Universidad de Concepción
Dirección de Postgrado
Facultad de Ingeniería - Programa de Magíster en Ciencias de la Computación

MARCO PARA LA CREACIÓN DE OPERADORES DE MUTACIÓN, GENERACIÓN Y EJECUCIÓN SELECTIVA DE MUTANTES

Tesis para optar al grado de
MAGÍSTER EN CIENCIAS DE LA COMPUTACIÓN

POR
Suilen Hernández Alvarado
CONCEPCIÓN, CHILE
septiembre, 2018

Directores: Gonzalo Rojas Durán, Nieves Rodríguez Brisaboa

... al único, al grande, al que hizo y hará posible todas las cosas, sin él nada tendría sentido... ¡¡¡Gracias!!!

Resumen

Las pruebas de software son una actividad fundamental dentro del proceso de control de calidad del software. Entre las múltiples técnicas existentes, las pruebas de mutación se consideran una aproximación efectiva para obtener un buen conjunto de casos de pruebas y, a partir de estos, detectar defectos en el código que podrían ser difíciles de identificar usando técnicas convencionales. Esta técnica consiste en inyectar errores artificiales por medio de operadores de mutación, que representan errores típicos cometidos por el programador.

El área de pruebas de mutación ha tomado interés dentro de la investigación en ingeniería de software, lo que direcciona a problemas interesantes que pueden explorarse. En un estudio realizado a través de una Revisión Sistemática de la Literatura, se ha comprobado la existencia de soluciones para el análisis de mutación tanto prácticas como teóricas, sin embargo, a pesar de que existen propuestas e implementaciones de operadores de mutación, un problema que se enfrenta es que no existe una definición de como representarlos para crear nuevos, lo que dificulta cubrir errores específicos que deseen estudiarse. A partir de este estudio, en esta tesis se presenta un Análisis Conceptual donde se identificaron y analizaron un conjunto de conceptos dentro del contexto de la mutación, con el objetivo de entender como estos pueden relacionarse para solucionar el problema identificado. Como resultado de este análisis, se definió un marco conceptual para representar nuevos operadores de mutación a través de la Programación Orientada a Aspectos. Sobre este marco se desarrolló una solución que se compone de un proceso de cuatro etapas, que engloban: la especificación y generación de operadores, la generación de los mutantes y su ejecución selectiva. Finalmente, se desarrolló una Prueba de Concepto donde se demostró la factibilidad de la solución propuesta. En esta prueba, se implementó un caso de ejemplo donde se identificaron 12 errores sobre una aplicación implementada en Java, y a partir de estos, se especificaron y generaron los operadores de mutación utilizando el lenguaje de aspectos AspectJ.

Tabla de Contenido

Índice de tablas

Índice de figuras

Capítulo 1

Introducción

1.1. Motivación

Las Pruebas de Software [8] son una actividad dentro del proceso de control de calidad del software; consisten en la verificación del comportamiento de un SUT (System Under Test) en un conjunto finito de casos de prueba. Su objetivo es proporcionar información sobre la calidad del producto a través de la búsqueda de posibles fallos de implementación [39].

Una de las técnicas dentro de las pruebas de software son las **Pruebas de Mutación**, enfoque basado en la inyección de errores artificiales que alteran el comportamiento del programa original. Estos errores se introducen mediante **operadores de mutación** [41] y la versión del SUT que los contiene se conoce como **mutante**. Esta idea fue propuesta por Richard Lipton en 1971 [29] y desarrollada con posterioridad por De Millo et. al [14] y Timothy Budd [11].

El proceso de pruebas de mutación se compone de las etapas que a continuación se relacionan, esta investigación se concentra en la primera etapa:

(i) **Generación de mutantes:** Consiste en crear mutantes a partir del programa original, aplicando operadores de mutación.

(ii) **Ejecución de casos de prueba:** Consiste en la ejecución de los casos de prueba contra el programa original y las versiones mutadas.

(iii) **Análisis de resultados:** Consiste en analizar los resultados de las ejecuciones de los casos de prueba y evaluar su calidad.

Para la aplicación de las pruebas de mutación es preciso contar con un catálogo de operadores que describan la mayor variedad de errores que se desea estudiar. Offutt et. al. [33] [34] exponen la importancia de la creación de nuevos operadores de mutación

que respondan a escenarios concretos como por ejemplo la sensibilidad de contexto en aplicaciones móviles y Polo et al. [47] refieren las ventajas de diseñar una arquitectura escalable que permita la inclusión de nuevos operadores de mutación.

Se han definido operadores de mutación de propósito general, llamados operadores tradicionales [1] [40] y otros dependientes del paradigma como los operadores orientados a objetos(OO) [2] [33], sin embargo, estos operadores no son suficientes para modelar errores específicos de una tecnología.

Desde el punto de vista práctico se han implementado distintas herramientas de análisis de mutación [33] [34] [21] [35] [24] [23] [36] orientadas al lenguaje Java, no obstante, todas tienen como denominador común la limitación a catálogos de operadores predefinidos donde agregar un nuevo operador supone la modificación del código fuente de la propia herramienta. Desde el punto de vista teórico se han hecho propuestas para la generación de mutantes a través de la programación orientada a aspectos donde también se han descrito operadores de mutación. Tal es el caso de los autores Bogacki et.al [9] [10] que usan programación orientada a aspectos para mutar programas Java, pero estas mutaciones sólo se limitan a tipos primitivos y valores nulos para objetos, y Polo et. al. [43] que describe sólo un conjunto de cinco operadores de mutación.

Un problema identificado en esta área es que no existe ninguna investigación que describa un marco genérico para representar operadores de mutación, que permita especificar y generar nuevos operadores con el fin de generar automáticamente mutantes de un SUT.

En esta propuesta se define un marco conceptual para especificar y generar operadores de mutación. Esta solución se basa en la representación abstracta del concepto de operador de mutación, definida a partir de la descripción de los elementos que lo componen. Es oportuno notar que la solución propuesta centra su atención en cómo especificar operadores de mutación y no en la calidad de estos para evaluar los casos de prueba.

[1] Son aplicables prácticamente a cualquier lenguaje de programación, definidos inicialmente para programas procedurales por Offutt et al. [40]

[2] Están específicamente diseñados para el paradigma orientado a objetos.

1.2. Hipótesis

Es posible generar operadores de mutación en base a una especificación computable.

1.3. Objetivos

1.3.1. Objetivo General

Proponer un marco que permita generar operadores de mutación de forma automática a partir de su especificación conceptual.

