SEP
SECRETARÍA DE EDUCACIÓN PÚBLICA

TECNOLÓGICO NACIONAL DE MÉXICO

DIVISIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN

OPCION I.- TESIS

TRABAJO PROFESIONAL

"DESARROLLO DE UNA HERRAMIENTA DE INGENIERÍA INVERSA PARA SCALA UTILIZANDO MECANISMOS DE REFLEXIÓN"

QUE PARA OBTENER EL GRADO DE:

MAESTRO EN SISTEMAS COMPUTACIONALES

PRESENTA:
MARIELA REMEDIOS LEZAMA SÁNCHEZ

DIRECTOR DE TESIS:
DR. ULISES JUÁREZ MARTÍNEZ

CODIRECTOR DE TESIS:
Mtro. CELIA ROMERO TORRES

ORIZABA, VER. MÉXICO JUNIO 2018

Índice general

Índice de figuras

Índice de tablas

Agradecimientos

A Dios por estar siempre conmigo, por ayudarme en todo momento y por darme la fortaleza que necesitaba para concluir esta etapa, pero sobre todo por ayudarme a alcanzar aquello que nunca imaginé poder ser.

A mi madre por sus oraciones, por hacer de mí la mujer que soy, por apoyarme y ayudarme a cumplir mis sueños. Gracias madre hacer todo lo posible, todo lo que va más allá hasta de ti misma para hacerme cumplir mis metas.

A mi padre por enseñarme que los logros entre más cuestan más saben. Por enseñarme a ser humilde.

A mi hermano por seguir mis pasos, por hacerme saber que hago un buen trabajo como hermana, como su ejemplo.

Al Dr. Ulises, por el apoyo para entrar a Maestría, por confiar en mí y por guiarme en estos años de estudio.

A mis compañeros: Betia, Daniel, Javier y Carlos por enseñarme y ayudarme en lo que se me dificultaba, por mostrarme el valor de la amistad.

A Bernardo por estar a mi lado y darme ánimos cuando estaba a punto de darme por vencida.

A las personas que creen en mí y que se alegran de mis logros.

Y al Consejo Nacional de Ciencia y Tecnología (CONACyT) por el apoyo económico que me permitió realizar estos estudios.

Resumen

La ingeniería inversa es el proceso de analizar un sistema para crear una representación del mismo, pero a un nivel más elevado de abstracción, con esto se refiere a recuperar la arquitectura de dicho sistema [3].

La ingeniería inversa es de gran importancia ya que ayuda a conocer de qué manera se implementó la arquitectura, es decir, si corresponde o no con la especificación que se hizo en el diseño; por otro lado, Scala es un lenguaje naturalmente funcional que se ejecuta sobre la Máquina Virtual de Java, lo cual hace que este lenguaje sea confiable y que gane popularidad en el mercado.

Con el advenimiento de la programación funcional, la comunicación entre componentes se realiza mediante funciones, lo cual repercute en la arquitectura del sistema, aquí nace la necesidad de saber el impacto que tiene la parte funcional en la arquitectura del sistema, por ello el objetivo de este proyecto es desarrollar una herramienta de ingeniería inversa que permita la representación objeto funcional de aplicaciones en Scala utilizando mecanismos de reflexión.

La herramienta tiene como entrada un archivo *.class* o *.scala*, de los cuales, mediante el mecanismo de Reflexión de Scala, se obtiene la información relevante de las entidades y miembros analizados (clases, *traits*, clases *case*, objetos, campos, métodos, parámetros y constructores). La construcción de las vistas importantes para la arquitectura y que son recuperables por ingeniería inversa se realizó utilizando el modelo 4+1; para crear la representación objeto funcional de manera correcta, se aplicó parte de la propuesta de notación descrita en [4]. Finalmente con esta información obtenida se estructuró un documento XMI

para visualizarla en un diagrama a través de la importación de este documento desde la herramienta de modelado *Enterprise Architect*.

Abstract

Reverse engineering is the process of analyzing a system to create a representation of itself, but at a higher level of abstraction, this refers to recovery of systems architecture [3]. Reverse engineering is of great importance since it helps to know how the architecture was implemented, that is, if it corresponds or not with the specification that was made in the design; on the other hand, Scala is a naturally functional language that runs on the Java Virtual Machine, which makes this language reliable and gains popularity in the market.

With the advent of functional programming, communication between components is done through functions, which affects the architecture of the system, there arises the need to know the impact that the functional part has on the architecture of the system, therefore the goal of this project is to develop a reverse engineering tool that allows object-functional representation of Scala applications using reflection mechanisms.

The tool has as input a *.class* or *.scala* file, from which, by means of the Scala Reflection mechanism, the relevant information of the analyzed entities and members is obtained (classes, traits, case classes, objects, fields, methods, parameters, and constructors). The construction of the important views for the architecture and that is recoverable by reverse engineering was done using the 4+1 model; to correctly create the representation of the functional object, a part of the notation proposal described in [4] was applied. Finally, with this information obtained, an XMI document was structured to visualize it in a diagram that could be imported from a modeling tool such as Enterprise Architect.

Introducción

En la parte concurrente y paralela de las aplicaciones las funciones son seguras para distribuirlas en varios núcleos debido a que evitan los datos mutables, esta distribución es posible de realizar debido al abaratamiento del hardware, por consecuencia, el cálculo de dichas funciones se agiliza, esta es la razón por la que la programación funcional gana popularidad.

Tradicionalmente los conectores entre componentes son llamadas a métodos, los cuales, por ejemplo, envían un valor. Actualmente, en sistemas basados en el paradigma funcional, los conectores envían una función.

El lenguaje Scala tiene sus inicios en el 2003, sin embargo, dada la popularidad que adquiere, este lenguaje se considera novedoso, por esta razón no se reporta alguna herramienta que ayude a conocer cómo impacta la parte funcional en la arquitectura de un sistema.

Lo anterior representa una necesidad y una oportunidad para desarrollar la herramienta de ingeniería inversa que muestre cómo se encuentra la arquitectura del sistema en Scala, representando de manera adecuada la parte funcional de la misma.

Este objetivo se logró mediante el estudio del lenguaje para conocer el paradigma objeto funcional y sus capacidades funcionales, además se realizó un análisis del mecanismo de reflexión que soporta Scala y así se determinaron las capacidades de ingeniería inversa con las que cuenta la herramienta. Para representar la arquitectura del sistema se utilizó el modelo 4+1 vistas, asimismo, mediante el estudio de la notación objeto-funcional desarrollada en [4], se logró una representación correcta de la parte funcional en la arquitectura.

La tesis está estructurada de la siguiente manera: En el capítulo 1 se muestra el marco teórico, el cual expone los conceptos principales que servirán para entender mejor el tema

de tesis, además de la problemática, objetivo general, objetivos específicos y la justificación del proyecto. En el capítulo 2 se describe el estado de la práctica, dicho capítulo contiene los resúmenes y la comparativa de los trabajos relacionados con el proyecto. En el capítulo 3 se describe la manera en que se desarrolló la herramienta. En el capítulo 4 se presentan los resultados obtenidos con la aplicación en un caso de estudio. Por último se presentan las conclusiones y trabajo futuro.

Capítulo 1

Antecedentes

En este capítulo se muestran conceptos para el entendimiento de la tesis, seguido de esto se presenta la problemática a resolver, así como los objetivos general y específicos y, por último, la justificación.

1.1. Marco teórico

En esta sección se definen los conceptos necesarios para comprender los temas que se abordan en este tema de tesis. Cabe mencionar que algunas de las referencias citadas tienen una antigüedad mayor a diez años, esto debido a que no se reportan nuevas definiciones de los conceptos y en otros casos se decidió tomar las definiciones de artículos y libros a los que más se hace referencia en la actualidad.

1.1.1. Ingeniería inversa

La ingeniería inversa se considera como el proceso de descubrir los principios tecnológicos de un dispositivo, objeto o sistema, mediante el análisis de su estructura, funcionamiento u operación. Este proceso consiste en tomar una entidad por separado y analizar a detalle su funcionamiento, usualmente con el fin de construir un dispositivo o programa nuevo que hará lo mismo, pero sin copiar todos los aspectos del original [3]. La ingeniería inversa es de gran

importancia ya que ayuda a conocer de qué manera se implementó la arquitectura, es decir, si corresponde o no con la especificación que se hizo en el diseño.

1.1.2. Ingeniería inversa de software

La ingeniería inversa es el proceso de analizar un sistema para crear una representación del mismo, pero a un nivel más elevado de abstracción [5]. La ingeniería inversa se ve como el proceso de analizar un sistema identificando sus componentes e interrelaciones para crear la representación física del mismo en otra forma o un nivel superior de abstracción.

1.1.3. Programación funcional

La programación funcional es un paradigma de programación que se basa en la evaluación de funciones matemáticas y evita los datos mutables y de estado. En la programación funcional los programas están compuestos de funciones que toman una entrada, producen valores y posiblemente otras funciones. Los bloques de construcción de la programación funcional no son objetos ni procedimientos (estilo de programación C), sino funciones. Una definición simple de programación funcional es "programación con funciones"[6].

La programación funcional no tiene efectos secundarios o mutabilidad. Los beneficios de no tener mutabilidad y efectos secundarios en los programas funcionales son que los programas son muy comprensivos, porque la actividad de la función es completamente local y no tiene efectos externos. Otra gran ventaja de la programación funcional es la facilidad de la programación concurrente.

1.1.4. Scala

Scala es un lenguaje de programación moderno multi-paradigma diseñado para expresar patrones de programación comunes de una forma concisa, elegante, y de tipificación segura. Integra fácilmente características de lenguajes orientados a objetos y funcionales.

Scala es un lenguaje puramente orientado a objetos, en el sentido de que cada valor es

un objeto. Los tipos y el comportamiento de los objetos se describen por clases y *traits* que se utilizan para definir tipos de objetos especificando la firma de los métodos soportados. A diferencia de Java, Scala permite que los rasgos se implementen parcialmente; es decir, es posible definir implementaciones predeterminadas para algunos métodos. En contraste con las clases, los rasgos no encapsulan el estado definiendo variables. Ni siquiera se les permite recibir argumentos de constructor.

Las abstracciones de clase se extienden mediante subclases y un mecanismo de composición basado en *mixin* como reemplazo para herencia múltiple. La composición *mixin* permite reutilizar todas las nuevas definiciones que no se heredan, en la definición de una nueva clase.

Scala es también un lenguaje funcional, en el sentido de que cada función es un valor. Scala proporciona una sintaxis compacta para definir funciones anónimas, lo que significa que se expresan funciones en la sintaxis literal de la función y que las funciones se representan por objetos, que se denominan valores de la función.

1.1.5. Reflexion

La reflexión es la capacidad integral de un programa para observar o cambiar su propio código, así como todos los aspectos de su lenguaje de programación (sintaxis, semántica o implementación), incluso en tiempo de ejecución [7].

En general, hay tres técnicas que una API *(Application Programming Interface, interfaz de programación de aplicaciones)* de reflexión utiliza para facilitar el cambio de comportamiento: la modificación directa de meta-objetos, las operaciones de uso de metadatos (como la invocación de métodos dinámicos) y la intercesión, en las que se permite interceder en varias fases de ejecución del programa. Se dice que un lenguaje de programación es reflexivo cuando proporciona a sus programas una reflexión completa, es decir, cuando cumple con las tres técnicas anteriores. La reflexión se utiliza para la observación y la modificación de un programa en tiempo de ejecución. En los lenguajes de programación orientados a objetos, la reflexión permite la inspección de clases, interfaces, campos y métodos en tiempo de ejecución sin conocer los nombres éstos en tiempo de compilación. También permite la instanciación

de objetos nuevos y la invocación de métodos. La reflexión también se utiliza para adaptar un programa dado a las diferentes situaciones de forma dinámica. La reflexión se utiliza a menudo como parte de las pruebas de software.

1.1.6. Bytecode

El *bytecode* es una instrucción que la JVM *(Java Virtual Machine, Máquina Virtual de Java)* espera recibir. Éste está contenido en un archivo *.class*, que es el formato binario que la JVM entiende, junto con una tabla de símbolos e información auxiliar [8].

1.1.7. Arquitectura de software

La arquitectura de software de un sistema o programa de computación es la estructura del sistema, que comprende componentes de software, las propiedades externamente visibles de esos componentes y las relaciones entre ellos. Las "propiedades externamente visibles"se refieren a aquellas otras partes que forman a otro componente, como sus servicios prestados, las características de rendimiento, manejo de errores y el uso de recursos compartidos [9].

En [10] la arquitectura de un sistema de software se define como el conjunto de decisiones de diseño principales sobre el sistema. El término principal implica un grado de importancia y actualidad que otorga una decisión de diseño al estado arquitectónico, es decir, que la convierte en una decisión de diseño arquitectónico. También implica que no todas las decisiones de diseño son arquitectónicas. De hecho, muchas de las decisiones de diseño tomadas en el proceso de ingeniería de un sistema (por ejemplo, los detalles de los algoritmos o estructuras de datos seleccionados) no afectarán la arquitectura de un sistema. A diferencia del diseño de software, la arquitectura comprende el nivel más alto de abstracción.

1.1.8. Componente

Un componente de software es una entidad que encapsula un subconjunto de funcionalidad y/o datos del sistema, restringe el acceso a ese conjunto vía una interfaz definida en forma

explícita y que posee dependencias explícitamente definidas en su contexto requerido de ejecución [10].

Un componente de software es una unidad de composición con interfaces contractualmente especificadas y explícitas sólo con dependencias dentro de un contexto. Un componente de software se despliega independientemente y está sujeto a la composición de terceros [11].

1.1.9. Conector

Los conectores de software son la abstracción arquitectónica encargada de gestionar las interacciones de los componentes. Un conector de software es un elemento arquitectónico encargado de efectuar una interacción reguladora entre los componentes.

Existen varios tipos de conectores, algunos de ellos se describen a continuación:

- Invocación a métodos: Este conector se implementa directamente en los lenguajes de programación, donde típicamente permite el intercambio sincronizado de datos y el control entre pares de componentes: el componente invocador (el llamador) pasa el hilo de control, así como los datos en forma de parámetros de invocación al invocado (el llamado); después de completar la operación solicitada, el receptor devuelve el control, así como cualquier resultado de la operación, al llamador.

- Acceso a datos: Estos conectores permiten a los componentes acceder a los datos que residen en un componente de almacén de datos, por lo tanto, proporcionan servicios de comunicación. El acceso a datos a menudo requiere la preparación del almacén de datos antes y la limpieza después de que haya completado el acceso. En caso de que haya una diferencia de formato de los datos requeridos y el formato en el que se almacenan y proporcionan los datos, los conectores de acceso a datos realizan la traducción de la información a la que se accede, es decir, la conversión.

- Distribución: Estas conexiones típicamente encapsulan las interfaces de programación de aplicaciones de la biblioteca de red para permitir que los componentes de un sistema distribuido interactúen. Un conector de distribución está normalmente acoplado con un

conector más básico para aislar los componentes interactivos que forman los detalles de distribución del sistema.

- Adaptador: Muchos sistemas de software se construyen a partir de componentes preexistentes, que no se hacen a medida del sistema dado. En tales casos, los componentes necesitan ayuda para integrarse e interactuar unos con otros. Se emplean conectores adaptadores para este fin. Dependiendo de sus características y del contexto en el que se utilizan, las envolturas y el código de pegamento son dos tipos comunes de conectores de adaptadores con los cuales el lector está familiarizado [10].

1.1.10. Diseño del software

El diseño crea una representación o modelo del software. El modelo de diseño proporciona detalles sobre estructuras de datos, interfaces y componentes que se necesitan para implementar el sistema. Además, incluye el punto de vista arquitectónico.

El diseño permite modelar el sistema o producto que se va a construir y es el lugar en el que se establece la calidad del software. El trabajo principal que se produce durante el diseño del software es un modelo de diseño que agrupa las representaciones arquitectónicas, interfaces en el nivel de componente y despliegue [12].

A diferencia de la arquitectura de software, en el diseño se toman en cuenta todas las decisiones de diseño, las cuales abarcan todos los aspectos del sistema en desarrollo, incluyendo [10]:

- Estructura del sistema.
- Comportamiento funcional.
- Interacción.
- Propiedades no funcionales.
- Implementación.

1.1.11. Documentación de arquitectura

La documentación de una arquitectura consiste principalmente en describir las vistas de esa arquitectura, además de registrar información que se aplica a más de una vista [13]. Un modelo que describe la arquitectura del software es el Modelo 4 + 1, el cual usa cinco vistas concurrentes.

La arquitectura lógica La arquitectura lógica apoya principalmente los requisitos funcionales, es decir, lo que el sistema brinda en términos de servicios a sus usuarios. El sistema se descompone en una serie de abstracciones clave, tomadas del dominio del problema en forma de objetos o clases de objetos. Aquí se aplican los principios de abstracción, encapsulamiento y herencia. Esta descomposición no sólo se hace para potenciar el análisis funcional, sino también sirve para identificar mecanismos y elementos de diseño comunes a diversas partes del sistema.

La vista de procesos La arquitectura del proceso tiene en cuenta algunos requisitos no funcionales, como el rendimiento y la disponibilidad. Se ocupa de cuestiones de concurrencia y distribución, de integridad del sistema, de tolerancia a fallos y de cómo las principales abstracciones de la vista lógica encajan dentro de la arquitectura del proceso, en la que el hilo de control es una operación para un objeto realmente ejecutado.

El software se divide en un conjunto de tareas independientes. Una tarea es un hilo de control separado, que se programa individualmente en un nodo de procesamiento. Se distinguen entonces: las tareas principales, que son los elementos arquitectónicos que son abordados de forma única y las tareas menores, que son tareas adicionales introducidas localmente por razones de implementación.

Las tareas principales interactúan a través de un conjunto de mecanismos de comunicación bien definido. Las tareas secundarias se comunican mediante el encuentro o la memoria compartida.

La vista de desarrollo La vista de desarrollo se centra en la organización real de los módulos de software en el ambiente de desarrollo del software. El software se empaqueta en partes pequeñas que se construyen por uno o un grupo mínimo de desarrolladores. Los

subsistemas se organizan en una jerarquía de capas, cada una de ellas brinda una interfaz estrecha y bien definida hacia las capas superiores.

Arquitectura física La arquitectura física toma en cuenta primeramente los requisitos no funcionales del sistema tales como la disponibilidad, confiabilidad, desempeño y escalabilidad. El software se ejecuta sobre una red de computadoras o nodos de procesamiento. Los elementos identificados tales como redes, procesos, tareas y objetos requieren mapearse sobre los nodos.

Escenarios Los elementos de las cuatro vistas trabajan conjuntamente en forma natural mediante el uso de un grupo de escenarios relevantes para los cuales se describen las secuencias de interacciones entre objetos y entre procesos. Los escenarios son de alguna manera una abstracción de los requisitos más importantes.

El diseño de los mismos se expresa mediante el uso de diagramas de escenarios y diagramas de interacción de objetos.

La vista de escenarios sirve a dos propósitos principales [14]:

- Como una guía para descubrir elementos arquitectónicos durante el diseño de arquitectura.

- Como un rol de validación e ilustración después de completar el diseño de arquitectura, en el papel y como punto de partida de las pruebas de un prototipo de arquitectura.

