[11] Oracle.com. Server jre (java se runtime environment) 8 downloads. "http://www.oracle.com/technetwork/java/javase/downloads/server-jre8-downloads-2133154.html", 2018. Accessed: 22 Feb. 2018.

[12] M Fowler. Inversion of control containers and the dependency injection pattern. "https://martinfowler.com/articles/injection.html#InversionOfControl", 2018. Accessed: 25 Feb. 2018.

[13] M. Lungu N. Schwarz and O. Nierstrasz. Seuss: Decoupling responsibilities from static methods for fine-grained configurability. *The Journal of Object Technology*, 11(1):3:1–23, 2012.

[14] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[15] Oracle.com. Project jigsaw. "http://openjdk.java.net/projects/jigsaw/", 2017. Retrieved: 29/11/2017.

[16] Oracle.com. Oracle jdk 9 documentation. "https://docs.oracle.com/javase/9/index.html", 2017. Retrieved: 29/11/2017.

[17] Arik Hadas. Language Oriented Modularity: From Theory to Practice. In *Companion Proceedings of the 15th International Conference on Modularity*, MODULARITY Companion 2016, pages 1–21, New York, NY, USA, 2016. ACM.

[18] Arik Hadas. Language Oriented Modularity: From Theory to Practice. In *Companion Proceedings of the 15th International Conference on Modularity*, MODULARITY Companion 2016, pages 1–21, New York, NY, USA, 2016. ACM.

[19] Florian Kübler, Patrick Müller, and Ben Hermann. SootKeeper: Runtime Reusability for Modular Static Analysis. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP 2017, pages 19–24, New York, NY, USA, 2017. ACM.

[20] Andreas Rentschler, Dominik Werle, Qais Noorshams, Lucia Happe, and Ralf Reussner. Designing Information Hiding Modularity for Model Transformation Languages. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, pages 217–228, New York, NY, USA, 2014. ACM.

[21] Quentin Stiévenart, Jens Nicolay, Wolfgang De Meuter, and Coen De Roover. Building a Modular Static Analysis Framework in Scala (Tool Paper). In *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala*, SCALA 2016, pages 105–109, New York, NY, USA, 2016. ACM.

[22] Matthias Schöttle, Omar Alam, Jörg Kienzle, and Gunter Mussbacher. On the Modularization Provided by Concern-oriented Reuse. In *Companion Proceedings of the 15th International Conference on Modularity*, MODULARITY Companion 2016, pages 184–189, New York, NY, USA, 2016. ACM.

[23] Ismael Figueroa and Romain Robbes. Is Functional Programming Better for Modularity? In *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU 2015, pages 49–52, New York, NY, USA, 2015. ACM.

[24] Shigeru Chiba. To Be Destructive or Not to Be, That is the Question on Modular Extensions. In *Proceedings of the 13th Workshop on Foundations of Aspect-oriented Languages*, FOAL '14, pages 1–2, New York, NY, USA, 2014. ACM.

[25] Walter Cazzola and Albert Shaqiri. Modularity and Optimization in Synergy. In *Proceedings of the 15th International Conference on Modularity*, MODULARITY 2016, pages 70–81, New York, NY, USA, 2016. ACM.

[26] Naoyasu Ubayashi and Yasutaka Kamei. Design Module: A Modularity Vision Beyond Code: Not Only Program Code but Also a Design Model is a Module. In *Proceedings of the 5th International Workshop on Modeling in Software Engineering*, MiSE '13, pages 44–50, Piscataway, NJ, USA, 2013. IEEE Press.

[27] Martin Fleck, Javier Troya, and Manuel Wimmer. Towards Generic Modularization Transformations. In *Companion Proceedings of the 15th International Conference on Modularity*, MODULARITY Companion 2016, pages 190–195, New York, NY, USA, 2016. ACM.

[28] Fahad R Golra, Antoine Beugnard, Fabien Dagnat, Sylvain Guerin, and Christophe Guychard. Addressing Modularity for Heterogeneous Multi-model Systems Using Model Federation. In *Companion Proceedings of the 15th International Conference on Modularity*, MODULARITY Companion 2016, pages 206–211, New York, NY, USA, 2016. ACM.

[29] Ana Maria Şutîi, Tom Verhoeff, and Mark van den Brand. Modular Multilevel Metamodeling with MetaMod. In *Companion Proceedings of the 15th International Conference on Modularity*, MODULARITY Companion 2016, pages 212–217, New York, NY, USA, 2016. ACM.

[30] Darya Melicher, Yangqingwei Shi, Alex Potanin, and Jonathan Aldrich. A Capability-Based Module System for Authority Control. In Peter Müller, editor, *31st European*

Conference on Object-Oriented Programming (ECOOP 2017), volume 74 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 20:1—-20:27, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[31] Nico Ritschel and Sebastian Erdweg. Modular Capture Avoidance for Program Transformations. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2015, pages 59–70, New York, NY, USA, 2015. ACM.

[32] Andrew Clement. [aspectj-users] tips about aspectj and java module system to improve modularity. https://www.eclipse.org/lists//aspectj-users/threads.html, 2018. Accessed: 1-Nov- 2018.