1.3.2. Objetivos Específicos

(i) Analizar las propuestas existentes de operadores de mutación y, específicamente cómo estos operadores son caracterizados.

(ii) Desarrollar una especificación para operadores de mutación que permita su generación y aplicación sobre un SUT.

(iii) Demostrar que la generación de un operador de mutación a partir de su especificación es posible con la tecnología existente.

1.4. Metodología

Para cumplir cada uno de los objetivos específicos, se plantea un enfoque metodológico que incluye los métodos de investigación: Revisión Sistemática de la Literatura, Análisis Conceptual y Prueba de Concepto. La utilización de estos métodos es reportada en un estudio realizado por Glass et.al. [20] dentro de la investigación en Ingeniería de Software.

(i) El primer objetivo se pretende abordar a través de una Revisión Sistemática de la Literatura. Este método permite realizar análisis exhaustivo del estado del arte e identificar los principales aportes teóricos y prácticos sobre el área de interés de nuestra investigación.

(ii) El segundo objetivo se pretende abordar a través del método Análisis Conceptual. Este método permite establecer nuevas relaciones e introducir nuevos términos y conceptos para aportar nuevos resultados. En esta investigación se tiene un grupo de conceptos a analizar y se pretende relacionarlos para obtener una solución.

(iii) El Tercer objetivo se pretende abordar a través del método Prueba de Concepto. Este método permite evaluar la viabilidad de una idea o concepto. En nuestro caso como lo que se pretende definir es una solución conceptual, es de interés demostrar si la idea a ejecutar es factible.

1.5. Estructura del Informe

Esta investigación se estructura en el siguiente orden:

Capítulo 1. Introducción: Se introduce a la motivación de la investigación, se expone el problema a tratar, y cómo será abordado.

Capítulo 2. Revisión del Estado del Arte: Se desarrolla el proceso y los resultados de la ejecución de una Revisión Sistemática de la Literatura sobre los operadores dentro de la técnica de pruebas de mutación.

Capítulo 3. Solución Propuesta: Se define un análisis conceptual y una propuesta de solución para la especificación y generación de operadores de mutación.

Capítulo 4. Prueba de Concepto: Se realiza una prueba de concepto sobre una aplicación de tecnología móvil para demostrar la viabilidad de la solución propuesta.

Capítulo 5. Conclusiones y Trabajo Futuro: Se describen los resultados de la investigación y se abordan posibles líneas para trabajos futuros.

Capítulo 2

Revisión del Estado del Arte

En este Capítulo se realiza una revisión bibliográfica donde se abordan los principales trabajos teóricos y aportes prácticos de investigaciones sobre la caracterización de operadores de mutación. Para esto se realiza una Revisión Sistemática de la Literatura basada en la definición de Petersen et. al. [42].

2.1. Método de Investigación

Una Revisión Sistemática de la Literatura (SLR) comprende la identificación, clasificación y análisis de la investigación relacionada a un área en particular. De acuerdo con Petersen et. al. [42] consiste en la definición de un conjunto de pasos como: (i)definir un grupo de preguntas de investigación, que conduzcan a la extracción de palabras claves, (ii) formar cadenas de búsquedas, (iii) filtrar el resultado considerando criterios de inclusión y exclusión, (iv) clasificar la información relevante y (v) realizar la extracción y mapeo de los datos.

2.2. Planificación del Mapeo

Para la realización del mapeo se consultaron las bibliotecas digitales: Springer Link, IEEE Xplore, Google Scholar y ACM Digital Library. El objetivo de este estudio es identificar trabajos relevantes respecto a las propuestas existentes tanto prácticas como teóricas que refieren a la caracterización de operadores de mutación. Es preciso señalar que la búsqueda de trabajos existentes en esta área se ha direccionado hacia propuestas que utilizan como solución tecnológica el lenguaje Java, ello justificado por el creciente uso de este lenguaje [1].

[1]De acuerdo al proyecto Octoverse, que analiza estadísticamente repositorios de GitHub, en el 2017 Java ocupa el 3er puesto. https://octoverse.github.com/

2.2.1. Definición de preguntas de investigación

Para poder cumplir con el objetivo del mapeo se debe trasladar este a preguntas de investigación capaces de obtener información. De nuestro estudio se derivaron cinco preguntas de investigación:

1. ¿Qué operadores de mutación se han definido para las pruebas de mutación?
2. ¿Qué herramientas de análisis de mutación existen orientadas al lenguaje Java?
3. ¿Cómo abordan las herramientas de análisis de mutación el soporte para la creación de nuevos operadores?
4. ¿Qué enfoques existen para definir y generar operadores de mutación orientados al lenguaje Java?
5. ¿Cómo aborda la programación orientada a aspectos la creación de nuevos operadores de mutación?

Estas preguntas de investigación permiten conocer el estado actual en el que se encuentra la investigación sobre los operadores de mutación, las herramientas de análisis de mutación para el lenguaje Java, como se aborda la caracterización de nuevos operadores de mutación y como se relaciona la programación orientada a aspectos dentro del campo de la técnica de pruebas de mutación.

2.2.2. Construcción de cadenas y estrategia de búsqueda

Las cadenas de búsqueda corresponden al texto que se ingresa en los buscadores de las bibliotecas digitales utilizadas. Estas cadenas fueron construidas a partir de las palabras claves o conceptos que representan las preguntas de investigación, estos conceptos son los siguientes:

- mutation operator
- mutation testing
- java

- aspect oriented programming
- mutation testing tool

A partir de estas palabras claves se construyeron un conjunto de cadenas, detalladas en la Tabla 2.1 que fueron combinadas con el operador booleano AND para mejorar el resultado.

Cadena de búsqueda
mutation operator AND mutation testing
mutation operator AND mutation testing AND java
mutation operator AND aspect oriented programming
mutation testing tool AND java
mutation operator AND java
mutation operator AND aspect oriented programming AND java

Tabla 2.1: Listado de cadenas de búsqueda. Fuente: Elaboración Propia

2.2.3. Criterios de inclusión y exclusión de documentos

Después de obtenidos los documentos tras aplicar las cadenas de búsqueda definidas, para seleccionarlos hemos especificado criterios de filtrado que refieren a los requisitos que debe cumplir un estudio para ser o no considerado, estos criterios se detallan en la Tabla 2.2.

Criterios de Inclusión	**Criterios de Exclusión**
- Operadores de mutación. - Herramientas de análisis de mutación para el lenguaje Java. - Uso de programación orientada a aspectos para alterar el comportamiento de una aplicación Java.	- Uso de la técnica de la mutación para mutar programas escritos bajo el paradigma de programación orientada a aspectos. - Herramientas de análisis de mutación para cualquier lenguaje de programación diferente a Java.