1.1.12. XMI

XMI (XML de Intercambio de Metadatos, *XML Metadata Interchange*) es una especificación para el Intercambio de Diagramas. La especificación para el intercambio de diagramas se escribió para proveer una manera de compartir modelos UML entre diferentes herramientas de modelado. En versiones anteriores de UML se utilizaba un Schema XML; pero este esquema no decía nada acerca de cómo debía representarse el modelo. Para solucionar este problema la nueva especificación para el intercambio de diagramas se desarrolló mediante un nuevo esquema XML que permite construir una representación SVG (*Scalable Vector Graphics*).

Típicamente esta especificación se utiliza solamente por quienes desarrollan herramientas de modelado UML.

El objetivo de XMI es permitir el fácil intercambio de metadatos entre las herramientas de modelado basadas en UML y los repositorios de metadatos basados en MOF en entornos heterogéneos distribuidos. XMI integra tres estándares de la industria:

- XML - Lenguaje de marcado extensible, un estándar W3C. Ver XML.
- UML - Lenguaje de modelado unificado, un estándar de modelado OMG. Ver UML.
- MOF - Meta Object Facility, un metamodelo OMG y un estándar de repositorio de metadatos. Ver MOF.

La integración de estos tres estándares en XMI permite a los desarrolladores de sistemas distribuidos compartir modelos de objetos y otros metadatos [15].

1.1.13. AspectJ

AspectJ es un lenguaje de programación orientado por aspectos construido como una extensión del lenguaje Java. Un compilador de AspectJ produce archivos de clase que se ajustan a la especificación del *bytecode* Java, lo que permite que cualquier JVM compatible ejecute esos archivos de clase. Al utilizar Java como el lenguaje base, AspectJ transfiere todos los beneficios de Java y facilita que los programadores de Java comprendan el lenguaje AspectJ.

AspectJ consta de dos partes: la especificación del lenguaje y la implementación del lenguaje. La parte de especificación del lenguaje define la sintáctica y semántica del lenguaje. La parte de implementación del lenguaje proporciona herramientas para compilar, depurar e integrar con los entornos de desarrollo integrados (IDE).

AspectJ evolucionó para facilitar el enfoque del desarrollo orientado a aspectos, como por ejemplo, una sintaxis alterna, basada en las anotaciones de Java, para expresar construcciones de corte en puntos (crosscutting), permitiendo el uso del compilador de Java en lugar de uno especializado [16].

1.2. Paradigma objeto funcional

El paradigma objeto funcional es un enfoque que tiene como base la orientación a objetos e integra propiedades funcionales.

Con el paradigma objeto funcional, los medios orientados a objetos se utilizan para fomentar la capacidad de cambio y comprensión del código local; los modificadores de acceso se usan para permitir solo dependencias sensibles, si es posible. Las características funcionales se utilizan para evitar la mutación y referencias de objetos.

En el enfoque OO los objetos mutables, es decir, aquellos que son modificados tras su creación, no fácilmente se usan de una manera segura en un contexto paralelo y el manejo de objetos a menudo es demasiado pesado. El enfoque funcional, por el contrario, brilla con características ligeras, así como su limitación y aislamiento de los efectos secundarios, es decir, aquellos que modifican una variable o cualquier estado de su entorno.

El paradigma objeto funcional reúne las ventajas de ambos enfoques originales, evitando sus desventajas. Al final, esto conduce a una productividad y una calidad de software potencialmente más elevadas [17].

1.3. Planteamiento del problema

Scala es un lenguaje de programación moderno, multi-paradigma, diseñado para expresar patrones de programación comunes de una forma concisa, elegante, y de tipificación segura. Integra fácilmente características de lenguajes orientados a objetos y funcionales [18]. Se dice que Scala es un lenguaje funcional, ya que toda función es un valor. Scala provee una sintaxis compacta para definir:

- Funciones anónimas
- Funciones de orden superior
- Funciones anidadas

- Funciones currificadas

Con el advenimiento del paradigma de programación objeto-funcional nace la necesidad de un estándar para la modelación de los elementos de dicha programación, problemática que se abordó en [4].

Además del paradigma funcional, Scala cuenta con un mecanismo de reflexión, el cual es prácticamente nuevo, ya que se introdujo en Scala 2.10, lanzada el 4 de enero del 2013, y la versión actual es la 2.12.4. Dado el potencial de Scala, teniendo ya una notación y ante la necesidad de contar con el soporte adecuado que facilite la adaptación de sistemas, es necesario desarrollar una herramienta de ingeniería inversa, utilizando el mecanismo de reflexión de este lenguaje y representando la estructura de los sistemas en un modelo objeto-funcional [18].

1.4. Objetivo general y objetivos específicos

En este apartado se dan a conocer el objetivo general y los objetivos específicos del tema de tesis. Dentro de estos objetivos se hace referencia a la notación propuesta en [4], a la cual se le nombrará como „Rodríguez„.

1.4.1. Objetivo general

Desarrollar una herramienta de ingeniería inversa que permita la representación objeto funcional de aplicaciones en Scala utilizando mecanismos de reflexión.

1.4.2. Objetivos específicos

- Estudiar el lenguaje Scala para conocer el paradigma objeto funcional y sus capacidades funcionales a través de la revisión de la documentación oficial y la bibliografía relacionada.

- Analizar el mecanismo de reflexión soportado por Scala comparándolo con Java para determinar las capacidades de ingeniería inversa a plantear en la herramienta.
- Estudiar la propuesta de notación objeto-funcional de „Rodríguez„para determinar las capacidades de representación objeto funcional que soportará la herramienta.
- Desarrollar una herramienta de ingeniería inversa mediante el análisis de artefactos adecuados para la representación funcional.
- Mostrar las capacidades de la herramienta en la representación objeto-funcional a partir del caso de estudio reportado por „Rodríguez„.

1.5. Justificación

La ingeniería inversa tiene por objetivo recuperar la estructura del sistema a partir del código intermedio y esto es factible utilizando el mecanismo de reflexión. Por medio de la ingeniería inversa que provee este mecanismo, se obtiene la información necesaria para recuperar dicha estructura.

En Scala se cuenta con el paradigma funcional y la API de Reflection, asimismo, dadas las necesidades para representar elementos funcionales ya se ha propuesto una notación.

Dado que se cuenta con los mecanismos para realizar la representación objeto funcional, es necesario estudiar la infraestructura de reflexión, el soporte funcional con el que cuenta este lenguaje, la notación objeto-funcional y así implementar y probar la herramienta para la correcta representación de la estructura del sistema.

Capítulo 2

Estado de la práctica

En este capítulo se muestra un resumen de los trabajos encontrados y el análisis comparativo, resultado de la investigación que se realizó sobre las herramientas de ingeniería inversa, con el objetivo de conocer los enfoques de las mismas, los problemas que resuelven y los resultados obtenidos. Dicha investigación ayudó a determinar las ventajas que tiene el proyecto a realizar en comparación con los ya encontrados.

2.1. Trabajos relacionados

Los trabajos relacionados con el tema de tesis se clasificaron de la siguiente manera: recuperación de dependencias, recuperación de diseño, análisis estático y dinámico y metodología.

2.1.1. Recuperación de dependencias

Uno de los desafíos a los que los desarrolladores de Java se enfrentan es la incapacidad para determinar el número correcto de las dependencias de clase. La capacidad de recuperar dependencias de clase ayuda a los desarrolladores a comprender el diseño de un sistema antes de modificarlo. Se propusieron herramientas para esta intención, pero pocas son capaces de analizar los tipos de dependencia asociados con el *bytecode* de Java. Para abordar este problema, en [19] se desarrolló una herramienta de software llamada *Bytecode-based Class*

Dependency Extraction Tool (Bytecode-CDET), que extrae las dependencias desde archivos de *bytecode* de *Java Virtual Machine* (JVM) (*.class*), los cuales son equivalentes a binarios en otras máquinas. El propósito principal de esta herramienta fue recuperar las dependencias de clase de los programas Java, lo cual ayudó a los desarrolladores a comprender las dependencias de un sistema existente. La herramienta comprendió las siguientes etapas:

1. **Análisis**: En este paso, la herramienta analizó el archivo JAR utilizando la API (Application Programming Interface, interfaz de programación de aplicaciones) Java Reflexión y Javassist, para traducir cada archivo en nodos de árboles de sintaxis abstractas (ASTs) particulares (es decir, constructores, variables y métodos).

2. **Extracción**: Bytecode-CDET extrajo las dependencias basadas en los nodos AST conseguidos en el primer paso. La herramienta obtuvo dependencias de los siguientes tipos: Variable Parámetro de constructor Interfaz Extensión Retorno del método Parámetro del método Cuerpo del Método.

3. **Generación**: Se desarrolló un componente generador XML para convertir los objetos extraídos en un documento XML.

4. **Importación**: Para representar visualmente las dependencias recuperadas, Bytecode-CDET importó automáticamente el documento XML generado.

5. **Exportación**: La herramienta permitió al usuario no sólo guardar el archivo XML a la máquina local, sino también exportar la información como un archivo de imagen.

A medida que las aplicaciones WEB evolucionan con el tiempo, se vuelven más complejas y difíciles de entender. Esto es cierto para la mayoría de las aplicaciones de software, pero es más agudo para los proyectos WEB, ya que suelen implicar varios idiomas en ambos lados del cliente y del servidor. Debido a todo esto, es muy difícil rastrear las dependencias entre elementos de una aplicación WEB cuando la documentación es inexistente u obsoleta. El soporte de herramientas es clave para facilitar o evitar la constante disciplina necesaria para la creación y actualización de la documentación. En [20] se presentó WAVI (WEB

Application Viewer), una herramienta de ingeniería inversa para aplicaciones WEB. WAVI permite analizar las aplicaciones WEB para extraer su estructura y visualizar el mismo utilizando diagramas intuitivos y comunes. La herramienta se basa en el análisis estático impulsado por una serie de filtros discriminatorios que ayudan a resolver las llamadas de función. La herramienta realizó los siguientes pasos:

1. Extraer información del código fuente: Como primer paso, el código fuente se procesó en un árbol de sintaxis. El segundo paso consistió en recorrer los nodos en busca de elementos que se consideraron relevantes para la documentación.

2. Presentación de una estructura de aplicación WEB: Para la representación se consideraron las siguientes salidas:

   - Salida textual: un archivo de datos JSON (JavaScript Object Notation, Notación de objetos JavaScript).
   - Salida gráfica 1: Gráficos dirigidos por la fuerza.
   - Salida gráfica 2: Diagramas de clases.

En [21] se mencionó que debido a que el tiempo de comercialización de aplicaciones es muy corto, éstas a menudo se desarrollan de una manera no disciplinada; es decir, este enfoque conduce en la mayoría de las veces a páginas WEB completas desarrolladas como una mezcla de SQL (*Structured Query Language*, lenguaje de consulta estructurada), PHP (*Hypertext Processor*, procesador de hipertexto), HTML (*HiperText Markup Language*, lenguaje de marcas de hipertexto) y JS (*JavaScript*, lenguaje de programación interpretado). A su vez, esto repercute en un esfuerzo considerable para mantener dichas aplicaciones. Para mejorar el desarrollo y la eficiencia de mantenimiento también estas páginas tienen que modelarse y su diseño tiene que elaborarse utilizando un estándar o lenguajes de modelado, como UML. Es por esto que se presenta una herramienta de ingeniería inversa llamada *phpModeler* que analiza de forma estática el código fuente de la aplicación WEB (*scripts* php, páginas HTML y bibliotecas JavaScript) y genera modelos que se utilicen como base para la recuperación de la arquitectura y que facilitan el mantenimiento de aplicaciones WEB.

La ingeniería inversa es la idea clave para la reconstrucción de cualquier sistema existente. Algunos de los investigadores trataron en el pasado el problema de la ingeniería inversa de un código orientado a objetos a diagramas de clase UML. Sin embargo, ninguno de estos investigadores trató todas las construcciones disponibles en los diagramas de clase UML. A diferencia del trabajo realizado anteriormente en ingeniería inversa en UML, el algoritmo que se propuso en [22] generó reglas para un conjunto completo de construcciones disponibles en diagramas de clases UML. Dicho algoritmo incluyó clases, relaciones, objetos, atributos, operaciones, herencia, asociaciones, interfaces y otros mecanismos extensibles. La entrada del algoritmo fue un código orientado a objetos y la salida fue el diagrama de clase UML correspondiente, para lo cual se realizaron los siguientes pasos: identificar las cosas estructurales, identificar las elementos de comportamiento, identificar los elementos agrupación, identificar los elementos por anotación, identificar objetos, identificar la dependencia, identificar asociación, identificar generalización, identificar la realización e identificar mecanismos ampliables. El algoritmo que se presentó es independiente de cualquier lenguaje de programación orientado a objetos.

2.1.2. Recuperación de diseño

Los encargados del mantenimiento de software necesitan herramientas para recuperar las opciones de diseño desde el código basado en sus patrones de origen. En [23] se mostró la *suite* de herramientas *Ptidej*, un ambiente de ingeniería inversa diseñado para facilitar el desarrollo de algoritmos de identificación de patrones. Las principales habilidades de la *suite* fueron su flexibilidad y extensibilidad para permitir que varios algoritmos funcionaran e interactuaran armoniosamente.

La detección de patrones de diseño es una metodología de ingeniería inversa que ayuda a los ingenieros de software a analizar y comprender el sistema recuperando su diseño y ayudando así en la preparación de actividades de reingeniería. Las herramientas que recuperan el diseño del sistema localizando instancias de patrones de diseño soportan la tarea tediosa y propensa a errores de comprender el software. En [24], se desarrolló un enfoque de detección

de patrones, llamado *Reclipse*, que integra un análisis estático estructural con un posterior análisis dinámico del comportamiento en tiempo de ejecución. El análisis estático se utiliza para detectar posibles instancias de patrón y cuando sea necesario se utiliza el análisis dinámico para verificar el comportamiento en tiempo de ejecución.

Para los desarrolladores es difícil comprender el software de otro desarrollador, ya que añade una gran cantidad de trabajo al modificar y extender las aplicaciones heredadas. Para resolver esto en [25] se desarrolló un prototipo de herramienta para extraer modelos de diseño del código fuente del programa para obtener mejor comprensión de código. Este prototipo permitió a los desarrolladores de software obtener una comprensión muy clara de una pieza existente de software y se añadieron cuatro características principales a la plataforma *UMLet* que incluye generación automática con relaciones, personalización en tiempo de ejecución de los diagramas, un panel de información que detalla las clases específicas y un enlace directo entre el diagrama y su fuente.

En [26] se describió un sistema que utiliza el generador de código IBM *VisualAge*, una herramienta que ahora ya no se soporta, sin embargo, la aplicación para la que se desarrolló el sistema sigue activa y estará en funcionamiento durante muchos años adelante. Se trata de un sistema para pagar los salarios de los empleados estatales, las pensiones de los empleados jubilados y los contratistas del estado. Para solucionar esto se tenían cinco alternativas y se optó por re-implementar el sistema en un lenguaje nuevo, al hacer esto se encontraron con tres enfoques para realizar la conversión del lenguaje, el enfoque de conversión semi-automatizada fue el que se eligió. En este enfoque surgió la necesidad de ingeniería inversa debido a que el desarrollador necesitaba la documentación del antiguo sistema. Cuando aplicaron ingeniería inversa a la herramienta *VisualAge* se ahorraron unas 6000 horas-persona. Uno de los desafíos de los desarrolladores de Fortran OO (*Object-oriented*, Orientado a objetos) es la incapacidad de obtener descripciones de software de alto nivel de las aplicaciones. La comunidad de ingeniería de software utiliza técnicas de ingeniería inversa para hacer frente a este reto. En [27] se propuso una herramienta de software para extraer diagramas de clase UML de código Fortran llamada *ForUML*. El proceso de transformación se basó en un enfoque de

ingeniería inversa. El diagrama de clases UML se representó con la herramienta de modelado UML llamada *ArgoUML*. Esta herramienta generó una representación visual de software implementado en Fortran OO. Los resultados experimentales mostraron que *ForUML* genera diagramas de alta precisión.

La mayoría de las herramientas automáticas de ingeniería inversa funcionan mal, centrándose en la producción de diagramas de clase sencillos sin representar correctamente las abstracciones de diseño, además, no revelan los procesos internos para producir el diagrama UML, entonces esto da lugar a resultados que no satisfacen las expectativas de los usuarios finales. Con el fin de proporcionar una herramienta de código abierto utilizable y disponible, abordar los problemas con eficiencia, y facilitar un formato de salida más conveniente, en [28] se creó la herramienta *srcYUML*. Esta herramienta es altamente eficaz y precisa para diagramas de clase de ingeniería inversa.

2.1.3. Análisis estático y dinámico

La demanda para diferentes herramientas de análisis asistente crece significativamente, incluyendo las herramientas de análisis estático y dinámico. En [29] se introdujo una herramienta de análisis dinámico de ingeniería inversa, llamada *ROPTool*. Esta herramienta usa la máquina virtual modificada QEMU (*Quick Emulator*, Emulador rápido) para obtener información importante ayudando a que la gente entienda el software, y supera las deficiencias de la tecnología tradicional de herramientas de análisis dinámico. *ROPTool* se utilizó para recopilar información de software dinámicamente.

En [30] se desarrolló *Linter*, una herramienta de análisis estático para Scala, la cual verifica posibles errores, código ineficiente y problemas de estilo de codificación. Esta herramienta se implementó como un complemento del compilador en lugar de una aplicación independiente. Internamente *Linter* consta de las siguientes cuatro fases de compilación:

1. El detector de inferencia de tipo registra todos los elementos del programa cuyos tipos tendrán que ser inferidos.

2. El patrón de coincidencias realiza la mayor parte de las comprobaciones de la correspondencia de los otros patrones.

3. El intérprete abstracto realiza chequeos basados en interpretación abstracta.

4. El verificador de parámetros implementa una comprobación que detecta si un parámetro de un método sin sobrescritura no se utiliza (esto sólo se comprueba después de que el compilador Scala realiza las pruebas de referencia o anulación).

2.1.4. Metodología de ingeniería inversa

La metodología de ingeniería inversa que se desarrolló en [31] se basó en la especificación inversa de los casos de uso vinculados a la ejecución del sistema. El objetivo en este artículo fue recuperar la traza entre el modelo de robustez que representa el análisis del caso de uso y sus clases de implementación real. Por lo tanto, necesitaban editar los casos de uso y los escenarios del sistema para que el medio ambiente pudiera procesar esta información, junto con el modelo de robustez y la ejecución para recuperar los enlaces de trazabilidad. Para resolver este problema desarrollaron un editor de casos de uso y de escenarios, esta herramienta se desarrolló como un complemento para Eclipse. En particular, se modeló la relación de generalización, se definió con precisión la diferencia entre casos de uso y escenarios y se presentó la forma en que se editan los modelos.

2.2. Análisis comparativo

En la Tabla 2.1 se muestra la comparativa de los trabajos relacionados con este proyecto, dicha tabla incluye el problema al que se enfrentaban los autores del artículo, las tecnologías que utilizaron, a aportación del trabajo y por último el estado de la aportación, es decir .

