SEP
SECRETARÍA DE EDUCACIÓN PÚBLICA

Subsecretaría de Educación Superior

Tecnológico Nacional de México

Instituto Tecnológico de Orizaba

"Año del Centenario de la Promulgación de la Constitución Política de los Estados Unidos Mexicanos"

DIVISIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN

OPCIÓN I.- TESIS

Trabajo Profesional

"Programación Orientada a Aspectos con Objetos Funcionales".

QUE PARA OBTENER EL GRADO DE:

Maestro en Sistemas Computacionales

PRESENTA:

I.S.C. Jesús Juárez de Felipe

DIRECTOR DE TESIS:

Dr. Ulises Juárez Martínez

CODIRECTOR DE TESIS:

M.C. María Antonieta Abud Figueroa

Ciencia - Técnica - Cultura

1957 - 2017 Aniversario

ORIZABA, VERACRUZ, MÉXICO. **OCTUBRE 2017**

"Año del Centenario de la Promulgación de la Constitución Política de los Estados Unidos Mexicanos"

FECHA: 10/10/2017
DEPENDENCIA: POSGRADO
ASUNTO: **Autorización de Impresión**
OPCIÓN: I

C. **JESUS JUAREZ DE FELIPE**
CANDIDATO A GRADO DE MAESTRO EN:
SISTEMAS COMPUTACIONALES

De acuerdo con el Reglamento de Titulación vigente de los Centros de Enseñanza Técnica Superior, dependiente de la Dirección General de Institutos Tecnológicos de la Secretaría de Educación Pública y habiendo cumplido con todas las indicaciones que la Comisión Revisora le hizo respecto a su Trabajo Profesional titulado:

"PROGRAMACION ORIENTADA A ASPECTOS CON OBJETOS FUNCIONALES".

Comunico a Usted que este Departamento concede su autorización para que proceda a la impresión del mismo.

A T E N T A M E N T E

RUBEN POSADA GOMEZ
JEFE DE LA DIV. DE ESTUDIOS DE POSGRADO

C.A. TITULACIÓN
SECRETARÍA DE EDUCACIÓN PÚBLICA
INSTITUTO TECNOLÓGICO DE ORIZABA

ggc

"Año del Centenario de la Promulgación de la Constitución Política de los Estados Unidos Mexicanos"

FECHA : 22/09/2017

ASUNTO: **Revisión de Trabajo Escrito**

C. M.C. MA. ELENA GARCÍA REYES
JEFE DE LA DIVISION DE ESTUDIOS
DE POSGRADO E INVESTIGACION.
P R E S E N T E

Los que suscriben, miembros del jurado, han realizado la revisión de la Tesis del (la) C. :

JESUS JUAREZ DE FELIPE

la cual lleva el título de:

"PROGRAMACION ORIENTADA A ASPECTOS CON OBJETOS FUNCIONALES".

Y concluyen que se acepta.

A T E N T A M E N T E

PRESIDENTE :	**DR.. ULISES JUAREZ MARTINEZ**	FIRMA
SECRETARIO :	**M.C. MARIA ANTONIETA ABUD FIGUEROA**	FIRMA
VOCAL :	**DR. JOSE LUIS SANCHEZ CERVANTES**	FIRMA
VOCAL SUP. :	**DRA. LISBETH RODRIGUEZ MAZAHUA**	FIRMA

EGRESADO(A) DE LA MAESTRIA EN **SISTEMAS COMPUTACIONALES**

OPCION: **I** **Tesis**

Dedicatoria

A mis padres, mi hermana y a Yessica por apoyarme en todo momento.

Agradecimientos

A mis padres y mi hermana por su apoyo incondicional.
A Yessica, por estar conmigo en los buenos y malos momentos, por apoyarme y motivarme a seguir adelante.
A mis compañeros, por su amistad durante esta etapa.
Al Dr. Ulises Juárez Martínez por su asesoría para la elaboración de este trabajo.
Al Consejo Nacional de Ciencia y Tecnología (CONACyT) por el financiamiento económico para la realización de mis estudios.

Muchas gracias

Índice general

Índice de tablas

Índice de figuras

Lista de Códigos

Resumen

El paradigma de programación orientado a objetos funcionales surge al combinar la programación orientada a objetos con la programación funcional. En este paradigma las funciones se consideran también objetos. El problema que surge con este nuevo paradigma es que no se cuenta con soporte para aplicar aspectos sobre estos nuevos mecanismos.
El presente trabajo muestra cómo aplicar cortes sobre los objetos funcionales en los lenguajes Java y Scala utilizando el lenguaje AspectJ, la herramienta Javassist y la técnica del Mixin Composition. Se analizaron las capacidades de corte de cada una de las herramientas y las capacidades que tienen para afectar a los objetos funcionales.
Se desarrolló un marco de trabajo que solventa algunos de los problemas que las herramientas actuales presentan para aplicar cortes sobre los objetos funcionales, el marco de trabajo desarrollado emplea anotaciones que se colocan sobre los objetos funcionales y así agregar un aviso en el objeto funcional.
Se analizaron los patrones de diseño orientados a aspectos que facilitan el desarrollo de diseños usando objetos funcionales, también se analizaron las ventajas y desventajas que tiene la aplicación de objetos funcionales en un sistema.

Abstract

The functional object oriented programming paradigm emerges to combining object-oriented programming with functional programming. In this paradigm, functions are also considered objects. The problem that arises with this new paradigm is that there is no support to apply aspects of these new mechanisms.
The present work shows how to apply cuts on the functional objects in the Java and Scala languages using AspectJ language, Javassist tool, and Mixin Composition technique. We analyzed the cutting capabilities of each tool and their ability to affect functional objects. A framework was developed that solves some of the problems that the present tools present to apply cuts on the functional objects, the developed framework uses annotations that are placed on the functional objects and thus add an advice in the functional object. We analyzed aspect oriented design patterns that facilitate the development of designs using functional objects, we also analyzed the advantages and disadvantages of the application of functional objects in a system.

Introducción

El paradigma de programación objeto funcional surge de la combinación del paradigma de programación orientado a objetos junto con el paradigma de programación funcional, en este nuevo paradigma cada función se considera un objeto. El paradigma funcional tiene varias ventajas como lo son una mejor modularización del código, así como también un mejor manejo de la concurrencia, por lo que en los objetos funcionales estas ventajas están presentes. El problema que aparece a partir del surgimiento de este nuevo paradigma es cómo aplicar la programación orientada a aspectos a los objetos funcionales, es decir, cómo programar aspectos que apliquen a este tipo de mecanismos, cómo aplicar patrones de diseño orientados a aspectos para construir aplicaciones.
Actualmente existen trabajos en los cuales se maneja la orientación a aspectos junto con lenguajes que trabajan el paradigma objeto funcional como lo son Scala y Java. También hay trabajos en donde se desarrollan patrones de diseño orientados a aspectos y se muestra su implementación en alguno de los lenguajes antes mencionados. Sin embargo, aún es necesario integrar adecuadamente todo este soporte para desarrollar soluciones eficientes para la industria.
La presente tesis se divide en cinco capítulos los cuales se describen a continuación. En el capítulo 1 se presentan conceptos relevantes para la comprensión del proyecto, entre los que se encuentran las definiciones de programación orientada a aspectos y programación funcional, se enumeran los principales patrones de diseño orientados a aspectos identificados hasta el momento, también se incluyen descripciones de los lenguajes Java, Scala y AspectJ, que son los que se utilizaron para el desarrollo del proyecto.
En el capítulo 2 se describen los trabajos relacionados con los temas tratados en el proyecto, donde se muestra el uso de AspectJ con Scala y Java, así como también los avances recientes en el campo del desarrollo de patrones de diseño orientados a aspectos, por último, se muestra la propuesta de solución, el planteamiento del problema, las herramientas a utilizar y la metodología necesaria para alcanzar los objetivos de la tesis.
En el capítulo 3 se muestra mediante distintos ejemplos cómo aplicar la programación orientada a aspectos usando el lenguaje AspectJ a los objetos funcionales en los lenguajes Java 8 y Scala, también se muestran opciones a AspectJ para aplicar aspectos sobre objetos funcionales como lo son el uso del lenguaje Scala con *mixin composition* y mónadas, ya que ambos mecanismos permiten igualar conceptos del paradigma orientado a aspectos. Por último, se presenta mediante un análisis qué patrones de diseño son más eficaces para el desarrollo de aplicaciones que emplean objetos funcionales.

En el capítulo 4 se presentan los resultados obtenidos mediante la demostración de un caso de estudio y se muestran las ventajas y desventajas de la aplicación de la programación orientada a aspectos sobre los objetos funcionales. En el último capítulo se dan las conclusiones y las recomendaciones.

Capítulo 1

Antecedentes

Este capítulo contiene los conceptos básicos para la correcta comprensión de la presente tesis. En primer lugar, se explican los conceptos más relevantes para la comprensión de esta investigación, como son la programación funcional, la programación orientada a aspectos, patrones de diseño orientados a aspectos y patrones de diseño objeto-funcionales. En segundo lugar, se presentan los conceptos más importantes de los lenguajes de programación que se utilizaron, entre los que se encuentran Java, Scala y AspectJ.
Al final de este capítulo se expone el planteamiento del problema, el objetivo general y los objetivos específicos, así como también la justificación de la presente investigación.

1.1. Marco teórico

A continuación se muestran conceptos relacionados con los objetos funcionales en los lenguajes Java y Scala, también explicaciones sobre los paradigmas de programación utilizados.

1.1.1. Efecto de borde

En programación el efecto de borde ocurre cuando en una expresión o función se modifica el estado interno, por ejemplo al modificar una variable (ya sea global o estática), escribir datos en la pantalla, leer o escribir en un archivo, entre otros. Los efectos de borde varían en función del paradigma de programación que se esté utilizando, siendo más comunes en la programación orientada a objetos y menos usados en la programación funcional.

Programación funcional

Así como otros tipos de programación se basan en los métodos o en los procedimientos, la programación funcional se basa en el comportamiento de las funciones en el sentido matemático. En matemáticas las funciones no tienen efecto de borde (*side effect*), por ejemplo, con la función x = sin (y), el resultado obtenido se asigna a x, entonces se dice

que la función no tiene efecto de borde o es pura. Gracias a esta propiedad se simplifica enormemente el analizar, realizar pruebas y depurar una función. Con esto se llaman a las funciones sin saber nada sobre el contexto en el que se invoca la función, sólo sería necesario saber qué otras funciones llamaría. Este olvido del contexto se conoce como transparencia referencial y permite que la invocación concurrente de las funciones sea sencilla y fiable, ya que las funciones siempre se comportarán de la misma manera. En la programación funcional existen funciones que se componen de otras funciones, éstas se tratan como valores y se conocen como funciones de primera clase. Cuando una función tiene otras funciones como argumentos o devuelve una función se llama función de orden superior. Otra característica que distingue a la programación funcional es que las variables son inmutables [1].

1.1.2. Asuntos transversales

Un asunto transversal es el comportamiento, con frecuencia de los datos, que se utiliza en el ámbito de la aplicación de una pieza de software. Por ejemplo, una restricción en alguna parte del software o simplemente el comportamiento que cada clase realiza [2].

Aspectos

Los aspectos son decisiones de diseño que son difíciles de capturar o dicho en otras palabras son unidades de descomposición no funcionales.

Puntos de unión

Un punto de unión es un punto identificable en la ejecución de un programa y en donde es posible invocar a un aviso. Algunos ejemplos de puntos de unión son [2]:

1. Cuando se llama a un método.
2. Cuando se ejecuta un método.
3. Cuando se invoca a un constructor.
4. Cuando se ejecuta un constructor.
5. Cuando se inicializa un objeto.
6. Cuando se signa un valor a un campo.

Avisos

Los avisos son los códigos que se ejecutan cuando se invoca un aspecto. Los avisos contienen su propio conjunto de reglas sobre cuándo es que se invocan en relación con el punto de unión que se desencadena [2].

Corte en puntos

Los cortes en puntos son predicados que determinan si un evento coincide con un punto de unión.

Programación orientada a aspectos

Mientras que la programación orientada a objetos (POO) es el paradigma más común empleado hoy para gestionar los asuntos fundamentales, no es suficiente para muchos asuntos transversales especialmente en aplicaciones complejas. La programación orientada a aspectos (POA) se encarga de la gestión de estos asuntos transversales. Una implementación típica orientada a objetos crea un acoplamiento entre el núcleo y los asuntos transversales que no es deseable, ya que la adición de nuevas características transversales e incluso ciertas modificaciones a la funcionalidad transversal existente requieren la modificación de los módulos básicos pertinentes. La POA es un nuevo paradigma de programación que proporciona una separación de los asuntos transversales mediante la introducción de una nueva unidad de modularización llamada aspecto, que separa los asuntos transversales en módulos. La POA permite implementar asuntos transversales en aspectos en lugar de fusionarlos dentro de los módulos básicos. El resultado es que la POA modulariza los asuntos transversales de una manera clara, produciendo una arquitectura de sistema que es más fácil de diseñar, implementar y mantener [3].

Patrones de diseño orientados a aspectos

Los patrones de diseño definen soluciones a problemas de diseño recurrentes, y se utilizan porque conducen a una solución rápida y probada a los problemas que más a menudo aparecen, así los patrones de diseño son plantillas que se utilizan para resolver determinados problemas de diseño. Entre los patrones de diseño orientados a aspectos más comunes están:

1. Huevo de Cuco.- Permite sobrescribir un tipo de objeto instanciado en una llamada al constructor para devolver un objeto de una clase diferente de forma transparente a la lógica de negocio original.

2. Director.- Sirve para definir un conjunto de roles que las clases de la aplicación utilizarán.

3. Control de Región.- Se utiliza para definir formalmente las regiones importantes dentro de la aplicación, para que los aspectos reutilicen esas definiciones y asegurar que sólo se aplican en las áreas correctas.

4. Política.- Se usa para especificar un conjunto de normas de desarrollo que se aplican a la estructura de la aplicación [2].

5. Objeto Trabajador.- Permite transformar aplicaciones secuenciales en aplicaciones concurrentes.

6. Agujero de Gusano.- Este patrón se usa cuando existe una pila de llamadas y permite la creación de una ruta directa entre distintos niveles de la pila de llamadas, así se crea un agujero de gusano y se evita viajar a través de cada capa.

7. Introducción de Excepciones.- Se usa para tratar de manera específica las excepciones comprobadas de manera sistemática.

8. Participante.- Se utiliza para modularizar características comunes basándose en asuntos transversales [3].

9. Administrador de Disponibilidad.- Sirve para dar seguimiento del estado de un componente en una aplicación, probablemente una fachada a un sistema externo [4].

10. Cumplimiento de contratos.- Al aplicar pre-condiciones y pos-condiciones (contratos) a la funcionalidad de un método se generan asuntos de corte. Este patrón separa la lógica de negocios del código del contrato [5].

11. Corte en punto elemental.- Sirve para definir la estructura general de un corte pero descomponiéndolo en cortes más básicos [6].

12. Parche.- Se utiliza para modificar, ampliar o sustituir funciones en código que está siendo reutilizado [7].

13. Espectador.- Sirve para examinar o verificar una traza de ejecución o un perfil dentro de un programa, y hace más fácil la depuración de éste [7].

14. Regulador.- Su función es proporcionar un control más especializado o un comportamiento adicional a algunas clases mientras se están ejecutando [7].

15. Extensión.- Sirve para extender las funciones de los programas que al momento del diseño no se conocen, pero se sabe que en el futuro se agregarán características adicionales [7].

16. Diseño heterárquico.- Se utiliza para separar los distintos asuntos de un diseño para que sean descritos independientemente [7].

1.1.3. Objeto funcional

Los objetos funcionales son objetos que no tienen un estado que cambie, esto quiere decir que no tienen efecto de borde, además se llaman como funciones, es decir, no se invocan como los objetos normales de la forma *objeto.método*, si no de la forma *objeto(parámetros)*. Los objetos funcionales se emplean en los lenguajes orientados a objetos para obtener los beneficios de la programación funcional. Existen varios lenguajes que implementan los objetos funcionales, entre ellos se encuentran JavaScript, PHP, Ruby, Scala, Java y C#.

1.1.4. Scala

Scala es un lenguaje de programación que como su nombre lo dice es escalable, esto quiere decir que crece con la demanda de los usuarios, por ello se usa para escribir sólo unas cuantas líneas de código, o para escribir un sistema complejo. Otra ventaja que presenta es que se ejecuta sobre la máquina virtual de Java (JVM, *Java Virtual Machine*) además de inter-operar con el código escrito en Java. Scala es capaz de utilizar las bibliotecas de Java. Scala combina los conceptos de la programación orientada a objetos y de la programación funcional en un lenguaje con tipificación estática. Esta combinación de estilos hace que sea posible expresar nuevos tipos de patrones de programación y abstracciones de componentes [8].

Traits

Los *traits* en Scala son clases similares a las interfaces en Java, encapsulan métodos y definiciones de campos que después usarán las clases, los *traits* poseen la limitante de que no tiene parámetros de constructor, así una clase es capaz de implementar varios *traits*. Scala también permite que los *traits* se apliquen parcialmente. Una definición de un *trait* se muestra en el código 1.1:

Código 1.1: Trait en Scala

```
1  trait Operaciones {
2   def Suma(x: Int): Int
3   def Resta(x: Any): Int
4  }
```

Funciones

El lenguaje Scala cuenta con *traits* para definir funciones que van del *trait* Function1 al *trait* Function22, para funciones desde 1 argumento hasta 22. En Scala las funciones son objetos, una función que toma un argumento es una instancia del *trait* Function1 ya sea que se declare la herencia de manera explícita o no, si se declara la herencia al *trait* correspondiente es necesario indicarle el valor de entrada y el valor de retorno de la función entre corchetes junto al nombre de *trait*, por ejemplo *Function1[String, String]*, para una función que recibe un valor *String* y de igual manera retorna un valor *String*. Los *traits* *Function* definen el método *apply()*, este método se implementa por defecto en todas las funciones y permite invocar a los objetos como si fueran funciones, un ejemplo de función se muestra en el código 1.2.

Código 1.2: Función en Scala

```
1  object Hola extends Function1[String, String] {
2    def apply(m: String): String = "Hola " + m
3  }
```

Para invocar a esa función se hace de dos formas distintas (código 1.3), la primera llamándola como a una función y sólo enviando los parámetros, y la segunda llamándola como un objeto e invocando a su método *apply()*:

Código 1.3: Formas de llamar a una función

```
1 Hola("Mundo")
2 Hola.apply("Mundo")
```

Funciones de orden superior

Las funciones de orden superior son funciones que toman como parámetro a otras funciones y se obtiene como resultado una función. En el código 1.4 se tiene la clase *Prueba* que usa a la función *suma* y le envía como parámetros además de dos números enteros la función *resultado*, esta función se convierte a cadena de caracteres al recibirse en la función *suma* (línea 5) y se utiliza en la implementación del comportamiento de la función:

Código 1.4: Función de orden superior

```
1  object Prueba {
2  def main(args: Array[String]) {
3   println( suma(resultado, 2, 10) )
4  }
5  def suma(a: Int => String, x: Int, y:Int)=a(x + y)
6  def resultado(x: Int) = "Resultado: " + x
7  }
```

Funciones currificadas

La currificación de una función consiste en transformar una función que toma varios parámetros en una cadena de funciones en donde cada una toma sólo un parámetro. En el código 1.5 se muestra la función *concatena*, esta función está currificada, por lo que en lugar de recibir los dos parámetros juntos de la forma *concatena(c1: String, c2: String)*, lo hace de manera independiente como si fueran dos funciones, dentro de esa función se ocupan los dos parámetros de manera normal.

Código 1.5: Función currificada

```
1  object Test {
2     def main(args: Array[String]) {
3        val c1:String = "Hola "
4        val c2:String = "Mundo"
5        println(concatena(c1)(c2))
6     }
7     def concatena(c1: String)(c2: String) = {
8        c1 + c2
9     }
10  }
```

1.1.5. Java

Java es un lenguaje orientado a objetos y que en su versión más reciente Java 8 ofrece soporte al paradigma de programación funcional. Una característica importante de Java

es que es independiente de la plataforma, esto quiere decir que los programas escritos en Java se ejecutan en muchos dispositivos de hardware. Java implementa un sistema de recolección de basura que se encarga de borrar de la memoria los objetos que ya no tienen referencias a sí mismos en el programa.

Lambdas

Las expresiones lambda son funciones que son pasadas como valores, se utilizan mediante una interfaz funcional que contiene un único método abstracto.
En el código 1.6 se emplea una lambda que recibe una cadena de caracteres y devuelve un saludo, se observa la interfaz funcional *Suma* con un único método *m*, y en el método *main* se crea el objeto funcional y se hace uso de él.

Código 1.6: Expresión lambda

```
1 public class Lambda {
2    interface Suma{
3       public String m(String x);
4     }
5    public static void main(String[] args) {
6      Suma obj = new Suma() {
7             public String m(String x) {
8                 return "Hola " + x;
9             }
10         };
11       System.out.println(obj.m("Mundo"));
12  }
13 }
```

Otra manera de implementar el comportamiento de las lambdas de una forma más reducida es mediante funciones anónimas, como se muestra en el siguiente ejemplo:

```
Suma b = (x) -> "Hola " + x;
```

Y lo que en el ejemplo anterior con el objeto llamado *obj* (código 1.6) se hacía con cuatro líneas de código ahora es sólo una. O en el caso de que no se esté familiarizado con las funciones y se coloque la palabra reservada *return*, esta se usa siempre y cuando se coloquen las llaves en la declaración del comportamiento:

```
Suma c = (x) -> {
        return "Hola "+ x;
      };
```

Para usar estas funciones es de la misma manera que en los ejemplos anteriores:

```
System.out.println(c.m("Mundo"));
```

1.1.6. AspectJ

AspectJ es un lenguaje de propósito general creado como una extensión orientada a aspectos para el lenguaje Java. Dado que AspectJ es una extensión de Java, todos los programas de Java también son programas válidos de AspectJ. Un compilador de AspectJ produce archivos de clases que se conforman de acuerdo con la especificación del código de bytes de Java permitiendo a cualquier código compatible con la Máquina virtual de Java el ejecutar los archivos de las clases. Mediante el uso de Java como lenguaje de base, AspectJ toma todos los beneficios de Java y facilita a los programadores de Java entender el lenguaje AspectJ. Este lenguaje consta de dos partes, la primera es la especificación del lenguaje y la segunda la implementación del lenguaje. La parte de especificación del lenguaje define la sintaxis del lenguaje AspectJ con el cual escribir el código, y la implementación del lenguaje permite la definición de los asuntos tranversales utilizando el lenguaje de programación Java, además AspectJ cuenta con un entrelazador que se encarga de entrecruzar el código de los aspectos con el código de la aplicación. La implementación del lenguaje proporciona herramientas para compilar y depurar además de proporcionar la integración con los más populares entornos de desarrollo [3].

Sintaxis de AspectJ

AspectJ cuenta con 17 primitivas de corte las cuales se muestran en la tabla 1.1 junto con una breve descripción.

AspectJ permite aplicar corte estático en donde se modifica la estructura de las clases y no solo el comportamiento. Con AspectJ es posible agregar nuevos campos y métodos a las clases, hacer que las clases implementen nuevas interfaces o hereden de nuevas clases y definir políticas mediante puntos de unión en donde el compilador genere advertencias y errores.

Además AspectJ cuenta con 3 tipos de avisos que se muestran en la tabla 1.2.

1.1.7. *Simple build tool* (SBT)

SBT es una herramienta de software libre para realizar construcciones de software de manera flexible, ya que se encarga de descargar todas las bibliotecas necesarias para que el proyecto funcione.

Para crear un proyecto es necesario crear un directorio base como se muestra a continuación:

```
src/
  main/
    resources/
      Archivos que se incluyen en el jar principal
    scala/
```

Tabla 1.1: Primitivas de corte del lenguaje AspectJ

Primitiva	**Descripción**
call	En la llamada de un método o un constructor
execution	En la ejecución de un método o un constructor
get	Al obtener el valor de un campo
set	Al asignar un valor a un campo
within	Para todos los eventos dentro de un tipo
withincode	Para todos los puntos de unión dentro de un método o constructor
staticinitialization	Al inicializar todas las partes estáticas
handler	Interviene al manejador de excepciones (*catch*)
preinitialization	A la invocación al súper- constructor
initialization	A la inicialización de los objetos
if	Condicionar el corte a través de valores expuestos
adviceexetucion	Interviene la ejecución de los avisos
args	Exponer el contexto de los argumentos del corte
target	Exponer el contexto de un objeto
this	Exponer el contexto de un objeto y lo que se desencadene de él
cflow	Atrapa la llamada original del flujo de control
cflowbelow	Atrapa todas las llamadas que se deriven de la llamada original

Tabla 1.2: Tipos de avisos de AspectJ

Aviso	**Descripción**
after	Después del punto de unión seleccionado
before	Antes del punto de unión seleccionado
around	En lugar del punto de unión seleccionado

```
      Código fuente en lenguaje Scala
   java/
      Código fuente en lenguaje Java
```

Los comandos más utilizados en SBT son:

1. *sbt run*.-Para compilar y ejecutar la aplicación.
2. *sbt compile*.- Para compilar el código.
3. *sbt test*.- Para ejecutar las pruebas.
4. *sbt package*.- Para crear un archivo jar.

1.2. Planteamiento del problema

Con la aparición de los objetos funcionales surge la interrogante sobre cómo encapsular adecuadamente los requerimientos no funcionales. Propiedades funcionales como las funciones de orden superior permiten igualar algunos conceptos de la programación orientada a objetos, por lo que se hace necesario revisar cómo se programan aspectos en este tipo de lenguajes híbridos, cómo se aplican los patrones de diseño [9],[10] orientados a aspectos (por ejemplo el patrón Objeto Trabajador [3] que convierten aplicaciones secuenciales en aplicaciones concurrentes) y cómo se obtienen ventajas (o desventajas) especialmente de un lenguaje como AspectJ que fue originalmente diseñado para Java. En el caso del lenguaje Scala, se reporta que el estilo de anotaciones de AspectJ es útil mediante una herramienta como Maven [11].

1.3. Objetivo general y específicos

A continuación se presentan el objetivo general y los objetivos específicos de la investigación.

1.3.1. Objetivo general

Aplicar la programación orientada a aspectos para solucionar problemas de asuntos de corte en el paradigma híbrido objeto-funcional mediante el estudio del lenguaje AspectJ y sus contrapartes en los lenguajes Java 8 y Scala.

1.3.2. Objetivos específicos

1. Comparar las capacidades funcionales de los lenguajes Java 8 y Scala considerando el desarrollo de aplicaciones secuenciales y/o concurrentes.

2. Analizar las propiedades de corte del lenguaje AspectJ para la identificación de conceptos equivalentes en el paradigma objeto-funcional.

3. Revisar e identificar los patrones de diseño, orientados a objetos, orientados a aspectos y objeto-funcionales que faciliten la especificación de diseños combinando aspectos y objetos funcionales.

4. Definir alternativas de solución a AspectJ para aplicar aspectos sobre funciones y objetos funcionales.

5. Identificar un caso de estudio que permita mostrar cómo trabajar la programación orientada a aspectos con el paradigma híbrido de objetos funcionales.

1.4. Justificación

Al enriquecer el paradigma de programación orientado a objetos con el paradigma de programación funcional surge un nuevo paradigma de programación basado en objetos funcionales, por lo que es necesario saber cómo aplicar la programación orientada a aspectos a estos nuevos objetos funcionales, qué ventajas y desventajas se obtienen al trabajar con ambos paradigmas, cómo programar aspectos para este tipo de aplicaciones, así como también crear buenas aplicaciones empleando patrones de diseño orientados a aspectos.

Capítulo 2

Estado de la práctica

Este capítulo contiene los trabajos más importantes relacionados con el uso de los lenguajes que manejan objetos funcionales junto con la aplicación de la programación orientada a aspectos, así como también artículos relevantes sobre los patrones de diseño orientados a aspectos.

2.1. Trabajos relacionados

En esta sección se describen los trabajos relacionados con el uso de los lenguajes Java y Scala junto con la programación orientada a aspectos.

2.1.1. Artículos en donde se empleó el lenguaje Scala junto con la programación orientada a aspectos

En [12] se presentó el diseño de una API (Interfaz de Programación de Aplicaciones) de meta programación para Scala cuya funcionalidad es capturar las tareas de meta programación generativa que dependen de definiciones existentes para generar otras, escribiendo meta código lo más cercano posible al código Scala. MorphScala es un lenguaje de dominio específico que introduce la técnica de meta programación *class morphing* a la programación en el lenguaje Scala. *Class morphing* trabaja mediante reflexión en tiempo de compilación sobre los campos o métodos de clases e interfaces. Las tecnologías de apoyo para MorphScala son los macros de Scala que proporcionan una potente API para las transformaciones en tiempo de compilación. Se presentó un boceto del diseño de meta clases en MorphScala, los casos de uso que se probaron demuestran qué tan seguro es el nivel de meta-clase, además de discutir la estrategia de implementación de MorphScala.

En [13] se presentó el desarrollo de JEqualityGen, creado para evitar realizar la implementación manual de comparaciones entre objetos por medio del método *equals* y usando el código *hash*, ya que hacer estas operaciones es tedioso y propenso a errores, así se logra aliviar al desarrollador de la carga de la aplicación. Las tecnologías de generación de código se emplean para hacer frente a este problema, haciendo que las implementaciones

resultantes sean rápidas, eficientes y fáciles de verificar. JEqualityGen funciona en dos estados, en primer lugar utiliza el código fuente de la reflexión en Meta AspectJ para generar aspectos que contienen las implementaciones del método, para después insertar éstos en la aplicación de destino. Aparte de la mejora sustancial de rendimiento que se registró en las pruebas, una ventaja de la generación de código es el análisis estático y que las herramientas de verificación formal trabajan con el código generado e inferir algunas propiedades del sistema. También es posible para herramientas como AspectJ colocar avisos directamente en el código generado. Otra ventaja del análisis estático del código es el emitir advertencias y errores en tiempo de generación de código, mientras que otros sistemas lanzan excepciones en tiempo de ejecución, que es mucho menos conveniente. JEqualityGen se probó con lenguajes distintos de Java que se ejecutan en la JVM como Scala.

El enfoque de *streaming* para la programación paralela es un popular paradigma de programación, en el paradigma de *streaming*, una aplicación se descompone en los núcleos de procesamiento, o bloques, que están conectados mediante comunicación explícita por canales. En [14] se planteó la creación de ScalaPipe, que es un generador de aplicaciones de *streaming* para plataformas heterogéneas, mediante el uso de una colección de lenguajes de dominio específico incrustados en el lenguaje de programación Scala. ScalaPipe permite la creación de aplicaciones de *streaming* que se ejecutan en una variedad de hardware. Además hace que sea fácil generar, modificar, optimizar e instrumentar grandes topologías complejas y asignaciones de recursos. ScalaPipe adopta un enfoque orientado a aspectos para el mapeo de recursos, es decir, los límites de los recursos se especifican como un borde entre dos bloques y el tipo de movimiento de recurso se indica. Además permite grandes topologías y asignaciones de recursos que se generan y modifican fácilmente sin cambios sustanciales en el código fuente de la aplicación, así el mismo código fuente se utiliza para diferentes tipos de datos por lo que se usa en múltiples plataformas según sea necesario. Se realizaron pruebas con diversos casos de estudio obteniendo buenos resultados.