Tabla 2.2: Criterios de inclusión y exclusión de documentos. Fuente: Elaboración Propia

2.2.4. Esquema de clasificación

Una vez acotados el conjunto de documentos a analizar se procede a crear un esquema para catalogarlos, esto a través de la construcción de un esquema de clasificación. El esquema de clasificación se basa en la búsqueda de palabras clave y conceptos de relevancia que contextualizan la investigación, que ayude a comprender su naturaleza, con el objetivo de construir un mapa de categorías que permita su clasificación. En nuestro caso se crearon 3 categorías para agrupar los estudios, descritas en la Tabla 2.3. Además, para detallar en mayor profundidad el análisis hemos definido en la Tabla 2.4 un conjunto de conceptos importantes con el objetivo de seleccionar los estudios de mayor relevancia a nuestra investigación.

Categoría	**Descripción**
Operadores de Mutación	Investigaciones que refieren a la propuesta de operadores de mutación
Herramientas de análisis de mutación	Herramientas para el análisis de mutación.
Programación orientada a aspectos aplicada a la técnica de mutación	Investigaciones sobre el uso de programación orientada a aspectos para alterar el comportamiento de un SUT.

Tabla 2.3: Clasificación de investigaciones. Fuente: Elaboración Propia

Conceptos
Operadores de Mutación
Programación orientada a aspectos
Prototipo/Herramienta de Análisis de Mutación
lenguaje Java

Tabla 2.4: Conceptos relevantes dentro del estudio. Fuente: Elaboración Propia

2.2.5. Extracción de datos y mapeo

Una vez definido el esquema de clasificación, catalogamos en este esquema cada una de las investigaciones. El objetivo principal es presentar la frecuencia de las publicaciones en relación con los conceptos y categorías definidas, para identificar si se responden las preguntas de investigación que nos hemos planteado. En la Tabla

2.5 se detallan las investigaciones por categorías y en la Figura 2.1 se reportan las frecuencias.

Categoría	**Investigaciones**
Operadores de Mutación	[14] [11] [12] [19] [7] [17] [45] [18] [13] [31] [32] [44] [38] [16] [15] [33] [34]
Herramientas de análisis de mutación	[33] [34] [21] [35] [24] [23] [37] [47]
Programación orientada a aspectos aplicada a la técnica de mutación	[9] [10] [43]

Tabla 2.5: Clasificación de investigaciones. Fuente: Elaboración Propia

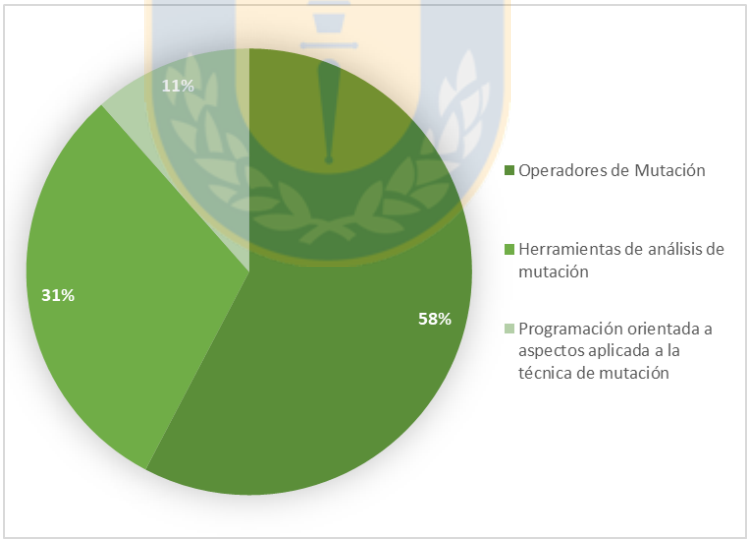

Figura 2.1: Clasificación de investigaciones por categorías. Fuente: Elaboración propia

En la Tabla 2.6 se tomaron los 26 estudios relevantes a nuestra investigación y se relacionaron con los conceptos encontrados que fueron especificados en la Tabla 2.4. Estas relaciones se presentan marcadas con 0 o 1 para denotar si está presente o no ese concepto en la investigación.

	Operadores de Mutación	**POA**	**Prototipo/Herramienta de Análisis de Mutación**	**Java**
[14]	1	0	0	0
[11]	1	0	0	0
[12]	1	0	0	0
[19]	1	0	0	0
[7]	1	0	0	0
[7]	1	0	0	0
[17]	1	0	0	0
[45]	1	0	0	0
[18]	1	0	0	0
[13]	1	0	0	0
[31]	1	0	0	0
[44]	1	0	0	1
[32]	1	0	0	0
[38]	1	0	0	1
[16]	1	0	0	1
[15]	1	0	0	1
[33]	0	0	1	1
[34]	0	0	1	1
[9]	1	1	0	1
[10]	1	1	0	1
[43]	1	1	1	1
[21]	0	0	1	1
[35]	0	0	1	1
[24]	0	0	1	1
[23]	0	0	1	1
[37]	0	0	1	1
[47]	0	0	1	1

Tabla 2.6: Relación entre conceptos encontrados dentro de los estudios de interés. Fuente: Elaboración Propia

Una vez definidos los conceptos a identificar por cada investigación, hemos encontrado que el concepto más frecuente es Operadores de Mutación, mostrado en la gráfica de la Figura 2.2, que aparece en 18 del total de investigaciones analizadas, sin embargo, estas investigaciones solo proponen operadores de mutación y no definen cómo crear nuevos.

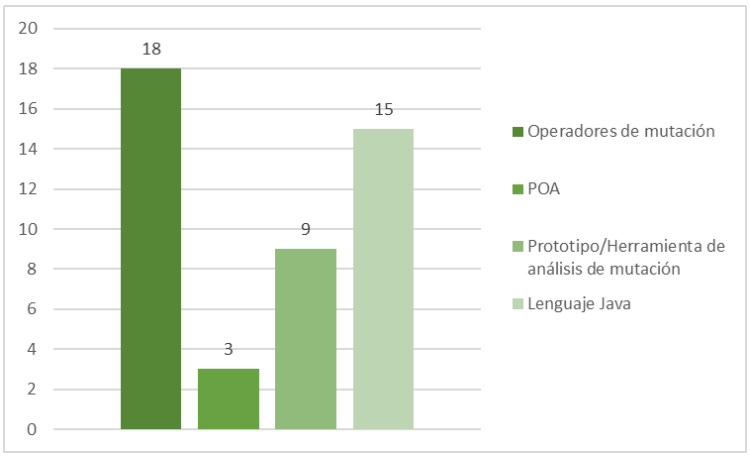

Figura 2.2: Cantidad de investigaciones por concepto. Fuente: Elaboración propia

Otra forma que hemos analizado los datos es encontrar cuántos conceptos hay presentes al mismo tiempo en cada investigación, en este caso este análisis aporta mayor utilidad porque permite extraer las investigaciones con mayor cobertura para encontrar lo buscado, que responda a información relacionada a nuestras preguntas de investigación. Este resultado es presentado en la Figura 2.3 donde puede apreciarse que solo 3 del total de las investigaciones cubren la mayor parte de los conceptos, por tanto estas representan los estudios de mayor relevancia a nuestro trabajo.