Tabla 2.1: Análisis comparativo de trabajos relacionados.

Artículo	**Problema**	**Tecnología**	**Aportación**	**Estado**
Nanthaamornphong et al. [19]	Uno de los desafíos a los que los desarrolladores de Java se enfrentan es la incapacidad para determinar el número correcto de las dependencias de clase.	▪ Java ▪ Reflexión ▪ Javassist	Herramienta Bytecode-CDET	Finalizado
Cloutier et al. [20]	Es muy difícil rastrear las dependencias entre elementos de una aplicación WEB cuando la documentación es inexistente u obsoleta.	▪ JavaScript	Herramienta WAVI	Mejoras a futuro
Maras et al. [21]	Las aplicaciones se desarrollan de una manera no disciplinada, lo cual repercute en el mantenimiento.	▪ Java ▪ UML	Herramienta complemento para el IDE de Eclipse.	Finalizado
Jain y Tayal [22]	Ningún investigador trató todas las construcciones disponibles en los diagramas de clase UML.	▪ C++ ▪ UML	Algoritmo para realizar diagramas UML a partir de código orientado a objetos.	Finalizado

Tabla 2.2: Análisis comparativo de trabajos relacionados (continuación).

Artículo	**Problema**	**Tecnología**	**Aportación**	**Estado**
Gueheneuc [23]	Los mantenedores de software necesitan herramientas para recuperar las opciones de diseño desde el código basado en sus patrones de origen.	▪ C++ ▪ Java	Herramienta Ptidej	Finalizado
Von Detten et al. [24]	Comprender el software es una tarea tediosa y propensa a errores.	▪ Java ▪ Matlab	Un complemento para el IDE Eclipse	Mejoras a futuro
Varoy et al. [25]	Entender el software de otro desarrollador es una tarea difícil para los desarrolladores.	▪ Java ▪ Metodología Scrum ▪ UML	Extensión de la herramienta UMLet	Mejoras a futuro
Sneed y Verhoef [26]	IBM VisualAge, es una herramienta ya no soportada, sin embargo, la aplicación que utiliza esta herramienta sigue activa.	▪ Java ▪ HTML ▪ JavaScript	Documentación de VisualAge	Mejoras a futuro

Tabla 2.3: Análisis comparativo de trabajos relacionados (continuación).

Artículo	**Problema**	**Tecnología**	**Aportación**	**Estado**
Nanthaamornphong et al. [27]	Uno de los desafíos de los desarrolladores de Fortran OO es la incapacidad de obtener descripciones de software de alto nivel de las aplicaciones.	▪ Fortran ▪ ArgoUML	Herramienta ForUML	Finalizado
Decker et al. [28]	La mayoría de las herramientas automáticas de ingeniería inversa funcionan mal y no revelan los procesos internos para producir el diagrama UML.	▪ SAX (Simple API for XML, Simple API para XML) ▪ C++	Herramienta srcYUML	Mejoras a futuro
Miao et al. [29]	La demanda para diferentes herramientas de análisis asistente crece significativamente, incluyendo las herramientas de análisis estático y dinámico.	▪ QEMU	Herramienta ROPTool	Mejoras a futuro
Potočnik et al. [30]	Es necesario verificar posibles errores, código ineficiente y problemas de estilo de codificación.	▪ Scala	Herramienta Linter	Finalizado

Tabla 2.4: Análisis comparativo de trabajos relacionados (continuación).

Artículo	**Problema**	**Tecnología**	**Aportación**	**Estado**
Repond et al. [31]	Se necesitaba editar los casos de uso y los escenarios	▪ Java ▪ UML	Herramienta complemento para Eclipse	Finalizado

En los primeros cuatro artículos se obtuvieron dependencias, con la diferencia de que en el trabajo de Nanthaa-mornphong et al. [19] son dependencias de clase en archivos Java, en Cloutier et al. [20] y en Maras et al. [21] se habla de dependencias entre archivos propios de la programación WEB y en Jain y Tayal [22] se obtienen dependencias, además de otros elementos propios del paradigma orientado a objetos. En los siguientes seis artículos: Gueheneuc [23], Von Detten et al. [24], Varoy et al. [25], Sneed y Verhoef [26], Nanthaa-mornphong et al. [27], Decker et al. [28] el resultado fue la recuperación del diseño a través de diagramas de clase. En Miao et al. [29] se habla de una herramienta de análisis dinámico mientras que en Potočnik et al. [30] se propuso una herramienta en Scala, la cual realiza un análisis estático. En el artículo [31] se desarrolló una metodología para ingeniería inversa, la cual abarca casos de uso y escenarios propios de UML.

La herramienta que se propone en esta tesis, desde ahora *ScalaReflect*, cuenta con:

- Un enfoque arquitectónico basado en el modelo 4+1.
- Una representación objeto funcional, basada en una notación para UML.
- El manejo del lenguaje Scala (ya se reportó una herramienta para Scala, pero fue hecha con el fin de verificar errores en el código).
- El uso de la *API Reflection* de Scala para realizar la ingeniería inversa.

2.3. Solución propuesta

Tomando en cuenta el enfoque de programación funcional que tiene el tema de tesis, las ventajas que ofrecen los entornos de desarrollo y las características de las metodologías de desarrollo, la solución más adecuada fue

- Scala como lenguaje de programación.
- IntelliJ con entorno de desarrollo integrado
- Scrum como metodología de desarrollo

2.3.1. Justificación de la solución seleccionada

Se eligió el lenguaje de programación Scala, debido a que el tema de tesis trata la parte funcional y este lenguaje integra de manera más natural la parte funcional.

Dentro de los ambientes de desarrollo para Scala se eligió IntelliJ debido a que, para efectos de este proyecto, la generación de diagramas de clase es importante, dado que sirve como comparativa entre el diagrama que genera la herramienta y el actual en el IDE.

En el contexto de las metodologías ninguna encaja en su totalidad con los requerimientos de desarrollo de la herramienta, en este proyecto el desarrollo lo realizó una sola persona es por esto que se decidió realizar una adaptación de las características y ventajas de la metodología Scrum. La adaptación consistió en lo siguiente:

- El desarrollo lo realizó una sola persona sin perder de vista las iteraciones y la pila de trabajo, características muy importantes de esta metodología.
- En las iteraciones se consideró la planificación, ejecución, revisión y retrospectiva de las mismas y en las pilas de producto se enlistarán los requisitos y funcionalidades de la herramienta.

De las herramientas de modelado, se optó por utilizar Enterprise Architec. Se tomó esta decisión debido a que desde la herramienta de modelado se importará un documento XMI generado desde *ScalaReflect*. Enterprise Architect maneja en el XMI sólo los elementos propios

para UML, lo cual hizo que la información recuperada desde reflexión pasara transparentemente al XMI.

Capítulo 3

Aplicación de la metodología

En este capítulo se describen las actividades que se realizaron para el desarrollo de la tesis.

3.1. Flujo de trabajo de la metodología de desarrollo

El flujo de trabajo que representa el desarrollo de la herramienta se formó de la siguiente manera (Figura 3.1):

1. Leer archivo .class.

2. Obtener información relevante, como las invocaciones a métodos a través del uso de la API Reflection de Scala, con el fin de saber cuáles son los componentes y establecer conexiones entre ellos.

3. Convertir esta información a formato XMI tomando en cuenta el Modelo 4+1 vistas, y la notación funcional propuesta en [4]. Se trabajó con el Modelo 4+1 ya que éste especifica las vistas mínimas para realizar una representación arquitectónica.

4. Importar el documento XMI en la herramienta de modelado UML Enterprise Architect y obtener la arquitectura del sistema.

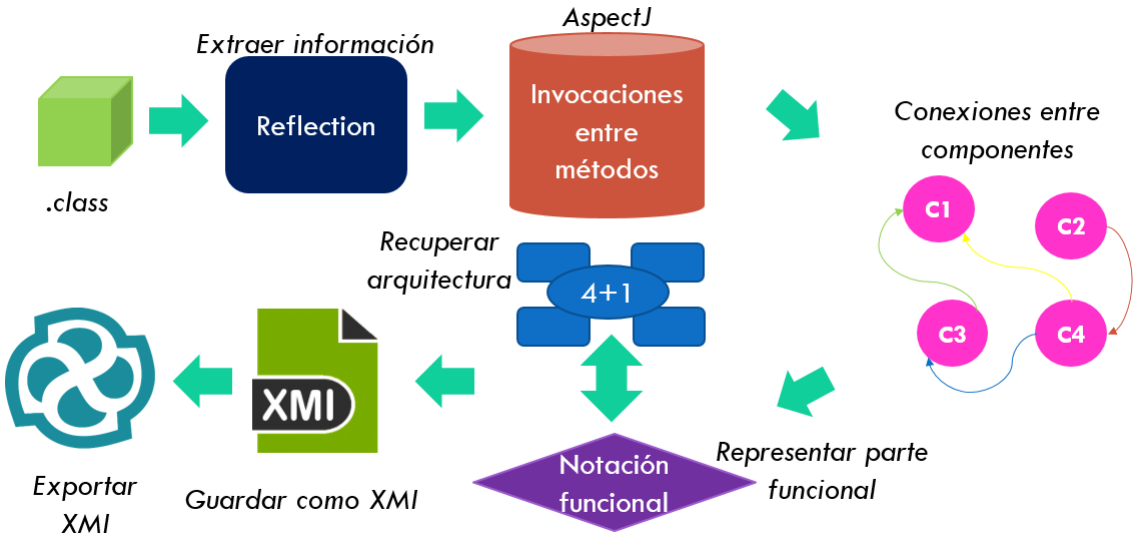

Figura 3.1: Flujo de trabajo de la metodología de desarrollo

3.2. Reflexión: Scala vs Java

El compilador de Scala traduce propiedades específicas de Scala (es decir, características que no pertenecen a Java) en algún equivalente de *bytecode* de Java para ejecutar en la JVM. Es por esto que el compilador de Scala crea clases sintéticas(es decir, clases creadas automáticamente) que se utilizan en tiempo de ejecución en lugar de las clases definidas por el usuario. Como consecuencia el uso de reflexión de Java en las clases de Scala arroja resultados incorrectos. A diferencia de Java, en el que se obtiene instancias de clase a tiempo de ejecución, en Scala se obtienen tipos a tiempo de ejecución. Los tipos de tiempo de ejecución de Scala llevan toda la información de tipo desde el tiempo de compilación, evitando desajustes entre el tiempo de compilación y el tiempo de ejecución. Un tipo a tiempo de ejecución es un metadato sobre una estructura del lenguaje (variable u objeto) que define el tipo de dato que se guarda en ella. Esta definición especifica de forma implícita el tipo de operaciones que se realizan sobre los datos.

3.2.1. API Reflection de Java

La reflexión es una característica del lenguaje de programación Java. Permite que un programa Java en ejecución examine o introspeccione sobre sí mismo y manipule las propiedades internas del programa. Por ejemplo, es posible que una clase Java obtenga los nombres de todos sus miembros y los muestre. La capacidad de examinar y manipular una clase Java desde sí misma no parece de gran importancia, pero en otros lenguajes de programación esta característica simplemente no existe. Por ejemplo, no hay forma en un programa Pascal, C o C ++ para obtener información sobre las funciones definidas dentro de ese programa. En la Tabla 3.1 se muestran las clases principales para obtener la información reflexiva.

El paquete `java.lang.reflect` proporciona clases e interfaces para obtener información reflexiva sobre clases y objetos.

Tabla 3.1: Package java.lang.reflect [1]

AccessibleObject	La clase AccessibleObject es la clase base para los objetos campos, métodos y constructores.
Array	La clase Array proporciona métodos estáticos para crear y acceder dinámicamente a las matrices de Java.
Constructor<T>	Proporciona información y acceso a un único constructor para una clase.
Executable	Una superclase compartida para la funcionalidad común de métodos y constructores.

Tabla 3.2: Package java.lang.reflect [1](Continuación)

Field	Proporciona información y acceso dinámico a un solo campo de una clase o una interfaz.
Method	Proporciona información y acceso a un único método en una clase o interfaz.
Modifier	La clase Modifier proporciona métodos estéticos y constantes para descodificar los modificadores de acceso de clases y miembros.
Parameter	Información sobre los parámetros del método.
Proxy	Proxy proporciona métodos estáticos para crear clases e instancias de proxy dinámico, y también es la superclase de todas las clases de proxy dinámico creadas por esos métodos.
ReflectPermission	La clase Permission para operaciones reflexivas.

3.2.2. API Reflection de Scala

En Scala toda la información disponible sobre la declaración de una entidad la contienen los símbolos. Un símbolo proporciona una gran cantidad de información que va desde el método básico `name` disponible en todos los símbolos, a otros más complicados, tal como conseguir el `baseClasses` de `ClassSymbol`, es decir la jerarquía de clases del símbolo. Otros casos de uso común de símbolos incluyen inspeccionar las firmas de los miembros, obtener los parámetros de tipo de una clase, obtener el tipo de parámetro de un método o averiguar

el tipo de campo (Tabla 3.2).

Tabla 3.3: Package scala.reflect.runtime.universe.Symbols [2]

ClassSymbol	Representa las definiciones de clase y *trait*.
MethodSymbol	Representa las declaraciones `def`.
ModuleSymbol	Representa declaraciones de objetos.
Symbol	Representa las declaraciones.
TermSymbol	Representa las declaraciones `val`, `var`, `def` y objetos, así como paquetes y valores de parámetros.
TypeSymbol	Representan las declaraciones de tipo, clase y *trait*, así como el tipo de los parámetros.

3.3. Vistas y diagramas recuperables por ingeniería inversa

Para encontrar las vistas y diagramas recuperables por ingeniería inversa, se realizó un análisis tomando en cuenta tres trabajos importantes de arquitectura de software, en los cuales los autores de cada escrito proponen distintos elementos de arquitectura. El análisis consistió en establecer una correspondencia entre estos elementos para finalmente compararlos con las vistas del modelo 4+1. Dicha correspondencia se representa en la Tabla 3.3.

Tabla 3.4: Correspondencia entre elementos de arquitectura

Cervantes Maceda et. al. [32]	**Clements et.al.** [33]	**Perry y Wolf** [34]	**Modelo 4+1 vistas** [14]
Dinámicos	Módulos	Procesamiento	Lógico
Lógicos	Componentes y conectores	Datos Conexión	Proceso
Físicos	Asignación		Desarrollo Físico

La ingeniería inversa es capaz de recuperar objetos, clases, módulos, componentes y conectores, los cuales pertenecen a los elementos dinámicos y lógicos según [32], módulos, componentes y conectores según [33] así como la vista lógica y de proceso según el modelo 4+1 [14]. Para efectos del tema de tesis las vistas que se cubren se especifican en el apartado 3.2.2.

3.3.1. Vistas

Debido al alcance que tiene la ingeniería inversa las vistas recuperables por este medio son:

- Vista lógica: Esta vista es factible de recuperar por ingeniería inversa debido a que específica cómo se asignan los requisitos funcionales a las clases y cómo se realizan sus interrelaciones. Además esta vista ayuda a los encargados del mantenimiento ya que usan estos modelos para entender el sistema con el fin de cambiar las funciones existentes, eliminar o agregar funciones.

- Vista de proceso: Los encargados del mantenimiento utilizan los modelos de proceso junto con los modelos lógicos para entender el sistema y las intenciones de los diseñadores originales con el fin de hacer modificaciones sin comprometer la integridad del sistema, es por esto que esta vista también es factible de recuperar por ingeniería inversa.

- Vista de desarrollo: Se ocupa de la gestión de la configuración del software y de los requerimeintos no funcionales tales como la capacidad de construcción, la capacidad de mantenimiento, la reutilización y la gestión de la configuración de las versiones del sistema. La vista de desarrollo a aborda la partición de la funcionalidad a través de subsistemas en apoyo del desarrollo.

Las demás vistas no son recuperables ya que la ingeniería inversa no recupera el mapeo del software en hardware y su distribución, tal es el caso de la vista física. Asimismo, la vista de escenarios no es recuperable ya que el alcance de la ingeniería inversa no permite recuperar casos de uso de la aplicación debido a que por medio de la codificación del sistema no es posible conocer la interacción con el usuario.

3.3.2. Diagramas

- Los modelos para una vista lógica son diagramas de clase o de entidad relación se recomienda el estilo arquitectónico orientado a objetos para esta vista, debido a su extensión en la representación de capacidades funcionales y requisitos de información. Estos diagramas de clase se crean utilizando el análisis orientado a objetos.

- La vista de proceso está compuesta de diagramas de clases y diagramas de colaboración que se centran en los objetos activos que representan los subprocesos y los procesos de un sistema. Los diagramas de colaboración se complementan con diagramas de actividad y estado que representan el objeto como una máquina de estados finitos.

- La vista de desarrollo está compuesta de diagramas de paquetes que representan la organización estática del software con respecto al entorno de desarrollo. Los componentes de una vista de desarrollo son módulos o subsistemas que componen el sistema.

Dado el alcance de reflexión para realizar la ingeniería inversa, de los diagramas con los que se representan estas vistas se trabajó con el diagrama de clases y el diagrama de paquetes, ya que el mecanismo de reflexión no da un soporte completo para conocer los procesos del

sistema, además que muchas de las llamadas entre objetos no necesariamente corresponden a la arquitectura. Sin embargo, se cuenta con el apoyo de AspectJ, el cual permite obtener información dinámica del sistema; dado el alcance del tema de tesis, se presentó una aportación que consistió en la realización de pruebas para comprobar que el uso de aspectos fue factible esta aportación se muestra en el apartado 3.6, sin embargo, la construcción de los diagramas correspondientes a los procesos del sistema se tomaron en cuenta como trabajo futuro.

3.4. Notación objeto funcional

En este trabajo de tesis se hizo uso de la notación objeto funcional [4] la cual propone una forma de modelar elementos de la programación funcional tales como: funciones de orden superior, funciones currificadas, tipos definidos por el usuario, evaluación perezosa, lambdas y mónadas.

3.4.1. Funciones de orden superior

Las funciones de orden superior son aquellas que son capaces de recibir una o más funciones como argumentos, o bien retornar una función como valor de retorno; esto recae en la necesidad de representar este tipo de argumento no común en UML: una operación (método o función). Para el Enfoque Funcional, todas las funciones poseen una firma, comúnmente representada a través del símbolo ->que separa cada argumento y el último indica el valor de retorno. En la figura 3.2 se muestra un ejemplo, donde `operation1` recibe una función `f` que tiene como parámetros un valor de tipo `int` y devuelve otro valor de tipo `int`; mientras que `operation2` recibe un parámetro de tipo `boolean` y devolverá una función `f` cuyos parámetros son: el primero de tipo `String` y el segundo de tipo `int`, devolviendo un tipo `double`.

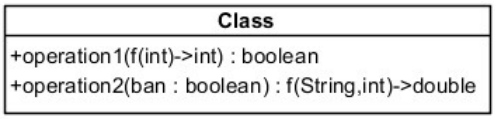

Figura 3.2: Ejemplo de notación para funciones de orden superior.