En las aplicaciones reactivas los eventos o cambios de estado como la interacción del usuario, los cambios de datos en un diseño de tipo Modelo-Vista-Controlador, los mensajes de red o la adquisición de valores a partir de sensores, entre otros, actualizan el estado del sistema o activan nuevos eventos y / o cálculos. A nivel de la organización del código, la modularización adecuada es difícil de lograr debido a que las reacciones se activan en varios lugares en el código. En tiempo de ejecución, el flujo de control normal se intercala con las reacciones de los eventos, lo que lleva a interacciones que son difíciles de prever. En [15] se desarrolló un lenguaje reactivo basado en Scala que integra los conceptos de la programación basada en eventos y funcional reactiva en el mundo orientado a objetos. REScala apoyó el desarrollo de aplicaciones reactivas mediante el fomento de un estilo funcional declarativo que complementa las ventajas del diseño orientado a objetos e integró a la perfección valores reactivos con un sistema avanzado de eventos. Así se explotan los beneficios de abstracciones reactivas sin perder las ventajas de diseño orientado a objetos. En REScala los eventos y valores reactivos son atributos de objeto, además que campos y métodos están expuestos como parte de la interfaz del objeto. Además cuenta con una amplia biblioteca de operaciones (API) para apoyar un diseño orientado a objetos

y funcional con eventos y valores reactivos. REScala mejoró eficazmente la creación de aplicaciones reactivas mediante el fomento de un estilo declarativo y funcional sin renunciar a las ventajas de diseño orientado a objetos.

Las aplicaciones reactivas son difíciles de implementar, las soluciones tradicionales se basan en los sistemas de eventos y el patrón observador, pero presentan varios inconvenientes, a pesar de ello se prefieren estos diseños por los beneficios de la orientación a objetos. Por otro lado, los enfoques reactivos basados en las actualizaciones automáticas de las dependencias proporcionan indudables ventajas, pero no encajan bien con los objetos mutables. La programación orientada a aspectos se utiliza para monitorizar las modificaciones en los objetos y mantener entidades dependientes actualizadas. Desde la POA se apoya la modularización adecuada de los asuntos transversales, las funcionalidades de actualización se separan a partir del código del objeto. Por ejemplo, el patrón observador es posible implementarlo en forma modular mediante el uso de técnicas de POA. En [16] se presentó la integración de eventos y comportamientos funcionales reactivos al lenguaje prototipo REScala y en las pruebas se obtuvieron resultados prometedores. Se planearon estrategias que se aplican para implementar la reactividad, se analizó la implementación del comportamiento reactivo en varias aplicaciones orientadas a objetos del mundo real, destacando los inconvenientes de abstracciones tradicionales, además se desarrolló un análisis de las soluciones existentes para sistemas reactivos y un plan de investigación que se ocupa de las cuestiones en que se encuentran los enfoques actuales y tienen el objetivo de la combinación de objetos y abstracciones reactivas en un lenguaje eficiente. Como trabajo futuro se planeó evaluar el lenguaje mediante la experiencia de los programadores, con experimentos para evaluar el impacto en el rendimiento de las abstracciones reactivas en comparación con las soluciones tradicionales, otros aspectos del diseño del lenguaje se evaluaron mediante el uso de métricas de software como de acoplamiento y cohesión, líneas de código, número de operaciones, entre otros.

En lenguajes como Java y Scala los eventos se tratan como notificaciones en los objetos escuchadores, en estos eventos imperativos los objetos acceden a los eventos mediante interfaces. Como alternativa, la programación orientada a aspectos cuenta con eventos implícitos en puntos de unión, así el aspecto observa en puntos identificables del flujo de control del programa y evita la necesidad de activar de forma explícita eventos en estos puntos, lo que simplifica y reduce el código. Por otra parte, los cortes en puntos se ven como declaraciones de eventos en los puntos de unión seleccionados. En [17] se presentó el concepto de evento orientado a objetos declarativo en donde los eventos se declaran como clases y se accede a ellos como atributos de los objetos. Los autores propusieron el diseño del lenguaje EScala como una extensión del lenguaje Scala que combina eventos imperativos desencadenados mediante mecanismos de programación orientada a aspectos que están específicamente diseñados para abordar la modularidad de cuestiones de diseños orientados a objetos manejados por eventos, en particular se integran a la perfección con la encapsulación de estilo orientado a objetos, razonamiento modular y *late binding*. Las implementaciones de la programación orientada a aspectos como AspectJ proporcionan extensiones de corte en puntos y avisos para insertar código de asuntos transversales en la base del programa a través de la transformación del *bytecode*.

En [18] se describió un marco de trabajo de programación orientada a aspectos totalmente funcional en el lenguaje Scala. Este lenguaje de programación posee tipificación estática con características orientadas a objetos y funcionales. El marco de trabajo se implementó como un lenguaje de dominio específico interno y posee una sintaxis intuitiva y expresiva. Este marco de trabajo permitió a los programadores especificar corte en puntos y aspectos utilizando un lenguaje específico de dominio incrustado dentro de Scala. Además se usaron las funciones de orden superior de Scala para interceptar llamadas de método con una sobrecarga sintáctica mínima impuestas sobre el programa base, también permite a los desarrolladores definir cortes en puntos especificando los tipos de clases y firmas de métodos y tener acceso a las variables de contexto, mientras que los aspectos insertan código de avisos antes o después del código con un punto de unión.

Los componentes de software se utilizan en varios dominios de aplicación, con el tiempo se propusieron modelos de componentes para cumplir con los requisitos específicos de cada dominio. La tendencia general seguida por estos enfoques es proporcionar modelos *ad-hoc* y herramientas para la captura de estos requisitos y para el apoyo de su implementación dentro de plataformas de ejecución dedicadas. El desafío es entonces proponer soluciones más flexibles en los componentes de reutilización. En [19] se presentó un marco de apoyo para la construcción y desarrollo de aplicaciones que cumplen diversos requisitos extra-funcionales y de dominio específico. Este marco está enfocado en el desarrollo de aplicaciones orientadas a componentes en donde los requisitos extra-funcionales se expresan como anotaciones en las unidades de composición en la arquitectura de la aplicación. Estas anotaciones se implementaron como contenedores basados en componentes abiertos y extensibles, logrando la separación completa de los requisitos funcionales y extra-funcionales. Así se creó el marco Hulotte, implementado con AspectJ, Scala y Java, que se basa en los principios de la programación orientada a aspectos en dos niveles. En primer lugar a nivel de aplicación, basándose en un lenguaje de puntos de unión cuyas capacidades de expresión permiten que el desarrollador declare anotaciones de dominio específico, en segundo lugar, en el nivel de la plataforma, donde se implementan las anotaciones de dominio específico por aspectos como conjuntos de componentes de grano fino en un camino simétrico y unificado. Este enfoque demostró sus beneficios en dos casos de estudio, en aplicaciones en tiempo real embebidas y en el dominio de la distribución del *middleware*. Sin embargo, tiene una limitación, la imposibilidad de implementar por completo algunas partes de los asuntos de dominio específico como arquitecturas de componentes.

El desarrollo actual de software tiene incorporado el estilo de programación orientada a aspectos de AspectJ. Este estilo se conoce como asimétrico, ya que se distingue fácilmente entre la base y los aspectos que afectan a ésta. En contraste con esto, en los enfoques orientados a aspectos simétricos las aplicaciones se componen de vistas parciales o aspectos sin que denote explícitamente ninguno de ellos como base, aunque éstos no tienen demasiada aceptación en la industria existen varios enfoques utilizados en la práctica que presentan ciertas características simétricas orientadas a aspectos. En [20] se analizaron las características de análisis y diseño de mecanismos de algunos lenguajes como los *traits* de Scala, *open clases* de Ruby, o *prototypes* de JavaScript, así como también las declaraciones y los avisos de AspectJ se usan de igual forma para emular la programación simétrica orientada

a aspectos. Se presentaron algunos casos de uso que muestran la base para el desarrollo de la programación simétrica con los mecanismos antes mencionados.

La programación orientada a aspectos introdujo nuevos tipos de fallos en el software y una de las opciones para abordarlos de manera sistemática es la mutación de pruebas. Sin embargo, este enfoque requiere para las pruebas adecuadas del apoyo de herramientas con el fin de que se realicen correctamente. En [21] se abordó este tema con la introducción de una herramienta llamada PROTEUM / AJ que realiza un conjunto de requerimientos para pruebas basadas en mutación y supera algunas limitaciones identificadas en las herramientas anteriores para programas orientados a aspectos. A través del desarrollo de un caso de estudio se demostró cómo PROTEUM / AJ se diseñó para apoyar las principales etapas de mutación de pruebas. Estas pruebas preliminares de la herramienta en un ciclo de pruebas completo proporcionaron evidencias de la viabilidad de su uso en el desarrollo de software y ayudaron a identificar las necesidades futuras.

La implementación de entrelazados dinámicos distribuidos es un asunto de corte transversal desde que la implementación se dividió en varios sub-asuntos y algunos de ellos son los asuntos transversales. Por ejemplo, se incluye a menudo un asunto de seguimiento, que permanece atento el progreso del programa de destino en ejecución de *hosts* remotos. Estos asuntos son entrelazados dinámicamente como el objetivo del programa de una manera transversal. Existen lenguajes dinámicos distribuidos pero no proporcionan apoyo suficiente para la implementación modular de tales tejidos dinámicos distribuidos. En [22] se propuso un nuevo lenguaje llamado Dandy, que permite a los desarrolladores implementar el entrelazado dinámico distribuido en un aspecto. El aspecto implementado es reutilizable y por lo tanto Dandy permite a los desarrolladores escribir una biblioteca de aspectos para entrelazar un aspecto dado en entornos distribuidos. Dandy integra algunas buenas ideas prestadas de trabajos existentes, como aspectos de primera clase, cortes en puntos a distancia y el entrelazado atómico. La contribución de este trabajo fue mostrar una biblioteca de aspectos para realizar entrelazados dinámicos distribuidos, que también son asuntos transversales.

En [23] se presentó una biblioteca orientada a aspectos codificada en AspectJ, que pretende imitar el estándar OpenMP para la programación multi-núcleo en Java. La construcción de la biblioteca apoya la semántica secuencial de OpenMP. La biblioteca permite además el uso de constructores paralelos relacionados en los sistemas orientados a objetos debido a una mejor compatibilidad con la herencia, lo que es más adecuado para introducir el paralelismo en los marcos orientados a objetos. Aunque la biblioteca requiere un cierto grado de recomposición con el fin de aplicarse a un código dado. La biblioteca se probó en varios casos y proporcionó un rendimiento similar al de las implementaciones usando técnicas tradicionales (por ejemplo hilos de Java).

2.1.2. Artículos donde utilizan juntos los lenguajes AspectJ y Java

Algunos autores sugieren que el lenguaje AspectJ es demasiado limitado y que no es posible seleccionar todos los puntos de unión de un programa. Por lo que se han propuesto

muchas mejoras que prácticamente no requieren cambios en el código original. En [24] se presentó una extensión del lenguaje AspectJ para seleccionar bloques y anotaciones en el lenguaje de extensión @Java.

Con el tamaño cada vez mayor de los sistemas de información las búsquedas dentro del código fuente se hacen más complejas, existen herramientas que ayudan a los desarrolladores de software en la recuperación de información relevante. En [25] se presentó una herramienta que permite la recuperación de información en el código fuente de AspectJ de una manera no estructurada.

La computación distribuida volvió a ser muy popular en estos días por su velocidad, precisión y capacidad de tolerancia a fallos. En [26] se trabajó con programas orientados a aspectos distribuidos, donde el paso de mensajes y la sincronización se manejan mediante los aspectos y se presentó un algoritmo dinámico de corte paralelo para programas orientados a aspectos distribuidos para hacer el proceso de cálculo mucho más rápido.

2.1.3. Artículos donde tratan el tema de patrones orientados a aspectos

La implementación de un patrón de diseño toma muchas formas de acuerdo con el lenguaje de programación que se esté utilizando. La mayor parte de la literatura presenta patrones de diseño en sus implementaciones orientadas a objetos. Otros estudios muestran la aplicación en los lenguajes orientados a aspectos más comunes como AspectJ y CaesarJ. En [27] se comparó la ejecución de tres patrones de diseño: *singleton*, observador y decorador en los lenguajes antes mencionados y se discutió la posibilidad de implementar los mismos en ParaAJ, que es una extensión del lenguaje AspectJ que implementa la idea de aspectos paramétricos. ParaAJ provee una implementación reutilizable de los patrones *singleton* y observador, pero no ayuda en el caso del patrón decorador, el problema se presenta con el patrón decorador debido al mecanismo de traducción actual de los aspectos de ParaAJ a los aspectos normales de AspectJ. Se enfatizó el poder de la generación de programas y sus ventajas en la reducción del esfuerzo de desarrollo y en facilitar la mantenibilidad de los programas.

Los paradigmas de programación definen la manera de pensar y diseñar al crear software. Los paradigmas orientados a objetos y funcional son dos de los más usados actualmente, el paradigma objeto funcional utiliza clases para mejorar el nivel de abstracción, además permite que los algoritmos se implementen funcionalmente. En [28] se propuso el uso sistemático de patrones de software para capturar esos nuevos problemas y sus soluciones. La motivación para esto surgió de algunas preguntas como ¿es posible mejorar el conocimiento de los patrones de software siguiendo el paradigma objeto funcional? y ¿cuáles patrones surgen en este contexto? Propusieron evaluar la posibilidad de evolución conocida de los patrones de software, identificando y documentando cada uno en orden a como se adaptan al paradigma objeto funcional implementando los patrones usando el lenguaje Scala. Los patrones son una buena forma de capturar conocimiento empírico en mejores prácticas, entonces si los desarrolladores emplean patrones para implementar software usando el paradigma objeto funcional en lugar del paradigma funcional u orientado a objetos, se espera

observar una reducción del tiempo de desarrollo, de la cantidad de errores introducidos y del tamaño de los códigos, este último logrado a través de mejores abstracciones provistas por el paradigma.

En la programación orientada a aspectos se distingue entre la aplicación asimétrica y simétrica de los aspectos. Los patrones se categorizan por elemento del dominio, en patrones de corte en puntos, y patrones de declaración de inter-tipo. Scala es un lenguaje de programación que une el paradigma orientado a objetos y el paradigma funcional. Scala contiene mecanismos lingüísticos que implementan características orientada a aspectos en forma simétrica, por ejemplo con la composición de *Mixin*. El planteamiento asimétrico distingue entre la base y el aspecto que afecta a esta base y la base no es consciente de los aspectos. AspectJ aplica los aspectos de manera asimétrica. En el enfoque simétrico se considera cómo combinar las cosas y tratar a todos los elementos de la misma forma. Los elementos se componen sin que se denote explícitamente como aspecto y base. En [29] se presentaron tres patrones de diseño orientados a aspectos y su implementación en el lenguaje Scala. El patrón huevo de cuco que su propósito es capturar la llamada a un constructor y en su lugar crear o proveer un objeto de otro tipo. El patrón director que define las funciones adicionales que se aplican a los tipos existentes sin cambiar la implementación. El patrón objeto trabajador que encapsula métodos en objetos para su ejecución posterior sin alterar el hilo principal de la aplicación.

2.2. Análisis comparativo de los trabajos

A continuación se muestra un análisis de los trabajos presentados anteriormente, mostrando a los autores, el nombre del trabajo, el problema a resolver y el resultado obtenido. La tabla 2.1 muestra los trabajos relacionados con el uso de programación orientada a objetos y lenguajes objeto-funcionales, y como se observa el lenguaje más usado es AspectJ.

Tabla 2.1: Trabajos relacionados con el uso de programación orientada a objetos y lenguajes objeto-funcionales.

Autores	**Trabajo**	**Problema**	**Resultado**
Joseph G. Wingbermuehle, Roger D. Chamberlain, Ron K. Cytron [14]	ScalaPipe: A Streaming Application Generator	Dificultad en la creación de aplicaciones de *streaming*.	Generador de aplicaciones de *streaming* para plataformas heterogéneas.

Tabla 2.1: Trabajos relacionados con el uso de programación orientada a objetos y lenguajes objeto-funcionales.

Autores	**Trabajo**	**Problema**	**Resultado**
Neville Grech, Julian Rathke, Bernd Fischer [13]	JEqualityGen: Generating Equality and Hashing Methods	Problemas en la implementación manual de comparaciones de objetos mediante el código *hash* o el método *equals*.	Generador de código fuente que genera automáticamente código para las comparaciones.
Guido Salvaneschi, Gerold Hintz y Mira Mezini [15]	REScala: Bridging Between Object-oriented and Functional Style in Reactive Applications	Casi todo el software orientado a objetos emplea el patrón observador para implementar aplicaciones reactivas.	Lenguaje reactivo que integra conceptos de programación basada en eventos y programación funcional reactiva.
Jaroslav Bálik y Valentino Vrani [20]	Symmetric Aspect-Orientation: Some Practical Consequences	La programación orientada a aspectos simétricos no tiene demasiada aceptación fuera del ambiente académico.	Ejemplos de cómo usar la programación orientada a aspectos simétrica con AspetJ y Scala.
Guido Salvaneschi y Mira Mezini [16]	Reactive Behavior in Object-oriented Applications: An Analysis and a Research Roadmap	Dificultad en el desarrollo de aplicaciones reactivas.	Integración de eventos y comportamientos funcionales reactivos al lenguaje prototipo REScala
Aggelos Biboudis y Eugene Burmako [12]	MorphScala: Safe Class Morphing with Macros	Dificultad al aplicar la meta programación.	API para Scala para capturar tareas de meta programación generativa que dependan de definiciones existentes para generar otras.

Tabla 2.1: Trabajos relacionados con el uso de programación orientada a objetos y lenguajes objeto-funcionales.

Autores	**Trabajo**	**Problema**	**Resultado**
Daniel Spiewak y Tian Zhao [18]	Method Proxy-Based AOP in Scala	No se aprovechan todas las funcionalidades de Scala en la programación orientada a aspectos.	Marco de trabajo completamente funcional orientado a aspectos en Scala.

En la tabla 2.2 se muestra un breve análisis de los artículos que trataron el tema de los patrones orientados a aspectos.

Tabla 2.2: Trabajos relacionados con el uso de patrones de diseño orientados a aspectos.

Autores	**Trabajo**	**Problema**	**Resultado**
Khalid Aljasser [27]	Implementing design patterns as parametric aspects using ParaAJ: The case of the singleton, observer, and decorator design patterns	De acuerdo al lenguaje que se utilice cambia la implementación del patrón de diseño usado.	Comparación de tres patrones de diseño orientados a aspectos en los lenguajes AspectJ, CaesarJ y ParaAJ.
Hugo Sereno Ferreira y Tiago Boldt [28]	Object-Functional Patterns: Rethinking Development in a Post-Functional World	En el paradigma objeto funcional además de utilizar clases los algoritmos se implementan funcionalmente.	Propuesta y análisis del uso de patrones sistemáticos para capturar los nuevos problemas y crear soluciones.
Pavol Pidanic [29]	Exploring Possibilities for Symmetric Implementation of Aspect-Oriented Design Patterns in Scala	No existe una implementación clara de la aplicación de patrones de diseño orientados a objetos en el lenguaje Scala.	Desarrollo de tres patrones de diseño orientados a objetos en el lenguaje Scala.

En los trabajos que se revisaron no se reporta alguno en dónde se utilice la programación orientada a aspectos junto con los objetos funcionales, el lenguaje AspectJ se usa junto con Scala y Java, ya sea mediante anotaciones o con la sintaxis propia de AspectJ, pero no para realizar cortes sobre objetos funcionales.

El que no se hayan encontrado trabajos es debido a que los objetos funcionales son relativamente nuevos y no se ha explorado mucho en ese campo o en combinarlo con otros paradigmas cómo es en este caso en dónde se trabajarán junto con la programación orientada a aspectos.

2.3. Propuesta de solución

A continuación se muestra el planteamiento de la propuesta de solución, el conjunto de tecnologías seleccionadas que permitió resolver la investigación y la metodología de desarrollo que permitió alcanzar todos los objetivos de la investigación.

2.3.1. Planteamiento de la solución

Los objetos funcionales son una nueva mejora en el lenguaje Java 8, en el lenguaje Scala existen desde su creación, la programación orientada a aspectos presenta múltiples ventajas, por lo que es necesario conocer cómo aplicar la programación orientada a aspectos sobre los objetos funcionales y las ventajas y desventajas que se obtienen de trabajar con los paradigmas funcional, orientado a objetos y orientado a aspectos.
La solución que se propone consiste en utilizar como lenguaje de programación a Scala y Java en su versión 8, junto con el lenguaje de programación AspectJ para trabajar la parte de la orientación a aspectos. Como herramienta de construcción a SBT (*Simple Build Tool*), ya que esta herramienta cuenta con soporte nativo para el lenguaje Scala junto con el entorno Eclipse, el cual cuenta con *plugins* para trabajar correctamente con Java y AspectJ.
Se decidió usar ambos lenguajes para analizar las ventajas y desventajas que presentan estos al usar objetos funcionales y en qué casos es mejor aplicar ya sea Java o Scala. Como lenguaje orientado a aspectos se decidió usar AspectJ, ya que este lenguaje permite trabajar junto con Scala mediante anotaciones y con Java mediante anotaciones o con la sintaxis de AspectJ.

2.3.2. Herramientas tecnológicas utilizadas

Para el desarrollo de este proyecto de investigación se seleccionaron las siguientes herramientas:

1. El entorno de desarrollo Eclipse tiene soporte para el lenguaje Java, y también para el lenguaje AspectJ mediante un *plugin*.

2. La herramienta de construcción de proyectos de software SBT tiene soporte para los lenguajes Scala y AspectJ.

3. El marco de trabajo Akka proporciona soporte para mecanismos concurrentes (actores, futuros) para los lenguajes Java y Scala [30].

4. Para la creación de los diagramas del caso de estudio se usó *Visual Paradigm* en su versión comunitaria.

2.3.3. Factibilidad de implementación de la solución propuesta

El desarrollo del proyecto de tesis se considera factible de acuerdo a los siguientes puntos:

1. El costo de adquisición de las herramientas necesarias para el desarrollo del proyecto es nulo, dado que todas las herramientas a utilizar son de libre uso o cuentan con una versión comunitaria.

2. En lo que se refiere al manejo de las herramientas tecnológicas ya se tiene conocimiento en el manejo de las herramientas de software y los lenguajes a utilizar.

Capítulo 3

Aplicación de la metodología

En este capítulo se presenta una comparación de las capacidades funcionales de los lenguajes Java 8 y Scala, considerando el desarrollo de aplicaciones secuenciales y/o concurrentes, también se muestra un análisis de los patrones de diseño orientados a aspectos, orientados a objetos y objeto funcionales que facilitan la especificación de diseños combinando aspectos y objetos funcionales.

3.1. Capacidades funcionales de los lenguajes Java 8 y Scala

Los lenguajes que se analizaron son Java y Scala haciendo énfasis en el desarrollo de aplicaciones concurrentes y/o estructuradas. El marco de trabajo Akka proporciona soporte para mecanismos concurrentes tanto para Java como para Scala con sólo importar la biblioteca necesaria, además de que los actores de Akka son el estándar para trabajar concurrencia con el lenguaje Scala [30].
Los actores son mecanismos computacionales que, en respuesta a un mensaje que reciben, pueden enviar un número finito de mensajes a otros actores, crear un número finito de nuevos actores o designar el comportamiento a utilizar para el siguiente mensaje que recibe. Para ello cada actor cuenta con un método *receive* que recibe el mensaje y decide qué hacer con él. Scala también cuenta con su propio API de actores, pero se recomienda usar la de Akka.
Los futuros en Akka son estructuras de datos que se utilizan para recuperar el resultado de alguna operación concurrente. Así al entrar ejecución el futuro, bloquea los recursos por un tiempo determinado mientras se ejecuta, para liberarlos al momento de obtener el resultado o generar un error si se excede el tiempo de bloqueo indicado.
Se realizó un análisis de las capacidades con que cuentan los lenguajes Java 8 y Scala, haciendo énfasis en sus capacidades concurrentes y funcionales, y se observó que aunque Java y Scala soportan herramientas muy parecidas para el desarrollo de aplicaciones concurrentes, en Scala este soporte es más maduro además de contar con soporte nativo para elementos importantes como los actores, dentro de la parte funcional el soporte de Scala

es mas maduro también, aunque ambos lenguajes cuentan con soporte para funciones anónimas, funciones de primera clase, funciones de orden superior y funciones currificadas, el soporte para estos dos últimos tipos de funciones sólo está de manera parcial, esto quiere decir que la sintaxis de este tipo de funciones no es tan natural como en el lenguaje Scala. Estos resultados se muestran en la tabla 3.1. Se decidió trabajar con ambos lenguajes para ver las ventajas y desventajas que presentan al trabajar con objetos funcionales y la orientación a aspectos.

Tabla 3.1: Análisis de los lenguajes de programación objeto-funcional

	Scala	**Java**
`Hilos`	Sí, hilos de Java	Sí
`Actores`	Sí, de manera nativa y mediante el marco de trabajo Akka	Sí, mediante el marco de trabajo Akka
`Futuros`	Sí	Sí, mediante el marco de trabajo Akka
`Funciones anónimas`	Sí	Sí
`Funciones de primera clase`	Sí	Sí
`Funciones de orden superior`	Sí	Sí, de manera parcial
`Funciones currificadas`	Sí	Sí, de manera parcial

3.2. Propiedades de corte aplicables a los objetos funcionales

En esta sección se muestran las capacidades de corte que el lenguaje AspectJ tiene sobre los objetos funcionales en Java 8 y Scala.

3.2.1. Java 8

En la siguientes secciones se presentan distintos tipo de objetos funcionales y se analizan las capacidades de corte que AspectJ tiene sobre ellos.

Lambdas

En el código 3.1 se emplea una lambda para realizar la operación de suma e imprimir el resultado, en la línea número 1 se observa una interfaz funcional llamada *Suma*, que posee solo un único método (línea 2), en el método *main* de la clase *Calculadora* se crea el objeto funcional, se le implementa su comportamiento (líneas 6 a 11) y por último se hace uso de él enviándole parámetros e imprimiendo su resultado en la línea 11.

Código 3.1: Objeto funcional en Java

```
1  interface Suma{
2    public int m(int x, int y);
3  }
4  public class Calculadora {
5   public static void main(String[] args) {
6    Suma a = new Suma() {
7         public int m(int x, int y) {
8      return x + y;
9         }
10   };
11   System.out.println(a.m(20, 10));
12  }
13 }
```

Para conocer hasta qué punto es capaz AspectJ de cortar a un objeto funcional es necesario conocer todos los puntos de unión que AspectJ detecta sobre cada uno de los objetos funcionales, para así diseñar cortes que sirvan al programador para cambiar la ejecución del programa, pero que a su vez esos cortes sean seguros y no comprometan la aplicación.
Java proporciona la herramienta *javap*, que permite ver el *bytecode* que contiene cada archivo con extensión *.class*, así analizando el *bytecode* se observan entre otra información las declaraciones de los métodos y los campos, los constructores y las clases internas, y muchos de ellos son posibles puntos de unión en dónde aplicar cortes para colocar avisos. Un fragmento a modo de ejemplo de como se muestra el *bytecode* usando la herramienta javap se muestra en el código 3.2:

Código 3.2: Ejemplo de *bytecode*

```
1  public static void main(java.lang.String[]);
2    descriptor: ([Ljava/lang/String;)V
3    flags: ACC_PUBLIC, ACC_STATIC
4    Code:
5      stack=4, locals=2, args_size=1
6         0: new           #2
            // class paq1/Lambda$1
7         3: dup
8         4: invokespecial #3
            // Method paq1/Lambda$1."<init>":()V
9         7: astore_1
10        8: getstatic     #4
11        11: aload_1
```

```
12          12: bipush        20
13          14: bipush        10
14          16: invokeinterface #5,  3
15            // InterfaceMethod paq1/Suma.m:(II)I
16          21: invokevirtual #6
17            // Method java/io/PrintStream.println:(I)V
18          24: return
```

En el código 3.2, en la línea 6 se muestra la creación de la clase Lambda$1, en la línea 14 la llamada al método *m* de la interfaz *Suma* y en la linea 16 la llamada al método *println*. Cabe mencionar que si se ocupa el entorno Eclipse y se quiere ver el *bytecode* de las clases que Eclipse generó del código, la opción de usar la herramienta *javap* sobre ellas no es recomendable, dado que el entrelazador de AspectJ ya trabajó sobre ellas y les agregó código que los aspectos necesitan para ejecutarse, por lo que la opción más recomendable es ubicar los archivos con el código fuente y usar la herramienta *javac* para compilar ese código y posteriormente usar la herramienta *javap* para así observar el *bytecode*, así sólo se observarán las líneas que pertenecen a la aplicación y se facilitará la ubicación de los nombres de los métodos y clases, ya que si el entrelazador colocó algún aspecto dentro de la clase, esta tendrá decenas de líneas adicionales en el *bytecode* y se dificultará mucho el análisis de las clases.
Otra manera de ver todos los puntos de unión es directamente aplicando un aspecto sobre la clase interesada y usar la primitiva de AspectJ *whitin* [2] sobre toda la clase e imprimir todos los puntos de unión que encuentre, como se muestra en el código 3.3.

Código 3.3: Cortes para observar todos los puntos de unión de una clase

```
1 @Aspect
2   public class MainAspect {
3     @Pointcut("within(paq.Main)")
4       public void m(){}
5     @Before("m()")
6       public void n(JoinPoint joinPoint) {
7       System.out.println(joinPoint);
8     }
9   }
```

Los puntos de unión que AspectJ es capaz de detectar sobre el código 3.3 para la clase *Calculadora* (código 3.1) son:

```
1 staticinitialization(paq.Calculadora.<clinit>)
2 execution(void paq.Calculadora.main(String[]))
3 call(paq.Calculadora.1())
4 staticinitialization(paq.Calculadora.1.<clinit>)
5 preinitialization(paq.Calculadora.1())
6 initialization(paq.Calculadora.1())
7 initialization(paq.Suma())
8 execution(paq.Calculadora.1())
9 get(PrintStream java.lang.System.out)
10 call(int paq.Suma.m(int, int))
11 execution(int paq.Calculadora.1.m(int, int))
```

```
12 call(void java.io.PrintStream.println(int))
```

Una vez obtenidos todos los puntos de unión que la clase contiene, es necesario determinar cuáles pertenecen al objeto funcional que se esté trabajando para diseñar los cortes. Sobre el código 3.3, el punto de unión número 2 corresponde a la ejecución de la clase *main*, el punto de unión número 7 es la inicialización de la interfaz *Suma*, el número 10 es la llamada al método *m* de la interfaz *Suma*, y el número 11 es la ejecución del método *m*.

En el código 3.4 se muestra cómo colocar avisos usando los cortes *initialization(paq.Suma())*, *call(int paq.Suma.m(int, int))* y *execution(int paq. Calculadora.1.m(int, int))*:

Código 3.4: Aplicación de cortes al objeto funcional del código 3.1

```
1 public aspect CalculadoraAspect {
2   pointcut uno():
3     initialization(paq.Suma.new(..));
4   pointcut dos(int x, int y):
5     call(* paq.Suma.m(int, int)) && args(x, y);
6   pointcut tres():
7     execution(int paq.Main.*.m(int, int));
8   before () :uno(){
9     System.out.println("Suma.new");
10   }
11  int around(int x, int y): dos(x, y) {
12    return proceed(x+1, y+2);
13   }
14  after() returning(Object r) :tres(){
15    System.out.println("Retorno: "+r.toString());
16   }
17 }
```

Analizando el *bytecode* con la herramienta *javap* se observó que el comportamiento del objeto funcional (Código 3.1), queda dentro de una clase interna en la clase *Calculadora* llamada *Calculadora$1*, el patrón que sigue para crear estas clases internas es colocar el nombre de la clase, un signo de pesos ($), y un número, tomando el uno para el primer objeto funcional de esa clase, el dos para el segundo y así sucesivamente. Se intentó acceder a la clase interna mediante un aspecto, pero AspectJ no fue capaz de detectarla, ya que en Java cuando se usa un objeto funcional se usa la instrucción de *bytecode invoke dynamic* la cuál se encarga del manejo de la clase interna que contiene el comportamiento del objeto funcional, esta instrucción no es detectada por AspectJ, a su vez, esta clase interna es creada a tiempo de compilación, y AspectJ coloca los aspectos a tiempo de compilación, por lo que cuando AspectJ analiza el código para ubicar los puntos de unión y colocar los avisos, el comportamiento del objeto no existe todavía, por estas razones AspectJ no cuenta con el soporte adecuado para este tipo de construcciones.