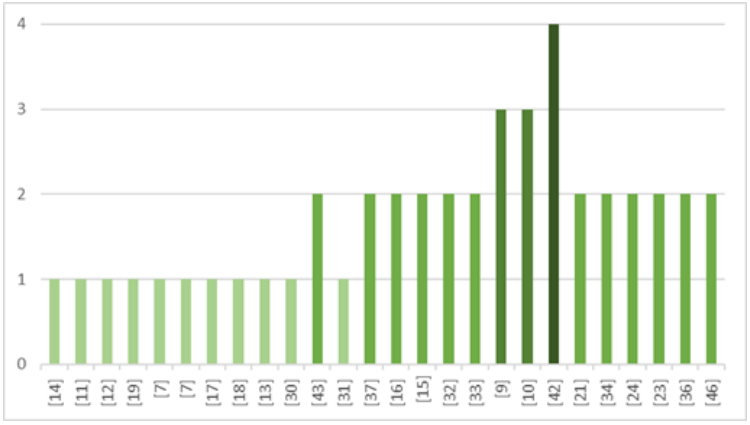

Figura 2.3: Cantidad de conceptos encontrados por cada investigación. Fuente: Elaboración propia

2.3. Reporte del Mapeo

2.3.1. Aportes Teóricos

Dentro del área de pruebas con mutación [22] han sido propuestos operadores de mutación orientados tanto para lenguajes de programación, niveles de pruebas, paradigmas como para diferentes plataformas. Sobre la idea original de las pruebas de mutación de Richard Lipton [14], De Millo et. al [11] implementan procedimientos para evaluar y medir la adecuación de los casos de prueba, basados en hacer construcciones alternativas al programa original, esta implementación fue orientada a programas escritos en Fortran, siendo este trabajo pionero en el área.

Posteriormente Delamaro et al. [12] [19] presenta por primera vez la aplicación de la mutación a nivel de integración, donde propone reproducciones de errores en programas C a nivel de las interacciones con interfaces.

Han sido propuestos además operadores de mutación orientados a lenguajes de programación específicos como C [7], C# [17], PHP [45], Python [18] y C++ [13]. También han sido definidos operadores para el paradigma de orientación a objetos [31] [32] por Ma y Offutt.

Posteriormente se han implementado catálogos de operadores en herramientas de análisis de mutación. Mateo et. al. [37][44] [47] en las herramientas Testooj y Bacterio han trabajado en el desarrollo de operadores de mutación para aplicaciones móviles en el lenguaje Java, donde Testooj toma los mutantes generados por MuJava [34] como entrada mientras que Bacterio incluye un generador de mutantes que inserta las fallas directamente en el bytecode.

Reales et al. [38] presentan una novedosa técnica de mutación para la integración multi-clase y las pruebas a nivel de sistema donde además definen 58 operadores de mutación clasificados en 5 categorías dependiendo de la falla a insertar, para su implementación hacen uso de la herramienta Bacterio.

Las pruebas de mutación también han sido abordadas a través del uso de POA, para la creación de operadores de mutación. La estrategia propuesta consiste en expresar los operadores como pointcuts para interceptar y manipular el código fuente del SUT. Bogacki et al. [9] [10] presentan un prototipo que simula las mutaciones

a través de la intercepción de los llamados al SUT que se realizan desde los casos de prueba, alterando los valores de retorno de los métodos. Este enfoque no genera mutantes físicos ni requiere la compilación de los mutantes, sin embargo, el conjunto de operadores soportados se limita a tipos primitivos y valores nulos para objetos, lo que reduce la simulación de errores. Otro trabajo en esta línea es el de Polo Usaola [43], donde se usa POA para alterar el comportamiento del código fuente del SUT, el enfoque de este trabajo define un conjunto de cinco operadores de mutación, sin embargo, no se especifica un modelo que de soporte a la creación de nuevos.

Recientemente han sido desarrollados operadores orientados a tecnologías móviles, Deng et al. [16] [15] propone 11 operadores de mutación para probar varias características de las aplicaciones de Android como código fuente, archivos XML y permisos.

2.3.2. Aportes Prácticos

En este epígrafe se analizan un conjunto de herramientas orientadas al análisis de mutación para el lenguaje Java. En la Tabla 2.7 se muestra una comparación tomando en cuenta las siguientes características:

(i) **Meta mutantes:** Generación de mutantes en una única copia del SUT.

(ii) **Tipos operadores:** Tipos de operadores de mutación que implementa la herramienta. Se especifica T(Tradicionales) y OO (Orientados a Objetos).

(iii) **Soporte operadores:** Si se implementa soporte para agregar nuevos operadores de mutación.

(iv) **Ejecución selectiva de mutantes:** Si se implementa algún mecanismo de ejecución de los mutantes generados.

Característica	MuJava	Jumble	Judy	Major	Bacterio
Año última versión	2013	2015	2016	2017	2017
Meta mutantes	no	si	si	si	no
Tipos operadores	T, OO	T, OO	T, OO	T	T
Soporte operadores	no	no	no	no	no
Ejecución selectiva	no	no	no	no	no

Tabla 2.7: Comparativa entre herramientas de Análisis de Mutación. Fuente: Elaboración Propia

- **MuJava** [33] [34]: Su propósito principal ha sido investigar operadores de mutación específicos para lenguajes de POO. Soporta un catálogo de operadores de mutación a nivel de método [31] y a nivel de clases [32] [30] del tipo tradicionales y orientados a objetos. Estos operadores están diseñados para propósito general, donde las mutaciones generan mutantes físicos por cada una de las mutaciones. Esta herramienta ha sido implementada en versiones distintas con diferentes niveles de mutación como mutación a nivel de código fuente y mutación a nivel de bytecode, con el objetivo de reducir el tiempo de generación de los mutantes. Hasta su última versión de abril de 2013 [6] esta herramienta no soporta la creación de nuevos operadores de mutación. Aunque fue liberada como código abierto, la creación de nuevos operadores supondría la modificación de su código fuente, lo que requeriría de conocimientos sobre la implementación de la herramienta, y debido a que es un sistema experimental, su extensibilidad no ha sido caracterizada con detalle.