3.4.2. Tipos definidos por el usuario

En los lenguajes Orientados a Objetos, cada clase representa un tipo, por ello la representación de tipos no requiere más que la representación de una clase. Sin embargo, en lenguajes como Scala, existen tipos (clases) especialmente definidos para servir como tipos de datos, este es el caso de la implementación de clases *case*. Para dar solución a la representación de estas clases especiales se propone la utilización del estereotipo *«type»*, en la figura 3.3 se muestra un ejemplo del uso de este estereotipo.

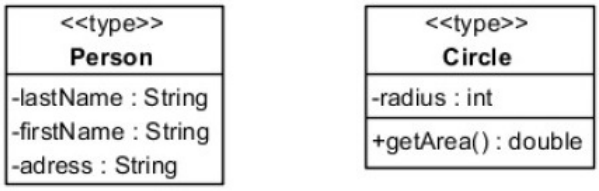

Figura 3.3: Ejemplo de notación para tipos definidos por el usuario.

Se elige esta notación, puesto que brinda una indicativo de que la clase sólo servirá como la definición de un tipo de dato, como es el caso de clases destinadas a ser registros en base de datos, además de que el uso de este estereotipo ya se encuentra difundido para este propósito. Aunque la palabra case es una forma de implementación en el lenguaje Scala, esto no define que sea igual para otros lenguajes, por ello no es factible usar esta palabra para definir su uso en la implementación.

3.4.3. Funciones currificadas

Una función se denomina currificada, cuando recibe sus parámetros uno a la vez; en los lenguajes OO esto no ocurre de manera común, solo algunos lenguajes soportan esta propiedad. En Scala es posible definir una función currificada de forma variable, es decir, cuántos y cuáles parámetros recibirá en cada llamada parcial, por ello es necesario especificar la distribución de los argumentos y el orden de las posibles llamadas parciales; la notación propuesta para esta propiedad es definir la separación de argumentos con paréntesis, de manera que se separe cada llamada parcial, como se muestra en la figura 3.4.

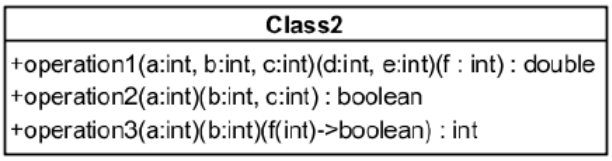

Figura 3.4: Ejemplo de notación para funciones currificadas.

A pesar de que las llamadas parciales retornan funciones, no es necesario especificar esto, puesto que se sobre entiende que al aplicar la currificación con llamadas parciales, la función devuelta será la misma pero con los argumentos faltantes para completar la llamada a dicha operación.

3.4.4. Evaluación perezosa

La evaluación perezosa permite que valores se evalúen únicamente cuando son necesarios, esta estrategia permite la manipulación de estructuras de datos infinitas, denominadas flujos (streams).

3.4.5. Expresiones lambda

Las expresiones lambda son funciones anónimas que se implementan cuando son necesarias, estas se utilizan comúnmente cuando se envían como parámetros a otras funciones.

Definir el uso de funciones anónimas en un modelo UML implica describir el comportamiento interno de un método, sin embargo, ese nivel de descripción no es común.

3.4.6. Mónadas

Las mónadas son un mecanismo de la programación funcional que permite la introducción de declaraciones imperativas y proporciona una manera de abstraer sobre diferentes tipos de cálculos. Un cálculo se considera como una función que típicamente producirá un valor, cada cálculo se caracteriza por una estructura específica de los parámetros y valores de retorno, al definir un constructor de tipos, se define el tipo de este cálculo [4].

Dada la información que proporciona la reflexión, en este trabajo se hará uso de la notación para funciones de orden superior y tipos definidos por el usuario.

Para obtener una representación de evaluación perezosa, lambdas y mónadas, es necesario conocer los enlaces que hay entre funciones, así como la estructura interna de la función, en otras palabras, obtener los elementos dinámicos del sistema. Dado el alcance del tema de tesis, en el apartado 3.6 se presenta un aporte que explica cómo obtener las conexiones entre funciones.

3.5. Mecanismo de reflexión en Scala

En Scala 2.10, se introdujo una nueva biblioteca de reflexión, no sólo para hacer frente a las deficiencias de ejecución de reflexión en Java, en los tipos específicos y genéricos de Scala, sino, para añadir un conjunto de herramientas con capacidades reflexivas generales.

3.5.1. Ambiente

El entorno de reflexión varía en función de si la tarea de reflexión es en tiempo de ejecución o en tiempo de compilación. La distinción entre un entorno que se utilizará en tiempo de ejecución o de tiempo de compilación se encapsula en un llamado universo. Otro aspecto importante del ambiente de reflexión es el conjunto de entidades con el que se tiene acceso

reflectante. Este conjunto de entidades se denomina espejo. Los espejos no sólo determinan el conjunto de entidades a las que se acceden reflexivamente, también proporcionan operaciones reflectantes para realizar en esas entidades.

3.5.2. Universos (*Universes*)

Un universo es el entorno en el que se realizará en proceso de reflexión. Hay dos tipos principales de universos, ya que existen capacidades de reflexión tanto en tiempo de ejecución como en tiempo de compilación, hay que utilizar el universo que corresponde a lo que la tarea se acerca. Ya sea:

`scala.reflect.runtime.universe`: Tiempo de ejecución

`scala.reflect.macros.Universe`: Tiempo de compilación

Un universo proporciona una interfaz para todos los conceptos principales utilizados en la reflexión, como *Types Trees* y *Annotations*.

Tipos *(Types)*

Como su nombre sugiere, las instancias de *Type* representan de información sobre el tipo de un símbolo correspondiente. Esto incluye sus miembros (métodos, campos, alias de tipo, tipos abstractos, clases anidadas, *traits*, etc.) declarados directamente o heredados, sus tipos base, su borrado, etc. Los tipos también proporcionan operaciones para probar la conformidad del tipo o la equivalencia.

Árboles *(Trees)*

Los árboles son la base de la sintaxis abstracta de Scala que se utiliza para representar programas. También se les llama árboles de sintaxis abstracta y comúnmente abreviados como AST. Es importante tener en cuenta que los árboles son inmutables a excepción de tres campos: *pos(Position), symbol(Symbol)* y *tpe(Type)*, que se asignan cuando un árbol se comprueba por el tipo.

Anotaciones *(Annotations)*

En Scala, las declaraciones se anotan utilizando subtipos de `scala.annotation.Annotation`. Además, dado que Scala se integra con el sistema de anotación de Java, es posible trabajar con anotaciones producidas por un compilador Java estándar.

La API distingue dos tipos de anotaciones:

- Anotaciones de Java: anotaciones sobre definiciones producidas por el compilador de Java, es decir, subtipos de `java.lang.annotation.Annotation`. Cuando se lee mediante la reflexión de Scala, el *trait* `scala.annotation.ClassfileAnnotation` se agrega automáticamente como una subclase a cada anotación de Java.
- Anotaciones de Scala: anotaciones sobre definiciones o tipos producidos por el compilador de Scala.

3.5.3. Espejos (*Mirrors*)

Toda la información proporcionada por la reflexión se hace accesible a través de los espejos. Dependiendo del tipo de información que se obtiene, o la acción reflectante que se desea realizar, se manejan diferentes tipos de espejos. Los espejos cargadores de clases se utilizan para obtener representaciones de tipos y miembros. Estos espejos traducen los nombres de símbolos. Desde un espejo cargador de clases, es posible obtener espejos para la invocación más especializados, que implementan las invocaciones reflectantes, como método o constructor, llamadas y acceso a campos.

3.6. Recuperación de arquitectura con Reflection

Dentro de la información de un *.class* se necesita conocer datos generales como su nombre, si se trata de una clase, objeto, *trait* o de una *case class* y sus modificadores de acceso (*public, private* etc.) además de su jerarquía de clases, sus declaraciones, constructores, anotaciones, métodos y campos. Para obtener esta información se construyeron diferentes clases

las cuales consisten en extraer la información a partir de un `String`, este *String* corresponde al archivo `.class`. Para extraer la información es necesario obtener un símbolo para la clase a partir de un espejo cargador de clases. Al trabajar con Reflexión se hacen las siguientes dos importaciones:

```
import scala.reflect.runtime.universe =>ru
import scala.reflect.runtime.universe._
```

El espejo cargador de clases se obtiene con la función `runtimeMirror` a partir del universo en tiempo de ejecución, por ello en la primera importación se le asigna a `ru` el universo, para así tener acceso a él:

```
val m = ru.runtimeMirror(getClass.getClassLoader)
```

- `getClass` devuelve la clase en tiempo de ejecución de un objeto.
- `getClassLoader` devuelve el cargador de clases para la clase.

Como lo que se recibe es un `String` se necesita obtener la clase de ese tipo, `Class.forName` es la función que permite realizar esto:

```
val cfn = Class.forName(rs)// El parámetro rs es el String que recibe
```

Ya que se tiene el cargador de clases y la clase, lo que prosigue es obtener su símbolo, lo cual se realiza con la función `classSymbol()`:

```
val c = m.classSymbol(cfn)
```

La información de la entidad que contiene el *.class* se extrae directamente del símbolo, en el caso de la demás información (jerarquía de clases, declaraciones, constructores, anotaciones, métodos y campos) la extracción se realiza a partir de un tipo (*Type*), lo cual se consigue de la siguiente manera:

```
val tc = c.toType// toType obtiene el tipo del símbolo especificado
```

3.6.1. Informacion de entidad

La información de la entidad se extrae desde la clase `InformationEntities` mostrada en el listado 3.1 recibe en su constructor un `String`, ésta se encarga de obtener el nombre del *.class*, los modificadores de acceso y de saber si se trata de una clase, *trait*, objeto o *case class*.

En las líneas de la 5 y 6 se muestra la obtención del símbolo de la clase. En la línea 7 se declaran las variables `s1` y `s2`, las cuales se utilizan más adelante para devolver el tipo `String` necesario. Para obtener el nombre de un símbolo en este caso el *.class* se utiliza la función name como se observa en la línea 8. Además se hace uso de la función `toString` para convertir name que es de tipo `TypeName` a `String` y así imprimirlo.

- `name` devuelve el nombre del símbolo

En la línea 9 se encuentra el método `getModifier()` el cual se encarga de obtener el modificador de acceso que tiene el *.class*.

- `isAbstract` devuelve *true* si el símbolo es de visibilidad abstracta.
- `isPrivate` devuelve *true* si el símbolo es de visibilidad privada.
- `isPublic` devuelve *true* si el símbolo es de visibilidad pública.
- `isProtected` devuelve *true* si el símbolo es de visibilidad protegida.

Seguido de esto se encuentra el método `getEntity()` que consta de un bloque `if-else` el cual se encarga de saber si el *.class* pertenece a una clase, *trait*, objeto o *case class* (Líneas 15-26).

- `isClass` devuelve *true* si el símbolo es *class*.
- `isTrait` devuelve *true* si el símbolo es *trait*.
- `isModule` devuelve *true* si el símbolo es *object*.

- `isCaseClass` devuelve *true* si el símbolo es *case class*.

Listado 3.1: Informacion de entidad.

```
1  package ScalaReflect
2  import scala.reflect.runtime.universe._
3  import scala.reflect.runtime.{universe => ru}
4  class InformationEntities(s: String) {
5    val cfn = Class.forName(s)
6    val c = ru.runtimeMirror(getClass.getClassLoader).classSymbol(cfn)
7    var s1,s2 = ""
8    val str = getModifier(c)+ "␣" + getEntity(c)+ "␣" + c.name.toString + "\n"
9    def getModifier(c: ClassSymbol): String ={
10     if (c.isAbstract) s1 = "abstract"
11     else if (c.isFinal) s1 = "final"
12     else if (c.isPublic) s1 = "public"
13     else if (c.isProtected) s1 = "protected"
14     else if (c.isStatic) s1 = "static"
15     else if (c.isPrivate) s1 = "private"
16     s1
17   }
18   def getEntity(c: ClassSymbol): String ={
19     if(c.isTrait) s2 = "trait"
20     else if(c.isModuleClass) s2 = "object"
21     else if(c.isCaseClass) s2 = "case␣class"
22     else if(c.isClass) s2 ="class"
23     s2
24   }
25 }
```

3.6.2. Jerarquía de clases

La clase `Hierarchy` (Listado 3.2) recibe un `String` y se encarga de obtener la jerarquía de clases del *.class* a través del método `baseClasses`. En las líneas 4 y 5 se obtiene el tipo

del *.class*, recordando que este es necesario para la extracción de información. Para obtener el nombre del tipo se utiliza la función `typeSymbol` como se muestra en la línea 6, en la línea 7 se hace uso de la función `baseClasses` la cual devuelve la lista de todas las clases base de este tipo (incluyendo su propio `typeSymbol`), empezando por la propia clase y terminando en la clase `Any`. Ya que `baseClasses` devuelve una lista con las clases heredadas separadas por comas, se aplica la función `mkSring()` con un salto de línea, para que las clases se visualicen de mejor manera. En la línea 9 se imprime la información extraída.

Listado 3.2: Jerarquía de clases.

```
1  package ScalaReflect
2  import scala.reflect.runtime.{universe => ru}
3  class Hierarchy(s: String) {
4    val cfn = Class.forName(s)
5    val tc = ru.runtimeMirror(getClass.getClassLoader).classSymbol(cfn).toType
6    val nc = tc.typeSymbol.name
7    val bc = tc.baseClasses
8    val mk = bc.mkString("\n")
9    val str = ""+ mk
10   print(mk)
11 }
```

3.6.3. Declaraciones

La clase `Declarations` muestra los miembros declarados en el *.class*. (Listado 3.3) La función `decls` devuelve un `Scope` que contiene directamente a los miembros que se declararon en el tipo dado, no devuelve los miembros que se le heredaron. La función `Scope` devuelve estos miembros con una sintaxis de declaración, `sorted` muestra el contenido del `Scope` en forma de lista y de manera ordenada, se añade la función `mkString()` para que la lista no se muestre en una sola línea si no que cada miembro se separe por un salto de línea (Línea 7).

Listado 3.3: Declaraciones.

```
1  package ScalaReflect
```

```
2 import scala.reflect.runtime.{universe => ru}
3 class Declarations(s: String) {
4   val cfn = Class.forName(s)
5   val t = ru.runtimeMirror(getClass.getClassLoader).classSymbol(cfn).toType
6   val name = t.typeSymbol.name
7   val decl = t.decls.sorted.mkString("\n")
8   println(name + ":\n" + decl)
9 }
```

3.6.4. Constructores

En el Listado 3.4 se ilustra la clase `Constructors`, que se encarga de obtener la información de los constructores.

- `collect` construye una nueva colección aplicando una función parcial a todos los elementos de esta colección iterable en el que se define la función (Línea 9).
- `MethodSymbol` devuelve el tipo de símbolos que representan un método, es decir, declaraciones `def` (Línea 10).
- `isConstructor` devuelve true si el símbolo es un constructor (Línea 11).
- `paramLists` devuelve todas las listas de parámetros del método (Línea 11).
- `map` construye una nueva colección mediante la aplicación de una función para todos los elementos de esta lista (Línea 11).

Listado 3.4: Constructores.

```
1 package ScalaReflect
2 import scala.reflect.runtime.universe._
3 import scala.reflect.runtime.{universe => ru}
4 class Constructors() {
5   def constInfo(rs: String): Iterable[_] = {
6     val cfn = Class.forName(rs)
```

```
7       val c = ru.runtimeMirror(getClass.getClassLoader).classSymbol(cfn).toType
8       var str = ""
9       c.decls.collect {
10        case m: MethodSymbol
11          if m.isConstructor => m.paramLists.map(_.map(_.name))
12          "| Name: ________| " + m.name + "\n| Info: ________| " + m.info + "\n"
13      }
14    }
15    def aCadenaC(s:String): String = {
16      val n = constInfo(s)
17      n.mkString
18    }
19  }
```

3.6.5. Anotaciones

En la clase `Annotations` (Listado 3.5) se obtiene la información referente a las anotaciones del .class.

- `asClass` se utiliza para convertir el símbolo en una clase (Línea 6).
- `annotations` devuelve las anotaciones del símbolo (Línea 9).

Listado 3.5: Anotaciones.

```
1  package ScalaReflect
2  import scala.reflect.runtime.{universe => ru}
3  class Annotations(s: String){
4    val cfn = Class.forName(s)
5    val m = ru.runtimeMirror(getClass.getClassLoader).classSymbol(cfn).toType
6    val a = m.typeSymbol.asClass
7    val name = a.name
8    var str = ""
9    val an = a.annotations
```

```
10    if(an.length ==0){
11     str += name + "␣Empty"
12    }else
13       str += "Annotations␣of"+ name +":␣"+an.mkString
14    println(str)
15  }
```

3.6.6. Campos

En la clase `Fields` se adquiere toda la información sobre los campos (Listado 3.6). Se obtiene el nombre del tipo de símbolo del *.class*, a continuación en la línea 9 se filtran los miembros declarados que no sean métodos, lo que da lugar al valor `fields`. El bloque `if-else` tiene como función saber si hay o no campos, lo que se logra con `isEmpty`, en el ciclo for se obtiene la firma del campo con el método `getFirm()`, sus propiedades con el método `getProperties()` y su modificador de acceso con el método `getModifier()`.

- `filter` selecciona los elementos que no cumplen con la condición (Línea 9).
- `isMethod` devuelve *true* si el símbolo es un método (Línea 9).
- `isEmpty` devuelve *true* si la colección *iterable* no contiene elementos (Línea 10).
- `typeSignature` devuelve la firma de este tipo de símbolo (Línea 21).
- `asTerm` Convierte el símbolo en un `TermSymbol` (Línea 25).
- `isVal` devuelve *true* si el símbolo es un valor (Línea 26).
- `isVar` devuelve *true* si el símbolo es una variable (Línea 27).
- `isFinal` devuelve *true* si el símbolo es *Final* (Línea 33).
- `isProtected` devuelve *true* si el símbolo es de visibilidad protegida (Línea 35).
- isStatic devuelve *true* si el símbolo es estático (Línea 36).

Listado 3.6: Campos.

```
1  package ScalaReflect
2  import scala.reflect.runtime.universe._
3  import scala.reflect.runtime.{universe => ru}
4  class Fields(s: String) {
5    val cfn = Class.forName(s)
6    val m = ru.runtimeMirror(getClass.getClassLoader).classSymbol(cfn).toType
7    val ncm = m.typeSymbol.name
8    var str = ""
9    val fields = m.decls.filter(mm => !mm.isMethod)
10   if(fields.isEmpty){
11     str = ""+ ncm +"Empty"
12   }else {
13     for (f <- m.decls.filter(mm => !mm.isMethod)) {
14       str += getFirm(f) + "\n" + getProperties(f) + "\n" + getModifier(f) + "\
           n" +"————————————————————————————————————" + "\n"
15       println(str)
16       println("————————————————————————————————————")
17     }
18   }
19
20   def getFirm(f: Symbol): String ={
21     "| Name:       | " + f.name + "\n" + "| Type:       | " + f.typeSignature
22   }
23   def getProperties(f: Symbol): String ={
24     var s = ""
25     val af = f.asTerm
26     if (af.isVal) s = "val"
27     else if (af.isVar) s = "var"
28     "| Properties: | Is " + s
29   }
30   def getModifier(m: Symbol): String ={
31     var s:String = ""
32     if (m.isAbstract) s = "Abstract"
```

```
33      else if (m.isFinal) s = "Final"
34      else if (m.isPublic) s = "Public"
35      else if (m.isProtected) s = "Protected"
36      else if (m.isStatic) s = "Static"
37      else if (m.isPrivate) s = "Private"
38      "| Modifier: ___| " + s
39    }
40  }
```

3.6.7. Métodos

La clase `Methods` Listado 3.7, se encarga de obtener la información relacionada con los métodos. Su estructura es similar a la de `Fields`, cambia en que filtra aquellos miembros que sean métodos pero no constructores. Obtiene la firma del método con `getFirm()` y el modoficador de acceso con `getModifier()`.