En el siguiente fragmento de *bytecode* se muestra la clase interna a la que AspectJ no es capaz de acceder:

```
InnerClasses:
```

```
static #2; //class paq/Main$1
```

El corte uno (código 3.4) se aplicó sobre el constructor de la interfaz *Suma* (Código 3.1) y sólo muestra un mensaje, el corte dos se realizó sobre la llamada al método *m* de la interfaz *Suma* y permite capturar los valores que se le mandan al método y modificarlos. Para el corte tres se tuvo que usar el comodín *, dado que la clase contiene un número 1 y en la sintaxis de AspectJ para las firmas de los métodos los números no son válidos, este aviso captura el valor de retorno de la función.

Al momento de generar cortes sobre los puntos de unión identificados es posible dañar la ejecución del programa si no se realizan de manera correcta, por ejemplo, en los puntos de unión mostrados anteriormente se podría comprometer la ejecución del programa si se hace un aviso *around* sobre los puntos de unión de los números 1 al 8, dado que se comprometería el proceso normal de la inicialización del objeto.

A su vez se tiene puntos de unión como los de ubicados en las líneas 10 al 12, en donde sí es posible realizar cortes aplicando avisos *around* sobre ellos y cambiar los valores que reciben o capturar los valores de retorno, y estos cortes serían seguros dado que no comprometen la ejecución del programa, sólo se les cambia el valor que reciben los métodos por otro del mismo tipo.

Un problema que se presenta al trabajar con expresiones lambda es cuando se vuelve a implementar el comportamiento de la misma interfaz funcional dentro de la misma clase, suponiendo que además del objeto que realiza una suma, se crea en la misma clase y usando la misma interfaz funcional otro objeto que realice una multiplicación, como se muestra en el código 3.5 usando la interfaz funcional *Suma* del código 3.1:

Código 3.5: Reimplementación de una interfaz funcional

```
1  Suma a = new Suma() {
2          public int m(int x, int y) {
3                  return x + y;
4          }
5       };
6  Suma b = new Suma() {
7          public int m(int x, int y) {
8                  return x * y;
9          }
10      };
```

Los puntos de unión de la clase en donde se encuentran los dos objetos de la interfaz *Suma* son:

```
1 call(int paq1.Suma.m(int, int))
2 execution(int paq1.Calculadora.1.m(int, int))
3 call(int paq1.Suma.m(int, int))
4 execution(int paq1.Calculadora.2.m(int, int))
```

Se observa en los cortes que si se quiere usar la llamada al método *m* de la interfaz *Suma*, este corte se repite para las dos lambdas, por lo que no se distinguiría una lamba en particular y se cortaría a las dos, también se observa que generó una clase interna para cada objeto funcional (líneas 2 y 4), llamadas *Calculadora$1* y *Calculadora$2*, si se quiere un corte sobre una lamba en particular, usar estos cortes es lo más adecuado, ya que solo es necesario verificar la lambda que aparece primero en el código y esa es la número 1, la siguiente es la número 2 y así sucesivamente, para después verificar en el aspecto el número de lambda que se quiere cortar, como se muestra en el código 3.6:

Código 3.6: Aplicación de cortes para identificar una expresión lambda en particular

```
1  public aspect MainAspect {
2    pointcut corte5(Object a):
3      call(int paq.Suma.sumar(int, int)) && target(a);
4    before(Object a): corte5(a) {
5      if(a.getClass().toString().contains("Lambda$1")){
6      System.out.println("Lambda 1");;
7      }
8     }
9    pointcut corte6():
10     execution(int paq.Main.lambda*$0(int, int));
11   before(): corte6() {
12     System.out.println("lambda 0 ");
13   }
14 }
```

En el *corte5* se atrapa al objeto con la primitiva *target* para ver su clase, convertirla en cadena y validar que sea la lambda que se busca, en el *corte6* se valida directamente en el corte al método de la lambda en la cual se tiene interés.

Funciones anónimas

Las funciones anónimas en Java se comportan de manera similar a las expresiones Lambda, sólo que con una sintaxis más reducida:

```
Suma b = (x, y) -> x + y;
```

Los puntos de unión que AspectJ detecta para esta función son los siguientes:

```
1 call(int paq.Suma.sumar(int, int))
2 execution(int paq.Main.lambda\$0(int, int))
```

El primer punto de unión es similar al primer ejemplo manejado (Código 3.1), que es la llamada al método de la interfaz, pero el segundo es diferente, este punto de unión es la ejecución del método lambda$0, este método se ubica en una clase interna dentro de la clase *main*. Para el segundo punto de unión un ejemplo de un aspecto es como se muestra en el código 3.7.

Código 3.7: Aplicación de un corte a una función anónima

```
1 public aspect CalculadoraAspect {
2   pointcut uno(int x, int y):
```

```
3      execution(int paq.Main.lambda$0(int, int)) && args(x,y);
4    int around(int x, int y): corte5(x, y) {
5      return proceed(x*10, y*5);
6    }
7 }
```

En el corte *uno* se capturan los argumentos del método y se modifican, multiplicando por diez al primer argumento y por cinco al segundo, y usando la variable *proceed* para continuar con la ejecución al programa pero con los parámetros modificados.

Como se mencionó anteriormente, AspectJ no cuenta con el soporte adecuado para interceptar las clases que se generan a tiempo de ejecución, esto es debido a que AspectJ no es capaz de generar el código de los aspectos para el código Java que se generará a tiempo de ejecución y que en tiempo de compilación solo existen referencias a este.

Para aplicar aspectos sobre objetos funcionales sería necesario un lenguaje orientado a aspectos dinámico, que fuera capaz de ver las clases generadas a tiempo de ejecución para así especificar cortes mientras la aplicación está ejecutándose, además de que debe generar el código necesario para que el aspecto funcione y entrelazarlo con el código fuente de la aplicación a tiempo de ejecución, también debería verificar que el código que se acaba de entrelazar no compromete la integridad de la aplicación y de ser así no aplicarlo. Trabajar en la creación de este lenguaje dinámico queda fuera del alcance de este proyecto dado que requeriría más tiempo del que se dispone para realizarlo.

Analizando el *bytecode* de una función anónima de la interfaz funcional *Suma* se observa que se llama a la instrucción *invokeDynamic*, esta sirve para generar un sitio de llamada dinámico, este sitio tiene la característica de que no tiene un método especifícado para invocar. Cuando se llama a la instrucción *invokeDynamic* ésta ejecuta un método de arranque, que vincula el nombre especificado por la instrucción *invokeDynamic* con el código que se va a ejecutar y que hace referencia a un método manejador, el método manejador hace referencia a la clase *LambdaMetaFactory*, esta clase es la encargada de generar objetos funcionales implementando una o más interfaces [31].

Funciones de orden superior

Java cuenta con soporte para funciones de orden superior, estas son funciones que toman como parámetros otras funciones o devuelven una función, en el ejemplo siguiente (Código 3.8) se tiene al objeto calculadora que como parámetros recibe dos enteros y una función, en la implementación del método hace uso de esa función enviando los dos enteros que recibió pero ahora sumados.

Código 3.8: Función de orden superior

```
1  import java.util.function.Function;
2  public class Main {
```

```
3    public void suma(int x, int y,
4    Function<Integer, String> o) {
5    System.out.println(o.apply(x + y));
6   }
7   public static void main(String[] args) {
8     Main calculadora = new Main();
9     Decorador dec = new Decorador();
10     calculadora.suma(65, 30, dec::m);
11   }
12  }
13  class Decorador {
14    public String m(int x) {
15      return "Resultado: " + x;
16   }
17  }
```

Los puntos de unión que AspectJ detecta para la función de orden superior omitiendo los correspondientes a la inicialización y ejecución de la clase Main son:

```
1 call(prueba.Decorador())
2 call(void prueba.Main.suma(int, int, Function))
3 execution(void prueba.Main.suma(int, int, Function))
4 get(PrintStream java.lang.System.out)
5 call(Integer java.lang.Integer.valueOf(int))
6 call(Object java.util.function.Function.apply(Object))
7 call(void java.io.PrintStream.println(String))
```

Los puntos de unión más relevantes sobre dónde aplicar cortes son los números 3 y el 6, el número 3 es la llamada al método *suma* de la clase *main*, el cual recibe dos enteros y un objeto de tipo *Function*, y el número 6 es la llamada al método *apply* dentro del método *suma*, el cual recibe como parámetro un objeto de tipo *Integer*.
Un par de ejemplos sobre la aplicación de cortes a los puntos de unión antes mostrados se observa en el código 3.9.

Código 3.9: Aplicación de cortes sobre una función de orden superior

```
1  public aspect MainAspect {
2   pointcut uno(int x, int y,
     java.util.function.Function<Integer, String> z):
3   execution(void prueba.Main.suma(int, int,
     java.util.function.Function)) && args(x, y, z);
4   pointcut dos(Object o):call(Object
     java.util.function.Function.apply(Object)) && args(o);
5   after(int x, int y,
     java.util.function.Function<Integer, String> z):
6   uno(x, y, z) {
7    System.out.println("Clase Función:"+z.getClass());
8    System.out.println("Objeto Función: "+z);
9   }
10  before(Object o): dos(o) {
11   System.out.println("Clase: "+o.getClass());
12   System.out.println("Objeto: "+o);
13  }
```

```
14 }
```

En el código 3.9 se muestran dos cortes, el primer corte llamado *uno* es sobre la ejecución del método *suma*, que recibe dos enteros y una función, el segundo corte, llamado *dos*, es sobre la llamada al método *apply* el cuál recibe un objeto, este objeto es la suma de los dos enteros recibidos en la primera función.

Funciones currificadas

El soporte a las funciones currificadas en Java 8 sólo está disponible de manera parcial dado que para enviarle los parámetros a una función currificada es necesario en Java invocar tantas veces el método *apply* dentro del mismo objeto como parámetros se le manden, por lo que sí se soporta la currificación pero con una sintaxis no tan natural.
En el código 3.10 se tiene la función *suma* a la que se le mandan dos enteros pero en lugar de hacerlo de la manera normal *suma(5, 20)*, se hace de manera currificada, enviando cada valor de manera independiente, en este caso invocando al método *apply* dos veces consecutivas y mandando un valor por cada invocación.
Para lograr que cada valor se mande de manera independiente se hace uso del tipo *Function<>*, este tipo representa a una función y como parámetros se le indica un valor de entrada y uno de retorno dentro de los símbolos <>, así es posible realizar anidaciones de tipo *Function* en donde algunas de ellas lo que reciban sea una función y otras un valor y así generar la estructura de una función currificada, para que reciba valores y retorne funciones hasta que reciba todos los valores que tenía que recibir y retorne el resultado.

Código 3.10: Función currificada

```
1 import java.util.function.*;
2  public class Main{
3   public Function<Integer, Function<Integer,Integer>> suma(){
4    return x -> y -> x + y;
5   }
6   public static void main(String args[]){
7    System.out.println(new Main().suma().apply(5).apply(20));
8   }
9  }
```

Los puntos de unión que AspectJ detecta para la función currificada en el código 3.10 son los siguientes:

```
1  call(Function prueba.Main.suma())
2  execution(Function prueba.Main.suma())
3  call(Integer java.lang.Integer.valueOf(int))
4  call(Object java.util.function.Function.apply(Object))
5  execution(Function prueba.Main.lambda\$0(Integer))
6  call(Integer java.lang.Integer.valueOf(int))
7  call(Object java.util.function.Function.apply(Object))
8  execution(Integer prueba.Main.lambda\$1(Integer, Integer))
9  call(int java.lang.Integer.intValue())
10 call(int java.lang.Integer.intValue())
```

```
11 call(Integer java.lang.Integer.valueOf(int))
12 call(void java.io.PrintStream.println(Object))
```

En los puntos de unión se observa que además de las dos llamadas al método *apply* en los puntos 4 y 7, se generan dos métodos más, llamados *lambda$0* y *lambda$1*, el primero recibe un valor entero y retorna una función y el segundo que recibe dos enteros y retorna el resultado. Cabe mencionar que por cada tipo *Function* que se esté empleando con su respectivo método *apply* se generará un método *lambda$*.

Streams

Los Streams en Java se definen como una secuencia de elementos y que además soportan operaciones para el procesamiento de sus datos sobre la que es posible aplicar varias funciones encadenadas [32]. Un ejemplo de la creación de un *stream* y la aplicación de algunas funciones sobre él, se muestra en el código 3.11:

Código 3.11: Stream en Java

```
1 public class Streams {
2    public static void main(String[] args) {
3     List<String>  lista = Arrays.asList("Programacion",
      "Orientada a aspectos","Orientada a objetos");
4      Stream<String> stream = lista.stream()
5       .map(s -> s.split(" "))
6       .flatMap(Arrays::stream)
7       .distinct();
8 } }
```

En el código 3.11 se crea una lista que contiene algunas frases (línea 3), en la línea 4 esa lista es convertida a un *stream*, en las líneas 5 a la 7 se aplican 3 operaciones sobre el *stream*, primero se dividen las frases en palabras tomando como punto de separación los espacios en blanco, después esos elementos se convierten a elementos del *stream*, por último solo se agregan al *stream* los elementos que son diferentes entre ellos. Para recorrer los elementos que contiene el *stream* y ver los resultados de las operaciones aplicadas sobre los datos una opción es usar el método *toArray()* que convierte un *stream* en un arreglo de objetos y recorrerlos con un ciclo *for*, ver código 3.12:

Código 3.12: Ciclo para recorrer un *Stream*

```
1 Object[] s = stream.toArray();
2  for(int i=0; i<s.length; i++){
3   System.out.println(s[i]);
4  }
```

Se analizaron los puntos de unión que se generan al ejecutar el código 3.11 y con base en ello se generaron los cortes mostrados en el código 3.13, en donde se atrapan los valores que están retornando las funciones que se aplicaron al *stream* y se muestran.

Código 3.13: Cortes sobre un Stream

```
1 public aspect StreamsAspect {
```

```
2  pointcut uno():
3   call(java.util.stream.Stream
    java.util.stream.Stream.map(java.util.function.Function));
4  pointcut dos():
5   call(java.util.stream.Stream
    java.util.stream.Stream.distinct());
6  pointcut tres():
7   call(java.util.stream.Stream
    java.util.stream.Stream.flatMap(
    java.util.function.Function));
8  pointcut cuatro():
9       call(java.util.stream.Stream java.util.List.stream());
10  after() returning(Object r) :uno(){
11    System.out.println(thisJoinPoint +
      "\n Retorno: "+r.getClass());
12  }
13  after() returning(Object r) :dos(){
14    System.out.print(thisJoinPoint +
      "\n Retorno: "+r.getClass());
15  }
16  after() returning(Object r) :tres(){
17    System.out.println(thisJoinPoint +
       "\n Retorno: "+r.getClass());
18  }
19  after() returning(Object r) :cuatro(){
20    System.out.println(thisJoinPoint +
      "\n Retorno: "+r.getClass());
21  }
22 }
```

El resultado de la ejecución del código 3.13 es :

```
call(Stream java.util.List.stream())
 Retorno: class java.util.stream.ReferencePipeline$Head
call(Stream java.util.stream.Stream.map(Function))
 Retorno: class java.util.stream.ReferencePipeline$3
call(Stream java.util.stream.Stream.flatMap(Function))
 Retorno: class java.util.stream.ReferencePipeline$7
call(Stream java.util.stream.Stream.distinct())
 Retorno: class java.util.stream.DistinctOps$1
```

Se analizó la opción de sustituir alguno de los valores de retorno por otro para cambiar los valores, pero esta opción queda descartada ya, que ninguna de las clases que se muestran en los resultados de la ejecución está disponible en la bibliotecas de Java, son clases que Java maneja de manera interna por lo que no es posible crear un objeto de alguna de ellas y sustituirlo.

Métodos estáticos en interfaces

Otra novedad que presenta Java 8 es la creación de métodos *default* y métodos estáticos dentro de las interfaces. Para ocupar el primer tipo de métodos es necesario crear un objeto

de esa interfaz y llamar al método de la interfaz, los métodos estáticos se llaman mediante el nombre de la interfaz y el nombre del método estático.

En el código 3.14 se muestra un ejemplo del empleo de un método estático y un método *default*:

Código 3.14: Empleo de métodos estáticos y *default* en interfaces

```
1   interface Inter1 {
2        default void m1(){
3         System.out.println("Metodo default");
4        }
5        public static void m2(){
6         System.out.println("Metodo estatico");
7        }
8   }
9   public class Interfaces implements Inter1 {
10       public static void main(String args[]){
11        new Inter1().m1();
12          Inter1.m2();
13       }
14  }
```

Se analizaron los puntos de unión que se generan al usar métodos estáticos y métodos *default* en las interfaces y se observó que para los métodos estáticos se crea un objeto del tipo de la interfaz, los métodos *default*, una vez que se declara la implementación de la interfaz en una clase, estos métodos pasan a formar parte de esa clase y se invocan por medio de un objeto de ella.

```
initialization(paq6.Inter1())
call(void paq6.Interfaces.m1())
call(void paq6.Inter1.m2())
```

Aplicación de *Reflection*

Una opción que permite observar las clases internas generadas a tiempo de ejecución junto con sus métodos, constructores y variables declaradas es usar la API de Java *Reflection*, con ella se observa todo lo que contiene la clase con el comportamiento del objeto funcional.

La ventaja que tiene *Reflection* es que funciona a tiempo de ejecución, cuando todo el código para el funcionamiento del objeto funcional ya está generado.

En el código 3.15 se muestra un programa en donde usando *Reflection* se visualizan los métodos, variables y constructores de una función anónima.

Código 3.15: Aplicación de *Reflection* en Java

```
1  import java.lang.reflect.*;
2  interface Suma{
3   public int m(int x, int y);
4  }
5  public class Main {
```

```
6   public static void main(String[] args) {
7    Suma a =(x, y)->  x + y;
8    Main b = new Main();
9    System.out.println("Clase principal: "+b.getClass());
10   System.out.println("Objeto funcional:"+a.getClass());
11   System.out.println("Clase anónima:"
      +a.getClass().isAnonymousClass());
12   Class<?> obj = a.getClass();
13   final Method[] metodos = obj.getDeclaredMethods();
14   System.out.println("Métodos");
15   if(metodos.length<1){
       System.out.println(" Sin métodos");}
16   else{
17    for (final Method metodo : metodos) {
18    System.out.println("  Método: " + metodo.getName());
19    Type[] tipos = metodo.getGenericParameterTypes();
20    System.out.print("  Parámetros:");
21    if(tipos.length<1){System.out.print("Sin parámetros");}
22      else{
23       for (Type tipo : tipos) {
24        System.out.print("  "+tipo.getTypeName());
25       }
26      }
27    System.out.println("\n  Tipo de retorno:" +
       metodo.getGenericReturnType().getTypeName());
28     }
29    }
30   System.out.println("Contructores");
31   if(metodos.length<1){
      System.out.println("Sin contructores");}
32    else{
33     final Constructor[] todosLosConstructores
         = obj.getDeclaredConstructors();
34      for (final Constructor<?>
         constructor : todosLosConstructores) {
35       System.out.println("Constructor:"+
         constructor.getName());
36       Type[] tipos = constructor.getGenericParameterTypes();
37       System.out.println("  Parámetros:");
38       if(tipos.length<1){System.out.print("Sin parámetros");}
39      else{
40       for (Type tipo : tipos) {
41        System.out.println("    "+tipo.getTypeName());
42       }
43      }
44     }
45    }
46   System.out.println("Campos");
47   Field[] vars = obj.getDeclaredFields();
48   if(vars.length<1){System.out.println("Sin campos");}
49   else{
50    for (Field variable : vars) {
```

```
51     String nomvar = variable.getName();
52     System.out.println("  Campo: " + nomvar);
53     Type tipo = variable.getGenericType();
54     String nombreTipoVariable = tipo.getTypeName();
55     System.out.println("  Tipo: " + nombreTipoVariable);
56     }
57    }
58   }
59 }
```

La ejecución del programa mostrado en el código 3.15 da como resultado el código 3.16.

Código 3.16: Resultados de la ejecución del código 3.15

```
Clase principal: class Anotaciones.Main
Clase objeto funcional:
  class Anotaciones.Main$$Lambda$1/168423058
Es clase anónima: false
Métodos
  Método: m
  Parámetros:  int  int
  Tipo de retorno: int
Contructores
  Constructor: Anotaciones.Main$$Lambda$1/168423058
  Parámetros: Sin parámetros
Campos
  Sin campos
```

En los resultados se observa que se generó una clase llamada *Anotaciones.Main$$ Lambda$1/168423058*, la cual contiene un método llamado *m*, que es el que se declaró en la interfaz funcional, además del constructor de la clase que no recibe parámetros. *Reflection* trabaja a tiempo de ejecución para indagar el contenido de las clases y así observar su contenido, por lo que es útil para indagar en las clases internas de los objetos funcionales.

3.2.2. Scala

En la siguientes secciones se presentan distintos tipos de objetos funcionales y se analizan las capacidades de corte que las anotaciones de AspectJ tienen sobre ellos.

Funciones

Los objetos funcionales en Scala se trabajan mediante diferentes formas. Scala cuenta con los *traits Function1* al *trait Function22* para definir funciones desde un argumento hasta veintidós, todas las funciones heredan de estos *traits*, ya sea que se utilice la palabra reservada *extends* seguido del *trait* que corresponda a su número de párametros junto con sus valores de entrada y de retorno, o si no se indica la herencia de manera explicita a algún *trait*, Scala lo hace de manera autómatica.

El código 3.17 muestra la aplicación de los objetos funcionales, en donde se emplea una clase *singleton* que será el objeto funcional con su método *apply*, que es el método que de manera predeterminada utilizará la función para implementar su comportamiento, a su vez la clase *Suma* hereda del *trait Function2* que representa una función que recibe dos parámetros y retorna uno, en el caso de este ejemplo recibe dos enteros y retorna uno *Function2[Int, Int, Int]*, siendo el ultimo *Int*, el indicador del valor de retorno.

Código 3.17: Objeto funcional en Scala

```
1  object Test{
2    def main(args: Array[String]){
3      println(Suma(20, 30));
4    }
5  }
6  object Suma extends Function2[Int, Int, Int] {
7    def apply(x: Int, y:Int): Int = x + y
8  }
```

También es posible ocupar los objetos funcionales mediante clases (*class*) en lugar de clases *singleton* (*object*).

Para la compilación y ejecución de los ejemplos se usó la herramienta de construcción SBT [33], además de usar las anotaciones de AspectJ [34], que sirven para aplicar la programación orientada a aspectos a programas escritos en el lenguaje Scala. Al ser Scala un lenguaje que se interpreta en la máquina virtual de Java, todos los archivos con extensión *.scala* al compilarse dan como resultado uno o más archivos con extensión *.class* por lo que también se obtienen los puntos de unión que las clases contienen con los mismos procedimientos que se mostraron en la sección 3.2.1 Java 8. Al compilar el código 3.17 se crean cuatro archivos con extensión *.class*, llamados *Test*, "*Test$*", *Suma* y *Suma$*, las clases que no llevan el símbolo *$* funcionan como clases envolventes para las clases con el mismo nombre que sí tienen el símbolo, la función de la clase *Test* es llamar al método principal de la clase *Test$*, la función de la clase *Suma* es llamar mediante su método *apply* a la clase *Suma$*.
Cuando se tienen dos clases con el mismo nombre, que se compilaron de una misma clase del código fuente y una de ellas tiene en su nombre el símbolo "*$*", es a esta clase a la que se deben aplicar los aspectos, ya que en ella se encuentra todo el comportamiento del programa, la que no lleva el símbolo "*$*" en su nombre funciona como tipo envolvente de la primera.

Dado que el lenguaje Scala está hecho para reducir el codígo que los programadores escriben, siempre hay código que no se muestra a simple vista, por eso para aplicar los cortes es necesario revisar el *bytecode* de Java y ver a cuál instrucción corresponde cada uno de los puntos de unión que muestra AspectJ para conocer los nombres, escribir correctamente las firmas y realizar los cortes. Un par de puntos de unión que AspectJ es capaz de detectar para la función *Suma* del código 3.17 y su correspondiente nombre en *bytecode* se muestran en la Tabla 3.2:

Tabla 3.2: Comparación de puntos de unión mostrados por AspectJ y nombres mostrados en el *bytecode*

Punto de unión	**Nombre en el *bytecode***
get(Suma. paq.Suma..MODULE$)	paq/Suma$.MODULE$:Lpaq/Suma$
call(int paq.Suma..apply$mcIII$sp(int, int))	paq.Suma$.apply$mcIII$sp:(II)I

Una vez que se conocen los puntos de unión existentes en el código y cuál es su nombre completo en *bytecode* es posible generar las firmas colocando los correspondientes nombres de las clases y cambiando por comodines los símbolos que AspectJ no admite. Un ejemplo de un corte para los puntos de unión mostrados en la tabla 3.2 se muestra en el código 3.18.

Código 3.18: Aplicación de cortes al objeto funcional del Código 3.17

```
1  @Aspect
2  class TestAspect {
3    @Before("get(paq.Suma$ paq.Suma$.MODULE*)")
4     def uno(joinPoint: JoinPoint) = {
5       println("MODULE")
6     }
7    @Around("call(* paq.Suma$.apply$mcIII$sp(..)) && args(x, y)")
8     def dos(joinPoint: ProceedingJoinPoint,
      x:Int, y:Int):Any = {
9             var a:Array[Object]=new Array[Object](2)
10           val z = new Integer(x+1)
11           val z2 = new Integer(y+2)
12           a(0)=z
13           a(1)=z2
14           joinPoint.proceed(a)
15    }
16   @AfterReturning(
17    pointcut = "call(* paq.Suma$.apply$mcIII$sp(..))",
18    returning= "result")
19     def tres(joinPoint:JoinPoint, result:Object) {
20           println("Retorno : " + result);
21   }
22 }
```

El corte *uno* es para el campo *module*, que es un objeto de tipo *suma*, el corte *dos* sirve para cambiar el valor que recibe la función, se capturan los valores pero para mandarlos la función sólo recibe un arreglo de objetos por lo que es necesario crear el arreglo y llenarlo con los valores nuevos. El corte *tres* captura el valor de retorno de la función y lo muestra. Para la clase *Suma$* los puntos de unión que AspectJ detecta son:

```
1  staticinitialization(principal.Suma..<clinit>)
2  call(principal.Suma.())
3  preinitialization(principal.Suma.())
4  initialization(principal.Suma.())
```

```
5  initialization(scala.Function2.mcIII.sp())
6  initialization(scala.Function2())
7  execution(principal.Suma.())
8  set(Suma. principal.Suma..MODULE\$)
9  call(void scala.Function2.class.\$init\$(Function2))
10 call(void scala.Function2.mcIII.sp.class.\$init\$(Function2.mcIII.sp)
   )
11 execution(int principal.Suma..apply\$mcIII\$sp(int, int))
```

En los puntos de unión arriba mostrados se aprecian todos los relacionados con la incialización de la clase *Suma*, además los puntos 9, 10 y 11 pertenecen a las llamadas y ejecución de los métodos que realizan las operaciones.

Funciones anónimas

Otra manera de crear los objetos funcionales en el lenguaje Scala es mediante las funciones anónimas como se muestra en el código 3.19.

Código 3.19: Función anónima en el lenguaje Scala

```
1  object Test{
2    def main(args: Array[String]){
3      val Sum = (z:Int , y:Int) => z + y
4      println(Sum.apply(30, 40))
5      println(Sum(30, 40))
6    }
7  }
```

Al compilar el código 3.19 se observa que se generan para la clase Test tres archivos con extensión *.class* llamados *Test*, *Test$* y *Test$$anonfun$1*, esta última clase es una clase interna dentro de la clase *Test$* y contiene la implementación de la función anónima. El problema que surge es con respecto a la clase interna *Test$$anonfun$1*, en donde AspectJ no es capaz de detectar ningún punto de unión, aunque al revisar el *bytecode* de esa clase se observa que contiene varios métodos y campos. Los puntos de unión que AspectJ encuentra para la clase textitTest del código 3.19 son los siguientes:

```
1 call(principal.Main..anonfun.1())
2 principal/Main\$\$anonfun\$1."<init>":()V
3 call(int scala.Function2.apply\$mcIII\$sp(int, int))
4 scala/Function2.apply\$mcIIsp:(II)I
```

En donde el primer punto de unión es la llamada al constructor de la clase interna, este tipo de clases se generan automáticamente una por cada función anónima que se utilice. El segundo punto de unión es la llamada al método *apply*, a la firma de este método internamente se le agregan los valores de entrada y de retorno, en este caso tres enteros, por eso en el nombre del método aparecen tres letras I juntas.

Un ejemplo de cortes para los puntos de unión antes mostrados se ve en el código 3.20:

Código 3.20: Aplicación de cortes a la función anónima del código 3.19

```
1   @Aspect
2   class TestAspect {
3     @Before("call(principal.Main$$anonfun$1.new())")
4     def uno(joinPoint: JoinPoint) =
5       println("anonfun.new")
6     @After("call(* scala.Function2.apply*(..))")
7     def dos(joinPoint:JoinPoint) =
8             println("Apply");
9   }
```

El corte *uno* aplica para la llamada al constructor de la clase interna, el corte *dos* es para la aplicación del aviso *after* sobre el método *apply*.

Al detectar Scala la sintaxis de una función anónima, se hace la llamada a una nueva clase interna nombrándola como la clase externa y agregándole el sufijo *$$anonfun$* seguido de un número según clases internas anónimas se tenga en la clase, en este caso es sólo 1. La clases internas generadas hacen uso de la clase *AbstractFunction*, esta clase de la biblioteca *scala.runtime* permite generar el comportamiento de las clases anónimas a tiempo de ejecución.

Probando con diversos objetos funcionales, con variados tipos de entrada y de retorno, se observó que el método *apply* toma distintos nombres de acuerdo a los valores que reciba y devuelva el método, en la tabla 3.3 se muestran algunos de los distintos nombres que toma este método.

Tabla 3.3: Distintos nombres del método *apply*

Datos de entrada	**Nombre del método**
Int	Function1.apply$mcII$sp(int)
Int, Int	Function2.apply$mcIII$sp(int, int)
Int, Int, Int	Function3.apply(Object, Object, Object)
Float	Function1.apply$mcFF$sp(float)
Float, Float	Function2.apply(Object, Object)
String	Function1.apply(Object)
Double	Function1.apply$mcDD$sp(double)
Double, Double, Double	Function3.apply(Object, Object, Object)
Char	Function1.apply(Object)
Boolean	Function1.apply(Object)
Long	Function1.apply$mcJJ$sp(long)
Long, Long	Function2.apply$mcJJJ$sp(long, long)
Long, Long, Long	Function3.apply(Object, Object, Object)
Byte	Function1.apply(Object)
Short	Function1.apply(Object)

Funciones de orden superior

Scala proporciona soporte para las funciones de orden superior, que son funciones que toman otras funciones como parámetros como se muestra en el código 3.21.

Código 3.21: Función de orden superior en Scala

```
1  object Test {
2    def main(args: Array[String]) {
3       println( suma( resultado, 10, 20) )
4    }
5    def suma(x: Int => String, y : Int, z : Int) = x(y + z)
6    def resultado[A](y: A) = "Resultado =" + y.toString()
7  }
```

En el código 3.21 se muestra una función de orden superior en donde a la función *Suma* se le manda como parámetro la función *resultado*. Los puntos de unión relevantes de este objeto funcional son:

```
1 call(String principal.Main..suma(Function1, int, int))
2 execution(String principal.Main..suma(Function1, int, int))
3 call(Object scala.Function1.apply(Object))
4 execution(String principal.Main..resultado(Object))
```

En donde los puntos de unión 1 y 2 corresponden a la función *suma* y los siguientes (3 y 4) a la función *resultado*. El parámetro que recibe la primera función es de tipo *Function1*. Un ejemplo de un aspecto para capturar la función que se le pasa como parámetro a la función *suma* se muestra en el código 3.22.