- **Jumble** [21]: Esta herramienta ha sido desarrollada para un entorno industrial, opera directamente a nivel de bytecode con el objetivo de acelerar las Pruebas de Mutación. Implementa un conjunto limitado de operadores de mutación que son versiones simplificadas de los operadores tradicionales AOR, ASR, COR, LOR, ROR y SOR [40] [3]. Hasta su última versión de mayo de 2015 [2] no soporta ningún módulo orientado a la creación de nuevos operadores de mutación, además no se encuentra disponible como fuente abierta para realizar modificaciones en su código fuente.

- **Judy** [35]: Introduce las mutaciones en una única copia del SUT evitando la

creación de múltiples copias mutadas. Sin embargo, estas mutaciones son introducidas creando copias de cada uno de los elementos a mutar por cada uno de los mutantes lo que supone un incremento en el tamaño de las clases mutadas que puede atentar contra el rendimiento y la confiabilidad en la compilación. Este problema obligó a los autores a dar una solución que limita el número máximo de mutaciones (MAXMUT) que podrían ser aplicadas a una clase dentro de una única iteración. Para la activación de los mutantes la herramienta implementa una optimización de la técnica Mutant Schemata [46], donde a través de un enfoque referido como FAMTA Light (Fast Aspect-Oriented Mutation Testing Algorithm) utilizan POA como alternativa para activar o desactivar las mutaciones introducidas en el SUT. Al igual que MuJava soporta un conjunto de operadores de mutación a nivel de método [31] y a nivel de clases[32] [30], así como operadores específicos al lenguaje java [8]. Su última versión de noviembre de 2016 [1] no soporta ningún módulo orientado a la creación de nuevos operadores de mutación, además no se encuentra disponible como fuente abierta para realizar modificaciones a su código fuente.

- **Major** [24] [23]: Las mutaciones están integradas al compilador de Java Open JDK, transformando e inyectando las mutaciones directamente en el árbol de sintaxis abstracta durante la compilación del código fuente original, evitando la recompilación de los mutantes. Cuenta con su propio lenguaje específico de dominio que proporciona mayor control para definir el contexto de la mutación. Hasta el momento soporta un conjunto limitado de operadores de mutación [5] orientados a la eliminación de sentencias, reemplazo y manipulación de sentencias condicionales. Hasta su última versión de mayo de 2017 [4] no soporta ningún modulo que permita la creación de nuevos operadores de mutación, además no se encuentra disponible como fuente abierta para realizar modificaciones a su código fuente.

- **Bacterio** [37]: Automatiza las tareas para realizar análisis de mutación e implementa un conjunto de técnicas de reducción de costos en la mutación y el tiempo de ejecución. Las mutaciones se realizan manipulando directamente el bytecode

lo que supone evitar los costos de compilación. Implementa una arquitectura extensible con una estructura bien definida para futuras implementaciones de nuevos operadores de mutación, sin embargo, para su adición es necesaria la manipulación de su código fuente. Actualmente soporta operadores tradicionales. Hasta su última versión 2017 [47] Bacterio no soporta ningún módulo orientado a la creación de nuevos operadores de mutación. Esta herramienta no se encuentra disponible como fuente de abierta para realizar modificaciones a su código fuente.

2.3.2.1. Análisis de herramientas: flexibilidad de creación de nuevos operadores de mutación

A continuación, se presenta una comparativa entre las herramientas de análisis de mutación analizadas y la solución propuesta, tomando en consideración la flexibilidad de cada una de estas herramientas para la creación de nuevos operadores de mutación específicos a determinados contextos. Para evaluar esta característica se presenta un caso de ejemplo basado en uno de los operadores de mutación detallados en el Capítulo 4 y se valora si es posible su replicación en cada una de las herramientas analizadas. Para comparar las soluciones de cada herramienta se han tomado en consideración 3 elementos, que se presentan en la Tabla 2.8.

En este Caso de Ejemplo se utiliza el operador de mutación correspondiente al Error ERR-06 F.1. Este error detiene el servicio de Google Play antes de solicitar la recepción de actualizaciones de posición del API de Google Services y provoca que se detenga la ejecución de la aplicación. Para que se ejecute este error es necesario manipular el comportamiento del método *starLocationUpdates(..)* perteneciente a la clase *MapsActivity*, que gestiona la visualización de los puntos de geolocalización sobre el mapa. Una de las instrucciones dentro de este método es la llamada a *requestLocationUpdates(..)* en la que se manipulan uno se sus parámetros para reproducir el error deseado. En la Figura 2.4 se muestra el diagrama con las clases interceptadas por este operador y en los fragmentos 2.1 y 2.2 se muestran, el método manipulado perteneciente a la clase *MapsActivity* y el Aspecto correspondiente al Operador ERR-06.

```
1  private void startLocationUpdates()
2  {
3    if (isLocationPermissionGranted())
4    {
5        LocationServices.FusedLocationApi.requestLocationUpdates(
6                            googleApiClient, locationRequest, this);
7    }
8  }
```

Fragmento 2.1: Método a mutar perteneciente a la clase MapsActivity. Fuente: Elaboración propia

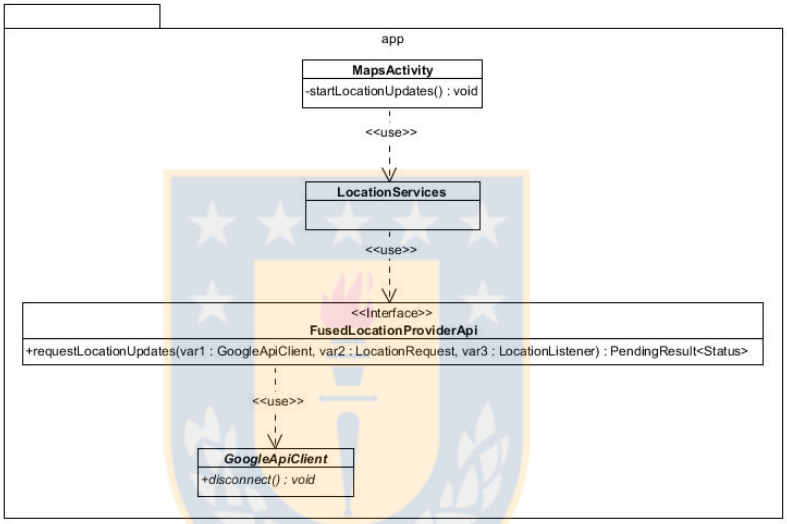

Figura 2.4: Diagrama de clases manipuladas por el Aspecto correspondiente al Operador ERR-06. Fuente: Elaboración propia

2.3.2.2. Análisis Comparativo

Para evaluar la flexibilidad de creación de nuevos operadores de mutación, se han medido 3 elementos en cada una de las herramientas de análisis de mutación:

(i) Si en el catálogo de operadores de mutación disponible en la herramienta existe algún operador que implemente el error deseado.