- `resultType` devuelve el tipo del símbolo.

Listado 3.7: Métodos.

```
1  package ScalaReflect
2  import scala.reflect.runtime.universe._
3  import scala.reflect.runtime.{universe => ru}
4  class Methods(s: String) {
5    val cfn = Class.forName(s)
6    val m = ru.runtimeMirror(getClass.getClassLoader).classSymbol(cfn).toType
7    val ncm = m.typeSymbol.name
8    var str = ""
9    val methods = m.decls.filter(m => m.isMethod && !m.isConstructor)
10   if(methods.isEmpty){
11     str =""+ ncm +"Empty"
12   }else
13     for(m <- m.decls.filter(m => m.isMethod && !m.isConstructor)){
14       var s = ""
```

```
15        val as = m.asTerm
16        if (as.isVal || as.isVar){
17          s = ""
18        }else{
19          s = getModifier(m)
20          str += ""+ getFirm(m)+"\n"+ s + "\n" +"
                ————————————————————————————————————" + "\n"
21          println(str)
22          println("————————————————————————————————————")
23        }
24      }
25
26    def getFirm(m: Symbol): String ={
27      "| Name:       | "+ m.name +"\n"+ "| Type:       | "+ m.typeSignature.
            resultType +"\n"+"| Arguments:  | "+m.typeSignature
28    }
29    def getModifier(m: Symbol): String ={
30      var s:String = ""
31      if (m.isAbstract) s = "abstract"
32      else if (m.isFinal) s = "final"
33      else if (m.isPublic) s = "public"
34      else if (m.isProtected) s = "protected"
35      else if (m.isStatic) s = "static"
36      else if (m.isPrivate) s = "private"
37      "| Modifier:   | " + s
38    }
39  }
```

3.7. Invocaciones a métodos con *AspectJ*

Para la construcción de la arquitectura de un sistema es necesario considerar tanto el comportamiento del sistema durante su ejecución como el mapeo de los elementos en tiempo de desarrollo y ejecución hacia elementos físicos. Por lo anterior, es necesario representar

elementos que hacen referencia a:

- Entidades dadas en el tiempo de ejecución, es decir, dinámicas, como objetos e hilos.
- Entidades que se presentan en el tiempo de desarrollo, es decir, lógicas, como clases y módulos.
- Entidades del mundo real, es decir, físicas, como nodos o carpetas.

Todos estos elementos se relacionan entre sí mediante interfaces u otras propiedades, y al hacerlo dan lugar a distintas estructuras. Es por ello que cuando se habla de la arquitectura de un sistema no se piensa en solo una estructura, sino se considera una combinación de estas, ya sean dinámicas, lógicas o físicas El mecanismo de reflexión permite recuperar las entidades lógicas; lo cual representa el alcance de este tema de tesis, sin embargo, se encontró que AspectJ da soporte para recuperar las entidades dinámicas, por ello se realizaron pruebas para conocer información dinámica a través de llamadas a métodos y funciones.

Estas llamadas se obtuvieron mediante el uso de *AspectJ* y la primitiva `call`. Scala es compatible con *AspectJ* mediante el uso de anotaciones, el uso de este lenguaje se presenta en el Listado 3.8.

- `call` primitiva que aplica el corte en la llamada de un método o un constructor.
- `before` aviso que se aplica antes del punto de unión seleccionado.
- `JoinPoint.StaticPart` contiene la información estática sobre un punto de unión. Está disponible desde el método `JoinPoint.getStaticPart()`, y se accede por separado usando `thisJoinPointStaticPart`.
- `JoinPoint.EnclosingStaticPart` Devuelve un objeto que encapsula las partes estáticas de este punto de unión.
- `getSignature()` Devuelve la ubicación de origen correspondiente al punto de unión.

- `getDeclaringType()` Devuelve un objeto `java.lang.Class` que representa la clase, interfaz o aspecto que declaró este miembro.
- `getName()` Devuelve la parte del identificador de esta firma.

La manera de recuperar las conexiones entre funciones, es conociendo la función que llamadora y la función llamada, así como la clase a la que pertenecen estas funciones (llamadora y llamada). Para esto se utilizó `JoinPoint.StaticPart` en la parte llamadora (Línea 8) y `EnclosingStaticPart` en la función llamada (Línea 10).

Listado 3.8: Aspecto aplicado para recuperar llamadas entre funciones.

```
1  package aspectos
2  import org.aspectj.lang.JoinPoint
3  import org.aspectj.lang.annotation.{Aspect, Before}
4  @Aspect
5  class MethodLogger {
6    @Before("call(* bClass.*.*(..))")
7    def logMeethod2(joinPointStaticPart: JoinPoint.StaticPart,
         joinPointEnclosingStaticPart: JoinPoint.EnclosingStaticPart ) = {
8      println("Llamando a "+joinPointStaticPart.getSignature.getName + " de " +
           joinPointStaticPart.getSignature.getDeclaringType)
9      println("***************" + joinPointStaticPart + "***************")
10     println("Desde " + joinPointEnclosingStaticPart.getSignature)
11   }
12 }
```

Con este mecanismo se recuperaron todas las llamadas entre funciones de clases, objetos y traits. La información obtenida tiene la siguiente apariencia

```
Llamando a function2ClassB de class bClass.ClassB ***************call(void
bClass.ClassB.function2ClassB(Function1))*************** Desde void
bClass.ClassA..main(String[])
```

En este ejemplo se aprecia la llamada a una función de la clase `ClassB` desde el `main` de la clase `ClassA`

3.8. XMI

Para representar la arquitectura recuperada con reflexión se hizo uso de un esquema XMI el cual fue elaborado para Enterprise Architect, esto con el fin de dar al usuario la comodidad de visualizar el diagrama en una herramienta de modelado.

Como se ve en el Listado 3.9 el documento XMI está conformado por los elementos *Model* y *Extension*. En *Model* se declaran los elementos tales como paquetes, clases, atributos y métodos de esa clase, así como las relaciones entre clases. En *Extension* se le asignan propiedades a los elementos declarados en *Model*, las etiquetas que destacan son <elements>(Línea 18),<connectors>(Línea 45) y <diagrams>(Línea 47).

Listado 3.9: Ejemplo de XMI.

```
1  <?xml version="1.0" encoding="windows-1252"?>
2  <xmi:XMI xmi:version="2.1" xmlns:uml="http://schema.omg.org/spec/UML/2.1"
       xmlns:xmi="http://schema.omg.org/spec/XMI/2.1">
3    <xmi:Documentation exporter="Enterprise Architect" exporterVersion="6.5"/>
4    <uml:Model xmi:type="uml:Model" name="EA_Model" visibility="public">
5      <packagedElement xmi:type="uml:Package" xmi:id="P968494" name="Package01"
           visibility="public">
6        <packagedElement xmi:type="uml:Package" xmi:id="P2038366" name="xmiEA"
             visibility="public">
7          <packagedElement xmi:type="uml:Class" xmi:id="C397834" name="ClassEA"
               visibility="P968494">
8            <ownedAttribute xmi:type="uml:Property" xmi:id="F3997333" name="str
                 " visibility="Private" association="">
9              <type xmi:idref=""/>
10           </ownedAttribute>
11           <ownedOperation xmi:id="M6520383" name="m" visibility="public">
12           </ownedOperation>
13         </packagedElement>
14       </packagedElement>
15     </packagedElement>
16   </uml:Model>
```

```
17  <xmi:Extension extender="Enterprise␣Architect" extenderID="6.5">
18    <elements>
19      <element xmi:idref="P968494" xmi:type="uml:Package" name="Package01"
            scope="public">
20        <properties sType="Package" nType="0" scope="public"/>
21        <extendedProperties package_name="Package01"/>
22      </element>
23      <element xmi:idref="P2038366" xmi:type="uml:Package" name="xmiEA" scope=
            "public">
24        <properties sType="Package" nType="0" scope="public"/>
25        <extendedProperties package_name="Package01"/>
26      </element>
27      <element xmi:idref="C397834" xmi:type="uml:Class" name="ClassEA" scope="
            public">
28        <model package="P2038366" ea_eleType="element"/>
29        <properties sType="Class" nType="0" scope="public" stereotype="type"
              isAbstract="false"/>
30        <extendedProperties package_name="xmiEA"/>
31        <attributes>
32          <attribute xmi:idref="F3997333" name="str␣" scope="Private">
33            <properties type="String"/>
34          </attribute>
35        </attributes>
36        <operations>
37          <operation xmi:idref="M6520383" name="m" scope="public">
38            <type type="Unit" const="" static="false" isAbstract="false"/>
39            <parameters>
40            </parameters>
41          </operation>
42        </operations>
43      </element>
44    </elements>
45    <connectors>
46    </connectors>
```

```
47      <diagrams>
48        <diagram xmi:id="D7933018">
49          <model package="P968494" localID="6" owner="P968494"/>
50          <properties name="clases" type="Logical"/>
51          <elements>
52            <element geometry="Left=1;Top=5;Right=100;Bottom=100;" subject="
                  C397834" seqno="1" style="DUID=FE805086;"/>
53          </elements>
54        </diagram>
55        <diagram xmi:id="D7124155">
56          <model package="P968494" localID="6" owner="P968494"/>
57          <properties name="paquetes" type="Package"/>
58          <elements>
59            <element geometry="Left=1;Top=5;Right=100;Bottom=100;" subject="
                  P2038366" seqno="1" style="DUID=FE805086;"/>
60          </elements>
61        </diagram>
62      </diagrams>
63    </xmi:Extension>
64  </xmi:XMI>
```

Para alinear el documento XMI al modelo 4+1 y representar correctamente la parte funcional se utilizó el atributo *stereotype* el cual representa un *trait*, una clase case (*type*) o un objeto (*object*). En el caso de las funciones el atributo *type* es el que ayudó a realizar la correcta representación de las mismas, esto fue posible dada la capacidad que mostró la reflexión al manejar dichas funciones. En el Listado 3.10 se muestra la clase `BuildPackageElementClass` la cual construye el elemento *Class* con sus atributos, operaciones y los parametros de dichas operaciones.

Listado 3.10: Clase BuildPackageElementClass.

```
1  package xmi
2
3  import ArraysXMI.{Args, Methodss}
4  import scala.collection.mutable.ArrayBuffer
```

```
5
6  class BuildPackageElementClass {
7    def buildPackageElementClass(id: String, name: String, visibility: String,
         oA: ArrayBuffer[Array[String]], oO: ArrayBuffer[Methodss]): String ={
8      var operations = ""
9      var attributes = ""
10     val packageElementClass = "\n\t\t\t\t" +
11       "<packagedElement xmi:type=\"uml:Class\" xmi:" +
12       "id=\""+id+"\" " +
13       "name=\""+name+"\" " +
14       "visibility=\""+visibility+"\">"
15     val packageElementClassClose = "\n\t\t\t\t</packagedElement>"
16     for(i <-0 to oO.length-1){
17       operations += buildOwnedOperation(oO(i).methodd(0),oO(i).methodd(1),oO(i
             ).methodd(2),oO(i).argss)
18     }
19     for(i<-0 to oA.length-1){
20       attributes += buildOwnedAttribute(oA(i)(0),oA(i)(1),oA(i)(2),oA(i)(4),oA
             (i)(5))
21     }
22     packageElementClass + attributes + operations + packageElementClassClose
23   }
24   def buildOwnedAttribute(id: String, name: String, visibility: String, idref:
         String, association:String): String ={
25     val ownedAttribute = "\n\t\t\t\t\t<ownedAttribute xmi:type=\"uml:Property
         \" " +
26       "xmi:id=\""+id+"\" " +
27       "name=\""+name+"\" " +
28       "visibility=\""+visibility+"\" " +
29       "association=\""+association+"\">" +
30       "\n\t\t\t\t\t\t<type xmi:idref=\""+idref+"\"/>"
31     val ownedAttributeClose = "\t\t\t\t\n\t\t\t\t\t</ownedAttribute>"
32     ownedAttribute + ownedAttributeClose
33   }
```

```
34   def buildOwnedOperation(id: String, name: String, visibility: String, oP:
       ArrayBuffer[Args]): String ={
35     var parameters = ""
36     val ownedOperation = "\n\t\t\t\t\t" +
37       "<ownedOperation xmi:" +
38       "id=\""+id+"\" " +
39       "name=\""+name+"\" " +
40       "visibility=\""+visibility+"\">"
41     val ownedOperationClose = "\n\t\t\t\t</ownedOperation>"
42     println("oP "+ oP.length)
43     for(i<-0 to oP.length-1){
44       parameters += buildOwnedParameter(oP(i).args(0),oP(i).args(1),oP(i).args
           (2),oP(i).args(3))
45     }
46     ownedOperation + parameters + ownedOperationClose
47   }
48   def buildOwnedParameter(id:String, name: String, direction: String, tipe:
       String): String ={
49     val ownedParameter = "\n\t\t\t\t\t<ownedParameter xmi:" +
50       "id=\""+id+"\" " +
51       "name=\""+name+"\" " +
52       "direction=\""+direction+"\" " +
53       "type=\""+tipe+"\"/>"
54     ownedParameter
55   }
56 }
```

Con base en el desarrollo de los pasos metodológicos descritos en el capítulo se logró dar cumplimiento al objetivo planteado en la presente tesis. También, con el soporte del lenguaje AspectJ fue posible identificar los elementos necesarios para soportar la recuperación de información dinámica del sistema, actividad que solo se reporta de forma parcial al estar fuera de los objetivos originales de la tesis.

Capítulo 4

Resultados

En este capítulo se mostrarán los resultados de la herramienta desarrollada, la funcionalidad y la aplicación del caso de estudio para mostrar las capacidades de la misma.

4.1. ScalaReflect

La herramienta de ingeniería inversa desarrollada en este proyecto lleva el nombre de ScalaReflect, dado el lenguaje de programación y el uso del mecanismo de reflexión para la ingeniería inversa.

4.1.1. Interfaz

La herramienta Figura 4.1 consta de tres menús: Archivo, Ingeniería Inversa y Ayuda y de un cuadro de texto donde se aprecia la información reflectada o el XMI según lo que se aplique. En el menú *Archivo* se despliegan las opciones *Abrir*, *Guardar como* y *Salir*; en el menú *Ingeniería inversa* se despliegan las opciones *Reflectar entidades* y *Exportar a XMI* finalmente en el menú *Ayuda* se despliegan la opciones *Ayuda* y *Acerca de*.

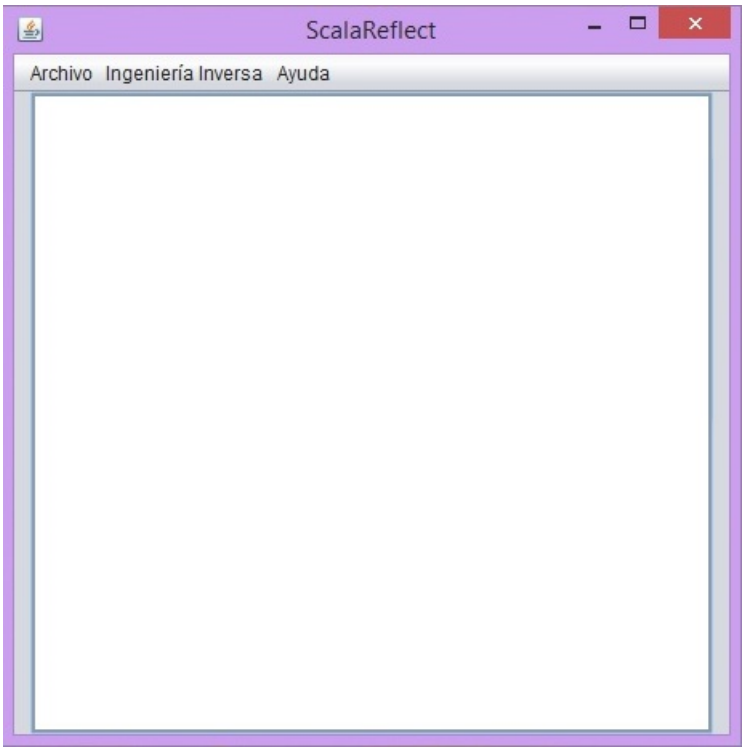

Figura 4.1: ScalaReflect: Interfaz de usuario

4.2. Proceso de ingeniería inversa

Para realizar el proceso de ingeniería inversa se selecciona el paquete donde están contenidas las entidades a analizar, para esto se va al menú Ingeniería Inversa - Reflectar entidades a continuación se abrirá un cuadro de diálogo donde ubicará la carpeta (paquete) a analizar Figura 4.2.

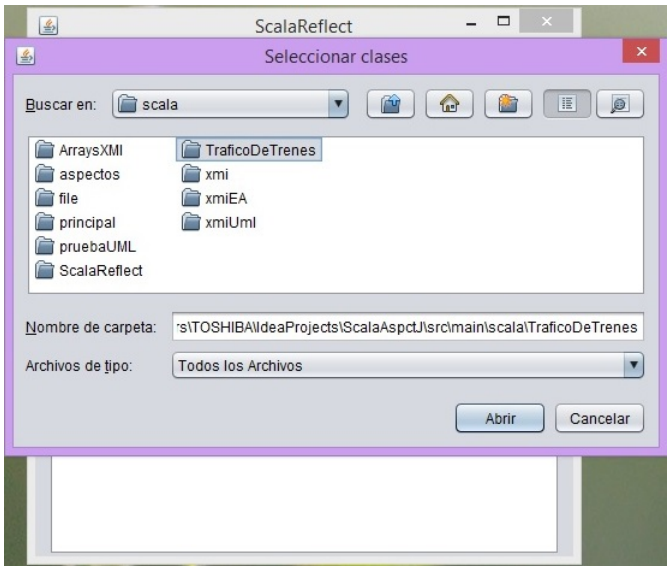

Figura 4.2: ScalaReflect: Seleccionar carpeta

La información recuperada se mostrará en el cuadro de texto, dicha información tendrá la siguiente apariencia:

```
-------Info--------
public case class Accion
-------Fields-------
     Name: desc
     Type: String
   Properties: Is val
   Modifier: Private
   ------------------
   ------Methods------
     Name: desc
     Type: String
   Arguments: =>String
    Modifier: public
------Constructors------
```

```
Name: <init>
Info: (desc: String, vel: Double)modelo.Accion
-------Hierarchy-------
class Accion
trait Serializable
trait Serializable
trait Product
trait Equals
class Object
class Any
```

Seguido de esto se exportará esta información a un documento XMI, para lo cual se va al menú Ingeniería Inversa - Exportar a XMI, se abrirá un cuadro de dialogo para elegir la ubicación en la cual se guardará el documento Figura 4.3, de igual manera el documento XMI se mostrará en el cuadro de texto.