Código 3.22: Aplicación de un corte a una función de orden superior

```
1  @Aspect
2  class MethodLogger {
3    @After("execution(String principal.Main$.Suma
     (scala.Function1, int, int)) && args(x, y, z)")
4    def uno(x:principal.Main$$anonfun$main$1,
     y:Int, z:Int, joinPoint: JoinPoint) = {
5      println(x.getClass)
6    }
7  }
```

En el corte *uno* al momento de declarar la firma del método se indica el tipo de la función, y en este caso su tipo es *scala.Function1*, ya que la función hereda de ese *trait*, al momento de recibir ese parámetro en el método del aviso éste es del tipo de la clase que a su vez está implementando al *trait scala.Funtion1*.

Funciones currificadas

Las funciones currificadas en Scala se emplean como se muestra en el código 3.23:

Código 3.23: Función currificada en Scala

```
1 object Main{
2  def main(args: Array[String]){
3   println(sumacur(5)(10))
4  def sumacur(c1: Int)(c2: Int) = {
5   c1 + c2
6   }
7  }
8 }
```

En donde los valores que recibe la función se envían a está como si fueran dos funciones. Se analizó el *bytecode* de esa clase y se observó que el método que recibe la función es igual a los métodos de otras funciones que no están currificadas. A continuación se muestra el nombre del método de la función currificada en *bytecode*:

```
principal/Main$.sumacur$1:(II)I
```

La función currificada se cambió por la función:

```
def sumacur(c1: Int, c2: Int) = {
   c1 + c2
}
```

Se observó que tiene el mismo comportamiento y el nombre del método en *bytecode* fue exactamente el mismo:

```
principal/Main$.sumacur$1:(II)I
```

Por lo que no hay ninguna diferencia entre los puntos de unión que AspectJ es capaz de detectar si la función está currificada o no:

```
call(int principal.Main..sumacur$1(int, int))
execution(int principal.Main..sumacur$1(int, int))
call(Integer scala.runtime.BoxesRunTime.boxToInteger(int))
```

Así se declaran aspectos para los puntos de unión de la misma manera como se hicieron en los ejemplos arriba mostrados (códigos 3.22, 3.20, 3.7):

Código 3.24: Corte a una función currificada

```
1  @Aspect
2  class MethodLogger {
3   @Around("execution(int
     principal.Main$.sumacur$1(int, int)) && args(x, y)")
4     def m(joinPoint: ProceedingJoinPoint, x:Int, y:Int):Any = {
5         println("Valores: "+x+" "+y)
6         joinPoint.proceed()
7   }
8  }
```

Streams

Los *streams* en Scala permiten el manejo de listas de manera perezosa, es decir, que se retrasa el cálculo de los valores hasta que éstos son necesarios, a su vez que es posible

aplicar diversas operaciones sobre ellos. Un ejemplo de un *stream* en Scala se muestra en el código 3.25:

Código 3.25: Stream en Scala

```
1 object Streams{
2  def main(args: Array[String]){
3    val str = 1 #:: 2 #:: 3 #:: 3 #:: Stream.empty
4    val str2 = str.filter{ _ > 2 }.map{ _ * 2}.distinct
5  }
6 }
```

En el código 3.25, en la línea 3 se crea un *stream* que contiene números del 1 al 3, para finalizar el *stream* es necesario colocarle un elemento vació, en la línea 4 se le aplican diversas operaciones al *stream*, primero un filtro para que seleccione los valores mayores a 2, después una función para que multiplique por 2 los valores del *stream*, por último una operación que solo obtiene los elementos no repetidos, el resultado de estas 3 operaciones se guardan en otro *stream*. Después de la ejecución del código los archivos *.class* que se generaron se muestran a continuación.

```
1 Streams
2 Streams$
3 Streams$$anonfun$1
4 Streams$$anonfun$2
5 Streams$$anonfun$3
6 Streams$$anonfun$3$$anonfun$apply$1
7 Streams$$anonfun$3$$anonfun$apply$1$$anonfun$apply$2
8 Streams$$anonfun$3$$anonfun$apply$1$$anonfun$apply$2$$anonfun
  $apply$3
```

En la lista anterior se observa la gran cantidad de clases internas que genera la compilación del código de un *stream*, en el caso de las últimas tres clases de las lista, estas son clases internas que se encuentran dentro de la clase interna número 5 llamada *Streams$$anonfun$3*. Cabe mencionar que las clases números 3 y 4 de la lista anterior pertenecen al primer *stream* del código 3.25 y del número 5 al 8 corresponden al segundo *stream*.

En la siguiente lista se muestran los puntos de unión que AspectJ es capaz de detectar sobre el código 3.25, en donde no es posible aplicar un corte dada la gran cantidad de métodos que emplea y que no se encuentran disponibles para su uso por el programador y así se repliquen y cambien en un aspecto, así como también por la gran cantidad de clases internas que poseen las implementaciones de *streams* y que estas clases no se soportan por AspectJ.

```
1  call(principal.Streams..anonfun.3())
2  call(Stream.ConsWrapper scala.collection.immutable.Stream.
   .consWrapper(Function0))
3  call(Integer scala.runtime.BoxesRunTime.boxToInteger(int))
4  call(Stream scala.collection.immutable.Stream.ConsWrapper.
   $hash$colo$colon(Object))
5  call(principal.Streams..anonfun.1())
6  call(Stream scala.collection.immutable.Stream.filter(Function1))
```

```
7  call(principal.Streams..anonfun.2())
8  get(Stream. scala.collection.immutable.Stream..MODULE$)
9  call(CanBuildFrom scala.collection.immutable.Stream.
   .canBuildFrom())
10 call(Object scala.collection.immutable.Stream.map(Function1,
   CanBuildFrom))
11 call(Stream scala.collection.immutable.Stream.distinct())
```

Aplicación de *Reflection* en Scala

El código 3.26 usa la API de Scala *Reflection* para acceder a la clase generada a tiempo de ejecución de una función anónima y ver los métodos, constructores y campos que contiene:

Código 3.26: Aplicación de *Reflection* en Scala

```
1  import scala.reflect.runtime._
2  object Test{
3   def main(args: Array[String]){
4    val sum = (z:Int , y:Int) => z + y
5    println("Clase Principal: "+Test.getClass)
6    println("Clase Anonima: "+Sum.getClass)
7    println("Aplicando Reflection")
8    println("Metodos")
9    val m = Sum.getClass.getMethods
10   if(m.length < 1){println("Sin metodos")}
11    else{
12   for( a <- m ){
13        println( "Nombre del metodo: " );
14        println("   "+a );
15        val a2 = a.getParameters
16        print("   Parametros: ")
17        if(a2.length<1){print(" Sin parametros")}
18        else{
19         for( a3 <- a2 ){
20          print("  "+a3)
21         }
22        }
23        val a4 = a.getReturnType
24        println("\n   Tipo de retorno: "+a4)
25    }
26    }
27    val c = Sum.getClass.getConstructors
28    println("Constructores")
29    if(c.length < 1){println("Sin constructores")}
30    else{
31     for( b <- c ){
32       println( "  Constructor: " + c );
33       println("  Tipo: "+b.getParameters)
34    }
35    }
```

```
36    val v = Sum.getClass.getDeclaredFields
37    println("Campos")
38    if(v.length < 1){println("Sin campos")}
39    else{
40    for( d <- v ){
41       println( "  Campo: " + v );
42       println("  Tipo: "+d.getType)
43    }
44    }
45  }
46  }
```

La ejecución del código 3.26 da como resultado:

```
Clase Principal: class Test$
Clase Anónima: class Test$$anonfun$1
Aplicando Reflection
Métodos
Nombre del método:
   public final int Test$$anonfun$1.apply(int,int)
   Parámetros:   int arg0  int arg1
   Tipo de retorno: int
Nombre del método:
   public final java.lang.Object
      Test$$anonfun$1.apply(java.lang.Object,java.lang.Object)
   Parámetros:   java.lang.Object arg0  java.lang.Object arg1
   Tipo de retorno: class java.lang.Object
Nombre del método:
   public int Test$$anonfun$1.apply$mcIII$sp(int,int)
   Parámetros:   int arg0  int arg1
   Tipo de retorno: int
Nombre del método:
   public
     java.lang.String scala.runtime.AbstractFunction2.toString()
   Parámetros:  Sin parámetros
   Tipo de retorno: class java.lang.String
Nombre del método:
   public
     scala.Function1 scala.runtime.AbstractFunction2.curried()
   Parámetros:  Sin parámetros
   Tipo de retorno: interface scala.Function1
Nombre del método:
   public
     scala.Function1 scala.runtime.AbstractFunction2.tupled()
   Parámetros:  Sin parámetros
   Tipo de retorno: interface scala.Function1
Nombre del método:
   public boolean
     scala.runtime.AbstractFunction2.apply$mcZDD$sp(double,double)
   Parámetros:   double arg0  double arg1
   Tipo de retorno: boolean
Nombre del método:
   public final void java.lang.Object.wait()
```

```
     throws java.lang.InterruptedException
   Parámetros:  Sin parámetros
   Tipo de retorno: void
Nombre del método:
   public final native void java.lang.Object.wait(long)
   throws java.lang.InterruptedException
   Parámetros:   long arg0
   Tipo de retorno: void
Nombre del método:
   public boolean java.lang.Object.equals(java.lang.Object)
   Parámetros:   java.lang.Object arg0
   Tipo de retorno: boolean
Nombre del método:
   public native int java.lang.Object.hashCode()
   Parámetros:  Sin parámetros
   Tipo de retorno: int
Nombre del método:
   public final native java.lang.Class
     java.lang.Object.getClass()
   Parámetros:  Sin parámetros
   Tipo de retorno: class java.lang.Class
Nombre del método:
   public final native void java.lang.Object.notify()
   Parámetros:  Sin parámetros
   Tipo de retorno: void
Nombre del método:
   public final native void java.lang.Object.notifyAll()
   Parámetros:  Sin parámetros
   Tipo de retorno: void
Constructores
  Constructor: [Ljava.lang.reflect.Constructor;@14ae5a5
  Parámetros: [Ljava.lang.reflect.Parameter;@7f31245a
Campos
  Campo: [Ljava.lang.reflect.Field;@6d6f6e28
  Tipo: long
```

Como se observa la clase del objeto funcional cuenta con gran cantidad de métodos que no se implementaron manualmente, se le agregan métodos para el manejo del hilo de ejecución como *wait* y también para dar funcionalidades como *hashCode*, entre otros. Se omitieron más de diez métodos dado que son más métodos *apply* que heredan de la clase *Function2* y que sirven para recibir más combinaciones de parámetros *int, long, double* y *object*. Con el uso de *Reflection* se visualizaron los campos, métodos y constructores que contienen las clases que se generan a tiempo de compilación.

3.3. Alternativas de solución a AspectJ para aplicar aspectos sobre funciones y objetos funcionales

El lenguaje Scala cuenta con algunos mecanismos que permiten igualar conceptos de la programación orientada a aspectos.

Mónadas

En [35] se muestran las semejanzas entre la programación con mónadas y la programación orientada a aspectos, ya que las mónadas permiten cambiar el comportamiento de un programa. En esta sección se muestra cómo aplicar mónadas en el lenguaje Scala para asimular mecanismos de la programación orientada a aspectos sobre las funciones.

Traits

Mediante el uso de composición de *Mixin* es posible agregar nuevos comportamientos a las clases, también usando este mecanismo es posible igualar algunos conceptos de la programación orientada a aspectos.
Para usar la composición es necesario que todos los *traits* extiendan de un *trait* en común, de ese *trait* en común el resto de los *traits* que conformarán la composición deberán implementar un método de él. También el objeto al que se quiere agregar más comportamiento es necesario que implemente uno o más *traits* que tendrán el comportamiento a agregar. Así es posible generar comportamientos similares a los avisos *before*, *around* y *after* sobre la ejecución de una función.
Como ejemplo se muestra el código 3.27, en donde se tiene una clase llamada *C*, que contiene un método *apply()*, que recibe una cadena de caracteres y le concatena otra para después imprimirla, también se tienen dos *traits* llamados *T1* y *T2*, cada uno hará las funciones de un aviso.
El diagrama del código 3.27 se muestra en el gráfico 3.3, en donde se aprecia de qué *traits* hereda comportamiento la función, así como también de qué *trait* heredan todos los demás.

Código 3.27: Composición de *Mixin* sobre un objeto funcional

```
1  trait T1 extends Function1[String,Unit] {
2  abstract override def apply(s: String):Unit = {
3    println("Before " + s)
4    super.apply(s)
5   }
6  }
7  trait T2 extends Function1[String,Unit] {
8  abstract override def apply(s: String):Unit = {
9     super.apply(s)
10     println("After " + s)
11  }
12 }
13 class C extends Function1[String, Unit] {
14   def apply(s: String): Unit = {
```

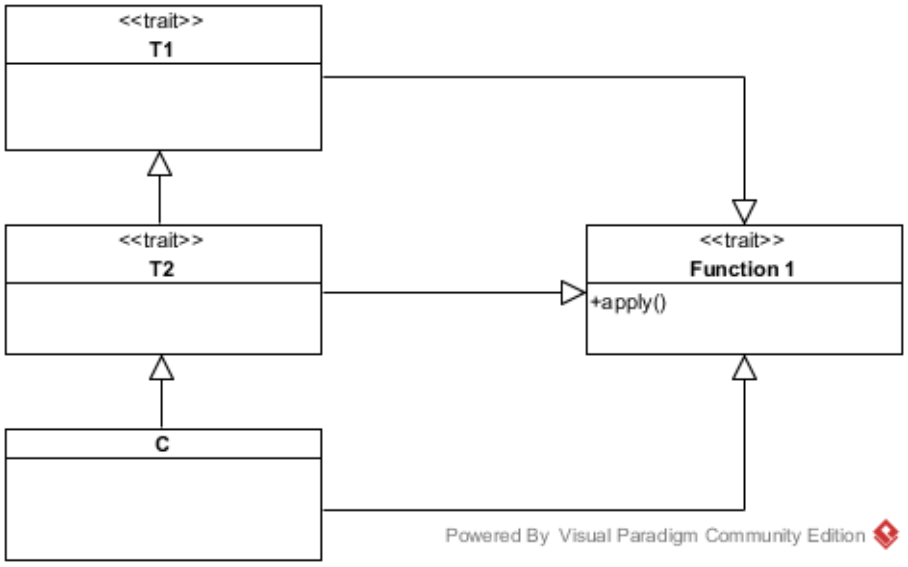

Figura 3.1: Diagrama *Mixin Composition*

```
15      println(s + " Mundo ")
16    }
17  }
18  object MixinComp {
19    def main(args: Array[String]){
20     val obj = new c with t2 with t1
21     obj.apply("Hola")
22   }
23  }
```

La ejecución del código 3.27 da cómo resultado:

```
Before Hola
Hola Mundo
After Hola
```

En el código 3.28 se observa que se agregó comportamiento a la función original del objeto funcional. Así en lugar de sólo mostrar la frase *Hola Mundo*, se le agregaron dos avisos, uno antes y uno después de la ejecución de la función. Para implementar un aviso *around* mediante *traits* se hace de la misma forma sólo que en la sobre escritura del método *apply()* no se emplea el método *super()*.

Código 3.28: Composición de *Mixin* sobre un objeto funcional

```
1 trait T3 extends Function1[String,Unit] {
2 abstract override def apply(s: String):Unit = {
3     println("Around " + s)
4   }
5 }
```

La ventaja que tiene la técnica del Mixin Composition con respecto a AspectJ es que sólo usa código Scala, no es necesario nada más. La desventaja es que solo se puede usar sobre clases no sobre *objects* y AspectJ se usa sobre ambos.

Javassist

Javassist [36] es un marco de trabajo que permite modificar y agregar nuevo *bytecode* a los programas. Entre las funciones que tiene este marco de trabajo está el permitir agregar nuevas clases a los programas, agregar nuevos métodos a las clases, agregar herencia a las clases, también permite modificar el *bytecode* de los métodos colocando código antes o después del cuerpo de los métodos y también sustituyendo el código del cuerpo del método por otro.

Los métodos de Javassist *inserbefore* e *insertAfter* permiten agregar instrucciones antes o después del cuerpo del método, estas dos instrucciones son usadas para aplicar cortes y colocar avisos, ya que permiten modificar la ejecución normal del programa, si los comparamos con AspectJ se tendrían los avisos *after* y *before* junto con la primitiva de corte *execution*. El método de Javassist *replace* permite cambiar el cuerpo de un método, comparando este método con AspectJ es similar al aviso *around* sin el uso de la variable *proceed* y junto con la primitiva de corte *execution*.

En el código 3.29 se muestra una lambda ubicada en el paquete llamado *Test* y en la clase llamada *Main*.

Código 3.29: Ejemplo de uso de Javassist

```
1  package Test;
2  public class Main {
3   public static void main(String args[]){
4                Suma a = new Suma() {
5                        public void m(int x, int y) {
6                                System.out.println(x + y);
7                        }
8                };
9                a.m(20, 10);
10  }
11 }
12 interface Suma {
13       public void m(int x, int y);
14 }
```

Para modificar el código 3.29 usando Javassist el proceso se realiza como se muestra en el código 3.30. En donde en la línea 4 se crea el *pool*, que es el objeto que obtiene representaciones de las clases ya compiladas, en la línea 5 se obtiene la clase a modificar, es necesario ser cuidadosos con la clase a modificar, ya que el comportamiento del objeto funcional queda en otra clase, por lo que se debe colocar el nombre correcto, de lo contrario no encontrará la clase, o de encontrarla no estará adentro el método que se esté buscando,

en la línea 6 del código se obtiene un método de la clase para modificarlo, en la línea 7 se usa el método *defrost* para tener acceso a la clase y modificarla, en la línea 8 se usa el método *insertBefore* para agregar un fragmento de código antes del código original del método, en la línea 9 se usa la instrucción *insertAfter* para agregar instrucciones después de la ejecución del método, por último en la línea 10 se guardan los cambios realizados a la clase.

Código 3.30: Ejemplo de uso de Javassist

```
1  public class JavassistPrueba {
2    public static void main(String[] args) {
3     try{
4         ClassPool pool = ClassPool.getDefault();
5         CtClass cc2 = pool.get("Test.Main$1");
6         cc2.defrost();
7         CtMethod m2 = cc2.getDeclaredMethod("m");
8         m2.insertBefore("{System.out.println(\"Before \"+$1);}");
9         m2.insertAfter("{System.out.println(\"After \");}");
10        cc2.writeFile();
11        }catch (NotFoundException e) {
12         e.printStackTrace();
13        } catch (CannotCompileException e) {
14         e.printStackTrace();
15        } catch (IOException e) {
16         e.printStackTrace();
17        }
18   }
19 }
```

Para realizar el reemplazo del cuerpo de un método sobre el código 3.30 se muestra como ejemplo el código 4.6, en donde se usa el método de Javassist *replace* para sustituir todo el comportamiento del objeto por otro.

Código 3.31: Ejemplo de reemplazo de un método

```
1   m2.instrument(new ExprEditor() {
3     public void edit(MethodCall m) throws CannotCompileException {
3       m.replace("{System.out.println($1 * $2);}");
4     }
5   });
```

Las modificaciones que realiza Javassist, al realizarse a nivel de *bytecode* son aplicables a programas escritos en lenguaje Scala, en el código 3.32 se muestra un objeto en Scala que ejecuta una función.

Código 3.32: Ejemplo de reemplazo de un método

```
1  object Test{
2    def main(args: Array[String]) {
3     A("Hello", "World");
4    }
5  }
6  object A {
```

```
7     def apply(x: String, y:String): Unit = println(x + " " + y)
8   }
```

En el código 3.33 se muestra cómo obtener la clase mostrada en el código 3.32 y que está escrita en lenguaje Scala, también se observa cómo obtener todos los métodos en un arreglo y recorrer ese arreglo, de esa forma se observan los nombres completos de los métodos *apply* y se selecciona el que se quiera modificar.

Código 3.33: Ejemplo de obtención de métodos

```
1  CtClass cc2 = pool.get("Test.A\$");
2  CtMethod[] v = cc2.getMethods();
3  for(int i=0; i<v.length; i++){
4    System.out.println(v[i]);
5  }
```

En el código 3.34 se muestra una función anónima en lenguaje Scala.

Código 3.34: Ejemplo de reemplazo de un método

```
1  object Test {
2   def main(args: Array[String]) {
3     val a = (z: String, y: String) => println(z + " " + y)
4     a.apply("Hello", "World")
5   }
6  }
```

La modificación de la función anónima del código 3.34 usando Javassist y Scala se muestra en el código 3.35, a modo de ejemplo se muestra el uso de los tres métodos, *before, after* y *replace*, aunque al ejecutar ese código el último método sobre escribiría a los otros dos.

Código 3.35: Ejemplo de Javassist en Scala

```
1  object Test{
2    def main(args: Array[String]) {
3      var pool = ClassPool.getDefault();
4      var cc2 = pool.get("Test.Main$1");
5      cc2.defrost();
6      var m2 = cc2.getDeclaredMethod("Test.Test\$\$anonfun\$1");
7      m2.insertBefore("{System.out.println(\"Before \" + $1); }");
8      m2.insertAfter("{System.out.println(\"After \"); }");
9      m2.instrument(new ExprEditor() {
10       def edit(m: MethodCall) {
11         m.replace("{ System.out.println($1 + $2); }");
12       }
13       });
14       cc2.writeFile();
15   }
16  }
```

La ventaja que tiene Javassist es que los códigos que se agreguen antes, después o en lugar del código original se insertan en el *bytecode* dentro del método, caso contrario a lo que sucede con AspectJ, que dentro del método se coloca solamente una llamada a otro

método que contiene el código del aviso.

La desventaja que presenta Javassist con respecto a AspectJ es que es necesario tener conocimientos acerca del *bytecode* y del comportamiento interno de los objetos funcionales para realizar modificaciones.

3.4. Patrones de diseño que facilitan diseños con objetos funcionales

En esta sección se muestra el análisis que se realizó a algunos patrones orientados a objetos y orientados a aspectos para ver si es posible realizar su aplicación de manera práctica sobre programas implementados con objetos funcionales.

3.4.1. Patrones orientados a objetos funcionales en Java

En [37] el autor presentó una lista de patrones de diseño adaptados para usar lambdas y algunos otros mecanismos funcionales, en donde sustituye las funciones generales que realiza el patrón por métodos estáticos, para acciones más específicas utiliza lambdas que en ocasiones se pasan como parámetros para ser ejecutadas en los métodos estáticos. También hace empleo de listas en donde guarda lambdas y usa llamadas a métodos estáticos. Los patrones que se presentan son:

1. Command
2. Strategy
3. Template Method
4. Observer
5. Decorator
6. Chain of Responsability
7. Visitor
8. Interpreter

Se revisó el resto de los patrones de diseño propuestos en [10] para analizar si es posible aplicarlos al uso de objetos funcionales. A continuación se muestra la lista de patrones y si es posible su implementación para trabajar con objetos funcionales:

1. Abstract Factory.- Este patrón no es aplicable dado que cuando se trabaja con lambdas o algún otro tipo de mecanismo funcional no se emplea la creación de familias de objetos.

2. Builder.- No es posible usar el mismo proceso de construcción para distintos objetos funcionales, en todo caso será la misma interfaz funcional pero con distintas implementaciones.

3. Factory Method.- Una variante de este patrón se emplea en la generación de lambdas dado que es un interfaz la que define el comportamiento del objeto mediante la definición de un único método abstracto.

4. Prototype.- El objeto funcional una vez creada su implementación tiene una única instancia, que se usa indistintamente, por lo que este patrón no aplicaría.

5. Flyweiht.- Este patrón sirve para aligerar el estado de los objetos, pero en el caso de los objetos funcionales que no deben mantener un estado, este patrón no es aplicable.

6. Singleton.- Un objeto funcional solo tiene una instancia, se tiene que re implementar la interfaz funcional para tener otro objeto, es posible tener más instancias del mismo objeto funcional aunque no tiene mucho caso eso, ya que el mismo objeto funcional se usa en varias partes sin que se afecte los resultados.

7. Adapter.- Este patrón sí es posible usarlo para que una lambda con un método particular adapte a otra que no posee dicho método.

8. Brigde.- El separar la abstracción de la implementación no es posible en un objeto funcional en Java, por lo que este patrón no aplica.

9. Composite.- Este patrón no aplica, dado que no se hace composición de objetos funcionales o lambdas.

10. Facade.- Este patrón sí se adapta para trabajar en vez de varios subsistemas con varias clases que contengan muchos objetos funcionales.

11. Proxy.- Este patrón es posible aplicarlo entre dos lambdas, que una actué como control de acceso hacia la otra.

12. Iterator.- Un objeto funcional no usa un iterador dado que no implementa los métodos necesarios, pero sí formaría parte de ese patrón y que sean objetos funcionales los que se iteren.

13. Mediator.- Solo en caso de que se estén trabajando con demasiados objetos funcionales este patrón resultaría útil, ya que el objeto mediator pasaría los llamados a otros objetos o incluso pasaría funciones como parámetros a otras funciones.

14. Memento.- Los objetos funcionales no mantienen un estado, así que no necesitarían retornar a un estado anterior.

15. State.- Los objetos funcionales no tiene un estado, así que este patrón no es aplicable dado que no se requiere cambiar su comportamiento de acuerdo a su estado.

3.4.2. Patrones en Scala

En [9] se presenta una gran cantidad de patrones, se muestra la implementación de los patrones definidos en [10], usando *traits* en lugar de interfaces y coincidencia de patrones para reducir el código. Además existen patrones en Scala sólo para la parte funcional entre los que se encuentran:

1. Functor.- En este patrón un contenedor hace referencia a un grupo de objetos y aplica una función a cada uno de ellos y regresando un nuevo contenedor con los objetos modificados.

2. Functor aplicativo.- Esta patrón es un functor, con la diferencia de que se agregan dos funciones, la primera de ellas que toma un objeto en un contexto y envuelve ese objeto en el mínimo contexto, y la segunda función que toma una o más funciones en un contexto y la aplica a los objetos en ese contexto.

3. Monoide.- Este patrón define un conjunto de objetos y una operación para combinarlos.

4. Mónada.- Similar a los monoides, este patrón identifica un contenedor de tipo T que provee un conjunto de operaciones que se agrupan en operaciones como functor, operaciones como functor aplicativo y operaciones como monoide.

5. Reducido.- Este patrón se utiliza cuando se tiene un conjunto de objetos y un requerimiento es reducir ese conjunto a un único objeto pero aplicando una función, ese único objeto representa el resultado final de aplicar la función dada a los primeros dos elementos para después tomar ese resultado y aplicarlo al resto de los elementos.

6. Cierre.- Este patrón se enfoca en un particular nodo de una estructura de árbol, este patrón actúa como un índice en la estructura, así se mueve la estructura con respecto a ese índice.

7. Lente.- Este patrón provee una manera de manejar actualizaciones a objetos inmutables, funciona creando una copia del objeto conteniendo los datos actualizados.

8. Vista.- Este patrón provee una diferente manera de representar una estructura de datos, las vistas representan una transformación de una colección de datos y son construidas aplicando una función a una colección resultado en una nueva colección.

9. Flecha.- Este patrón es una generalización de las mónadas y provee un conjunto de comportamientos que son aplicados a un contenedor.

También en Scala se manejan otros patrones de diseño que están más relacionados con la sintaxis del lenguaje como son:

1. Inmutabilidad.- La inmutabilidad indica que los datos de un objeto una vez creados no se cambian. En el lenguaje Scala se tiene la clase *Inmutable* para representar todos estos elementos.

2. Trait marcador.- Estos *traits* no poseen métodos, funciones o tipos, y son usados como anotaciones para indicar el rol de algunas entidades dentro del dominio.

3. Singleton.- Este patrón describe un tipo que solo tiene una instancia. En Scala se tiene el tipo *object* para definir objetos *Singleton*.

4. Delegación.- Este patrón provee la manera de extender y reusar la funcionalidad de un tipo, sin usar herencia y combinando los objetos de tal manera que un objeto delegue comportamientos a otros objetos subordinados.

Dado que Scala es un lenguaje que maneja los objetos funcionales de manera más natural, todos los patrones orientados a objetos son aplicables para objetos funcionales en Scala, solo es necesario cuidar que no se genere efecto de borde y si se está usando el método *apply* se estén usando los *traits* correctos.

3.4.3. Patrones orientados a aspectos en Java y Scala

Se analizaron los patrones de diseño orientados a aspectos para observar si es posible aplicarlos sobre objetos funcionales en Java y Scala y usando las anotaciones de AspectJ para la creación de los aspectos.

1. Huevo de Cuco.- Este patrón no es posible implementarlo sobre objetos funcionales de la misma manera que se aplica sobre objetos normales, ya que aunque el objeto que se quiere sustituir comparte una misma interfaz funcional, el nombre de las clases del objeto original y el objeto reemplazo son distintos, se encontró una forma de implementar un comportamiento similar al que realiza este patrón, es sustituir la llamada a la clase interna mediante un aviso *around*, tomar los parámetros que recibe el objeto original y pasarlos a una lambda que se encuentre dentro del aspecto, tomar el resultado y retornarlo al objeto original, así el objeto original nunca sabrá que fue intervenido, pero el resultado será el esperado por la modificación.

2. Director.- Aplicar el patrón director sobre un objeto funcional no es posible, ya que no se pueden agregar nuevos roles a un objeto funcional, el objeto tiene solo un método en su comportamiento, y esta es la función del patrón director el agregar un nuevo rol.

3. Política.- Este patrón sí es aplicable al trabajar con objetos funcionales, en el caso de las lambdas una opción es emplearlo para evitar que se creen lambdas de interfaces que no se definieron como interfaces funcionales y se muestren advertencias.

4. Control de Borde.- Este patrón sí es aplicable al trabajo con objetos funcionales dado que se definirían regiones donde se encuentren lambdas o en donde se haga uso de éstas.

5. Control de excepciones.- Si algún objeto funcional durante su ejecución genera una excepción que no estaba prevista, este patrón es útil para el control de la excepción.

6. Objeto trabajador.- Este patrón es posible usarlo para volver concurrentes varios objetos funcionales.

7. Espectador.- Esta patrón se usa para manejar la depuración de programas, por lo que es aplicable a cualquier tipo de objeto, funcional o no.

8. Regulador.- Permite manejar la ejecución de un programa sacando a un aspecto parte del comportamiento, es aplicable al trabajo con objetos funcionales e incluso genera un mayor control del comportamiento del programa, ya que no todo estaría en la clase interna.

9. Parche.- Esta patrón se usa para agregar una nueva funcionalidad a la clase sin modificar el código de la clase, para ello se usa corte estático, por lo que sí es aplicable al trabajo con funciones.

10. Administrador de disponibilidad.- Este patrón se usa aplicado atrás de un patrón Fachada, que se encarga de conectar a un sistema remoto que puede o no estar accesible, por lo que el aspecto se encarga de verificar la disponibilidad o mandar una excepción en caso de que no esté disponible el sistema externo. Este patrón no aplica directamente al trabajo con objetos funcionales, ya que trabaja sobre subsistemas.

11. Corte en punto elemental.- Este patrón permite usar aspectos como plantillas para así reutilizarlos, por lo que se usa independientemente del tipo de objetos que se trabajen en el diseño.

12. Diseño heterárquico.- Este patrón no es aplicable al trabajo con funciones, ya que su fin es mejorar estructuras complejas de objetos separándolas en aspectos y no con composición o herencia.

13. Extensión.- Este patrón sí es viable usarlo en el trabajo con objetos funcionales, dado que permite extender la funcionalidad del objeto funcional usando avisos *around* y sustituyendo el comportamiento inicial por otro más complejo en caso de ser necesario.