(ii) Si la herramienta cuenta con soporte para adicionar nuevos operadores de mutación.

(iii) Si la herramienta está publicada bajo licencia de código abierto y cuenta con la

```
1  public privileged aspect
       Aspecto_OpLoc06_MapsActivityStartLocationUpdates
2  {
3      PendingResult<Status> around
4      (FusedLocationProviderApi fusedlocationproviderapi,
           GoogleApiClient p0, LocationRequest p1, LocationListener p2):
5      withincode(void MapsActivity.startLocationUpdates())
6
7      && call (PendingResult<Status> FusedLocationProviderApi.
           requestLocationUpdates(GoogleApiClient, LocationRequest,
           LocationListener))
8      && target(fusedlocationproviderapi)
9      && args(p0, p1, p2)
10     {
11         p0.disconnect();
12         return proceed(fusedlocationproviderapi, p0, p1, p2);
13     }
14 }
```

Fragmento 2.2: Aspecto correspondiente al Operador ERR-06. Fuente: Elaboración propia

documentación suficiente para hacer modificaciones en su código fuente, donde sea posible agregar nuevos operadores de mutación a su catálogo.

Como se detalla en la Tabla 2.8 la evaluación realizada arroja lo siguiente:

(i) **Operador disponible:** Las herramientas analizadas no disponen de ningún operador de mutación [31] [40] [3] [31] [32] [30] [5] [47] que genere el tipo de mutación que se ha analizado en el caso de ejemplo.

(ii) **Soporte para operadores:** Las herramientas analizadas no implementan soporte [6] [2] [1] [4] [47] para adicionar nuevos operadores a su catálogo. Su funcionalidad está acotada al uso de los operadores de mutación que tienen implementados.

(iii) **Código abierto:** La herramienta MuJava es la única que actualmente dispone de su código fuente publicado [6] lo que posibilita su modificación. Sin embargo, al ser un sistema experimental no se encuentra suficientemente documentado.

Como se detalla en la Tabla 2.8 sobre las herramientas analizadas se midieron los elementos: Operador disponible, Soporte para operadores y Código abierto. Solo la herramienta MuJava permite la modificación de su código fuente, sin embargo, ésta por ser un sistema experimental no se encuentra documentada con detalle, lo

que dificulta su modificación. Por otra parte, ninguna de las herramientas cuenta con soporte para adicionar nuevos operadores de mutación y su catálogo es acotado, lo que no permite reproducir otros errores que se identifiquen.

Elemento	**MuJava**	**Jumble**	**Judy**	**Major**	**Bacterio**
Operador disponible	no	no	no	no	no
Soporte para operadores	no	no	no	no	no
Código abierto	si	no	no	no	no

Tabla 2.8: Comparativa entre herramientas: flexibilidad de creación de nuevos operadores de mutación. Fuente: Elaboración Propia

2.4. Conclusión

En este Capítulo se ha descrito una Revisión Sistemática de la Literatura, donde hemos presentado y analizado a través de un mapeo un conjunto de trabajos con el objetivo de responder a las preguntas de investigación que nos hemos planteado. En nuestro caso no hemos encontrado investigaciones que cubran una respuesta clara, sin embargo, se encontraron conceptos importantes dentro de las investigaciones que nos muestran que, aunque no se cubre nuestro objetivo de estudio existen trabajos que se direccionan hacia partes de las respuestas, lo que supondría que en el área de pruebas con mutación aún existen problemas a resolver.

En consecuencia, concluimos que, el análisis de mutaciones está bien establecido en la investigación de ingeniería de software, pero en la práctica se necesita una mejor escalabilidad y soporte de las soluciones existentes para la representación, especificación y generación de nuevos operadores de mutación. De acuerdo con nuestro análisis señalamos lo siguiente:

(i) No existe un modelo que permita representar operadores de mutación con el fin de especificarlos y generarlos.

(ii) La mayor parte de los trabajos analizados se direccionan hacia propuestas de nuevos operadores de mutación, lo que argumenta la creciente importancia de identificar nuevos errores que puedan generar nuevos operadores.

(iii) Existen tipos de errores que no son posibles de especificar con los operadores disponibles, por ejemplo, algunos errores específicos a aplicaciones móviles.

(iv) La adición de nuevos operadores a los catálogos ya existentes en las herramientas de análisis de mutación supondría la modificación de su código fuente.

(v) Debido al uso experimental, la mayoría de las herramientas disponibles carecen de una completa documentación técnica y no disponen de código abierto, lo que hace que su diseño no esté caracterizado a detalle, suponiendo un alto costo de aprendizaje sobre su implementación para la creación de nuevos operadores.

Capítulo 3

Solución Propuesta

En este Capítulo, se desarrolla un análisis conceptual, donde a partir de la presentación de un conjunto de conceptos, se propone una solución para especificar operadores de mutación, donde estos puedan ser generados y aplicados sobre un SUT. Esta solución está basada en el paradigma POA.

3.1. Análisis Conceptual

3.1.1. Justificación

A partir del análisis realizado en el Capítulo Revisión del Estado del Arte 2, donde no se encontraron elementos que satisficieran una respuesta precisa para nuestra investigación, en este capítulo se presentan y analizan un grupo de conceptos representativos a nuestro contexto, con el propósito de construir relaciones entre ellos que permitan combinar conocimientos con el fin de cumplir nuestro objetivo. En nuestro caso, hemos adoptado el método de investigación Análisis Conceptual (CA), y más específicamente Análisis Constructivo [27]. El Análisis Constructivo es una forma de abordar el método CA que pretende introducir nuevos términos y conceptos, y establecer nuevas relaciones entre partes de una teoría inicial existente para aportar algún nuevo resultado que amplíe nuestra teoría conceptual.