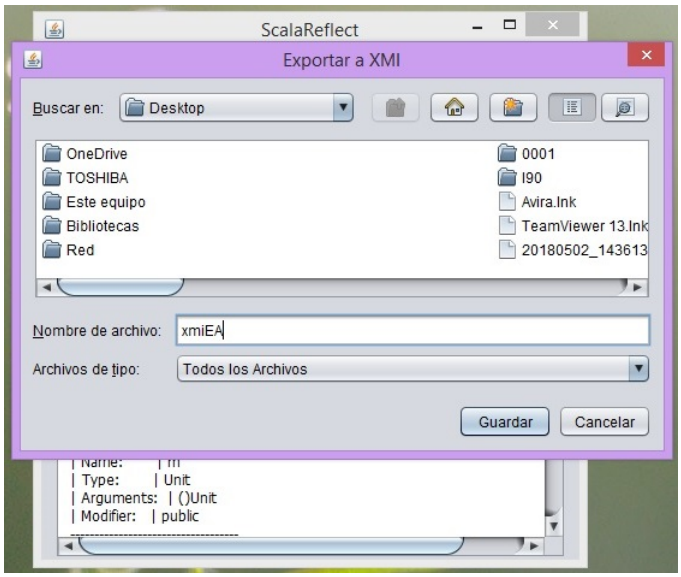

Figura 4.3: ScalaReflect: Exportar a XMI

Lo siguiente es importar el documento XMI desde la herramienta Enterprise Architect, para esto crea un proyecto nuevo e importa el XMI para poder visualizar las entidades

analizadas Figura 4.4.

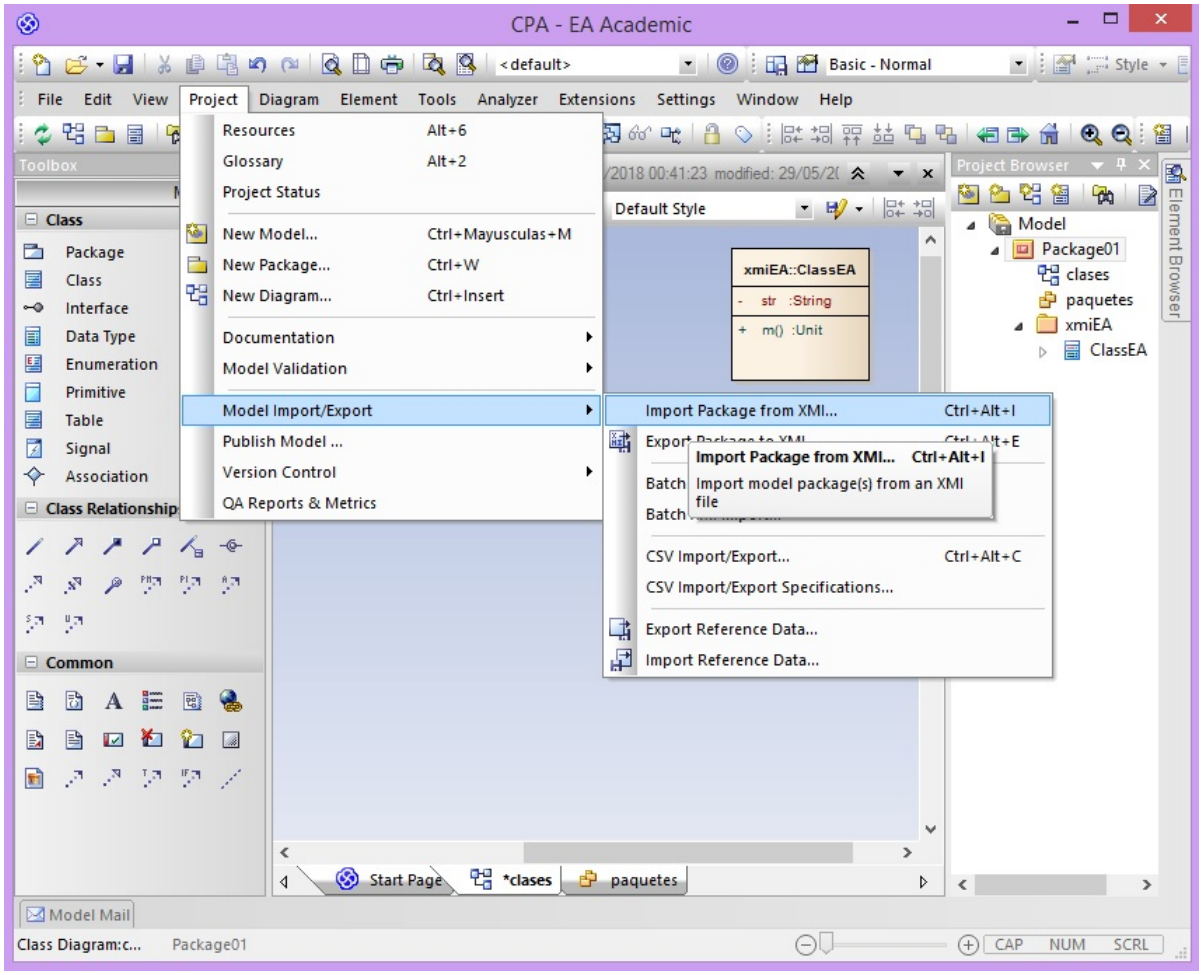

Figura 4.4: Enterprise Architect: Importar XMI

4.3. Representación objeto funcional

En este aparatado se muestra el resultado de la representación objeto funcional que realiza la herramienta. Esto se detalla desde tres perspectivas que se manejan en la herramienta:

- Reflexión: La información que se recupera desde este mecanismo.
- XMI: como se traduce la información reflectiva en este documento.

- Diagrama de clases: El resultado de la representación objeto funcional desde la herramienta Enterprise Architect.

4.3.1. Tipos definidos por el usuario

Para los tipos definidos por el usuario, es decir las clases case la información obtenida desde reflexión es la siguiente:

```
------Package xmiEA
-------Info--------
public case class ClaseCase
-------Fields-------
    | Name: | num
    | Type: | Int
 | Properties: | Is val
 | Modifier: | Private
 ------------------
```

Esta información en el XMI se estructura de la siguiente manera:

Listado 4.1: XMI para la clase case ClaseCase.

```
1  <element xmi:idref="C8909000" xmi:type="uml:Class" name="ClaseCase"
       scope="public">
2    <model package="P9069156" ea_eleType="element"/>
3    <properties sType="Class" nType="0" scope="public" stereotype="type"
         isAbstract="false"/>
4    <extendedProperties package_name="xmiEA"/>
5    <attributes>
6      <attribute xmi:idref="F8445014" name="num_" scope="Private">
7        <properties type="Int"/>
8      </attribute>
9    </attributes>
10 </element>
```

En la herramienta Enterprise Architect, esta clase case se representa con el estereotipo «type» como se muestra en la figura 4.5.

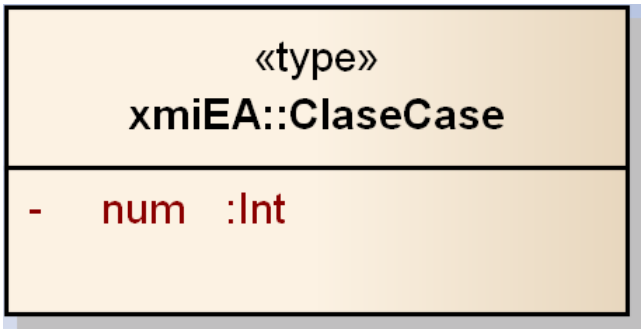

Figura 4.5: Clase ClaseCase

4.3.2. Funciones currificadas y funciones de orden superior

Para las funciones currificadas y las funciones de orden superior la información obtenida desde reflexión se enlista a continuación:

```
------------------
 | Name: | suma2
  | Type: | Int
| Arguments: | (x: Int)(y: Int)Int
| Modifier: | public
------------------
 | Name: | apply2
  | Type: | Int
| Arguments: | (f: Int =>Int, n: Int)Int
| Modifier: | public
------------------
```

Estos datos de las funciones en el XMI se estructuran de la siguiente manera:

Listado 4.2: XMI para funciones suma2 y apply2.

```
1                <ownedOperation xmi:id="M6171138" name="suma2" visibility=
                     "public">
2                    <ownedParameter xmi:id="P6640530" name="x" direction="
                         in" type="Int"/>
3                    <ownedParameter xmi:id="P6640530" name="y" direction="
                         in" type="Int"/>
4            </ownedOperation>
5            <ownedOperation xmi:id="M5734257" name="apply2" visibility="public">
6              <ownedParameter xmi:id="P1724207" name="f" direction="in" type="
                   Int => Int"/>
7              <ownedParameter xmi:id="P5666289" name="n" direction="in" type="
                   Int"/>
8            </ownedOperation>
```

En el diagrama de clases la representación de las funciones se muestra en la Figura 4.6.

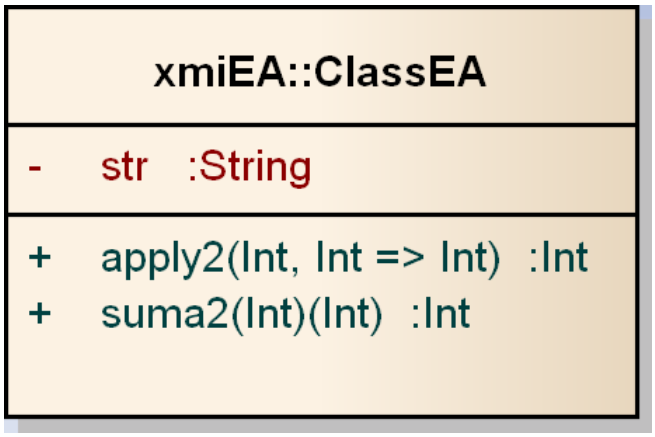

Figura 4.6: Funciones `suma2` y `apply2`

Desde las tres perspectivas que se manejaron la representación funcional fue exitosa. En reflexión de obtuvo la información de los parámetros que manejan las funciones currificadas y de orden superior, así como el tipo de entidad con la que se trata para el caso de los tipos definidos por el usuario. La información obtenida fue manejada de manera correcta para vaciarla en los atributos que estructuran el XMI; finamente se obtuvo un modelo correcto en

la herramienta Enterprise Architect. Se construyó un diagrama UML con elementos propios de este lenguaje para la representación objeto funcional en Scala.

4.4. Diagrama de paquetes

Un diagrama de paquetes en UML representa las dependencias entre los paquetes que componen un modelo. Es decir, muestra cómo un sistema está dividido en agrupaciones lógicas y las dependencias entre esas agrupaciones. En este tema de tesis fue factible construir un diagrama de paquetes gracias al soporte que con el que cuenta la herramienta para indagar entre capetas y buscar todos los archivos *.scala* o *.class* contenidos en ellas; y la forma en que reflexión extrae esa información. Para construir este diagrama se agrega al elemento `<diagrams>` el elemento `<diagram>` dentro del cual se especifican los paquetes a representar mediante la etiqueta `model`. Esta estructura se muestra a continuación.

Listado 4.3: XMI para distribución de paquetes.

```
1          <diagrams>
2        <diagram xmi:id="D3304780">
3          <model package="P7830068" localID="6" owner="P7830068"/>
4          <properties name="clases" type="Logical"/>
5          <elements>
6            <element geometry="Left=1;Top=5;Right=100;Bottom=100;" subject="
                 C2598591" seqno="1" style="DUID=FE805086;"/>
7            <element geometry="Left=2;Top=10;Right=100;Bottom=100;" subject="
                 C8127320" seqno="1" style="DUID=FE805086;"/>
8            <element geometry="Left=3;Top=15;Right=100;Bottom=100;" subject="
                 C7870891" seqno="1" style="DUID=FE805086;"/>
9            <element geometry="Left=4;Top=20;Right=100;Bottom=100;" subject="
                 C8572253" seqno="1" style="DUID=FE805086;"/>
10         </elements>
11       </diagram>
12         </diagrams>
```

En la figura 4.7 se muestra la representación de los paquetes en Enterprise Architect.

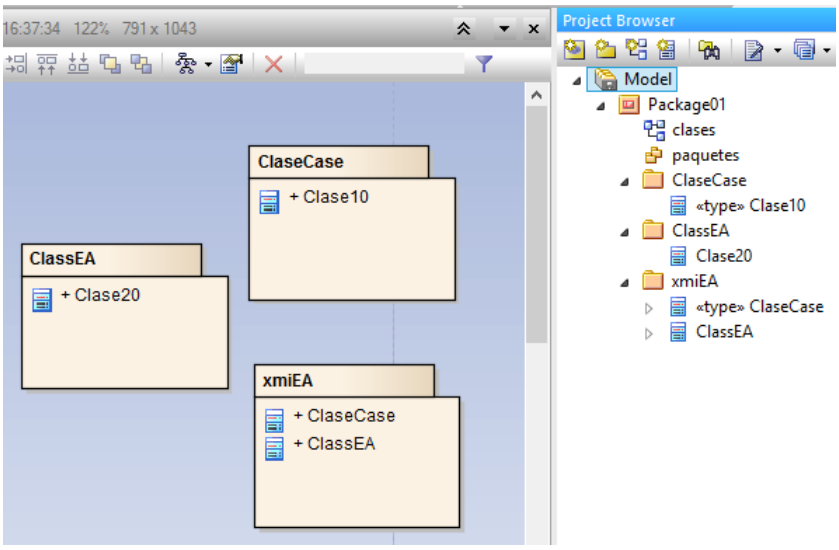

Figura 4.7: Diagrama de paquetes

4.5. Caso de estudio

El caso de estudio trata del Sistema de control: gestión de tráfico de trenes, este sistema fue construido en Scala, lo cual significa que contiene propiedades funcionales que servirán para para mostrar las capacidades de la herramienta. A continuación se presenta el diagrama de clases correspondiente al sistema Figura 4.8.

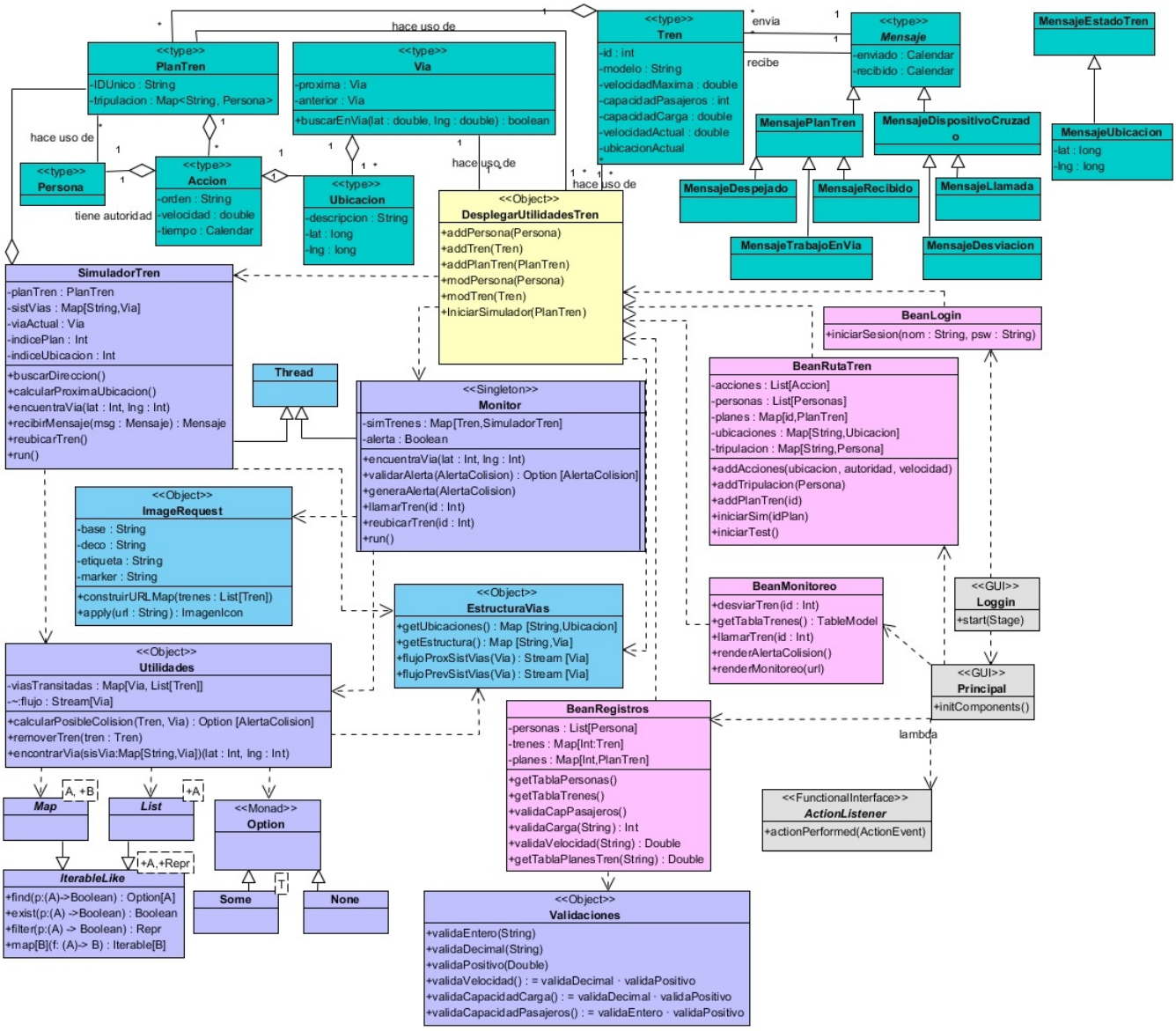

Figura 4.8: Diagrama de clases del Sistema de control

El sistema consta de seis paquetes:

- `bean`
- `datos`
- `gui`

- `modelo`

- `simuladorTrenes`

- `utilidades`

Estos paquetes se analizaron por separado para una mejor visualización de los diagramas. Los resultados de la ingeniería inversa aplicada a este sistema se muestran a continuación.