14. Agujero de gusano.- Este patrón sí es aplicable al trabajo con funciones, aunque dado las limitaciones que presenta AspectJ al trabajar con el comportamiento de las funciones se tiene que capturar la primera llamada y sustituir las demás llamadas intermedias con avisos *around* y pasar los parámetros a la última llamada.

Capítulo 4

Resultados

En el siguiente capítulo se presenta un caso de estudio para mostrar cómo trabajar la programación orientada a aspectos con los objetos funcionales en los lenguajes Java y Scala, también se analizan las ventajas y desventajas que se obtienen al trabajar con ambos paradigmas, además se muestra un marco de trabajo creado para solventar algunos de los problemas que presentan herramientas como AspectJ o Javassist al trabajar con objetos funcionales.

4.1. Caso de estudio

El caso de estudio que se realizó se tomó de [38] y consiste en un Sistema de Control de Tráfico de Trenes (SCTT).

4.1.1. Descripción del caso de estudio

El Sistema de Control de Tráfico de Trenes (SCTT) tiene como funciones el enrutamiento y monitorización de los trenes, la planificación del tráfico, la predicción de fallas, el seguimiento de la ubicación del tren, la monitorización del tráfico de una zona determinada, la prevención de colisiones y por último el control de las fallas que presenten los trenes. A partir de estas funciones, se definieron seis casos de uso, como se muestra a continuación.

1. Ruta del Tren: Establecer una ruta de tren que defina el recorrido para un tren en particular.
2. Sistemas de monitorización del tren: Vigilar que los sistemas de control del tren funcionen adecuadamente.
3. Predecir fallos: Realizar un análisis de las condiciones del sistema de los trenes para predecir fallos.
4. Monitorización del tráfico: Monitorizar la localización de los trenes que utilizan el SCTT según su zona geográfica.

5. Prevenir Colisiones: Analizar las ubicaciones para detectar posibles colisiones.

6. Control de Fallas: Proporcionar los medios para el control de las fallas que presenten los trenes.

Los requerimientos no funcionales que afectan al sistema son los siguientes:

1. Transportar pasajeros y carga.

2. Los trenes soportan velocidades de hasta 90 kilómetros por hora.

3. Proporcionar un nivel de disponibilidad del sistema de 90 %.

4. Responder a las entradas del operador en 1 segundo.

Existen dos tipos de personas que interactúan con el sistema. Estos usuarios tienen las siguientes funciones dentro del SCTT.

1. Despachador.- Establece rutas de tren, realiza el seguimiento del progreso de los trenes, recibe las posibles colisiones a generarse y decide qué hacer, recibe las fallas que presentan los trenes y decide la acción a realizar, además administra zonas geográficas, usuarios, trenes y estaciones.

2. Ingeniero de trenes.- Vigila el estado del tren y lo opera.

Los actores y las funciones que realizan cada uno de los actores dentro del sistema quedan ilustrados en el diagrama de casos de uso mostrado en la Figura 4.1.

El Sistema de Control de Trafico de Trenes se dividió en dos subsistemas que trabajan conjuntamente, el primero es el sistema que controla el despachador, en este se administran las zonas geográficas, rutas de tren, usuarios y estaciones, también se muestra un mapa donde se visualiza una zona geográfica y se muestran los trenes que en ella circulan, a su vez el sistema del despachador recibe las fallas que presentan los trenes y las muestra, también tiene un sistema detector de colisiones que se encarga de verificar la cercanía de los trenes que corren sobre una misma vía y si se encuentran demasiado cerca mostrará una posible colisión en la pantalla para que el despachador la atienda.

El segundo subsistema es el de a bordo del tren, este muestra los datos que recogen los sensores de los sistemas físicos del tren, también muestra la ubicación del tren en un mapa y si el tren tiene una ruta seleccionada la muestra en una tabla y agrega al mapa las estaciones por donde debe de pasar el tren.

Para entender mejor cómo funciona el sistema se muestra el diagrama entidad relación que representa la base datos en la Figura 4.2, donde se observa que se tienen unas tablas para almacenar las zonas geográficas, las estaciones, los usuarios del sistema, los trenes y las fallas, además se tiene una tabla para almacenar las rutas de los trenes y una tabla

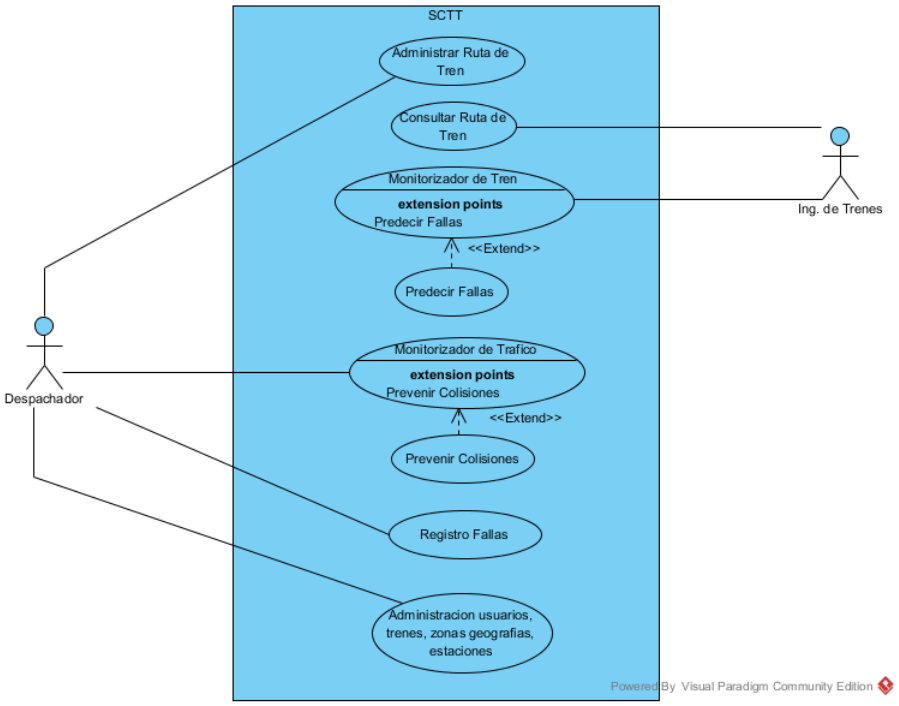

Figura 4.1: Diagrama de Casos de Uso del Sistema de Control de Tráfico de Trenes

llamada ruta tren detalle que almacena las distintas acciones que debe realizar cada tren en una ruta, ejemplos de estas acciones es parar en cierta estación y cargar vagones o recoger pasajeros.

El diagrama de clases del modelo del sistema se muestra en la Figura 4.3, es muy similar al diagrama de entidad relación aunque en este se omitió la clase Falla, dado que esta clase es solo para enviar información entre los subsistemas.

El sistema del despachador y los sistemas de a bordo de cada tren tienen que comunicarse entre ellos, el sistema de a bordo se comunica con el sistema del despachador para enviarle en tiempo real la ubicación del tren, también le comunica al despachador las fallas que le ocurren al tren. El despachador le comunica a cada tren la acción a realizar en caso de que en su ruta se produzca una colisión o sea necesario tomar acciones en caso de producirse una falla. En la Figura 4.4 se muestra un diagrama de secuencia que ilustra mejor la comunicación de los sockets en el SCTT.

Se realizaron dos casos de estudio del mismo sistema, uno en lenguaje Scala y otro en lenguaje Java, los sistemas cuentan con objetos funcionales para una parte de su funcionalidad. A ambos casos de estudio se le realizaron dos modificaciones en el sistema del despachador usando distintas herramientas, las modificaciones realizadas son las siguientes.

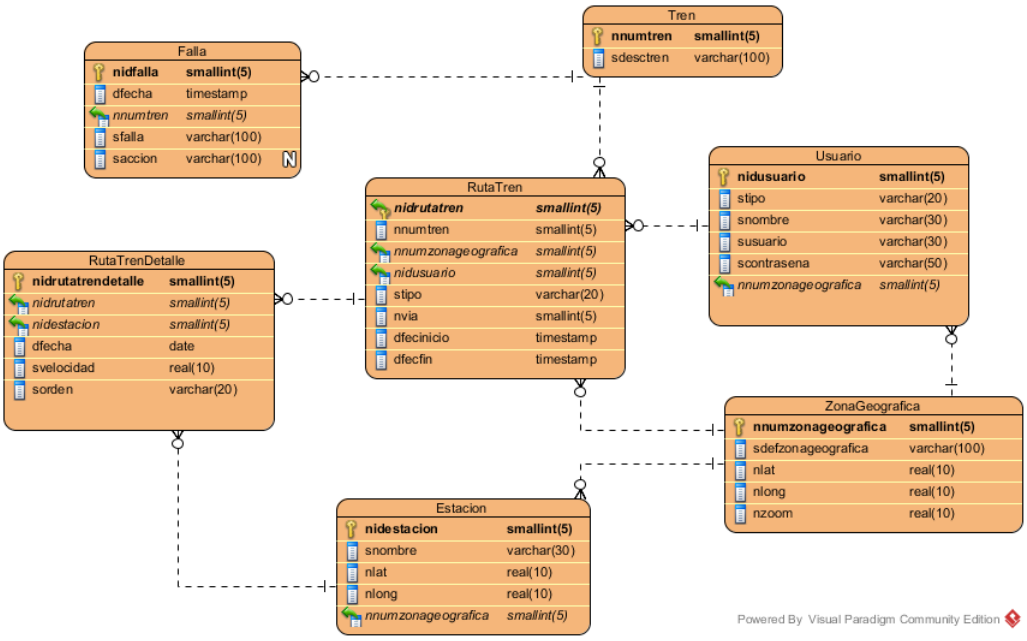

Figura 4.2: Diagrama entidad relación del SCTT

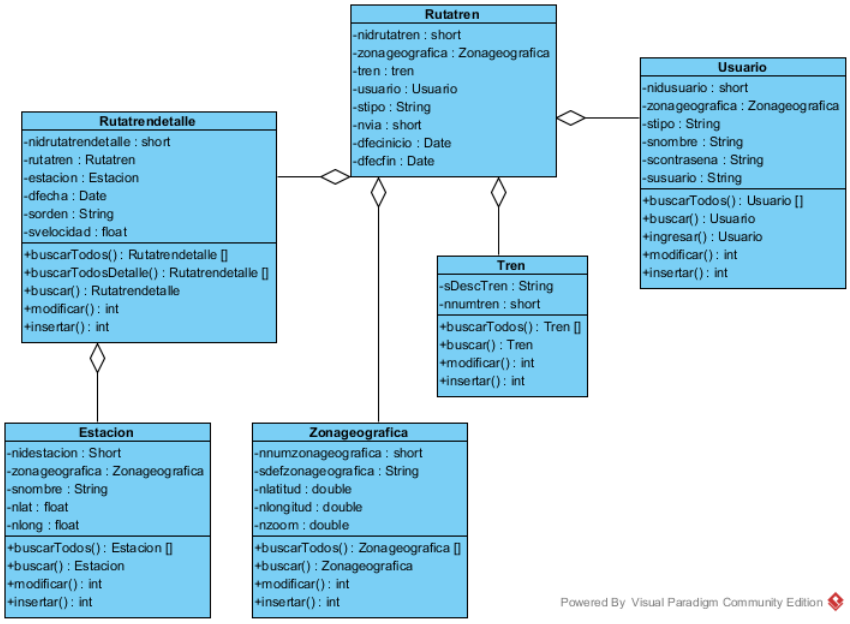

Figura 4.3: Diagrama de clases del SCTT

1. Cambiar los cálculos que detectan las colisiones
2. Mostrar un mensaje de confirmación al intentar eliminar un registro de la base de datos

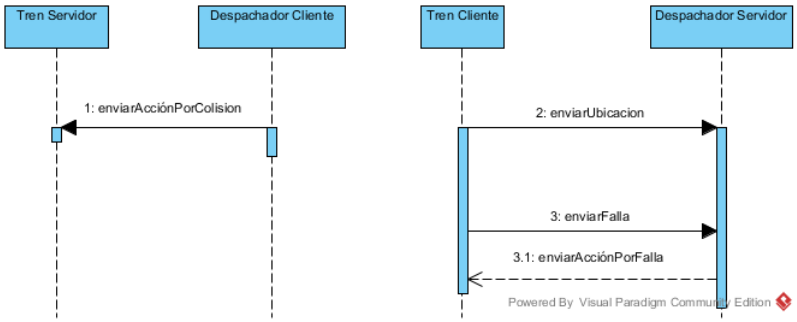

Figura 4.4: Diagrama de Secuencia de los Sockets del SCTT

4.1.2. Java 8

El caso de estudio en Java usa JavaFX para la interfaz gráfica del sistema, para la base de datos usa PostgreSQL, además como el sistema emplea mapas para mostrar las ubicaciones de las estaciones y los trenes se usó la biblioteca GMapsFX para incluir los mapas proporcionados por Google a la aplicación. Las bibliotecas que se utilizaron en la aplicación y sus versiones se listan a continuación:

1. Java 1.8.0_121
2. GMapsFX 2.0.8
3. PostgreSQL JDBC Driver 9.2-1002

La ventana principal del Sistema de Control de Tráfico de Trenes para el despachador se muestra en la Figura 4.5, en ella se observa el mapa donde se muestran las ubicaciones de cada tren, en las tabla del lado inferior derecho se muestran las fallas conforme las vaya recibiendo el sistema despachador, la tabla del lado inferior izquierdo muestra las colisiones que el sistema detecta al analizar las ubicaciones de los trenes, además del lado izquierdo de la ventana se encuentran las opciones para dirigirse a la administración de los demás objetos del sistema, por ejemplo rutas de tren o estaciones.

La ventana de administración de los trenes se muestra en la Figura 4.6, en ella se observa una tabla donde se muestran todos los trenes del sistema y cada uno tiene la opción de modificar o eliminar ese registro, además en la parte inferior se encuentran cuadros de texto para colocar los datos y agregar un nuevo tren. Las demás ventanas para la administración de los demás registros se omitieron porque su funcionamiento e interfaz gráfica son similares.

La ventana principal del sistema de a bordo del tren se muestra en la Figura 4.7, en donde se observa el mapa que muestra la ubicación del tren, a un lado se encuentra un cuadro de texto donde el ingeniero de tren ingresa la ruta que el tren tiene que cubrir, una

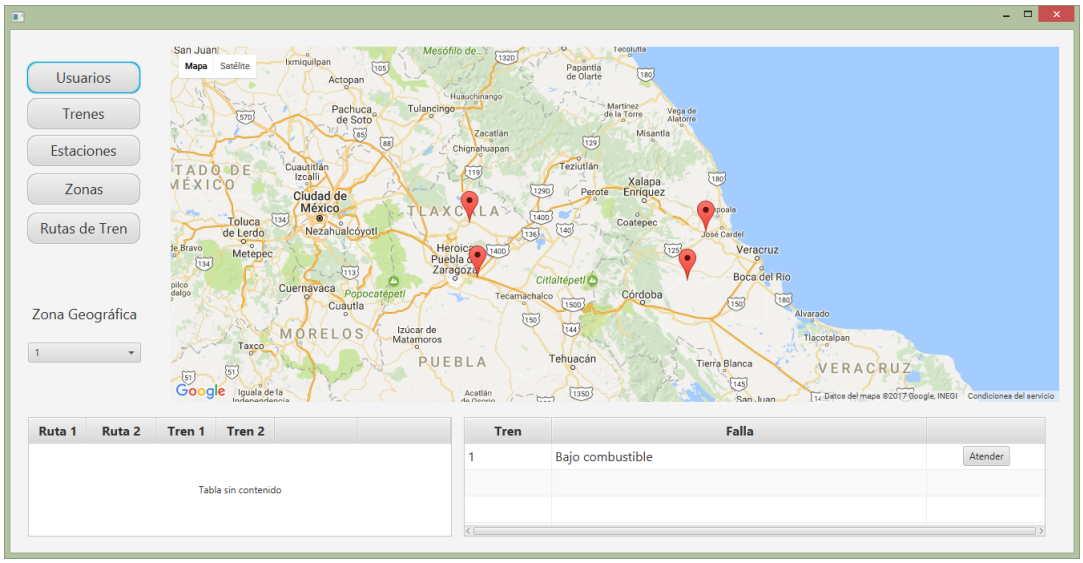

Figura 4.5: Ventana principal del sistema del despachador

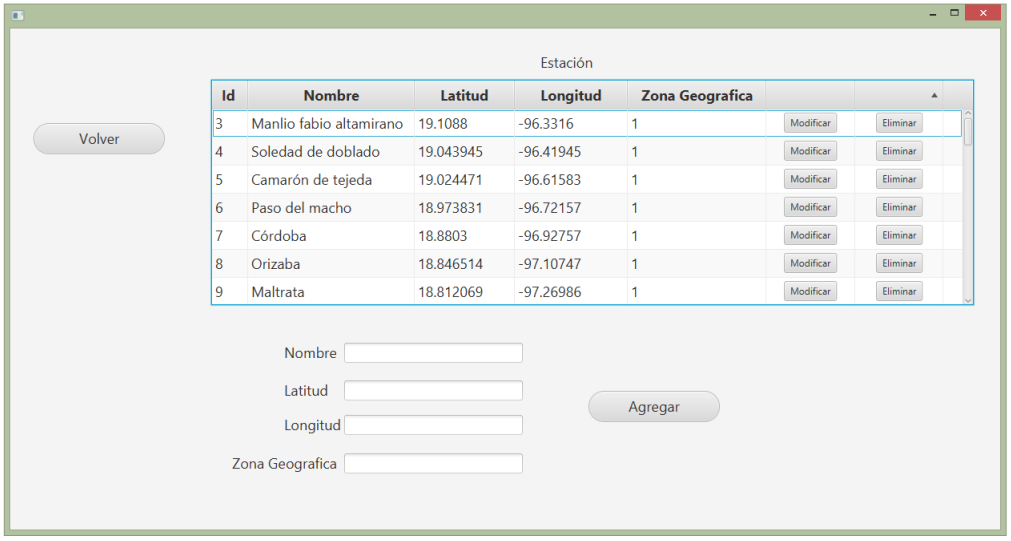

Figura 4.6: Ventana de administración de trenes del sistema del despachador

vez ingresada la ruta el sistema le mostrará en la tabla de la parte inferior y agregará al mapa las estaciones donde el tren tiene alguna actividad.

En la Figura 4.8 se observa la ventana encargada de mostrar los datos de los sistemas mecánico, eléctrico y químico del tren. Los datos se muestran en color verde, en caso de presentarse una falla cambia a color rojo el dato que presente un valor por debajo o por encima de lo esperado.

Al caso de estudio se le agregaron objetos funcionales en su diseño, tomando como base el proceso [39], en dónde se indica que las operaciones más usadas van dentro de

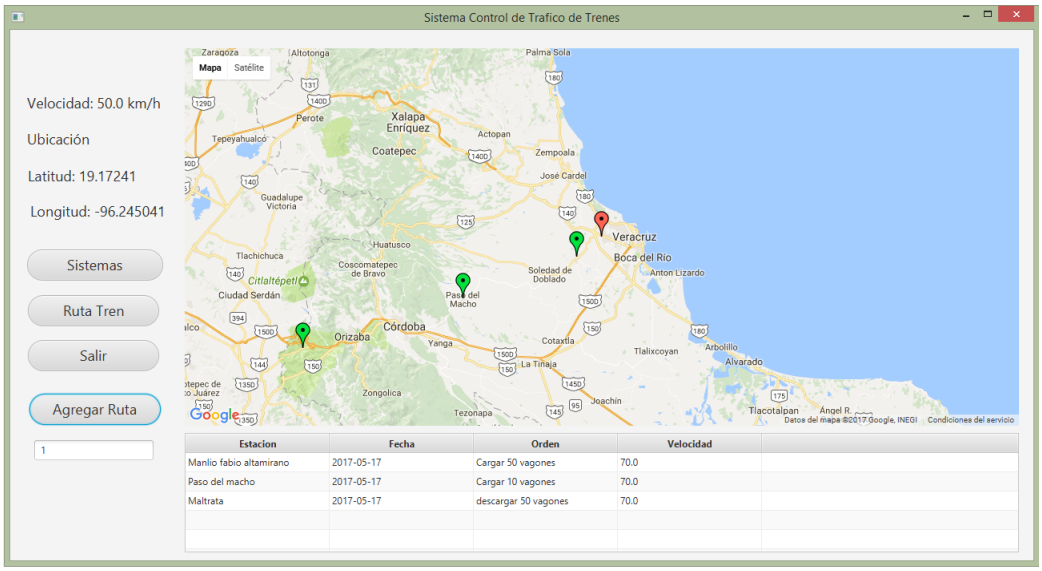

Figura 4.7: Ventana principal del sistema de a bordo

Figura 4.8: Ventana de datos de los sistemas del tren

objetos funcionales. Así se separó la funcionalidad más utilizada en diez lambdas y cuatro funciones anónimas.

Las funciones que cumplen las lambdas son ejecutar consultas en la base de datos, ejecutar comandos en la base de datos, también se crearon lambdas que reciben como entrada una posición del arreglo que retorna la consulta de la base de datos y se encargan de extraer los datos y generar un objeto, por último se tiene una lambda llamada elimi-

narObjeto que tiene la función de eliminar registros de la base de datos.

Las funciones anónimas del sistema tienen el trabajo de validar los datos de entrada del sistema, por ejemplo, al ingresar el número de una ruta de tren o una coordenada la función anónima valida la cadena de texto y verifica si se trata de un número entero, una fecha o una coordenada, de no coincidir marcará un error, se tiene otra función anónima que se encarga de calcular la distancia entre dos trenes usando sus coordenadas y verificar que no se produzcan colisiones.

En el código 4.1 se muestra una interfaz funcional llamada Eliminar que contiene un método llamado eliminar, abajo de ella se muestra la lambda que usa la interfaz, esta lambda se encarga de eliminar un registro de la base de datos, recibe como parámetros el identificador del registro y el tipo de registro a eliminar, para así tomando esos datos crear el comando y usar otra lambda llamada ejecutarComando, para realizar la eliminación en la base de datos.

Código 4.1: Ejemplo de Lambda en el SCTT

```
1  @FunctionalInterface
2  public interface Eliminar {
3    public int eliminar(short x, String tipo);
4  }
5  public static Eliminar eliminarObjeto = new Eliminar()  {
6    @Override
7    public int eliminar(short x, String tipo) {
8    int nRet = 0;
9    String sQuery = "";
10    if (x>=0) {
11       switch(tipo) {
12        case "Estacion" :
13          sQuery = "DELETE FROM Estacion WHERE nidestacion = " + x;
14          break;
15        case "Rutatren" :
16          sQuery = "DELETE FROM Rutatren WHERE nidRutatren = " + x;
17          break;
18        case "Rutatrendetalle" :
19          sQuery = "DELETE FROM Rutatrendetalle WHERE
                      nidRutatrendetalle = " + x;
20          break;
21        case "Tren" :
22          sQuery = "DELETE FROM Tren WHERE nnumtren = " + x;
23          break;
24        case "Usuario" :
25          sQuery = "DELETE FROM Usuario WHERE nidUsuario = " + x;
26          break;
27        case "Zonageografica" :
28          sQuery = "DELETE FROM Zonageografica WHERE
                      nnumzonageografica = " + x;
29          break;
```

```
30          case "RutatrendetallePorRuta" :
31            sQuery = "DELETE FROM Rutatrendetalle WHERE
                        nidRutatren =" + x;
32            break;
33        }
34          nRet = ejecutarComando.ejecutar(sQuery);
35      }
36       return nRet;
37      }
38  };
```

En el código 4.2 se muestra una interfaz funcional llamada Validar y la función anónima que la implementa, esta función valida que la cadena de caracteres que recibe coincida con el patrón de una fecha que incluya horas y minutos.

Código 4.2: Ejemplo de función anónima en el SCTT

```
1   @FunctionalInterface
2   public interface Validar{
3     public boolean validar(String x);
4   }
5   public static Validar validaDate = (String x) -> {
6     boolean ban = false;
7     if (x.matches("\\d{2}/\\d{2}/\\d{4}\\s\\d{2}:\\d{2}")) {
8       ban = true;
9     }
10    return ban;
11  };
```

Otra función anónima es la mostrada en el código 4.3 en donde se observa la interfaz funcional y la función que la implementa, esta función se encarga de calcular la distancia entre dos trenes basándose en sus coordenadas.

Código 4.3: Ejemplo de función anónima en el SCTT

```
1   @FunctionalInterface
2   interface Distancia{
3     public double calcular(double lat1, double lng1,
                             double lat2, double lng2);
4   }
5       static Distancia c = (lat1, lng1, lat2, lng2) -> {
6      double respuesta;
7      double res1,res2;
8      res1=lat2-lat1;
9      res2=lng2-lng1;
10     res1=Math.pow(res1, 2)+Math.pow(res2, 2);
11     respuest=Math.sqrt(res1);
12     return respuesta;
13  };
```

Aplicación de la programación orientada a aspectos al caso de estudio usando AspectJ

Para ejecutar la aplicación y aplicarle los aspectos se ocupó el entorno de desarrollo Eclipse junto con el *plugin* AspectJ Development Tools [40]. También se usó el *plugin* e(fx)clipse para ejecutar el proyecto dado que usa JavaFx. Las versiones de Eclipse y de los *plugins* que se instalaron se muestran a continuación.

- Version: Mars.2 Release (4.5.2)
- AspectJ development tools 2.2.4
- e(fx)clipse IDE 2.4.0

Las versiones que se muestran fueron las más actuales de cada una de las herramientas y además mostraron compatibilidad entre ellas, dado que hay versiones de Eclipse que no aceptan el *plugin* de AspectJ y el *plugin* e(fx)elipse de manera simultanea.

Para usar JavaFx y AspectJ en un proyecto en Eclipse es necesario crear un proyecto de AspectJ y sobre él colocar el código de la aplicación que utiliza JavaFX, y para su ejecución sólo es necesario ejecutar directamente la clase que inicia la ventana principal del sistema.

En el código 4.4 se muestra cómo aplicar un corte sobre la lambda mostrada en el código 4.1, la modificación a realizar es la de agregar un diálogo de confirmación que pregunte si se desea realizar la acción, para que el usuario tenga la oportunidad de cancelar la opción en dado caso que no desee eliminar algún registro.

El corte llamado eliminar realiza la modificación mencionada en el párrafo anterior, el corte se realiza a la llamada al método eliminar, éste contiene la funcionalidad del objeto y se ejecuta cada vez que la lambda es llamada, una vez realizado el corte se creó un aviso de tipo *around* que sustituye la llamada original del método, una vez que el aviso se ejecuta muestra un diálogo preguntando si se desea eliminar el registro, si el usuario acepta, se usa la variable *proceed* para continuar con la ejecución normal del programa y eliminar el objeto, en caso de que el usuario no acepte eliminar, la opción no se realiza y el método retornará un valor de cero. Con esa modificación se evita que los usuarios borren registros por accidente.

El segundo corte mostrado en el código 4.4 es un corte diseñado para afectar a la función anónima 4.3, la cual realiza el cálculo de la distancia entre dos trenes, la modificación a realizar es la sustitución de los cálculos originales por otros que son más exactos, ya que los segundos toman en cuenta la curvatura de la tierra. Para realizar el corte y al ser una función anónima es necesario identificar el método estático creado con el comportamiento del objeto funcional, dado que es la primera y única lambda en su clase su nombre es

lambda$0, una vez identificado el método se procede a cortarlo y a colocar un aviso *around* sobre él para sustituir la llamada al comportamiento original por los nuevos cálculos colocados dentro del aviso, así cuando la función anónima sea llamada los cálculos se realizarán con el nuevo código.

Código 4.4: Aplicación de cortes usando AspectJ al SCTT

```
1  public aspect MainAspect {
2    pointcut eliminar(short x, String y):
3          call(int Funcional.Eliminar.eliminar(short, String))
              && args(x, y);
4          int around (short x, String y) : eliminar(x, y) {
5          Alert alert = new Alert(AlertType.CONFIRMATION);
6          alert.setTitle("Confirmacion");
7          alert.setHeaderText("Seleccione una opcion");
8          alert.setContentText("¿Desea eliminar el registro?");
9          Optional<ButtonType> result = alert.showAndWait();
10         if (result.get() == ButtonType.OK){
11               return proceed(x, y);
12         } else {
13               return 0;
14       }
15    }
16     public pointcut calcular(double lat1, double lng1,
                                        double lat2, double lng2) :
17      execution(* Controlador.DetectorColisiones.lambda\$0(..)) &&
                      args(lat1,lng1,lat2,lng2);
18      double around(double lat1, double lng1, double lat2,
                      double lng2):calcular(lat1,lng1,lat2,lng2){
19      double radioTierra = 6371;
20      double dLat = Math.toRadians(lat2 - lat1);
21      double dLng = Math.toRadians(lng2 - lng1);
22      double sindLat = Math.sin(dLat / 2);
23      double sindLng = Math.sin(dLng / 2);
24      double va1 = Math.pow(sindLat, 2) + Math.pow(sindLng, 2)
                           * Math.cos(Math.toRadians(lat1)) *
                           Math.cos(Math.toRadians(lat2));
25      double va2 = 2 * Math.atan2(Math.sqrt(va1),
                           Math.sqrt(1 - va1));
26      double distancia = radioTierra * va2;
27      return distancia;
28      } }
```

En la Figura 4.9 se muestra cómo el corte eliminar tiene efecto sobre la aplicación del despachador, así al seleccionar eliminar un objeto se mostrará un diálogo de confirmación y si se acepta eliminará el objeto, si se cancela el objeto no se eliminará.

El aspecto sobre la función anónima no es observable a simple vista, los resultados se observan en el algoritmo que se cambió, ya que con el nuevo código se obtiene una mejoría en los cálculos de las distancias de los trenes, que se vuelven más exactos.

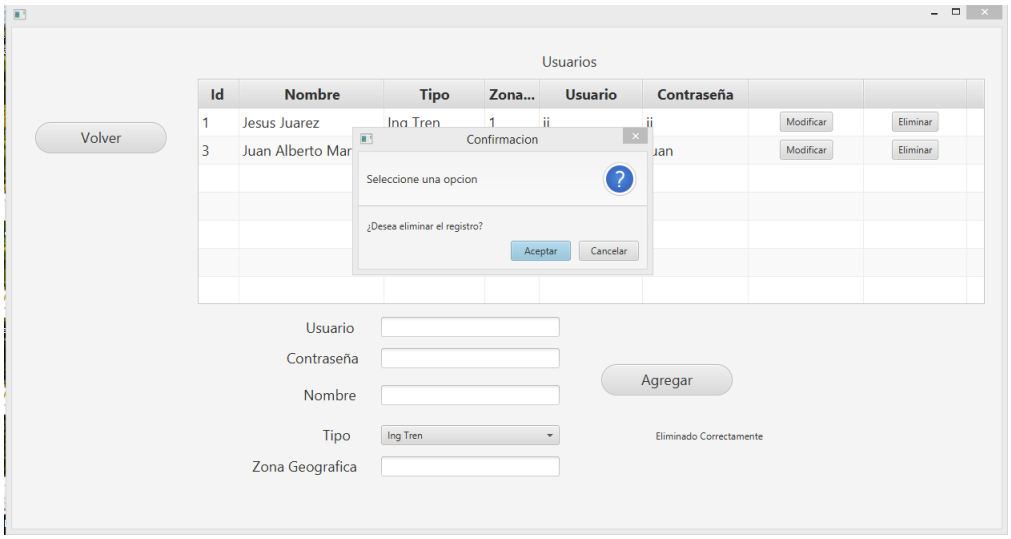

Figura 4.9: Eliminación de un usuario

Aplicación de la programación orientada a aspectos al caso de estudio usando Javassist

Se intentó realizar la misma modificación de mostrar un diálogo de confirmación al intentar eliminar registros en el sistema del despachador, pero usando una herramienta diferente, en este caso Javassist, pero al intentar modificar el *bytecode* se descubrió que se excedieron las capacidades de modificación que Javassist posee.
Javassist tiene la limitante de no modificar las clases internas dado que no le es posible a Javassist acceder a ellas, y como se menciona anteriormente el comportamiento de los objetos funcionales, específicamente el de las lambdas, queda dentro de una clase interna, por lo que la capacidad de modificación de las lambdas con Javassist tiene limitantes.
Para realizar la modificación se intentó usar el método de Javassist replace, y sí es posible reemplazar el comportamiento del método, pero debe ser de manera total, no es posible como en AspectJ retomar la ejecución del programa si se cumple determinada condición y cambiar la ejecución si no se cumple lo esperado.
Se intentó tomar los parámetros que recibe el método, pero al trabajar con clases internas no es posible que los parámetros se obtengan de ninguna forma, ya sea de manera global con la variable $$, o de manera individual con $1, $2, $n.
De igual manera se intentó llamar a la lambda desde el código de reemplazo, pero Javassist no tiene la capacidad de llamar métodos de clases internas, así que llamar a la lambda de nuevo no es posible. Por lo que en este caso solo se tienen las opciones de agregar un comportamiento antes o después de la ejecución del método, o reemplazar el cuerpo del método por otro, pero sin que se capturen los parámetros que se le mandan al método en la llamada original.
En el código 4.5 se tiene un ejemplo de una modificación usando Javassist en donde una

vez que el usuario eliminó un registro sólo le muestra un mensaje avisándole la acción que acaba de ocurrir.