Para crear nuestra propuesta, primero, hemos analizado un conjunto de conceptos individualmente para entender cómo pueden relacionarse entre ellos y cómo es posible a través de esas relaciones construir una nueva solución. Este análisis se grafica en la Figura 3.1, donde se muestran tres fases, en la primera fase se enmarcan los conceptos: *Mutación*, *Operador de Mutación*, *POA* y *Operadores de Mutación con POA*, analizados a detalle en el epígrafe 3.1.2. En la segunda fase se muestran las relaciones

que existen entre estos conceptos, en este caso, para implementar la técnica de mutación se necesitan operadores de mutación, por otra parte, tenemos la programación orientada a aspectos, que relacionándola con el concepto operador de mutación se tiene que es posible realizar mutaciones a un programa usando este paradigma. En la tercera fase se expone cómo de la relación de los conceptos Operador de Mutación y POA se puede crear un nuevo marco para expresar la representación de un operador de mutación a través de la identificación de sus elementos y relaciones, construidos a partir de un lenguaje de aspectos.

Figura 3.1: Representación de relaciones entre conceptos. Fuente: Elaboración propia

3.1.2. Conceptos

3.1.2.1. Mutación

La mutación es una técnica de prueba basada en la introducción de errores. Consiste en realizar cambios en un programa a nivel de código fuente o compilado e implica el diseño de casos de prueba para detectar estos cambios [48].

Esta técnica se ha utilizado para validar casos de prueba. Su objetivo consiste en encontrar errores en un SUT que, si no son detectados por los casos de prueba, pueda asumirse que estos últimos están mal diseñados.

Estos errores son inyectados artificialmente al SUT por operadores de mutación y reciben el nombre de mutaciones. Las mutaciones son una copia del programa original que se está probando al que se le han introducido cambios sin impedir que el programa compile correctamente, generando así una versión defectuosa del programa original.

En la Tabla 3.1 se muestra a modo de ejemplo una función que realiza una operación aritmética de 2 números y 3 mutantes generados para dicha operación. Estos mutantes se obtienen mediante una modificación sintáctica que sustituye el operador aritmético suma (+) por los operadores resta (-), multiplicación (*) y división (/).

Versión de la función	**Implementación**

Original	int suma(int a, int b){
	return a+b;
	}

Mutante 1	int suma(int a, int b){
	return a-b;
	}

Mutante 2	int suma(int a, int b){
	return a*b;
	}

Mutante 3	int suma(int a, int b){
	return a/b;
	}

Tabla 3.1: Ejemplo de una función y sus versiones mutadas. Fuente: Elaboración Propia

3.1.2.2. Operador de Mutación

La generación de los mutantes a partir del programa original se realiza aplicando operadores de mutación. Cada operador introduce en el SUT un determinado tipo de error, estos fallos deben ser réplicas de los errores comunes que puedan cometer los programadores de forma involuntaria. Existen tipos de operadores clasificados de acuerdo con los contextos para lo que fueron implementados:

- Tradicionales: Son aplicables prácticamente a cualquier lenguaje de programación.

- Dependientes del paradigma: Están orientados a distintos paradigmas de programación, como por ejemplo Programación Orientada a Objetos.

- Específicos del lenguaje: Se definen para un lenguaje de programación en particular.

- Orientados a tecnologías: Son diseñados específicamente para alguna una tecnología específica.

En las Tablas 3.2 y 3.3 se describen algunos de los operadores de mutación Tradicionales y específicos al Paradigma Orientado a Objetos (POO).

Operador de Mutación	**Descripción**
ABS (absolute value)	Sustitución de una variable por el valor absoluto de dicha variable
ACR (array reference for constant replacement)	Sustitución de una referencia variable a un arreglo por una constante
AOR (arithmetic operator replacement)	Sustitución de un operador aritmético
CRP (constant replacement)	Sustitución del valor de una constante
ROR (relational operator replacement)	Sustitución de un operador relacional
RSR (return statement replacement)	Sustitución de la instrucción return
SDL (statement deletion)	Eliminación de una sentencia
UOI (unary operator insertion)	Inserción de operador unario

Tabla 3.2: Operadores de Mutación Tradicionales. Fuente: [40]

Operador	Descripción
AMC (access modifier change)	Reemplazo del modificador de acceso
AOC (argument order change)	Cambio del orden de los argumentos pasados en la llamada a un método
CRT (compatible reference type replacement)	Sustitución de una referencia a una instancia de una clase por una referencia a una instancia de una clase compatible
EHC (exception handling change)	Cambio de una instrucción de manejo de excepciones por una sentencia que propague la excepción y viceversa
EHR (exception handling removal)	Eliminación de una instrucción de manejo de excepciones
HFA (hiding field variable addition)	Adición en la subclase una variable con el mismo nombre que una variable de su superclase
MIR (method invocation replacement)	Reemplazo de una llamada a un método por una llamada a otra versión del mismo método
OMR (overriding method removal)	Eliminación en la subclase la redefinición de un método definido en una superclase
POC (parameter order change)	Cambio del orden de los parámetros en la declaración de un método
SMC (static modifier change)	Adición o eliminación del modificador static

Tabla 3.3: Operadores de Mutación para POO. Fuente: [33]

3.1.2.3. Programación Orientada a Aspectos

La programación orientada a aspectos (POA) es un paradigma que intenta formalizar y representar los elementos transversales a todo el sistema denominándolos Aspectos. Surgió de una investigación realizada en el Centro de Investigación Xerox PARC durante los años 80 y 90 [25].

Un **Aspecto** es una pieza de código que implementa un interés común y modifica el comportamiento de aquellas zonas del código relacionadas con el propósito del aspecto. Estas zonas del código cuyo comportamiento se captura se denominan **joinpoints** (o punto de enlace). El código que implementa el cambio que debe ejecutarse

cuando se captura un joinpoint se denomina **advice**. El código del advice puede ejecutarse antes (**before**), después (**after**) o en vez del (**around**) código del joinpoint. Estos modificadores y otros conforman el **pointcut** (o punto de corte) e indican en qué momento de la ejecución del joinpoint se debe ejecutar el código del advice. A través de estos elementos es posible alterar el comportamiento del código original sin modificarlo. También permite especificar con un fino nivel de granularidad el contexto de la alteración del comportamiento de la aplicación.

En POA, es posible definir joinpoints sobre la ejecución que se desea capturar en un programa, estos representan los lugares donde los aspectos añaden su comportamiento. En la Tabla 3.4 se detallan las palabras claves para indicar el tipo de captura que el pointcut debe hacer, seguida de la información de identificación del método, constructor u otro elemento a capturar.