En la figura 4.9 se visualizan las clases del paquete `bean` las cuales son:

- `BeanRutaTren`

- `BeanLoggin`

- `BeanRegistros`

- `BeanMonitoreo`

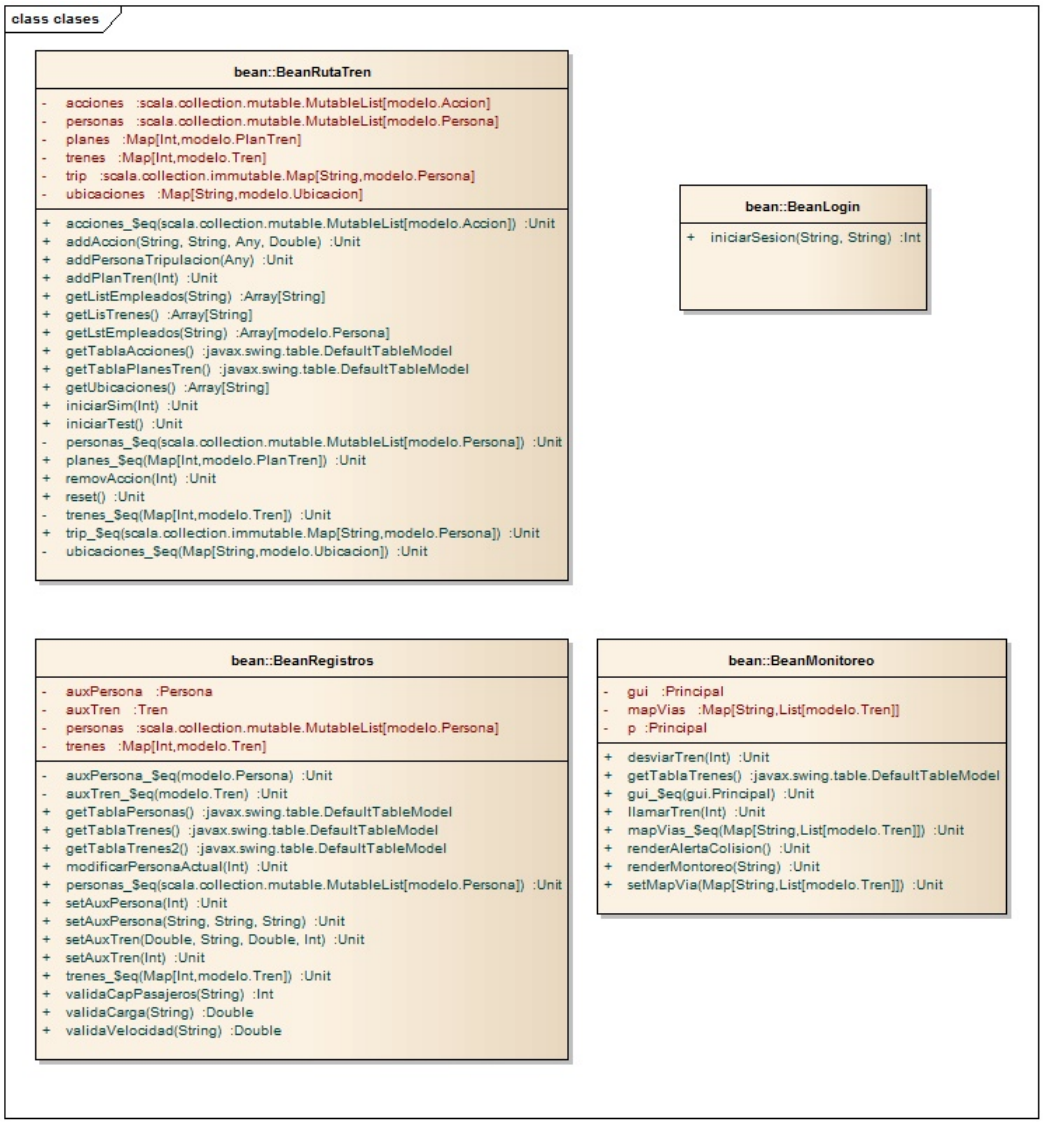

Figura 4.9: Diagrama de clases perteneciente al paquete `bean`

En la figura 4.10 se visualizan los objetos del paquete `datos` los cuales son:

- `EstructuraVias`
- `DesplegarUtilizadesTren`

Se observa el uso de estereotipos para especificar que se trata de un objeto.

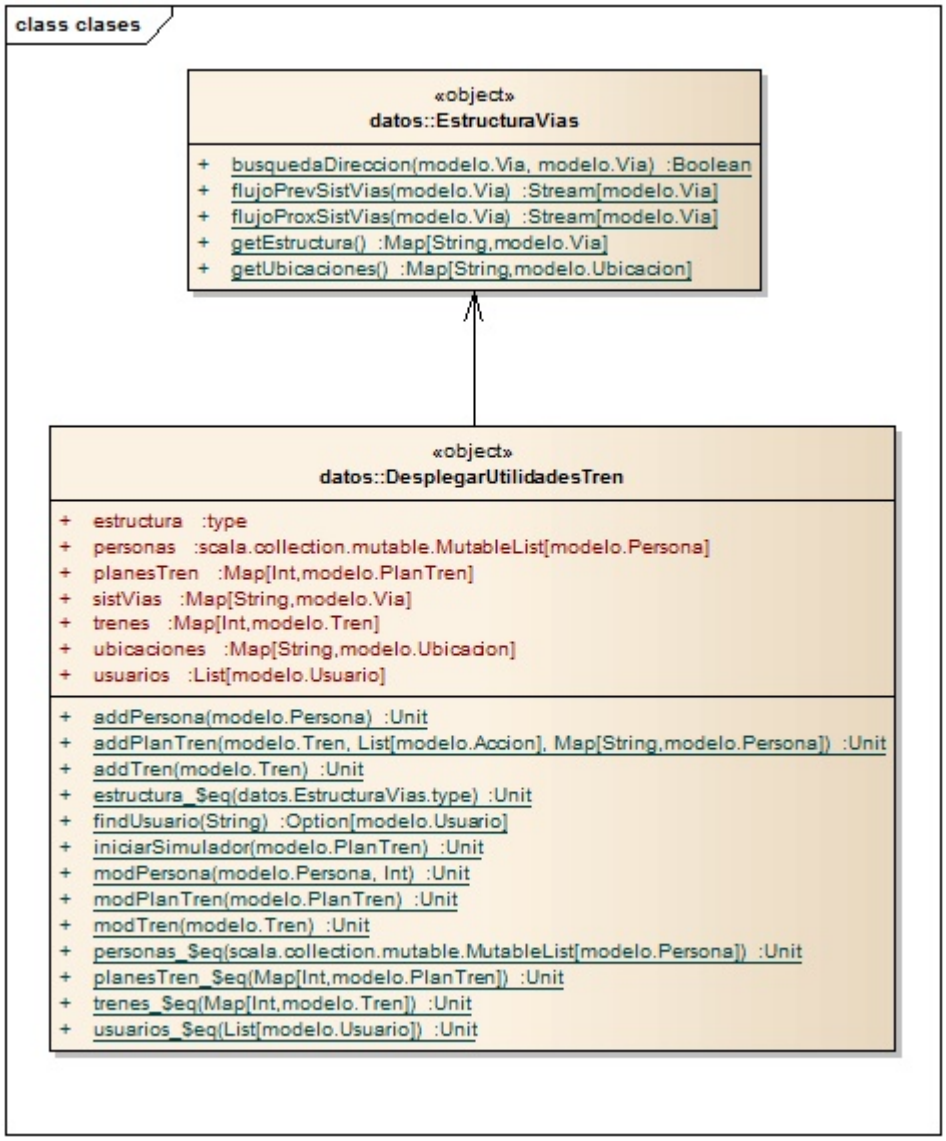

Figura 4.10: Diagrama de clases perteneciente al paquete `datos`

En la figura 4.11 y 4.12 se visualizan las clases del paquete `GUI` las cuales son:

- `Principal`
- `TextAreaRenderer2`
- `ScalaRunner`
- `Iniciar`

- `ButtonColumn`

- `Loggin`

Se observa que en la clase Iniciar se escuentra el método `main` de la aplicación, ademas en la clase `Principal` Figura 4.12 se observa un grupo de funciones *lambdas*.

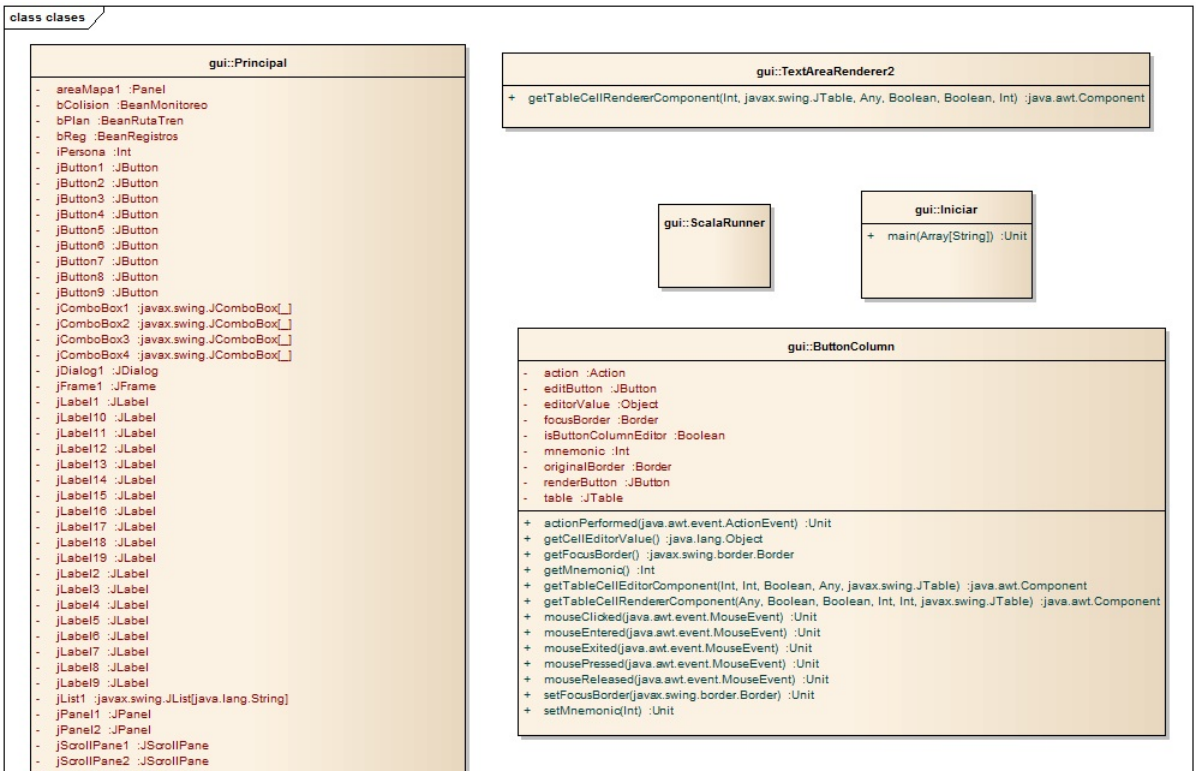

Figura 4.11: Diagrama de clases perteneciente al paquete `GUI`, parte 1

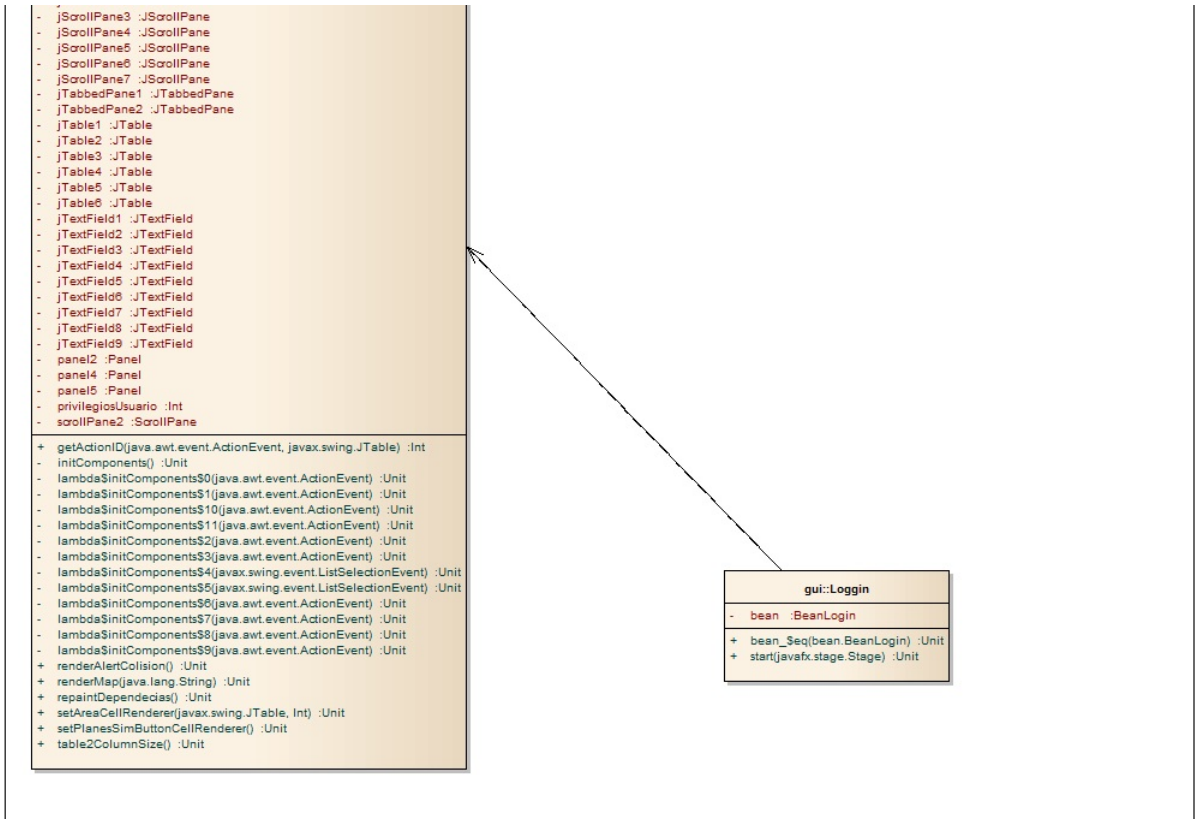

Figura 4.12: Diagrama de clases perteneciente al paquete `GUI`, parte 2

En la figura 4.13 y 4.14 se visualzan las clases y clases *case* del paquete `modelo`.

Clases

- MensajeDespejado
- MensajeDesviacion
- MensajeDispositivoCruzado
- MensajeEstadoTren
- MensajeLlamada
- MensajeParada
- MensajePlanTren
- MensajeRecibido
- MensajeReubicado
- MensajeTrabajoEnVia
- MensajeUbicación
- MensajeVelocidad

Clases *case*

- Accion
- Mensaje
- Persona
- PlanTren
- Tren
- Ubicacion
- Usuario
- Via

Se observa el uso del estereotipo «type» para especificar que se trata de una clase *case*.

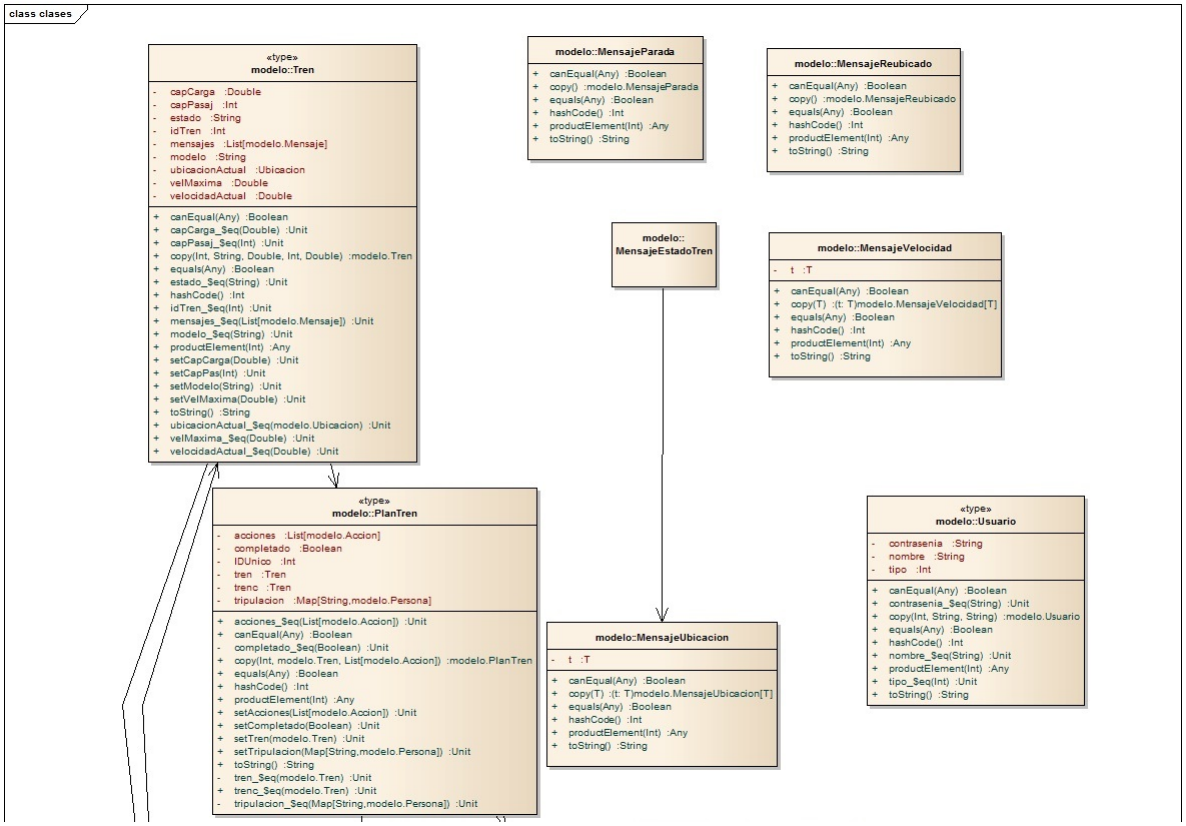

Figura 4.13: Diagrama de clases perteneciente al paquete modelo, parte 1

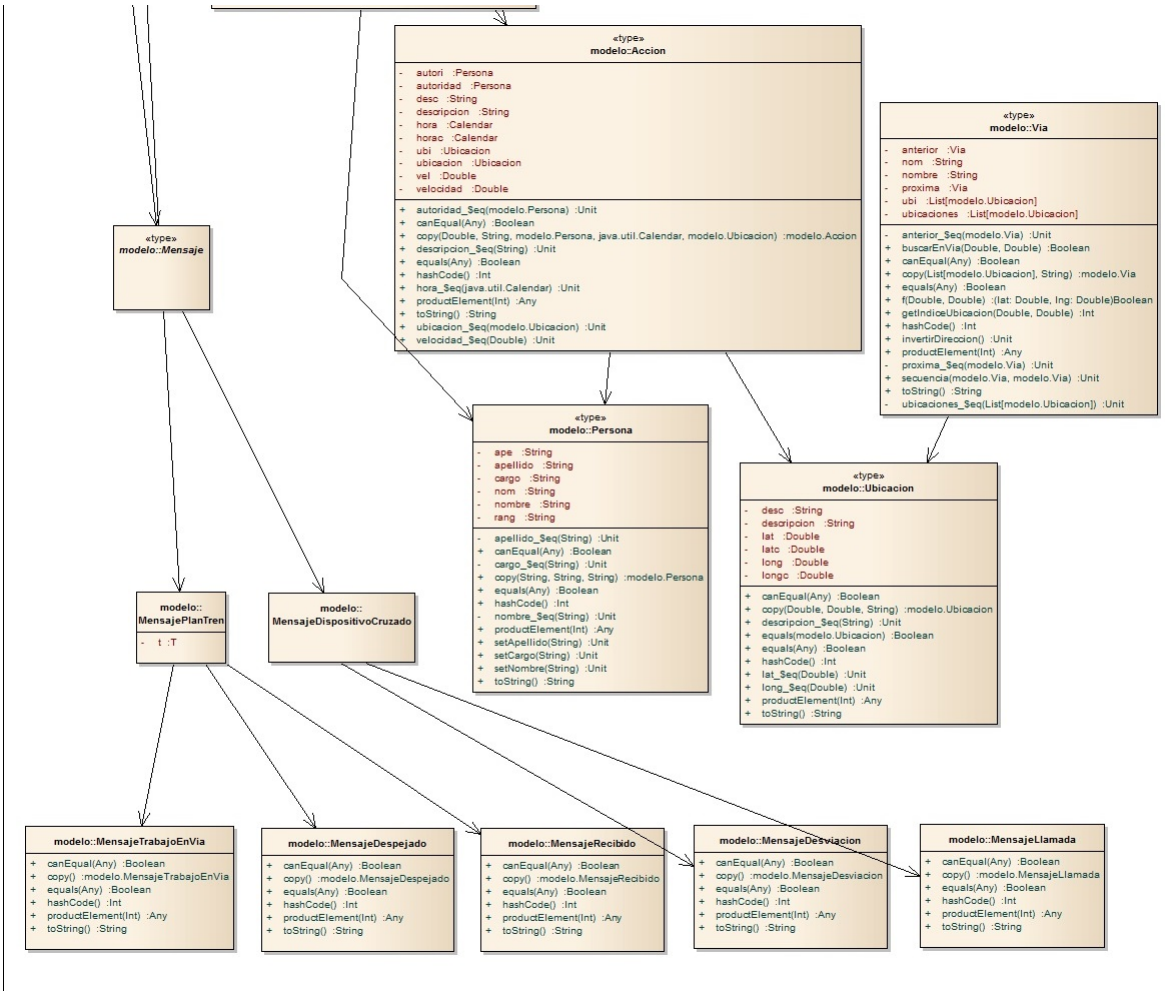

Figura 4.14: Diagrama de clases perteneciente al paquete modelo, parte 2

En la figura 4.15 se visualiza la clase `SimuladorTren` del paquete simuladorTrenes. En esta clase se observa la función `encuentraVia()`.