Código 4.5: Ejemplo de Javassist para modificar una lambda

```
1  public class JavassistPrueba {
2        public static void main(String[] args) {
3          try {
4           ClassPool pool = ClassPool.getDefault();
5               pool.importPackage("javax.swing.JOptionPane");
6               pool.importPackage("Funcional.Lambdas");
7               CtClass cc2 = pool.get("Funcional.Lambdas\$1");
8               cc2.defrost();
9               CtMethod m2 = cc2.getDeclaredMethod("eliminar");
10              m2.insertAfter("{JOptionPane.showMessageDialog
11                          (null, \"Acaba de eliminar un registro\");
   }");
12              cc2.writeFile();
13         } catch (Exception e) {
14                      e.printStackTrace();
15          }
16       }
17 }
```

Para realizar la modificación a la función anónima del código 4.3 que consiste en cambiar el código que hace los cálculos, la limitante que presenta Javassist para las lambdas no está presente, dado que sólo aplica a las clases internas y las funciones anónimas funcionan con un método dentro de la misma clase.

En el código 4.6 se modifica la función anónima llamada calcular, la cual se encuentra en una clase llamada DetectorColisiones en el paquete controlador, a esta clase se le reemplazará el código que realiza los cálculos por otro, por lo que al usar el método replace es necesario cambiar los parámetros de entrada por la variables de Javassist $n por cada valor de entrada que reciba el método y $_ para el valor de retorno de la función.

Código 4.6: Ejemplo de Javassist para modificar una función anónima

```
1  public class JavassistPrueba {
2    public static void main(String[] args) {
3         try {
4          ClassPool pool = ClassPool.getDefault();
5          CtClass cc2 = pool.get("Controlador.DetectorColisiones");
6          cc2.defrost();
7          CtMethod m2 = cc2.getDeclaredMethod("lambda$static$0");
8          m2.instrument(
9         new ExprEditor() {
10        public void edit(MethodCall m)
11         throws CannotCompileException
12        {
13          m.replace("{ "+
                    "double radioTierra = 6371;" +
                    "double dLat = Math.toRadians($2 - $0);  "+
```

```
                    "double dLng = Math.toRadians($3 - $1);  "+
                     "double sindLat = Math.sin(dLat / 2.0);  "+
                     "double sindLng = Math.sin(dLng / 2.0);  "+
                     "double va1 = Math.pow(sindLat, 2.0) +
                       Math.pow(sindLng, 2.0)  * Math.cos(Math.
                          toRadians($0))
                       *  Math.cos(Math.toRadians($2)); "
                     "double va2 = 2 * Math.atan2(Math.sqrt(va1),
                       Math.sqrt(1.0 - va1));  "
                     "$_ = radioTierra * va2;}");
14          }
15         });
16                    cc2.writeFile();
17                } catch (Exception e) {
18                        e.printStackTrace();
19                }
20            }
21   }
```

4.1.3. Scala

Para realizar el caso de estudio Sistema de Control de Tráfico de Trenes en el lenguaje Scala se usó SBT para la compilación y ejecución de los proyectos, la versión de Scala utilizada es la 2.11.8, además para la parte gráfica se usó ScalaFx. Un fragmento del archivo de configuración de SBT para el proyecto se muestra en el código 4.7, en donde se observan las bibliotecas que se incluyen para que el proyecto funcione.

Código 4.7: Fragmento del archivo de configuración de SBT

```
1 addCompilerPlugin("org.scalamacros" % "paradise" % "2.0.1" cross
   CrossVersion.full)

2 libraryDependencies ++= List(
   "org.scalafx" %% "scalafx" % "8.0.20-R6",
   "org.scalafx" %% "scalafxml-core-sfx8" % "0.2.2",
   "com.lynden" % "GMapsFX" % "2.0.8",
   "org.postgresql" % "postgresql" % "9.2-1002-jdbc4"
   )

3 resolvers ++= Seq("snapshots","releases").map(Resolver.sonatypeRepo)

4 unmanagedJars in Compile +=
  Attributed.blank(file(System.getenv("JAVA_HOME") + "/jre/lib/ext/
  jfxrt.jar"))
```

En la Figura 4.10 se muestra la ventana principal del sistema despachador. Las demás ventanas del sistema despachador y del sistema de a bordo se omitieron, dado que se usó ScalaFX para la interfaz gráfica y al ser ScalaFX un envolvente de JavaFX el resultado gráfico final se use una u otra biblioteca es muy similar.

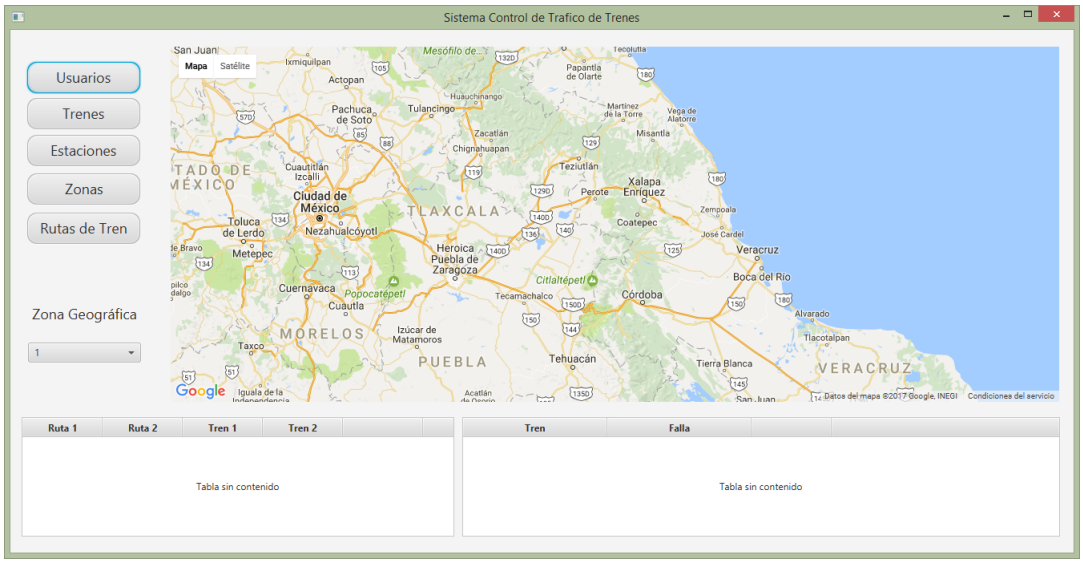

Figura 4.10: Ventana principal del sistema del despachador

El sistema de control de tráfico de trenes para los dos subsistemas que lo componen, cuenta con objetos funcionales que realizan tareas muy utilizadas en distintas partes de la aplicación, a continuación en el código 4.8 se muestra una de ellas, esta función llamada Eliminar se encarga de borrar registros de la base de datos, la función se creó con la finalidad de reducir código en las clases que componen el modelo de datos de la aplicación, así en lugar de que cada clase, por ejemplo la clase Tren o la clase Estación tengan su propio método eliminar, esos métodos van en una función que es usada por todos los que la necesiten, la función recibe el identificador del elemento a eliminar y su tipo, esta función usa otra función llamada EjecutarComando, que se encarga de ejecutar la instrucción en la base de datos.

Código 4.8: Función en Scala

```
1  object Eliminar{
2    def apply(id:Short, tipo:String):Int={
3     var sQuery: String = ""
4      tipo match {
5      case "Estacion" => sQuery =
        "DELETE FROM Estacion WHERE nidestacion = " + id
6      case "Rutatren" => sQuery =
        "DELETE FROM Rutatren WHERE nidRutatren =  " + id
7      case "Rutatrendetalle" => sQuery =
        "DELETE FROM Rutatrendetalle WHERE nidRutatrendetalle =" + id
8      case "Tren" => sQuery =
        "DELETE FROM Tren WHERE nnumtren = " + id
9      case "Usuario" => sQuery =
        "DELETE FROM Usuario WHERE nidUsuario = " + id
10     case "Zonageografica" => sQuery =
        "DELETE FROM Zonageografica WHERE nnumzonageografica = " + id
11     case "RutatrendetallePorRuta" => sQuery =
```

```
        "DELETE FROM Rutatrendetalle WHERE nidRutatren =" + id
12    }
13   EjecutarComando(sQuery)
14   }
15  }
```

En el código 4.9 se muestra un objeto que contiene dos funciones anónimas, ambas tienen la función de recibir cadenas de caracteres y validar si el contenido de la cadena que reciben es un numero entero o es una coordenada, y devuelven verdadero si coincide y falso si no lo hace, estas funciones se crearon para validar los datos de entrada que el usuario ingresa en la aplicación.

Código 4.9: Funciones anónimas en Scala

```
1   object Validaciones{
2     private val intRegex = """(\d+)""".r
3     private val latRegex = """(-?\d+\.\d+)""".r
4     var intValidate = (str:String)=>{
5      str match {
6       case intRegex(str) => true
7       case _ => false
8      }
9     }
10   var latValidate = (str:String)=>{
11     str match {
12      case latRegex(str) => true
13      case _ => false
14     }
15    }
16  }
```

En el código 4.13 se muestra otra función anónima con la que cuenta el sistema del despachador, la función llamada detectar colisión se encarga de calcular la distancia entre dos trenes usando sus coordenadas, esta función es muy usada por el sistema que detecta las condiciones.

Código 4.10: Función anónima en Scala

```
1  object Colisiones{
2    var DetectarColision = (lat1:Double, lng1:Double,
                    lat2:Double, lng2:Double) => {
3    var respuest: Double = 0.0
4    var res1: Double = 0.0
5    var res2: Double = 0.0
6    res1 = lat2 - lat1
7    res2 = lng2 - lng1
8    res1 = Math.pow(res1, 2.0) + Math.pow(res2, 2.0)
9    respuest = Math.sqrt(res1)
10     respuest
11  }
12 }
```

Aplicación de la programación orientada a aspectos al caso de estudio usando AspectJ

Para que SBT ejecute los aspectos que van a afectar a la aplicación es necesario agregar un archivo *xml* que contendrá los nombres de las clases donde se encuentran los aspectos y la clases donde éstos se van a aplicar. El archivo *xml* empieza indicando que corresponde a AspectJ (línea 1), de la línea 2 a la 4 se muestra cómo se declaran las clases donde se colocan los aspectos, de la línea 5 a la 7 se muestra cómo se declaran las opciones del entrelazador, la bandera *XnoInline* (línea 5) se usa para declarar que se genere código que trabaje en cualquier compilador de Java.

```
1 <aspectj>
2   <aspects>
3     <aspect name="paquete.clase"/>
4   </aspects>
5   <weaver options="-XnoInline">
6     <include within="aspectosPaquete.*"/>
7     <include within=clasesPaquete.*"/>
8   </weaver>
9 </aspectj>
```

El archivo mencionado anteriormente se incluirá en la siguiente ubicación dentro del proyecto:

```
main/
   resources/
         META-INF/
             Archivo.xml
```

También es necesario agregarle al archivo de configuración del proyecto las dependencias que ocupa AspectJ (línea 1) y es necesario especificar las opciones de Java para que trabajen juntos AspectJ con la Máquina Virtual de Java (línea 2).

```
1 libraryDependencies ++= Seq(
    "org.aspectj" % "aspectjweaver" % "1.8.6",
    "org.aspectj" % "aspectjrt"     % "1.8.6"  )
2 javaOptions += "-javaagent:" +
    System.getProperty("user.home")+
    "/.ivy2/cache/org.aspectj/aspectjweaver/jars
    /aspectjweaver-1.8.6.jar"
```

Se realizaron modificaciones usando aspectos al caso de estudio del sistema despachador, las funciones que se van a modificar son las mostradas en el código 4.13 y 4.8. El código 4.11 muestra dos cortes para modificar las funciones antes mencionadas, el corte llamado eliminar modifica la ejecución de la función que elimina registros, una vez que se llama a la función el aviso entra en acción y muestra un diálogo de confirmación el cual el

usuario acepta para eliminar el registro, o cancelar para que la función no elimine algo, el corte llamado calcular reemplaza el comportamiento de la función 4.13 por otro.

Código 4.11: Aspectos sobre el caso

```
1  @Aspect
2  class MainAspect {
3   @Around("execution(int Modelo.Eliminar*.apply(..))")
4     def eliminar(joinPoint: ProceedingJoinPoint):Int = {
5      var message = "Desea eliminar el registro?";
6      var title = "Confirmacion";
7      var i:Int =0;
8      var reply = JOptionPane.showConfirmDialog(null, message, title,
                 JOptionPane.YES_NO_OPTION);
9      if (reply == JOptionPane.YES_OPTION)
10     {  i = joinPoint.proceed().asInstanceOf[Int] }
11         i
12    }
13      @Around("call(* scala.Function4.apply(..)) &&
                  args(lat1, lng1, lat2, lng2)")
14    def calcular(lat1:Object, lng1:Object, lat2:Object,
                  lng2:Object):Object = {
15     val radioTierra: Double = 6371
16     val dLat: Double = Math.toRadians(lat2.asInstanceOf[Double] -
                        lat1.asInstanceOf[Double])
17     val dLng: Double = Math.toRadians(lng2.asInstanceOf[Double] -
                        lng1.asInstanceOf[Double])
18     val sindLat: Double = Math.sin(dLat / 2)
19     val sindLng: Double = Math.sin(dLng / 2)
20     val va1: Double = Math.pow(sindLat, 2) +
          Math.pow(sindLng, 2) *
          Math.cos(Math.toRadians(lat1.asInstanceOf[Double])) *
          Math.cos(Math.toRadians(lat2.asInstanceOf[Double]))
21     val va2: Double = 2 * Math.atan2(Math.sqrt(va1),
                Math.sqrt(1 - va1))
22     val distancia: Double = radioTierra * va2
23     distancia.asInstanceOf[AnyRef]
24    }
25   }
```

En la Figura 4.11 se muestra cómo el aviso entra en funcionamiento cuando se intenta eliminar algún registro.

La modificación que realizó el corte calcular no es apreciable gráficamente, dado que el cambio que surte es de mejora de la precision al detectar una colisión.

Aplicación de la programación orientada a aspectos al caso de estudio usando Mixin Composition

El Mixin Composition es una técnica que permite agregar nuevos comportamientos a las clases, para usarla sobre la función 4.13 es necesario hacer algunas modificaciones al código, para empezar, la función ya no debe ser de tipo *object*, es necesario que sea de

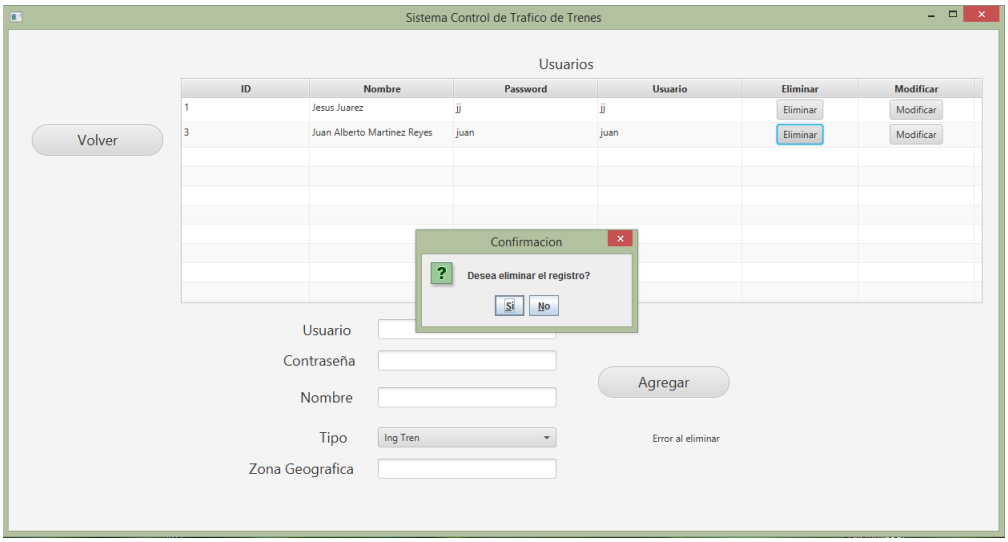

Figura 4.11: Eliminación de un usuario en el SCTT

tipo *class*, además es necesario crear un *trait* que contendrá el código de la modificación, el *trait* y la clase que contiene la función deben heredar de un mismo tipo en este caso es *Function2[Short, String, Int]*.

En el código 4.12 se muestra cómo se modificó la función eliminar y también se observa el *trait* llamado T1, el cual es ejecutado antes de la función y muestra el cuadro de diálogo, y si se acepta, la clase eliminar se llama.

Código 4.12: Aplicación de Mixin Composition a una función anónima

```
1  class Eliminar extends Function2[Short, String, Int] {
2   def apply(id:Short, tipo:String):Int={
3    var sQuery: String = ""
4    tipo match {
5     case "Estacion" => sQuery =
       "DELETE FROM Estacion WHERE nidestacion = " + id
6     case "Rutatren" => sQuery =
       "DELETE FROM Rutatren WHERE nidRutatren =  " + id
7     case "Rutatrendetalle" => sQuery =
       "DELETE FROM Rutatrendetalle WHERE nidRutatrendetalle =  " + id
8     case "Tren" => sQuery =
       "DELETE FROM Tren WHERE nnumtren = " + id
9     case "Usuario" => sQuery =
       "DELETE FROM Usuario WHERE nidUsuario = " + id
10    case "Zonageografica" => sQuery =
       "DELETE FROM Zonageografica WHERE nnumzonageografica = " + id
11    case "RutatrendetallePorRuta" => sQuery =
       "DELETE FROM Rutatrendetalle WHERE nidRutatren =" + id
12   }
13    EjecutarComando(sQuery)
14   }
```

```
15  }

16 trait T1  extends Function2[Short, String, Int]{
17  abstract override def apply(id:Short, tipo:String):Int = {
18    var message = "Desea eliminar el registro?";
19    var title = "Confirmacion";
20        var i:Int =0;
21    var reply = JOptionPane.showConfirmDialog(null, message, title,
                  JOptionPane.YES_NO_OPTION);
22    if (reply == JOptionPane.YES_OPTION)
23    { i = super.apply(id, tipo) }
24         i
25   }
26 }
```

El llamado a la función también cambia, dado que es necesario crear un nuevo objeto de la clase que contiene la función y además agregar la palabra reservada *with* que realiza la composición del *trait* que contiene el código a agregar con la clase de la función.

```
val obj = new Modelo.Eliminar2 with Modelo.T1
var i = obj.apply(x, "Usuario")
```

En el caso de las funciones anónimas no es posible usar el Mixin Composition sobre éstas, dado que la forma en como se manejan las funciones anónimas es mediante campos, y es necesario crear un objeto para hacer la composición, ya que no funciona con *objects*.

En el código 4.13 se muestra cómo sería la tranformación de la función anónima que detecta las colisiones a una clase, y se muestra un *trait* que anula el comportamiento de esa clase y cambia el código que hace los cálculos, así cuando se llame a la función el que trabajará será el *trait* y no la clase.

Código 4.13: Función anónima en Scala

```
1 class DetectarColision extends Function4[Double, Double,
      Double, Double, Double] {
2   def apply(lat1:Double, lng1:Double,
        lat2:Double, lng2:Double):Double={
3    var respuest: Double = 0.0
4    var res1: Double = 0.0
5    var res2: Double = 0.0
6    res1 = lat2 - lat1
7    res2 = lng2 - lng1
8    res1 = Math.pow(res1, 2.0) + Math.pow(res2, 2.0)
9    respuest = Math.sqrt(res1)
10   respuest
11  }
12 }

13 trait T2  extends Function4[Double, Double, Double,
      Double, Double] {
14  abstract override def apply(lat1:Double, lng1:Double,
      lat2:Double, lng2:Double):Double = {
```

```
15     val radioTierra: Double = 6371
16     val dLat: Double = Math.toRadians(lat2 - lat1)
17     val dLng: Double = Math.toRadians(lng2 - lng1)
18     val sindLat: Double = Math.sin(dLat / 2)
19     val sindLng: Double = Math.sin(dLng / 2)
20     val va1: Double = Math.pow(sindLat, 2) +
           Math.pow(sindLng, 2) * Math.cos(Math.toRadians(lat1)) *
           Math.cos(Math.toRadians(lat2))
21     val va2: Double = 2 * Math.atan2(Math.sqrt(va1),
           Math.sqrt(1 - va1))
22     val distancia: Double = radioTierra * va2
23     distancia
24   }
25 }
```

Aplicación de la programación orientada a aspectos al caso de estudio usando Javassist

Al probar las capacidades de modificación que Javassist tiene sobre los objetos funcionales en el lenguaje Scala se encontró que tiene varias limitantes, al usar el método *replace* no es posible capturar los valores de entrada del método, en Javassist los valores se capturan de manera global con la variable $$ y de forma individual con las variables $1, $2, $n, pero al ejecutar el programa de Javassist para hacer el reemplazo del método y se incluyen estas variables, el compilador marca un error de que las variables no existen, ese es un error de Javassist, dado que si se ocupa un método normal en Java y se realiza la misma modificación, Javassist modifica sin problemas.

Una opción aunque un tanto limitada para modificar *bytecode* en Scala usando Javassist es reemplazar el cuerpo del método, pero sin capturar los parámetros de entrada del método original, con los métodos de Javassist *insertBefore* e *insertAfter* no se presentan inconvenientes.

En el código 4.14 se muestra un programa en Scala que modifica a dos funciones mostradas en los códigos 4.13 y 4.8, en ambas lo que se hace es mostrar los valores que estas funciones reciben. El método a modificar es el mismo en ambas funciones, es el método *apply*, lo importante al modificar es identificar la clase correcta, en el primer caso para la función que estaba en un *object* y su nombre es Eliminar la clase es Eliminar$, en la función anónima la clase contiene el nombre del objeto donde se encuentra la clase y se le agrega el sufijo $$anonfun$ y en este caso es el número 1 dado que es la primera función anónima que se encuentra en ese objeto.

Código 4.14: Uso de Javassist en Scala

```
1  object JavassistScala {
2    def main(args: Array[String]): Unit = {
3      try {
4        val pool: ClassPool = ClassPool.getDefault
5        val cc: CtClass = pool.get("Modelo.Eliminar$")
```

```
6         cc.defrost()
7         val m: CtMethod = cc.getDeclaredMethod("apply")
8         m.insertBefore("{System.out.println($1+\" \"+$2);}")
9         cc.writeFile()
10      } catch {
11        case e: Exception => e.printStackTrace()
12      }
13      try {
14        val pool2: ClassPool = ClassPool.getDefault
15        val cc2: CtClass = pool2.get("Modelo.Colisiones$$anonfun$1")
16        cc2.defrost()
17        val m2: CtMethod = cc2.getDeclaredMethod("apply")
18        m2.insertBefore("{System.out.println($1+\" \"+
                            $2+$3+\" \"+$4);}")
19        cc2.writeFile()
20      } catch {
21        case e: Exception => e.printStackTrace()
22      }
23    }
24  }
```

4.1.4. Comparación de las herramientas para aplicar aspectos sobre objetos funcionales

En todas las herramientas y técnicas que se usaron para aplicar aspectos sobre los objetos funcionales se logró la colocación de avisos antes, después y en lugar de la llamada original de la función, aunque con algunas restricciones.

En la Tabla 4.1 se muestra una comparación de las herramientas que se usaron para aplicar cortes sobre los objetos funcionales en los lenguajes Java y Scala, también se muestran las ventajas y desventajas que tiene el uso de los objetos funcionales.

Como se observa en la Tabla 4.1, la herramienta que presenta mejores resultados es AspectJ, por ejemplo si se usa la primitiva de corte *within* junto con comodines, se puede hacer un análisis de los puntos de unión de varias clases con nombres similares o analizar las clases de un paquete e identificar el nombre del método y de la clase que contiene el comportamiento del objeto funcional. Algo que con Javassist no es posible y se tiene que hacer de manera manual, revisando las clases que se generaron durante la compilación y usando una herramienta por ejemplo javap para ver el *bytecode*.

En Scala con el uso del Mixin Composition se obtuvieron buenos resultados, la desventaja es que sólo es aplicable al trabajo con funciones y se requiere realizar modificaciones al código fuente.

Ventajas y desventajas del uso de objetos funcionales

En el lenguaje Java la principal desventaja que se encontró del uso de de los objetos funcionales es el empleo de la interfaz funcional. Por ejemplo, si se quieren crear varias lambdas pero con distintos valores de entrada o de retorno, se debe crear una interfaz funcional para cada uno, de la misma forma si se desea usar esas interfaces funcionales en otros paquetes, es necesario que esas interfaces funcionales sean públicas, por lo que se debe crear un archivo Java para cada interfaz, así aunque el código de cada interfaz funcional sea pequeño se tiene que crear su archivo, por lo que la cantidad de archivos crece si se crean varias lambdas.

En el lenguaje Scala, dado que fue desarrollado con el uso de funciones desde su origen, éstas están bien integradas en el lenguaje y no presentan inconvenientes al momento de usarlas en el código.

Tabla 4.1: Comparación de las herramientas usadas

Herramienta	**Ventajas**	**Desventajas**
Mixin Composition (Scala)	Sólo se usa lenguaje Scala, no hay ninguna herramienta externa que modifique al programa	Es necesario modificar el código de la función, crear el nuevo trait y modificar la llamada a la función para que agregue el trait que hace la composición, además aplica sobre funciones anónimas
Javassist	La herramienta es fácil de utilizar	Es necesario conocer el nombre de la clase y del método donde colocar el objeto, al reemplazar la llamada de la función no es posible recuperar los parámetros que se le envían, la modificación es a nivel de *bytecode*, así que no es observable en el código
AspectJ	Facilidad para realizar los cortes, ya que se cuentan con entornos de desarrollo que ayudan en la tarea, el uso de comodines facilita la inspección de las clases y ubicación del punto de unión correcto	Es necesario conocer el nombre de la clase y del método del objeto funcional a modificar

La ventaja que manejan los objetos funcionales tanto en Java como en Scala es que permiten una reducción del código, así se manda el comportamiento más usado a funciones y en cualquier parte de la aplicación que se necesitan se usan, por lo que el código repetido se reduce.
Para colocar los cortes sobre los objetos funcionales se usó el enfoque asimétrico, eso quiere decir que una vez terminado el programa se incluyeron los aspectos, considerando esto y también que las modificaciones que se hicieron son sobre el objeto funcional, ya sea para agregar, eliminar o modificar un comportamiento en específico al objeto, la necesidad de usar patrones de diseño orientados a aspectos se elimina, dado que sólo se está trabajando sobre el comportamiento del objeto.

Además el empleo de patrones de diseño solo es aplicable si se usa el lenguaje AspectJ, si se usa Javassist para una modificación, los patrones de diseño no aplican, dado que Javassist realiza la modificación en el *bytecode* del objeto funcional y no se vuelve a saber de Javassist. En todo caso la técnica del Mixin Composition sí debería ser considerada un patrón de diseño porque se cambia no sólo la estructura del objeto, también la estructura del programa por los cambios que se tienen que hacer para que se realice la composición.

4.2. Marco de Trabajo para la aplicación de avisos sobre objetos funcionales

Dadas la dificultades que se encontraron para aplicar cortes y avisos sobre los objetos funcionales mencionadas en la sección 3.2, y como un objetivo adicional a los propuestos originalmente en el tema de tesis, se decidió crear una herramienta que proporcione un mejor soporte sobre los objetos funcionales y que mejore las desventajas de las opciones mostradas anteriormente y que son Javassist, AspectJ y Mixin Composition.
Se desarrolló un marco de trabajo, el cual se nombró como AspectFunctional, este marco de trabajo permite aplicar programación orientada a aspectos sobre los objetos funcionales. Siendo más específicos, este marco de trabajo permite colocar avisos sobre los métodos que contienen el comportamiento de los objetos funcionales, facilitando así la tarea de aplicar cortes sobre los objetos.
Los cortes con los que el marco de trabajo cuenta son:

1. Cambiar el cuerpo del método del objeto funcional
2. Modificar o cambiar los parámetros de entrada del método del objeto funcional
3. Agregar un comportamiento antes de la ejecución del objeto funcional
4. Agregar un comportamiento después de la ejecución del objeto

Usando los puntos arriba mostrados se creó una anotación para cada uno de ellos y se muestran en la Tabla 4.2 junto con su correspondiente descripción.
El marco de trabajo AspectFunctional funciona mediante anotaciones, que se colocan so-

Tabla 4.2: Anotaciones de AspectFunctional

Anotación	**Descripción**
@Replace(code=)	Permite cambiar el comportamiento completo del método
@Change(code=)	Permite cambiar los parámetros de entrada del método
@Next(code=)	Permite agregar un comportamiento después del comportamiento del método
@Previus(code=)	Permite agregar un comportamiento antes del comportamiento del método

bre los métodos de los objetos funcionales, y sobre los campos de las funciones anónimas, así el programador no tiene que crear otro archivo para indicar el corte y el aviso.

En la siguiente lista se describe el proceso completo y paso por paso que utiliza el marco de trabajo para realizar las modificaciones sobre los objetos funcionales.

1. Compila el código fuente de la aplicación a modificar.
2. Copia las clases compiladas de la carpeta por defecto llamada bin a la carpeta llamada mdf, que contendrá el código modificado, esto con la finalidad de mantener separado el código modificado del código original.
3. Busca en el directorio mdf las clases.
4. Busca en cada uno de los archivos mediante *Reflection* si existen anotaciones pertenecientes a AspectFunctional.
5. Si existen anotaciones, identifica el tipo de anotación y extrae el código que contiene.
6. Usa Javassist para modificar el *bytecode* de acuerdo al tipo de anotación detectada e inserta el código que se ingresó en la anotacíon.
7. Ejecuta la aplicación.

El proceso de funcionamiento del marco de trabajo queda ilustrado de mejor manera en la Figura 4.12, en donde se muestran todos los pasos que se realizan para modificar el bytecode.

El único pre requisito para usar el marco de trabajo es necesario tener Java 8 instalado. Este marco de trabajo se ejecuta mediante un archivo por lotes que es el encargado de arrancar la ejecución del programa llamado *Core*, y que es el encargado de realizar las modificaciones, dependiendo de la opción que se le mande es la cantidad de parámetros que se espera recibir, si se le indica que sólo modifique, el programa esperará que se le mande solo el parámetro cp, si se espera que compile y modifique, es necesario indicarlo en

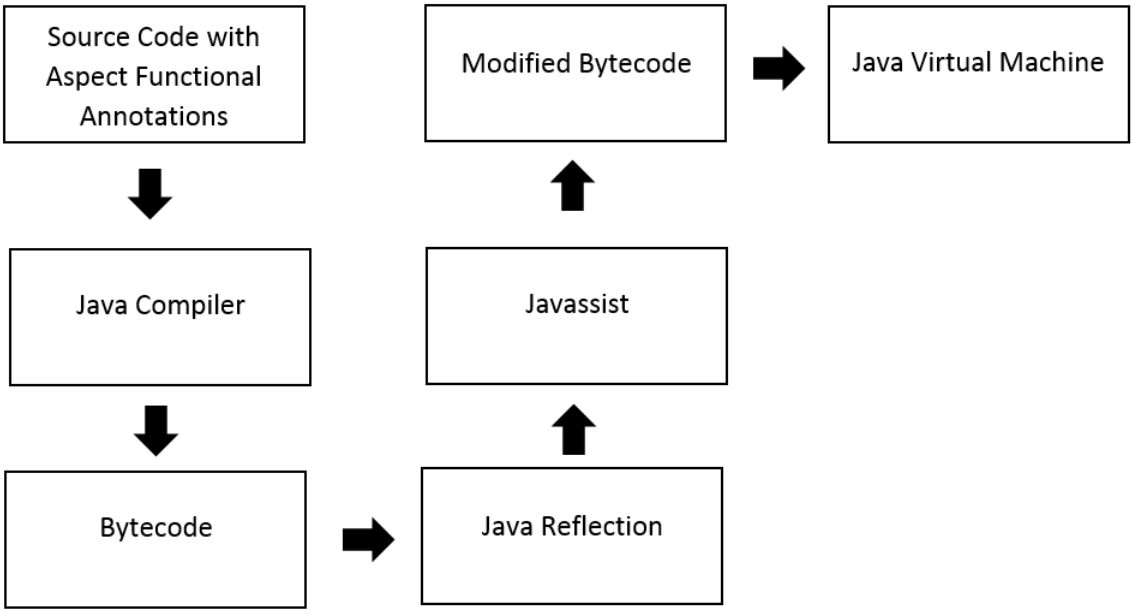

Figura 4.12: Proceso de trabajo de AspectFunctional

el parámetro opt. A continuación se muestra un ejemplo del paso de parámetros al marco de trabajo.

```
aspectFunctional.bat -opt(a) -cp(C:/Users/user/programa)
-jar(C:/Users/Desktop/library.jar) -args(5, 8) -main(paq.claseMain)
```

A continuación se explica la función que cumple cada uno de los parámetros.

1. cp: Dentro de este parámetro se manda la ruta principal del proyecto.
2. jar: Este parámetro representa a los archivos jar que son necesarios para la compilación y ejecución del proyecto.
3. args: Dentro de él se incluyen los argumentos necesarios para la ejecución del proyecto y que recibirá el método main.
4. opt: Este parámetro indica la opción a realizar por el marco de trabajo.
5. main: Indica la ruta de la clase principal del proyecto.

El parámetro main es necesario cuando se necesita ejecutar la aplicación modificada, si se indica este parámetro durante la compilación, el marco de trabajo lo agregará, y compilará la aplicación a partir de la clase principal, de lo contrario sólo compilará todos los archivos Java que se encuentren dentro de la carpeta src.
Las funciones con las que cuenta el marco de trabajo son las de modificar las clases compiladas, compilar el código y ejecutar el proyecto, el parámetro opt recibe una letra que indica el conjunto de operaciones a realizar y sus distintos valores se muestran en la Tabla

4.3.

Tabla 4.3: Valores del parámetro option

Opción	**Descripción**
m	Modificar
c	Compilar y modificar
r	Modificar y ejecutar
a	Compilar, modificar y ejecutar

El parámetro mínimo que recibe el marco de trabajo para funcionar es cp, si no lo recibe marcará un error y se detendrá el programa, ya que el marco de trabajo no tendrá la ruta para ubicar el programa y modificarlo. Si no se le manda el parámetro opt, el marco de trabajo toma por defecto el modificar las clases previamente compiladas.
No es necesario indicar en el parámetro jar la ruta de las anotaciones de AspectFunctional, el marco de trabajo lo hace por defecto, aunque sí es obligatorio agregar cualquier otro archivo jar que sea necesario para la compilación y/o ejecución del proyecto.

La anotación @Previus permite sustituir a la anotación @Change, ya que agregando un comportamiento antes del objeto también se cambiaría el valor de entrada si se indica en la anotación, aunque no se recomienda su uso dado que se perdería legibilidad en el código, ya que los nombres de las anotaciones indican la modificación a realizar.
La palabra *code* dentro de las anotaciones hace referencia al código que el programador quiere que se coloque dentro del método, los avisos hablando en términos de programación orientada a aspectos, y que se ejecutan cuando se llama al objeto.
Para indicar los valores de entrada que recibe un método éstos se representarán usando los símbolos $1, $2, y así sucesivamente por cada valor de entrada que reciba un método, hasta el número 255, que es el número máximo de parámetros que recibe un método [41].

AspectFuncional emplea *Reflection* para acceder al contenido a las anotaciones, y así saber el método y la clase a la que están afectando. Una vez obtenida la información de las clases éstas pasan a ser modificadas usando los métodos de Javassist *replace*, *inserBefore* e *inserAfter*.
Para leer las anotaciones es necesario que el programa esté compilado, pero este no debe ser ejecutado hasta que las anotaciones hayan sido procesadas y el código de los avisos haya sido agregado en los métodos señalados. Para ello se creó un programa llamado Core, el cuál es el encargado de extraer las anotaciones y realizar las modificaciones correspondientes para la generación de los aspectos. Así como también de compilar y ejecutar el programa.
AspectFunctional separa las diferentes salidas que se obtienen de un programa en diferentes directorios, los distintos directorios y su contenido son:

- src para los archivos java

- bin para las clases compiladas
- mdf para las clases modificadas

Para el manejo de los errores, dado que el proceso del marco de trabajo debe de realizarse de manera total, al generarse algún error el marco de trabajo detendrá su ejecución, mostrará el error que se generó y eliminará los archivos de la carpeta llamada mdf. Algunos de los mensajes de error que se generan se muestran en la Tabla 4.4, el resto de los errores aparecen en el anexo E.

Tabla 4.4: Valores del parámetro option

Error	**Descripción**
Error to modify in annotation replace	Descripción: Error al modificar usando la anotación replace
Error to extract annotations	Descripción: Error al extraer anotaciones

Además de mostrar los mensajes de error, se imprimen en la pantalla los mensajes que el compilador de Java o Javassist muestran, lo que permite identificar de manera más fácil en cuál anotación se encuentra el error, además de saber si el código de la anotación no es correcto y así realizar las correcciones pertinentes de manera más precisa. Las anotaciones no se colocan en variables locales, aún cuando éstas sean funciones anónimas, ya que no son accesibles mediante reflection, por lo que el marco de trabajo no identificará que hay una función anónima en la clase y no se producirán cambios en el *bytecode*.
Al colocar anotaciones sobre campos de algún tipo que no sea una interfaz funcional el marco de trabajo las ignorará, ya que valida que esos campos sean del tipo de alguna interfaz. Como se mencionó anteriormente, en las lambdas al momento de la compilación se genera una clase interna, las anotaciones que se colocan a nivel de método en la lambda se mueven a la clase interna, esto lo hace de manera automática el compilador y es lo que aprovecha el marco de trabajo para ubicar la clase y el método a modificar con la anotación.
En las clases donde existen funciones anónimas se genera un método dentro de la clase en donde se encuentran. En estos casos el marco de trabajo ubica cuántos campos con anotaciones existen y cuántos métodos generados para funciones anónimas existen, si coinciden, genera, los cambios de lo contrario, marcará un error. Por esto es recomendable que si se usan funciones anónimas, las anotaciones se coloquen a nivel de campo, y si se usan lambdas, las anotaciones se colocan a nivel de método.
Si se coloca más de una anotación sobre un campo o un método, el marco de trabajo aplicará todas las que se indiquen, solo es necesario ser cuidadoso de que las anotaciones no sean excluyentes entre ellas, por ejemplo, al usar una anotación para colocar un código después del método y en la siguiente anotación reemplazar el cuerpo del método, el código de la primera anotación queda eliminado por la acción de la segunda anotación. Por lo

que la precedencia de las anotaciones consiste en tomarlas y aplicarlas en el orden en que aparecen en el código, es decir, de arriba hacia abajo.
Dado que las anotaciones se colocan a nivel de métodos, éstas no quedan limitadas solo a los métodos de las lambdas, ya que pueden aplicarse a todos los métodos de la aplicación, y el marco de trabajo modificará estos métodos.
El marco de trabajo da soporte también al lenguaje Scala, sólo que en este lenguaje el párametro code de las anotaciones debe ser colocado en lenguaje Java y no en lenguaje Scala. Esto es así porque se ocupa Javassist para realizar los cambios en el *bytecode* y esta herramienta solo recibe código Java.
Al funcionar el marco de trabajo sólo con anotaciones y que el código correspondiente a la modificaciones se coloque dentro de los métodos de los objetos funcionales, se tiene la opción de usar los programas modificados o el código fuente con las anotaciones con otros marcos de trabajo o con lenguajes como AspectJ. Con la condición de que se use primero AspectFunctional, dado que se pueden presentar problemas si se usan primero otras herramientas, por ejemplo AspectJ, en donde el proceso de entrelazado coloca mucho código dentro de las clases, por lo que el marco de trabajo presentaría problemas para identificar dónde colocar el código de las anotaciones.

El marco de trabajo AspectFunctional mejora dos actividades fundamentales en el proceso de desarrollo de software [42], la primera actividad que mejora son las pruebas en el software, dado que permite colocar avisos de manera rápida y permite observar los parámetros que reciben los métodos, así se ubican los errores de manera más rápida y precisa, la segunda actividad que mejora el marco de trabajo es la evolución del software, ya que reduce el tiempo necesario para adaptar los programas a nuevas necesidades, dado que si por ejemplo se requiere cambiar el comportamiento de un método, se tienen varias opciones a elegir, la primera es agregar a ese método un comportamiento adicional que cambie la funcionalidad, la segunda es cambiar por completo el cuerpo de ese método y la tercera es alterar solo los parámetros que ese método recibe, ya sea modificando los parámetros que recibe el método o sustituyendo esos parámetros por otros. Este marco de trabajo aplica programación orientada a aspectos de manera asimétrica, es decir, los aspectos se encargan de describir un comportamiento dinámico adicional a un sistema.

Las anotaciones al colocarse a nivel de método sirven también para modificar los métodos tradicionales, no sólo los pertenecientes a objetos funcionales. En el código 4.15 se muestra cómo cambiar el parámetro que recibe un método, sólo es necesario colocar la anotación sobre el método y en el parámetro *code* de la anotación colocar el código que realiza el cambio.

Código 4.15: Ejemplo de uso de la anotación @Change

```
1 @Change(code="$1 = \"World\";")
2   public void m(String name){
3           System.out.println("Hello "+ name);
4       }
```

Para usar las anotaciones sobre las funciones anónimas éstas se colocan no sobre el método, ya que en las funciones anónimas las anotaciones a nivel de método no se permiten, por lo que se colocan las anotaciones sobre el campo que se quiera aplicar la modificación.

Código 4.16: Ejemplo de uso de la anotación @Previus sobre una función anónima

```
1 @Previus(code="System.out.println(\"val 1:\" +$1+ \"val 2: \"+$2);")
2 Add b =  (x, y) ->  x + y;
```

Si es necesario agregar una cantidad de código que resulte complicado colocarlo dentro de la anotación, una opción es agregar una llamada a otro método dentro de la anotación, como se muestra en el código 4.17, donde se agrega una llamada a un método dentro de una lambda.

Código 4.17: Ejemplo de la anotación @Next

```
1  Add a = new Add() {
2    @Next(code="new paq.c().m($1, $2);")
3     public int m(int x, int y) {
4             return x + y;
5     }
6    };
```

Es posible agregar más de una anotación sobre la misma lamba o función anónima, como se muestra en el código 4.18 en donde se coloca código antes y después del comportamiento original del método.

Código 4.18: Ejemplo del uso de dos anotaciones sobre el mismo objeto

```
1  Add a = new Add() {
2   @Previus(code="System.out.println(\"val 1: \"+$1+\"val 2: \"+$2);")
3   @Next(code="System.out.println(\"end m \");")
4     public int m(int x, int y) {
5             return x + y;
6     }
7    };
```

El uso del marco de trabajo tiene la ventaja, comparándolo con AspectJ de que la sintaxis es más reducida, ya que no es necesario crear un aspecto, definir un corte y colocar un aviso, además tiene la ventaja de que el contenido del aviso queda dentro de la anotación y al observar el código fuente se aprecian los cambios que se realizarán, por lo que el código es más claro. Como desventaja se tiene que el soporte es más limitado, dado que solo se tiene el punto de unión de la ejecución del método con los avisos *after*, *before* y *around*.

El marco de trabajo al funcionar mediante Javassist, tiene todas las limitaciones mencionadas anteriormente. Las anotaciones son compatibles con el lenguaje Scala y son usadas de la misma manera que en el lenguaje Java. A continuación se muestran algunos ejemplos de su uso sobre los mecanismos de Scala. En el código 4.19 se observa una función anónima y se muestra cómo exponer el parámetro que recibe la función.

Código 4.19: Ejemplo de la anotación @Next en Scala

```
1  @Next(code="System.out.println($1);")
```

```
2   val af = ( a : Double) => {
3   /* function code */ };
```

Las anotaciones también son aplicables a los métodos en el lenguaje Scala, como se observa en el código 4.20.

Código 4.20: Ejemplo de Javassist en Scala

```
1  @Previus(code="System.out.println(\"in method m\");")
2  def m(){
3   \\method code
4  }
```

En los códigos 4.21, 4.22, 4.23 y 4.24 se muestra cómo realizar modificaciones a las funciones que se tomaron del caso de estudio y a las cuales se les realizaron cortes.

Código 4.21: Uso de la anotación Next sobre una lambda en Java

```
1  public static Eliminar eliminarObjeto = new Eliminar()  {
2    @Override
3     @Next(code = "JOptionPane.showMessageDialog
                    (null, \"Acaba de eliminar un registro\"); ")
4    public int eliminar(short x, String tipo) {}
5  }
```

Código 4.22: Uso de la anotación Replace sobre una función anónima en Java

```
1    String cadena ="double radioTierra = 6371;" +
                "double dLat = Math.toRadians($2 - $0);  "+
                "double dLng = Math.toRadians($3 - $1);  "+
                "double sindLat = Math.sin(dLat / 2.0);  "+
                "double sindLng = Math.sin(dLng / 2.0);  "+
                "double va1 = Math.pow(sindLat, 2.0) " +
                "Math.pow(sindLng, 2.0)*Math.cos(Math.toRadians($0))"+
                "*  Math.cos(Math.toRadians($2)); " +
                "double va2 = 2 * Math.atan2(Math.sqrt(va1),"+
                "Math.sqrt(1.0 - va1));  "+
                "$_ = radioTierra * va2;";
2    @Replace(code = cadena)
3    static Distancia c = (lat1, lng1, lat2, lng2) -> { };
```

Código 4.23: Uso de la anotación Previus sobre una función en Scala

```
1  @Previus(code = System.out.println($1+\" \"+$2);)
2  object Eliminar{
3    def apply(id:Short, tipo:String):Int={
4  }
```

Código 4.24: Uso de la anotación Previus sobre una función anónima

```
1  object Colisiones{
2  @Previus(code = System.out.println($1+\" \"+$2 + $3+\" \"+$4);)
3  var DetectarColision = (lat1:Double, lng1:Double,
                           lat2:Double, lng2:Double) => {
4  }
5 }
```

Como se observa en los ejemplos, es más fácil realizar modificaciones al usar las anotaciones y no es necesario conocer acerca del funcionamiento interno de Java o Scala.

Capítulo 5

Conclusiones

En el presente capítulo se presentan las conclusiones correspondientes al trabajo de tesis.

5.1. Conclusiones

La programación orientada a aspectos permite encapsular adecuadamente los asuntos de corte, también permite realizar modificaciones a los programas de manera más fácil, lo que permite la adecuación del programa a nuevos requerimientos.
Los objetos funcionales usan de instrucciones de *bytecode* que a tiempo de ejecución permiten generar código que sirve para el correcto funcionamiento de los objetos funcionales. Dentro de este código, que en la mayoría de las ocasiones pertenece a clases internas, queda el comportamiento de los objetos, por lo que para aplicar un corte es necesario conocer el nombre correcto de la clase interna y del método a cortar.
Se revisaron mecanismos y herramientas para aplicar cortes sobre los objetos funcionales y actualmente no existe una herramienta que permita aplicarlos de manera fácil para el programador, si se emplea Javassist o AspectJ, es necesario tener conocimientos del *bytecode* y del comportamiento interno de los objetos funcionales para realizar algún corte, si se emplea el *Mixin Composition*, este sólo es aplicable a funciones en el lenguaje Scala. El uso de *Mixin Composition* para aplicar programación orientada a aspectos en el lenguaje Scala tiene la ventaja de que no es necesario conocer otro lenguaje de programación, como es en este caso las anotaciones de AspectJ, por el contrario, tiene la desventajas de que sí es necesario modificar el código fuente de la aplicación.
Como un objetivo adicional y dado que se encontraron limitaciones en las herramientas para aplicar aspectos sobre los objetos funcionales se decidió crear un marco de trabajo que dé un soporte más completo, y se llamó AspectFunctional.
Al usar el marco de trabajo AspectFunctional se obtienen más ventajas al aplicar cortes sobre los objetos funcionales, como que no es necesario tener conocimiento del *bytecode* para diseñar los cortes, también se realiza una separación entre las clases modificadas por el marco de trabajo y las clases sin modificar, que el programa que tiene las anotaciones

puede ser usado en varios entornos de desarrollo, aumenta la facilidad de realizar una modificación sobre un objeto, ya que solo se coloca la anotación, todas estas ventajas se obtienen debido a que la herramienta fue diseñada específicamente para el trabajo con objetos funcionales.
En el análisis del estado de la práctica no se encontraron trabajos parecidos a éste; por lo tanto, este trabajo es uno de los primeros en mostrar cómo trabajar la programación orientada a aspectos junto con los objetos funcionales.

5.2. Trabajo futuro

Como trabajo futuro se tiene mejorar la interfaz del marco de trabajo AspectFunctional para que sea más fácil de usar, por ejemplo mediante un *plugin* y no desde línea de comandos. Mejorar las capacidades de corte del marco de trabajo para que abarque más avisos sobre los objetos funcionales y corregir las limitaciones que el marco de trabajo posee.

Probar el marco de trabajo Spring que cuenta con un módulo que permite aplicar aspectos sobre los programas, así como analizar las capacidades que el lenguaje Ptolemy tiene sobre los objetos funcionales.

Publicaciones

J. Juárez, U. Juárez, M. A. Abud, L. Rodríguez, J. L. Sánchez. Propiedades de Corte Aplicables a los Objetos Funcionales. Research in Computing Science, Volumen 126, noviembre 2016. ISSN: 1870-4069.

Glosario

H

Heterarquía.- Se refiere a la posibilidad de coexistencia de jerarquías distintas, tanto sucesivas como simultáneas, en el funcionamiento de un sistema determinado

Apéndice A Anotaciones de AspectJ con Scala

Para declarar un aspecto sólo es necesario declarar una clase con la anotación Aspect en la parte superior de la clase:

```
@Aspect
class C {}
```

Si es necesario declarar algún tipo de ejecución especial, se deben colocar esos parámetros dentro de la anotación Aspect entre paréntesis:

```
@Aspect("perthis(execution(* C.(..)))")
@Aspect("pertarget(execution(* C.(..)))")
@Aspect("percflow(execution(* C.(..)))")
```

Para declarar un corte se emplea la anotación *Pointcut* y entre paréntesis la primitiva de corte con la firma necesaria, inmediatamente abajo de la anotación se debe declarar un método, que es al que se debe hacer referencia para usar el corte:

```
@Pointcut("call(* paquete.clase.metodo(..))")
def m() {}
```

Para la declaración de avisos se emplean las anotaciones *Before*, *Around*, *After* y entre paréntesis se le pasa el nombre del método que quedó asignado al corte.
Ejemplo de un aviso *before*:

```
@Before("m()")
def n1() {
  println("Before")
}
```

Ejemplo de un aviso *after*:

```
@After("m()")
def n2() {
 println("After")
}
```

Ejemplo de un aviso *around*:

```
@Around("m()")
def n3():Any {
  println("Around")
}
```

Ejemplo de un aviso *afterReturning*:

```
@AfterReturning(pointcut = "m()",returning= "result")
def n4(result:Object) {
  println(return value : " + result)
}
```

En este último tipo de aviso se tienen que definir dos variables dentro de la anotación, la primera es *pointcut* y en ella se indica el método que tiene asignado el corte, y la segunda es la variable *returning*, en ella se coloca la variable que recibirá el valor de retorno. Al método del aviso se le debe colocar como valor de entrada el nombre de la variable *returning* junto con su tipo de dato.
Ejemplo de un aviso *afterThrowing*:

```
@AfterThrowing(pointcut = "m()", throwing = "e")
def m3(e : Exception)  {
 e.printStackTrace();
}
```

Para este aviso se tiene que definir además de la variable *pointcut* la variable *throwing*, esta última variable es donde se atrapa la excepción que se produzca, y el método del aviso la tiene que recibir como parámetro, siendo su tipo de dato alguna excepción.
También se manejan los cortes anónimos solo colocando la primitiva con la firma dentro de la anotación del aviso, en lugar de declarar un Pointcut y colocar el método que tiene asignado este:

```
@Before("execution(* p.c*(..)) && this(x)")
def m() {
  println("Before");
}
```

La primitiva de corte *if* no está soportada en el sistema de anotaciones, así que si se requiere usar una estructura de este tipo, es necesario hacerlo mediante una función en Scala por ejemplo:

```
@Pointcut("call(* *.*(int)) && args(i) && if()")
public static boolean m(int i) {
  return i > 0;
}
```

Y esto sería el equivalente a realizar un corte con la sintaxis normal:

```
pointcut m(int i) : call(* *.*(int)) && args(i) && if(i > 0);
```

La variable *proceed* es necesaria cuando se requiere retomar el curso original del programa al usar el aviso *around*, para usarla en el sistema de notaciones esta debe ser recibida en el método del aviso como una variable del tipo *ProceedingJoinPoint*, para así dentro del método poder usar el método de la variable *proceed()*.

```
@Around("execution(* *.*.apply(..))")
def m(joinPoint: ProceedingJoinPoint): Any = {
  joinPoint.proceed()
}
```

La variable *joinPoint* sirve para mostrar el punto de unión en donde se realiza algún corte, para usarla es necesario crear una variable de tipo *JoinPoin* y que el método que tiene el aviso la reciba como parámetro:

```
@Before("within(*.*)")
def m(joinPoint: JoinPoint) = {
  println(joinPoint)
}
```

Para declarar precedencia entre dos aspectos se tiene la anotación *DeclarePrecedence* que recibe como parámetros los aspectos en el orden como deben ser considerados:

```
@Aspect
@DeclarePrecedence("paq.clase1, paq.clase2")
class C { }
```

Para la declaración de políticas se tiene dos anotaciones, la primera sirve para la declaración de advertencias y es la anotación *DeclareWarning* en donde se tiene que declarar un objeto *singleton* y dentro una variable de tipo cadena de caracteres que es la que contendrá el aviso cuando el corte sea ubicado:

```
@DeclareWarning("call(* paq.class.m(..))")
object mensaje{
  val mensaje:String = "Advertencia";
}
```

La segunda anotación sirve para la declaración de errores y es *DeclareError*, también es necesario declarar dentro de un objeto *singleton* una variable que contendrá el aviso del error:

```
@DeclareError("call(* paq.class.m(..))")
object mensaje{
   val mensaje:String = "AdvertenciaError"
}
```

En las pruebas que se realizaron con estas dos anotaciones para la declaración de políticas, se observó que no tuvieron ningún efecto al momento de la compilación y ejecución usando SBT.
La anotación *SuppressAjWarnings* sirve para que el compilador ignore los mensajes de advertencia, por ejemplo, cuando la firma de un método no coincide con ninguna de las clases donde el aspecto tiene efecto y se coloca arriba del aviso que presenta la advertencia. Pero al usar SBT esta anotación no aplica, dado que si no coincide la firma al ejecutar el aspecto, este no muestra nada, ni siquiera el mensaje de advertencia. La anotación *AdviceName* ya no se utiliza y se sustituyó por la primitiva de corte *adviceexecution()*.
La anotación *DeclareParents* es una instrucción de corte estático, y sirve para que una clase implemente una interfaz, para usar esta anotación en Scala, es necesario un *trait* que suplirá a la interfaz, la anotación requiere que se declaren dos campos, el primero de ellos llamado *value* en donde se coloca la firma de la clase a la cual se le realizará el corte estático, y un segundo parámetro llamado *defaultImpl* en donde se coloca el nombre de la clase estática que implementa a la interfaz, abajo de la anotación es necesario declarar

un campo del tipo de la interfaz, para usar esos campos y métodos es necesario crear un objeto de la clase que implementó la interfaz y convertirlo al tipo de la interfaz, para así con ese objeto llamar a los métodos y campos.

```
trait Md {
  var x = "Hi"
  def m(): Unit = println("m")
}

class C {}

object Test {
  def main(args: Array[String]) {
    var c: C = new C()
    var obj: Md = c.asInstanceOf[Md]
    obj.m()
    println(obj.x)
  }
}


class MdImpl extends Md {}

@Aspect
class MethodLogger {
  @DeclareParents(value = "principal.C",
    defaultImpl = classOf[MdImpl])
   private var m: Md = null
}
```

La anotación @DeclareMixin tiene el mismo funcionamiento que la anotación @DeclareParents, aunque implementación distinta. la anotación recibe como parámetro la firma de la clase a la cual se quiere agregar el nuevo comportamiento, y abajo de la anotación se coloca un método estático que retorna un objeto de la clase estática que implementa a la interfaz. Una vez hecho esto, se procede a usar los objetos como en el código anterior.

```
class MdImpl extends Md {}

@Aspect
class MethodLogger {
  @DeclareMixin("principal.C")
  def implInter(): Md = {
    return new MdImpl();
  }
}
```

Apéndice B Anotaciones de AspectJ con Java

Para crear un aspecto es necesario crear una clase Java y declararla como un aspecto con la anotación:

```
@Aspect
public class MainAspect {}
```

Una vez creado el aspecto se deben declarar los cortes en puntos dentro de él:

```
@Pointcut("within(paq.clase)")
public void m(){}
```

Abajo de la declaración del corte se debe colocar un método, este método estará ligado al corte y este es el que se debe indicar en la declaración de los avisos. A continuación se muestra la declaración de los avisos *around, after* y *before*, de igual manera abajo de la anotación de cada aviso se debe colocar un método que contendrá el comportamiento del aviso:

```
@Before("m()")
public void n1() {
  System.out.println("Before");
}

After("m()")
public void n2() {
  System.out.println("After");
}

@Around("m()")
public void n3() {
  System.out.println("Around";
}
```

El sistema de anotaciones también maneja los cortes anónimos, para declararlos solo es necesario colocar la firma dentro del aviso en lugar de declarar un corte mediante la anotación *Pointcut*:

```
@Before("call(int Paq.Class.m(int, int))")
public void n(int x, int y) {
  System.out.println(x+"  "+ y);
}
```

Si lo que se necesita es capturar el valor de retorno de un método, se emplea el aviso *afterReturning*, el cuál necesita dos variables, la primera llamada *pointcut*, en donde se coloca el punto de unión, y la segunda, llamada *result*, que es una variable que recibirá el valor de retorno:

```
@AfterReturning(pointcut = "m()", returning= "result")
public void m2(Object result) {System.out.println(return value : " +
    result)
}
```

Para capturar algún tipo de excepción que se genere durante la ejecución se cuenta con el aviso *afterThrowing*, este necesita además de definir una variable con el corte, una variable llamada *throwing* donde se capturará la excepción que se llegue a producir:

```
@AfterThrowing(pointcut = "m()", throwing = "e")
public void m3(Exception e)  {
   e.printStackTrace();
}
```

Para usar la variable *procced* en el aviso *around* es necesario que el método del aviso reciba una variable del tipo *ProceedingJoinPoint*, para que dentro del método se use el método *proceed()* de esta variable y continuar con la ejecución del programa:

```
@Around("call(int paq.class.m(int, int)) && args(x, y)")
public void n(int x, int y, ProceedingJoinPoint j)
 throws Throwable {
   j.proceed(x +1, y + 1);
}
```

El sistema de anotaciones no soporta la primitiva de corte *if*, por lo que sí es necesario crear un corte como el que se muestra a continuación:

```
pointcut s(int i) : call(* *.*(int)) && args(i) && if(i > 0);
```

El corte usando anotaciones se debe hacer indicando la primitiva *if* en el corte pero sin ninguna condición en su interior, la condición se coloca en un método estático que devolverá un valor cierto o falso:

```
@Pointcut("call(int paq.class.m(int, int))
&& args(i, j) && if()")
public static boolean s(int i, int j) {
  return i > 0;
}
```

Para la declaración de políticas se tienen dos anotaciones, la primera para declarar advertencias y es *DeclareWarning* y la segunda para declarar errores y es la anotación *DeclareError*, las dos se muestra a continuación y es necesario declarar una variable estática de tipo cadena de caracteres que mostrará el mensaje, ya sea de error o de advertencia al coincidir el corte:

```
@DeclareWarning("call(int paq.class.m(int, int))")
static final String mensaje1 = "Advertencia";

@DeclareError("call(int paq.class.m(int, int))")
static final String mensaje2 = "Error";
```

Para usar la variable de AspectJ *thisJoinPoint* es necesario además de importarla, declararla como parámetro que el método del aviso recibirá:

```
@Before("m()")
public void n(JoinPoint joinPoint) {
  System.out.println(joinPoint);
}
```

Cuando el compilador muestra advertencias, como por ejemplo cuando el punto de unión no ha sido identificado y se muestra un mensaje, existe la anotación *SuppressAjWarn ings* que hace que el compilador ignore estos mensajes de advertencia y no los muestre:

```
@SuppressAjWarnings
@Before("m(x, y)")
public void n(int x, int y, JoinPoint joinPoint) {
  System.out.println(x+"  "+ y);
}
```

Para el corte estático AspectJ cuenta con la anotación *DeclareParents*, que sirve para implementar interfaces, en el siguiente código se muestra el uso de ésta anotación.

```
@Aspect
class MainAspecto {
  public static class Impl implements I {
   @Override
   public void m() {
     System.out.println("m");
   }
}

@DeclareParents(value = "C", defaultImpl = Impl.class)
private I var;

}

interface I {
  public int x = 5;
  void m();
}

class C {}

public class Main {
  public static void main(String[] args) {
    C c = new C();
    I obj = ((I) c);
    obj.m();
    System.out.println(obj.x);
  }
}
```

La anotación *DeclareParents* requiere que se declaren dos campos, el primero de ellos *value* en donde se coloca la firma de la clase a la cual se le realizará el corte estático, y

defaultImpl en donde se coloca el nombre de la clase estática que implementa a la interfaz, abajo de la anotación es necesario declarar un campo del tipo de la interfaz, para usar esos campos y métodos es necesario crear un objeto de la clase que implementó la interfaz mediante la anotación y convertirlo al tipo de la interfaz, para así con ese objeto llamar a los métodos y campos.
También es posible colocar lambdas dentro de la interfaz y para usarlas se sigue el mismo procedimiento explicado en el párrafo anterior.

```
interface I {
  Suma a = new Suma() {
   public int m(int x, int y) {
     return x + y;
   }
  };
}
```

La anotación para corte estático *DeclareMixin* es similar a la anotación vista anteriormente llamada *DeclareParents*, se tomó el código de la anotación anterior y se modificó para adaptarlo a la nueva anotación, la anotación recibe como parámetro la firma de la clase a la cual se quiere agregar el nuevo comportamiento, y abajo de la anotación se coloca un método estático que retorna un objeto de la clase estática que implementa a la interfaz. Al usar esta anotación se obtiene el mismo resultado que con la anotación *DeclareParents*.

```
@Aspect
class MainAspecto {
  public static class Impl implements I {
    @Override
    public void m() {
      System.out.println("m");
    }
}

  @DeclareMixin("C")
  public static I implInter() {
    return new Impl();
  }
}

interface I {
   public int x = 5;
   void m();
}

class C {}

public class Main {
  public static void main(String[] args) {
    C c = new C();
    I obj = ((I)c);
    obj.m();
    System.out.println(obj.x);
```

```
   }
}
```

Apéndice C Patrones de diseño orientados a aspectos con objetos funcionales en Java

A continuación se muestran códigos de ejemplo de cada uno de los patrones de diseño orientados a aspectos que son aplicables a los objetos funcionales en el lenguaje Java.