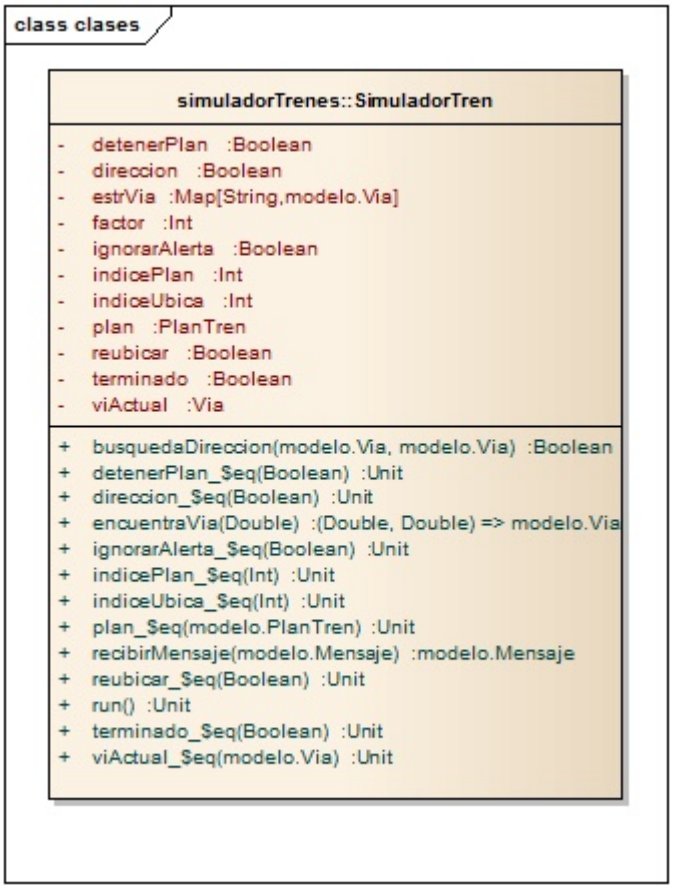

Figura 4.15: Diagrama de clases perteneciente al paquete simuladorTrenes

En la figura 4.16 se visualizan los objetos del paquete utilidades. En el objeto `Monitor` se observa la función `buscVia()` y en el objeto `Utilidades` la función `encontrarVia()`.

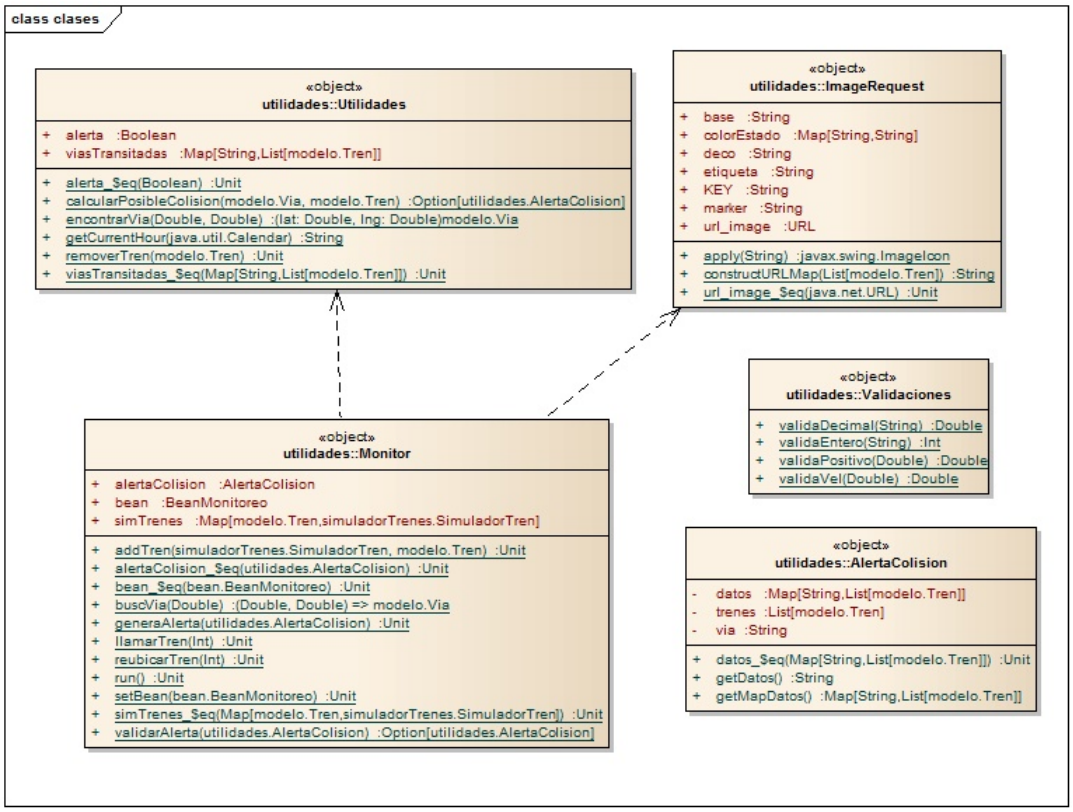

Figura 4.16: Diagrama de clases perteneciente al paquete utilidades

Como se vio en los resultados, la herramienta realiza una correcta representación de los elementos funcionales de Scala esto gracias al uso de estereotipos y el manejo de tipos que realiza la reflexión.

La herramienta da una correcta representación de la implementación real del sistema, ya que al comparar el diagrama original del Sistema de control se aprecia que hay muchos otros elementos por representar.

El potencial de *ScalaReflect* es de gran ayuda para los encargados del mantenimiento ya que ellos usan estos modelos para entender el sistema con el fin de cambiar las funciones existentes, eliminar funciones o agregar funciones.

4.6. Comparativa ScalaReflect vs Class Visualizer

La herramienta *Class Visualizer* es un generador de diagramas de clase interactivos a partir *bytecode* de Java [35]. Se eligió esta herramienta para compararla con *ScalaReflect*, ya que cuenta con las siguientes características: es gratuita, simple, rápida y fácil de usar. Se utilizó esta herramienta para encontrar ventajas y desventajas de *ScalaReflect*, y al mismo tiempo se tomaron en cuenta aspectos que se presentaron como trabajo a futuro.

Las características principales de *Class Visualizer* son:

- Diagrama de clase generado automáticamente
- Vista previa instantánea de la clase UML
- Navegador de elementos y relaciones de clase
- Lista de clases
- Jerarquía de clases
- Sin repositorio, sin problemas de sincronización
- Guarda contenido cargado como un proyecto

Mientras que las características principales de *ScalaReflect* son:

- Información de entidades recuperadas, en la que destaca:
    - Paquete al que pertenece la clase
    - Campos
    - Métodos
    - Constructores
    - Parámetros
    - Anotaciones

    - Funciones
    - Jerarquía de clases

- Representación objeto funcional
- Construcción de documento XMI, para guardar el diagrama resultante y para exportarlo.
- Diagrama de clases, donde se visualizan las entidades y sus relaciones
- Diagrama de paquetes donde se visualiza la forma en que se agrupan las clases.

A partir de estas características se realizó una comparativa entre las dos herramientas la cual se muestra en la tabla 4.1

Tabla 4.1: Comparativa entre ScalaReflect y Class Visualizer

Características	**ScalaReflect**	**Class Visualizer**
Vista previa clase UML	SI	SI
Diagrama de clases	SI	SI
Diagrama de paquetes	SI	NO
Jerarquía de clases	SI	SI
Representación funcional	SI	NO
Enfoque arquitectónico	SI	NO
Guardado de contenido	SI	SI
Lenguaje	Scala	Java
Navegador de elementos	SI	SI
Relaciones de clase	SI	SI
Lista de clases	SI	SI
Exportación a XMI	SI	NO

ScalaReflect cuenta con una vista previa desde el diagrama de clases, a diferencia de *Class Visualizer* se selecciona la clase para que de la vista previa de clase UML. *ScalaReflect* tiene la

capacidad de construir un diagrama de paquetes el cual permite navegar entre los elementos del diagrama. En cuanto a la jerarquía de clases *ScalaReflect* la obtiene mediante reflexión y se presenta en forma de texto, mientras que *Class Visualizer* la presenta gráficamente. *ScalaReflect* cuenta con un el soporte para representar la parte funcional del sistema mientras que la otra herramienta no cuenta con este soporte. En *Class Visualizer* no se menciona que haya un enfoque arquitectónico, solo se describe como un generador de diagrama de clases. *ScalaReflect* cuenta con el soporte para la construcción de un documento XMI el cual sirve para dos propósitos: el primero para guardar el contenido recuperado en la herramienta y el segundo para ser exportado desde Enterprise Architect y así mostrar el diagrama de clases de una manera más organizada, por el contrario *Class Visualizer* guarda el contenido como un proyecto pero no tiene la capacidad de exportarlo a un documento XMI.

Capítulo 5

Conclusiones

Trabajar con un enfoque arquitectónico además de que apoya a los futuros usuarios de *ScalaReflect*, ayudó a la construcción de la herramienta, ya que trabajar alineándose al Modelo 4+1 facilitó el desarrollo, ya que este modelo sirvió como una guía, la cual dice lo que se va a representar y cómo se va a representar.

El mecanismo de reflexión de Scala es el pilar de esta herramienta, debido al gran soporte que ofreció para construir los diagramas aquí presentados. Fungió un papel muy relevante para la representación funcional, la cual es importante de recuperar ya que al estar implementada en los sistemas Scala, impacta en cuestiones de diseño y de arquitectura.

La herramienta realiza una correcta representación de los elementos funcionales de Scala esto gracias al uso de estereotipos y el manejo de tipos que realiza la reflexión. Con la implementación de XMI, se le dará la facilidad al encargado de mantenimiento de visualizar los diagramas en una herramienta diferente y no necesariamente en la que se está llevando a cabo el proceso de ingeniería inversa. Enterprise Architect fue la herramienta ideal para este trabajo ya que maneja en el XMI sólo los elementos propios para UML.

ScalaRelflect apoyará en el mantenimiento de sistemas desarrollados en Scala, ya que será más fácil identificar qué aspectos de la implementación son los que necesitan ajustes o en cuáles de ellos se introducirá un nuevo requerimiento. Los encargados del mantenimiento usan estos diagramas para entender el sistema con el fin de cambiar las funciones existentes,

eliminar funciones o agregar funciones. Mediante el uso de los modelos de vista lógicos, los encargados del mantenimiento localizan las clases involucradas en un área funcional dada y entender las intenciones de los diseñadores originales para que las modificaciones se ajusten al diseño original.

ScalaReflect cuenta con:

- Un enfoque arquitectónico basado en el modelo 4+1.
- Una representación objeto funcional, basada en una notación para UML.
- El uso de la API Reflection de Scala para realizar la ingeniería inversa

Ya que el mecanismo de reflexión no da un soporte completo para conocer los procesos del sistema, se presentó una aportación que consistió en la realización de pruebas para comprobar que el uso de aspectos fue factible, como trabajo futuro se piensa construir diagramas dinámicos a partir de la información que proporciona AspectJ, para que la herramienta cumpla con los diagramas de actividades y secuencia pertenecientes a la vista de procesos del modelo 4+1.

Producto Acádemico

Mariela Lezama Sánchez, Ulises Juárez Martínez, Celia Romero Torres, Gustavo Peláez Camarena, Antonieta Abud Figueroa *Prototipo de ingeniería inversa basado en reflexión para sistemas desarrollados en Scala*. XIV Congreso Internacional sobre Innovación y Desarrollo Tecnológico. Cuernavaca, Morelos, México. 2018 ISBN 978-607-95255-8-3

Figura 5.1: XIV Congreso Internacional sobre Innovación y Desarrollo Tecnológico

Bibliografía

[1] Oracle and/or its affiliates. Package java.lang.reflect. https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/package-summary.html, 2018.

[2] EPFL. Symbols. https://www.scala-lang.org/api/2.12.0-M5/scala-reflect/scala/reflect/api/Symbols.html, 2016.

[3] R. Warden. *Software Reuse and Reverse Engineering in Practice*. Chapman and Hall, London, England, 1992.

[4] N. Rodríguez Munguía. Propuesta de notación para el modelado de elementos de programación funcional en uml. *CIITI*, 2016.

[5] E. Chikofsky y J. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[6] N. Raychaudhuri. *Scala in action*. Manning Publications Co., Shelter Island, New York, 2013.

[7] J. Malenfant. A tutorial on behavioral reflection and its implementation. *Reflection 96' Conference Proceedings*, 1(1):1–20, 1996.

[8] T. Lindholm. The java® virtual machine specification. https://docs.oracle.com/javase/specs/jvms/se7/html/index.html, Febrero 2013.

[9] P. Clements y R. Kazman L. Bass. *Software Architecture in Practice.* Addison-Wesley, 3rd edition, 2012.

[10] N. Medvidovic y E. M. Dashofy R. N. Taylor. *Software Architecture Foundations, Theory, and Practice.* Wiley India Pvt, 2010.

[11] C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley Longman Publishing Co., Inc, Boston, MA, USA, 2nd edition, 2002.

[12] R. S. Pressman. *Ingeniería de Software Un enfoque práctico.* Mc Graw Gill, Delegación Álvaro Obregón, 2010.

[13] L. Bass y J. Carriere F. Bachmann. *Documentation in Practice: Documenting Architectural Layers.* Addison Wesley, 2000.

[14] P. Kruchten. The 4 +1 vew model of architecture. *IEEE Software*, 12(6):42–50, 1995.

[15] Douglas K Barry. Xml metadata interchange (xmi). https://www.service-architecture.com/articles/web-services/xml_metadata_interchange_xmi.html.

[16] Ramnivas Laddad. *AspectJ in Action.* Manning Publications, second edition edition, August 2009.

[17] Martin Lau. The essence of object-functional programming and the practical potential of scala. https://blog.codecentric.de/en/2015/08/essence-of-object-functional-programming-practical-potential-of-scala/, September 2015.

[18] Scala. Scala documentation. http://docs.scala-lang.org/tutorials/tour/tour-of-scala.html.

[19] T. Kitpanich y P. Thongnuan A. Nanthaamornphong, A. Leatongkam. Bytecode-based class dependency extraction tool: Bytecode-cdet. In *2015 7th International Conference on Information Technology and Electrical Engineering (ICITEE)*, pages 6–11, 2015.

[20] S. Kpodjedo y G. El Boussaidi J. Cloutier. Wavi: A reverse engineering tool for web applications. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–3, 2016.

[21] M. Stula y I. Crnkovic J. Maras. phpmodeler - a web model extractor. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 660–661, 2009.

[22] A. Jain y D. K. Tayal Vinita. On reverse engineering an object-oriented code into uml class diagrams incorporating extensible mechanisms. *ACM SIGSOFT Software Engineering Notes*, 33(5):1–9, 2008.

[23] Y. G. Gueheneuc. Ptidej: A flexible reverse engineering tool suite. In *2007 IEEE International Conference on Software Maintenance*, pages 529–530, 2007.

[24] M. Meyer y D. Travkin M. von Detten. Reverse engineering with the reclipse tool suite,. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, pages 299–300, 2010.

[25] J. Sun y S. Manoharan E. Varoy, J. Burrows. From code to design: A reverse engineering approach,. In *2016 21st International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 181–186, 2016.

[26] H. Sneed y C. Verhoef. Reverse engineering a visual age application. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 487–496, 2015.

[27] K. Morris y S. Filippone A. Nanthaamornphong. Extracting uml class diagrams from object-oriented fortran: Foruml. In *SE-HPCCSE '13 Proceedings of the 1st International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering*, pages 9–16, 2013.

[28] M. Collard y J. Maletic M. Decker, K. Swartz. A tool for efficiently reverse engineering accurate uml class diagrams,. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 607–609, 2016.

[29] M. Qi-Guang Y. Yuan-Zhu Hui-Liu Ying-Cao Yun-Wang L. Zhong-Lin y Z. Xian-Guo. Roptool: A reverse engineering assistant tool for dynamic analysis. In *2010 International Conference On Computer Design and Applications*, pages 216–220, 2010.

[30] U. Čibej y B. Slivnik M. Potočnik. Linter-a tool for finding bugs and potential problems in scala code. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing - SAC '14*, pages 1615–1616, 2014.

[31] P. Dugerdil y P. Descombes J. Repond. Use-case and scenario metamodeling for automated processing in a reverse engineering tool. In *Proceedings of the 4th India Software Engineering Conference on - ISEC '11*, pages 135–144, 2011.

[32] P. Velazco Elizondo y L. Castro Careaga H. Cervantes Maceda. *Arquitectura de software Conceptos y ciclo de desarrollo.* Cengage Learning, 2016.

[33] J. Ivers L. Reed y R. Nord P. Clements F. Bachmann L. Bass D. Garlan. *Documenting software architectures: Views and beyond.* MA: Addison-Wesley Professional, 2011.

[34] D. E. Perry y A. L. Wolf. Foundations for the study of software architecture. *SOFTWARE ENGINEERING*, 17(4):40–52, 1992.

[35] Jonatan Kazmierczak. Class visualizer. https://www.class-visualizer.net/overview.html, 2011, 2017.