Variante del patrón Huevo de Cuco

En el siguiente código se muestra la lambda llamada *a* y una función anónima llamada *b*, junto con la interfaz funcional de la que dependen ambas funciones.

```
public class Main {
  public static void main(String[] args) {
    Suma a = new Suma() {
      public int m(int x, int y) {
        return x + y;
      }
    };

    Suma b = (x, y) -> x + y;

    System.out.println(a.m(20, 10));
  }
}

interface Suma {
  public int m(int x, int y);
}
```

Como no es posible sustituir un objeto por otro como lo plantea el patrón huevo de cuco, lo que se hace con el aspecto, que se muestra a continuación, es usar un aviso *around* para sustituir la ejecución del objeto original y cambiarla por el resultado de otro objeto funcional, en este caso llamado *b*, se muestran dos cortes llamados *cuco* y *cuco2*, el primero es para sustituir el resultado de la lambda, el segundo aplica para la función anónima.

```
public aspect MainAspect {
```

```
  public pointcut cuco(int x, int y) :
    execution(int huevoDeCuco.Main.*.m(int, int)) && args(x, y);

  public pointcut cuco2(int x, int y) :
    execution(int huevoDeCuco.Main.lambda$0(int, int)) && args(x, y);


  int around(int x, int y) :cuco(x, y){
    return b.m(x, y);
  }

  Resta b = new Resta() {
    public int m(int x, int y) {
      return x - y;
    }
  };
}
```

Política

A contunuación se muestra una lambda llamada *a*, que es usada para sumar dos números.

```
public class Main {
  public static void main(String[] args) {
    Suma a = new Suma() {
      public int m(int x, int y) {
        return x + y;
      }
    };

    System.out.println(a.m(20, 10));
  }
}
```

Ahora suponiendo que queremos que el programador no use ciertas interfaces como interfaces funcionales, podemos usar el patrón política para que muestre una advertencia si se crean lambdas usando las interfaces funcionales que se designen como prohibidas.

```
public aspect MainAspect {

  declare warning :
    initialization(politica.Suma.new(..))
      : "no is a functional interface";

}
```

Control de Borde

Las siguientes dos clases llamadas *Funcional* y *OO* contienen objetos funcionales.

```
public class Funcional {

  public static void main(String[] args) {
    Suma a = new Suma() {
      public int m(int x, int y) {
        return x + y;
      }
    };

  System.out.println(a.m(20, 10));
  System.out.println(new OO().Resta(20, 10));

}

}

public class OO {
  public int Resta(int x, int y) {
    return x - y;
  }
}
```

En el siguiente aspecto se muestra cómo aplicar el patrón control de borde sobre las clases mostradas en el código anterior, en donde se observan distintos cortes, algunos aplican sobre las clases, uno sobre el método *Main* y uno más sobre una lambda.

```
public aspect MainAspect {

  public pointcut FuncionalRegion() :
    within(controlDeBorde.Funcional);

  public pointcut OORegion() :
    within(controlDeBorde.OO);

  public pointcut MainMethod() :
    withincode(public void controlDeBorde.Funcional.main(..));

  public pointcut a() :
    withincode(public void controlDeBorde.Funcional.a(..));

  pointcut regionsOfInterest() :
    OORegion( ) ||
    MainMethod( ) ||
    a();

}
```

Control de Excepciones

El siguiente código muestra una lambda que se encarga de mostrar dos cadenas de caracteres.

```
public class Main {

  A a = new A() {
    public void ma(String x, String y) {
      System.out.println(x + y);
    }
  };

  public static void main(String[] args) {
    new Main().a.ma("Hello", "World");
  }
}

interface A {
  public void ma(String x, String y);
}
```

En el siguiente ejemplo se muestra cómo aplicar el patrón control de excepciones a la llamada sobre el método del objeto funcional.

```
abstract aspect Aspect {

  abstract pointcut operations();

  before() throws MyException : operations() {
    b.mb();
  }

  B b = new B() {
    public void mb() {
      try {
        throw new MyException();
       } catch (Exception e) {
         System.out.println(e);
       }
    }
  };
}

public aspect MainAspect extends Aspect {
  pointcut operations() : call(* controlDeExcepciones.A.*(..));
}

class MyException extends Exception {
}

interface B {
  public void mb();
```

```
}
```

Objeto Trabajador

En el siguiente código se muestran dos lambdas, llamadas a y b.

```
public class Main {

  public static void main(String[] args) {
    Suma a = new Suma() {
      public void m(int x, int y) {
        System.out.println(x + y);
      }
    };

    Resta b = new Resta() {
      public void n(int x, int y) {
        System.out.println(x - y);
      }
    };

    a.m(20, 10);
    b.n(20, 10);
  }

}

interface Suma {
  public void m(int x, int y);
}

interface Resta {
  public void n(int x, int y);
}
```

Si se quiere que esas dos funciones sean concurrentes, se emplea el patrón objeto trabajador como se muestra en el siguiente aspecto. En donde cada que se ejecute una de las funciones pasarán a ejecutarse dentro de un hilo.

```
abstract aspect MainAspectAbstract {
  abstract pointcut corte();

  void around() : corte() {
    Runnable worker = new Runnable() {
      public void run() {
        proceed();
      }
    };
    Thread asyncExecutionThread = new Thread(worker);
    asyncExecutionThread.start();
  }
```

```
}

public aspect MainAspect extends MainAspectAbstract {

  public pointcut corte():
    execution(void objetoTrabajador.Main.*.m(int, int))
    || execution(void objetoTrabajador.Main.*.n(int, int));

}
```

Regulador

A continuación se muestra una lambda que sirve para sumar dos números.

```
public class Main {
  static Suma a = new Suma() {
    public int m(int x, int y) {
      return x + y;
    }
   };

  public static void main(String[] args) {
    System.out.println(a.m(20, 10));
  }

}
```

Si lo que se quiere es controlar la ejecución de esa lambda se emplea el patrón regulador, como se muestra en el código siguiente, en donde se validan los valores que el objeto funcional recibe y si cumplen con una determinada condición, el programa continúa normalmente, de lo contrario, retornará un valor de cero.

```
public aspect MainAspect {

  int around(int x, int y): execution
    (int regulator.Main.*.m(int, int)) && args(x, y) {
  int res = 0;
  if (x > 10 && y > 5) {
    res = proceed(x, y);
  }
  return res;
  }

}
```

Espectador

Tomando como base la lambda mostrada en el código del ejemplo anterior y suponiendo que se quieren conocer los valores de entrada y de retorno de ese método, se tiene el patrón

espectador, mostrado en el siguiente código, en donde se observa el corte llamado *Aspect*, que corta a la lambda, abajo del corte se tienen dos avisos, el primero para ver los valores de entrada al objeto funcional, y el segundo para ver su valor de retorno.

```
public aspect MainAspect {

  public pointcut espect(int x, int y) :
    execution(* espectador.Main.Suma*.m*(..)) && args(x, y);


  before(int x, int y) : espect(x, y) {
    System.out.println("Entrada "
    + thisJoinPointStaticPart.getSignature().getName() + " " + x
    + " " + y);
  }

  after() returning(int x) : espect2() {
    System.out.println("Retorno "
    + thisJoinPointStaticPart.getSignature().getName() + " " + x);
  }

}
```

Parche

Si se quiere agregar una nueva lambda a una función se hace de la misma manera como se agregan campos usando corte estático con AspectJ.

```
privileged aspect MainAspect {
  Suma parche.Main.b = new Suma() {
    public int m(int x, int y) {
      return x + y;
    }
  };
}
```

En el siguiente código se muestra el uso de dos lambdas, la primera que se encuentra dentro de la clase, y la segunda que se agregó usando el patrón parche.

```
public class Main {
  static Suma a = new Suma() {
    public int m(int x, int y) {
      return x + y;
    }
  };

  public static void main(String[] args) {
    System.out.println(a.m(20, 10));
    System.out.println(new Main().b.m(10, 10));
  }
}
```

Se observó que no es posible agregar una función anónima a una clase, ya que se genera un error de tipo *NoSuchMethodError*, porque AspectJ no puede colocar la función anónima dentro de la clase que se le indica, y al llamarla desde el programa principal se genera el error.

Extensión

En el siguiente ejemplo se tiene una lambda que se encarga de mostrar un mapa y recibe el tamaño de acercamiento del mapa y la latitud y longitud para mostrarlo.

```
public class Main {
  ShowMap a = new ShowMap() {
    public void show(int zoom, double lat, double lon) {
      // code to show
    }
  };

  public static void main(String[] args) {
    new Main().a.show(10, 19.34, 18.45);
  }
}

interface ShowMap {
  public void show(int zoom, double lat, double lon);
}
```

Si se quiere extender la funcionalidad de la lambda del código anterior, se emplea el patrón extensión, en el siguiente ejemplo si el valor zoom sobrepasa a diez, se hace uso de otra lambda para mostrar la misma ubicación pero en un mapa distinto que tiene más detalle.

```
public aspect MainAspect {

  ShowNewMap b = new ShowNewMap() {
    public void show(int zoom, double lat, double lon) {
     // code to show
    }
  };

  public pointcut extend(int zoom, double lat, double lon):
    execution(void extencion.Main.*.show(int, double, double)) &&
    args(zoom, lat, lon);

  void around(int zoom, double lat, double lon): extend(zoom, lat, lon)
     {
    if (zoom > 10) {
      b.show(zoom, lat, lon);
    } else {
    proceed(zoom, lat, lon);
    }
  }
```

```
}

interface ShowNewMap {
  public void show(int zoom, double lat, double lon);
}
```

Agujero de Gusano

En el siguiente código se muestran varias lambdas, la primera llamada *a* recibe dos objetos y envía uno a la lambda llamada *b*, esta al recibir el objeto lo envía a la lambda llamada *c* que se encarga de imprimirlo.

```
public class Main {

  C c = new C() {
    public void mc(Object z) {
      System.out.println(z);
    }
  };

  B b = new B() {
    public void mb(Object w) {
      c.mc(w);
    }
  };

  A a = new A() {
    public void ma(Object x, Object y) {
      b.mb(y);
    }
  };

  public static void main(String[] args) {
    new Main().a.ma("Hello", "World");
  }
}

interface A {
  public void ma(Object x, Object y);
}

interface B {
  public void mb(Object w);
}

interface C {
  public void mc(Object z);
}
```

En el siguiente código se muestra el empleo del patrón agujero de gusano para reducir el viaje del objeto entre los métodos, con este patrón se eliminan las llamadas intermedias dejando solo la inicial y la final.

```
public aspect MainAspect {

  pointcut espacioLlamadas(A obj, Object n, Object v):
    execution(* agujerodegusano.A.*(Object, Object))
    && this(obj)
    && args(n, v);

  pointcut espacioLlegada(C obj, Object v):
    execution(* agujerodegusano.C.*(Object))
    && this(obj)
    && args(v);

  pointcut agujeroGusano(A c1, Object n, Object v1, C c2, Object v2):
    cflow(espacioLlamadas(c1, n, v1))
    && espacioLlegada(c2, v2);

  void around(A c1, Object n, Object v1, C c2, Object v2):
    agujeroGusano(c1, n, v1, c2, v2) {
    if (v1.getClass().toString().equals("class java.lang.String"))
      proceed(c1, n, v1, c2, v2);
    else {
      System.err.println("Error");
    }
  }
}
```

Apéndice D Patrones de diseño orientados a aspectos con objetos funcionales en Scala

En esta sección se muestra la implementación de patrones de diseño orientados a aspectos usando el lenguaje Scala y anotaciones de AspectJ.

Variante del patrón Huevo de Cuco

El siguiente ejemplo muestra una función que realiza una suma, y un objeto que usa la función.

```
object Test {
  def main(args: Array[String]) {
    println(A.apply(10, 20, 30))
  }
}

object A {
  def apply(x: Int, y: Int, z: Int): Int = {
    x + y + z
  }
}
```

Se intentó implementar el patrón huevo de cuco de la manera tradicional, realizando un corte al constructor del objeto y cambiándolo por otro, pero esto genera un error al no ser objetos del mismo tipo, esto sucede aunque a los dos objetos, el original y el que sustituye, heredaran de un mismo trait. Al no lograr implementar el patrón de la manera tradicional, se creó una variante del patrón que tiene el mismo efecto sobre el programa al que afecta, pero no sustituye el objeto, sustituye la llamada, intercepta los valores que se le envían y llama a otro objeto que realiza una operación diferente de la original, la implementación de esta variante del patrón se muestra en el código siguiente.

```
@Aspect
class MethodLogger {

  @Pointcut("call(* principal.A*.apply*(..)) && args(x, y, z)")
```

```
  def espacioLlegada(x: Int, y: Int, z: Int) {}

  @Around("espacioLlegada(x, y, z)")
  def worm(x: Int, y: Int, z: Int): Int = {
    return B.apply(x, y, z)
  }

}

object B {
  def apply(x: Int, y: Int, z: Int): Int = {
    x * y * z
  }
}
```

Política

Se tiene un objeto llamado *Test* que usa una función, suponiendo que no se quiere emplear funciones en ese objeto, se tiene el patrón política que permite lanzar una advertencia cuando se genera determinado corte, así en este caso se lanza una advertencia en caso de que se llame al método *apply* dentro de ese objeto.

```
object Test {
  def main(args: Array[String]) {
    A(10, 20)
  }
}
object A {
  def apply(x: Int, y: Int): Int = {
    x + y
  }
}

@Aspect
class MethodLogger {
  @DeclareWarning("call(* principal.*$.apply$(..))")
  object mensaje {
    val mensaje: String = "No usar funciones en esta clase";
  }
}
```

Control de Borde

A continuación se muestran distintas funciones que son usadas en un objeto llamado *Test*.

```
object Test {
  def main(args: Array[String]) {
```

```
    println(A(10, 20))
    println(B(10, 20))
    println(C(10, 20))
  }
}
object A {
  def apply(x: Int, y: Int): Int = {
    x + y
  }
}
object B {
  def apply(x: Int, y: Int): Int = {
    x - y
  }
}
object C {
  def apply(x: Int, y: Int): Int = {
    x * y
  }
}
```

Si se quieren definir regiones para colocar posteriormente avisos sobre ellas se usa el patrón control de borde, en el siguiente código se muestra su aplicación para definir cuatro regiones sobre el código anterior.

```
@Aspect
class MethodLogger {

    @Pointcut("within(principal.Test$)")
     def funcional() {}

    @Pointcut("withincode(* principal.A$.apply*(Int, Int))")
     def A() {}

        @Pointcut("withincode(* principal.Test$.main(..))")
     def MainMethod() {}

        @Pointcut(" funcional( ) || MainMethod( ) || A()")
     def regionsOfInterest() {}

}
```

Control de Excepciones

En el siguiente ejemplo se muestra una función que realiza una suma y un objeto que la emplea.

```
object Test {
  def main(args: Array[String]) {
    A(10, 20)
```

```
  }
}
object A {
  def apply(x: Int, y: Int): Int = {
    x + y
  }
}
```

Si se quiere aplicar el patrón control de excepciones al ejemplo anterior se hace de la siguiente manera.

```
@Aspect
class MethodLogger {

  @Pointcut("call(* principal.A$.apply*(..))")
  def operations() {}

  @Before("operations()")
  @throws(classOf[Exception])
  def logMethod() = {
    println(B(10, 20))
  }

}

class MyException extends Exception {}

object B {
  @throws(classOf[Exception])
  def apply(x: Int, y: Int): Int = {
    throw new MyException();
    return x - y
  }
}
```

Espectador

Tomando como base el objeto *Test* y *A* del código del patrón anterior se creó un aspecto que implementa el patrón espectador, y consiste en mostrar los valores de entrada y de retorno que recibe la función.

```
@Aspect
class MethodLogger {

  @Pointcut("execution(* principal.A$.apply*(..)) && args(x, y)")
  def espect(x: Int, y: Int) {}

  @Before("espect(x, y)")
  def logMethod(x: Int, y: Int, joinPoint: JoinPoint) = {
    println("Entrada " + joinPoint.getSignature().getName()
```

```
            + " " + x + " " + y);
  }

  @AfterReturning(
    pointcut = "espect(x, y)",
    returning = "result")
  def logAfterReturning(x: Int, y: Int,
      joinPoint: JoinPoint, result: Object) {
    println("Retorno " + joinPoint.getSignature().getName()
      + " " + result);
  }
}
```

Objeto Trabajador

En el código siguiente se tienen dos funciones y un objeto que controla a ambas.

```
object Test {
  def main(args: Array[String]) {
    A.apply(10, 20, 30)
    B.apply(10, 20, 30)
  }
}

object A {
  def apply(x: Int, y: Int, z: Int): Unit = {
    println(x + y + z)
  }
}

object B {
  def apply(x: Int, y: Int, z: Int): Unit = {
    println(x * y * z)
  }
}
```

Si se necesita que las operaciones se hagan de manera concurrente, se usa el patrón objeto trabajador, el cual se muestra en el siguiente ejemplo, este patrón coloca en un hilo cada objeto para su ejecución.

```
@Aspect
class MethodLogger {

  @Pointcut("call(* principal.A*.apply*(..))")
  def Aaspect() {}

  @Pointcut("call(* principal.B*.apply*(..))")
  def Baspect() {}

  @Around("Aaspect() || Baspect()")
  def around(joinPoint: ProceedingJoinPoint): Unit = {
```

```
    var worker = new Runnable() {
      override def run(): Unit = {
        joinPoint.proceed()
      }
    };
    var asyncExecutionThread = new Thread(worker);
    asyncExecutionThread.start();
  }

}
```

Regulador

El siguiente código es para mostrar la implementación del patrón regulador, el cual sirve para controlar de mejor manera la ejecución de los programas, transfiriendo comportamientos a aspectos en lugar de que un método o una función haga todo el trabajo, por ejemplo el siguiente código que tiene una condición, imprime un mensaje y realiza una operación.

```
object A {
  def apply(x: Int, y:Int): Int =  {
    println("Realizando resta");
    if(x<y){
      return 0
    }else{
      x - y
    }
  }
}
```

Al aplicar el patrón de diseño parte del comportamiento se pasa a un aspecto dejando al objeto solo con el trabajo de realizar la operación aritmética, como se observa en el siguiente código.

```
object Test{
  def main(args: Array[String]){
    A(10, 20)
  }
}
object A {
  def apply(x: Int, y:Int): Int =  {
    x - y
  }
}

@Aspect
class MethodLogger {
  @Around("execution(* principal.A$.apply*(..)) && args(x, y)")
  def regulator(x:Int, y:Int, joinPoint: ProceedingJoinPoint):Int = {
    var a:Array[Object]=new Array[Object](1)
```

```
    val z1 = new Integer(x)
    val z2 = new Integer(y)
    a(0)=z1
    a(1)=z2
    println("Realizando resta");
    if(x<y){
      return 0
    }else{
      joinPoint.proceed(a)
    }
  }
}
```

Parche

Este patrón requiere de corte estático para implementarse, ya que necesita agregar una nueva funcionalidad, en el siguiente ejemplo se tiene la clase llamada *MethodLogger*, la cual contiene un campo llamado m que es del tipo de un *Trait* y se usa la anotación @DeclareParents para indicar que la clase

```
class MdImpl extends T {}

@Aspect
class MethodLogger {
  @DeclareParents(value = "principal.C",
    defaultImpl = classOf[MdImpl])
   private var m: T = null
}

trait T {
  val Sum = (z: Int, y: Int) => z + y
}

class C {}

object Test {
  def main(args: Array[String]) {
    var c: C = new C()
    var obj: T = c.asInstanceOf[T]
    println(obj.Sum(20, 30))
  }
}
```

Al *trait* es posible agregarle una lambda y usarla como se usaron los métodos en los códigos anteriores.

```
trait T {
  object A {
    def apply(x: Int, y: Int): Int = x + y
  }
}
```

Extensión

Al siguiente código que consiste en el empleo de una función se le aplicará un aspecto que usa el patrón extensión para agregar una funcionalidad adicional, en este caso al cumplir la condición llamará a una función distinta de la original, de no cumplirse la condición el programa continua su ejecución de manera normal.

```
object Test {
  def main(args: Array[String]) {
    A(10, 20)
  }
}
object A {
  def apply(x: Int, y: Int): Int = {
    x + y
  }
}

@Aspect
class MethodLogger {

  @Pointcut("execution(* principal.A$.apply*(..)) && args(x, y)")
  def extend(x: Int, y: Int) {}

  @Around("extend(x, y)")
  def extendAround(x: Int, y: Int, joinPoint: ProceedingJoinPoint):
    Any= {
    if (x >= 10) {
      return B(x, y)
    } else {
      var a: Array[Object] = new Array[Object](2)
      val m = new Integer(x)
      val n = new Integer(y)
      a(0) = m;
      a(1) = n;
      joinPoint.proceed(a)
    }
  }
}
```

Apéndice E Errores mostrados por el Marco de trabajo AspectFunctional

A continuación se presentan todos los errores que el marco de trabajo AspectFunctional muestra junto con la descripción del error.

- Mensaje: Error: Whitout options
  Descripción: No se mandaron opciones al marco de trabajo
- Mensaje: Error in opt option
  Descripción: Error al procesar los datos de la opción opt
- Mensaje: Error in cp option
  Descripción: Error al procesar los datos de la opción cp
- Mensaje: Error in jar option
  Descripción: Error al procesar los datos de la opción jar
- Mensaje: Error in args option
  Descripción: Error al procesar los datos de la opción args
- Mensaje: Error in main option
  Descripción: Error al procesar los datos de la opción main
- Mensaje: Error invalid option
  Descripción: La opción pasada como parámetro no es válida
- Mensaje: Error in entry arguments
  Descripción: No se recibieron los parámetros mínimos (cp y opt)
- Mensaje: Error invalid opt
  Descripción: El parámetro opt no coincide con las opciones válidas
- Mensaje: Error to extract options
  Descripción: Ocurrió una falla al extraer el contenido de las opciones
- Mensaje: Error to create directory mdf
  Descripción: Error al crear el directorio donde se colocarán las clases modificadas

- Mensaje: Error to create directory bin
  Descripción: Error al crear el directorio donde se colocarán las clases compiladas
- Mensaje: Error to run
  Descripción: Error al ejecutar el programa
- Mensaje: Error to compile
  Descripción: Error al compilar el programa
- Mensaje: Error in Previus annotation code
  Descripción: La anotación Previus no contiene código
- Mensaje: Error in Change annotation code
  Descripción: La anotación Change no contiene código
- Mensaje: Error in Next annotation code
  Descripción: La anotación Next no contiene código
- Mensaje: Error in Replace annotation code
  Descripción: La anotación Replace no contiene código
- Mensaje: Error in Response annotation code
  Descripción: La anotación Response no contiene código
- Mensaje: Error to modify
  Descripción: No se encontró la clase o el método para modificar
- Mensaje: Error to modify in annotation previus
  Descripción: Error al modificar usando la anotación previus
- Mensaje: Error to modify in annotation change
  Descripción: Error al modificar usando la anotación change
- Mensaje: Error to modify in annotation next
  Descripción: Error al modificar usando la anotación next
- Mensaje: Error to modify in annotation replace
  Descripción: Error al modificar usando la anotación replace
- Mensaje: Error to save modified class
  Descripción: Error al guardar la clase modificada
- Mensaje: Error to print in console
  Descripción: Error al imprimir los resultados en la consola de la ejecución o la compilación
- Mensaje: Error to obtain methods or class
  Descripción: Error al obtener la clase o los métodos de esa clase

- Mensaje: Error to extract annotations
  Descripción: Error al extraer anotaciones

Bibliografía

[1] D. Wampler. *Functional Programming for Java Developers*. O'Reilly, Sebastopol, CA, 2011.

[2] R. Miles. *AspectJ Cookbook* . O'Reilly, Sebastopol, CA, 1th edition, 2004.

[3] R. Laddad. *AspectJ in Action Practical Aspect-Oriented Programming* . Manning Publications Co., Greenwich, CT, 1th edition, 2003.

[4] J. Horne. The availability manager design pattern. *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, 2006.

[5] Henrique M. Rebêlo, Ricardo Lima, Uirá Kulesza, Roberta Coelho, Alexandre Mota, Márcio Ribeiro, and José Elias Araújo. The contract enforcement aspect pattern. *Proceedings of the 8th Latin American Conference on Pattern Languages of Programs*, 2010.

[6] M. Bynens, B. Lagaisse, W. Joosen, and E. Truyen. The elementary pointcut pattern. *Proceedings of the 2Nd Workshop on Best Practices in Applying Aspect-oriented Software Development*, 2007.

[7] James Noble, Arno Schmiedmeier, David J. Pearce, and Andrew P. Black. Patterns of aspect-oriented design. In *Proceedings of the 12th European Conference on Pattern Languages of Programs (EuroPLoP '2007), Irsee, Germany, July 4-8, 2007.*, pages 769–796, 2007.

[8] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala* . Artima, Mountain View, CA, 1th edition, 2007.

[9] J. Hunt. *Design Patterns: Patterns for Practical Reuse and Design* . Springer Publishing Company, Incorporated, 2013.

[10] E. Gamma, R. Johnson, R. Helm, and J. Vlissides. *Design Patterns; Elements of Reusable Object-Oriented Software* . Professional Computing Series, Addison Wesley, 1994.

[11] Apache maven project. `https://maven.apache.org/`. Noviembre 2016.

[12] B. Aggelos and B. Eugene. Morphscala: Safe class morphing with macros. In *Proceedings of the Fifth Annual Scala Workshop*, SCALA '14, pages 18–22, New York, NY, USA, 2014. ACM.

[13] N. Grech, J. Rathke, and B. Fischer. Jequalitygen: Generating equality and hashing methods. *SIGPLAN Not.*, 46(2):177–186, October 2010.

[14] J. G. Wingbermuehle, R. D. Chamberlain, and R. K. Cytron. Scalapipe: A streaming application generator. *SIGPLAN Not.*, 46(2):177–186, October 2010.

[15] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. Rescala: Bridging between object-oriented and functional style in reactive applications. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, pages 25–36, New York, NY, USA, 2014. ACM.

[16] Guido Salvaneschi and Mira Mezini. Reactive behavior in object-oriented applications: An analysis and a research roadmap. In *Proceedings of the 12th Annual International Conference on Aspect-oriented Software Development*, AOSD '13, pages 37–48, New York, NY, USA, 2013. ACM.

[17] Vaidas Gasiunas, Lucas Satabin, Mira Mezini, Angel Núñez, and Jacques Noyé. Escala: Modular event-driven object interactions in scala. In *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*, AOSD '11, pages 227–240, New York, NY, USA, 2011. ACM.

[18] Daniel Spiewak and Tian Zhao. Method proxy-based aop in scala. *JOT: Journal of Object Technology*, 8(7):149–169, 2009.

[19] Fréderic Loiret, Romain Rouvoy, Lionel Seinturier, Daniel Romero, Kévin Sénéchal, and Ales Plsek. An aspect-oriented framework for weaving domain-specific concerns into component-based systems. *Journal of Universal Computer Science*, 17(5):742–776, mar 2011.

[20] Jaroslav Bálik and Valentino Vranić. Symmetric aspect-orientation: Some practical consequences. In *Proceedings of the 2012 Workshop on Next Generation Modularity Approaches for Requirements and Architecture*, NEMARA 12, pages 7–12, New York, NY, USA, 2012. ACM.

[21] Fabiano Cutigi Ferrari, Elisa Yumi Nakagawa, Awais Rashid, and José Carlos Maldonado. Automating the mutation testing of aspect-oriented java programs. In *Proceedings of the 5th Workshop on Automation of Software Test*, AST '10, pages 51–58, New York, NY, USA, 2010. ACM.

[22] Michihiro Horie, Satoshi Morita, and Shigeru Chiba. Distributed dynamic weaving is a crosscutting concern. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 1353–1360, New York, NY, USA, 2011. ACM.

[23] Bruno Medeiros and João L. Sobral. Implementing an openmp-like standard with aspectj. In *Proceedings of the 3rd Workshop on Modularity in Systems Software*, MISS '13, pages 1–6, New York, NY, USA, 2013. ACM.

[24] W. Cazzola and E. Vacchi. Fine-grained annotations for pointcuts with a finer granularity. *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1706–1711, 2013.

[25] R. C. Gil, E. K. Piveta, D. de Brum Saccol, and C. de Faveri. A tool for searching in unstructured code aspectj. *Proceedings of the Annual Conference on Brazilian Symposium on Information Systems: Information Systems: A Computer Socio-Technical Perspective - Volume 1*, pages 39–46, 2015.

[26] J. Singh, S. Panda, P. M. Khilar, and D. P. Mohapatra. A graph-based dynamic slicing of distributed aspect-oriented software. *SIGSOFT Softw. Eng. Notes*, 41(2):1–8, May 2016.

[27] K. Aljasser. Implementing design patterns as parametric aspects using paraaj: The case of the singleton, observer, and decorator design patterns. *Computer Languages, Systems and Structures*, 45:1 – 15, 2016.

[28] T.B. Sousa and H.S. Ferreira. Object-functional patterns: Re-thinking development in a post-functional world. In *Quality of Information and Communications Technology (QUATIC), 2012 Eighth International Conference on the*, pages 348–352, Sept 2012.

[29] Pavol Pidanic. Exploring possibilities for symmetric implementation of aspect-oriented design patterns in scala. In *Proceedings in Informatics and Information Technologies IIT.SRC 2015 Student Research Conference*, pages 262–267. Vydavatestvo STU, 2015.

[30] Akka. `https://akka.io`. Abril 2016.

[31] Java virtual machine support for non-java languages. `http://docs.oracle.com/javase/7/docs/technotes/guides/vm/multiple-language-support.html`. Agosto 2016.

[32] Introducción a expresiones lambda y api stream en java se 8 parte 2. `http://www.oracle.com/technetwork/es/articles/java/expresiones-lambda-api-stream-java-2737544-esa.html`. Julio 2016.

[33] The interactive build tool. `http://www.scala-sbt.org`. Julio 2016.

[34] An annotation based development style. `https://eclipse.org/aspectj/doc/next/adk15notebook/ataspectj.html`. Julio 2016.

[35] Wolfgang De Meuter. Monads as a theoretical foundation for aop. In *In International Workshop on Aspect-Oriented Programming at ECOOP*, page 25. Springer-Verlag, 1997.

[36] Javassist java bytecode engineering toolkit since 1999. `http://jboss-javassist.github.io/javassist/`. Diciembre 2016.

[37] Gang of four patterns in a functional light. part 1 to 4. `https://www.voxxed.com/blog/author/mario_fusco/`. Octubre 2016.

[38] G. Booch. *Object-Oriented Analysis and Design with Applications*. Pearson Education, USA, 2 edition, 2007.

[39] U. Juárez N. M. Rodríguez. Definición de un proceso para el desarrollo de software orientado a objetos funcionales. *XIII Congreso Internacional sobre Innovación y Desarrollo Tecnológico*, 2016.

[40] Eclipse ajdt. `http://www.eclipse.org/ajdt`. Julio 2016.

[41] Limitations of the java virtual machine. `https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.11`. Diciembre 2016.

[42] I. Sommerville. *Ingenieria del Software*. Pearson Addison Wesley, Madrid España, 7 edition, 2005.