Elemento a capturar	**Palabra clave**
Llamadas a método	call(signatura del método)
Ejecución de método	execution(signatura del método)
Llamadas a constructor	call new(signatura del constructor)
Ejecución de inicializador	initialization(signatura del constructor)
Ejecución de constructor	execution new(signatura del constructor)
Ejecución de inicializador estático	static initialization(nombre del tipo)
Lectura de campo	get(signatura del campo)
Asignación de valor a campo	set(signatura del campo)
Ejecución del manejador de excepciones	handler(excepción a capturar)

Tabla 3.4: Palabras claves usadas en los tipos de captura del pointcut. Fuente: Elaboración Propia

El pointcut puede incluir además operadores, caracteres comodín y otras palabras clave para especificar con exactitud los joinpoints que deben ser atrapados:

- **Operadores:**
  - ! negación.
  - || OR lógico.
  - && AND lógico.

- () paréntesis.

- **Caracteres comodín:**
  - * asterisco, para hacer referencia a cualquier cosa.
  - .. dos puntos, para sustituir un solo elemento.
  - . un punto, para hacer referencia a subpaquetes.
  - + signo más, para hacer referencia a subtipos.

- **Palabras clave:**
  - *target* hace referencia al objeto sobre el que se ejecutará el código capturado, es decir, al objeto que ejecuta el joinpoint.
  - *args* hace referencia a los argumentos del método capturado.

El entrelazado de un aspecto con un programa se realiza a través del proceso de **weaving** o **entretejido**. Este proceso es el encargado de combinar adecuadamente el código del programa original, es decir, los joinpoints, con el código que implementa el cambio que se desea introducir, en este caso, los advices de manera que, cuando el sistema esté en funcionamiento, el advice y el joinpoint se ejecuten conjuntamente. El entretejido puede ser estático o dinámico, el primer caso maneja aquellos elementos que pueden ser determinados antes que el programa comience su ejecución, normalmente en tiempo de compilación, mientras que el dinámico ocurre en tiempo de ejecución.

Para hacer uso de POA es necesario:

(i) **Lenguaje base:** Un lenguaje de propósito general donde se definan las funcionalidades básicas.

(ii) **Lenguaje orientado a aspectos:** Un lenguaje para programar aspectos.

(iii) **Herramienta para realizar entretejido:** Un programa encargado de hacer el entretejido que combine el programa y los aspectos.

```
1  package aspectos;
2  public class OperacionAritmetica {
3
4      public double sumar(double x, double y) {
5          return x+y;
6      }
7      public double restar(double x, double y) {
8          return x-y;
9      }
10     public double multiplicar(double x, double y) {
11         return x*y;
12     }
13     public double dividir(double x, double y) {
14         return x/y;
15     }
16 }
```

Fragmento 3.1: Clase OperacionAritmetica. Fuente: Elaboración propia

3.1.2.4. Operadores de Mutación con Aspectos

En lugar de manipular manualmente el código fuente de un programa para modificar su comportamiento con el objetivo de reproducir fallos, con POA esto puede lograrse sin alterar directamente el código fuente original. Para ello a través de distintos elementos se escribe un aspecto que intercepte el programa original causándole modificaciones. Esta abstracción en la que se especifica la escritura de un cambio de comportamiento del programa simulando un error representa un operador de mutación [43].

Supongamos que se tiene la clase OperacionAritmetica mostrada en el Fragmento 3.1, que implementa un conjunto de métodos para operaciones aritméticas básicas. Se desea aplicar el operador de mutación AOC (Argument Order Change) al método dividir para intercambiar los valores de los argumentos, de manera que, si se escribe dividir(10, 2), realmente se ejecute dividir(2, 10) simulando este error del programador. En este caso, se utiliza la cláusula *around* que sustituye el cuerpo de un método por el código que se incluya en el *advice*. El Fragmento 3.2 muestra el código del aspecto que contiene el *pointcut* que captura la llamada y el código del *advice* que realiza la modificación a este método de la clase OperacionAritmetica. En el *advice* se han añadido con fines ilustrativos 3 instrucciones para realizar el intercambio de los valores de los parámetros utilizando una variable temporal y luego, se ha incluido una llamada a la función *proceed(m, x, y)*, que produce una llamada al método original.

```
1  public privileged aspect Aspecto_OperacionAritmetica
2  {
3      double around(OperacionAritmetica operacion, double x, double y)
           :
4      call (double operacion.dividir(double, double))
5      && target(operacion)
6      && args(x, y)
7      {
8          double aux=x;
9          x=y;
10         y=aux;
11         return proceed(operacion, x, y);
12     }
13 }
```

Fragmento 3.2: Aspecto que implementa el operador AOC para interceptar el método dividir de la clase OperacionAritmetica. Fuente: Elaboración propia

Obsérvese el código del aspecto presentado en el Fragmento 3.2:

- En la línea 3 la cláusula *around* sustituye el cuerpo del método por el código incluido en el advice.
- Los argumentos pasados son: (i) el tipo del objeto sobre el que se ejecuta la operación, en este caso el objeto *operacion* de la clase *OperacionAritmetica* y (ii) los tipos y nombres de los parámetros del método que se va a capturar.
- Después del símbolo de dos puntos (:), en la línea 4 se indica que se va a atrapar la llamada con la palabra clave *call* a la operación que se indica como argumento del propio *call*; esto es, que se atrapará la llamada a *double OperacionAritmetica.dividir(double, double)*. Después de *call* en las líneas 5 y 6 identificamos el rol de los elementos que pasamos como argumentos al *around*: el *target* es el objeto sobre el que se ejecuta la operación, y los argumentos son x e y.
- A partir de la línea 7 en el advice se escribe el código que se desea ejecutar cuando se llame la operación *dividir(..)*, que en este caso será intercambiar los valores de los parámetros del método.

Al entretejer el programa original en el Fragmento 3.1 con el aspecto especificado en el Fragmento 3.2 se ejecutará un resultado diferente; en este caso, en vez de dividir los valores (x, y) en ese orden, el aspecto los intercambiará, introduciendo en su ejecución un error que no existe en la clase original.

3.2. Descripción de la Propuesta

3.2.1. Visión general de las etapas

Esta propuesta describe un enfoque que se compone de 4 etapas, mostrado en la Figura 3.2. Estas etapas engloban un modelo para dar soporte a la especificación y generación de nuevos operadores de mutación, y con el objetivo de probar la generación de dichos operadores se ha propuesto además una estrategia para la generación de mutantes y su ejecución selectiva. En los epígrafes 3.2.2, 3.2.3, 3.2.4 y 3.2.5 se describe en detalle cada una de las etapas.

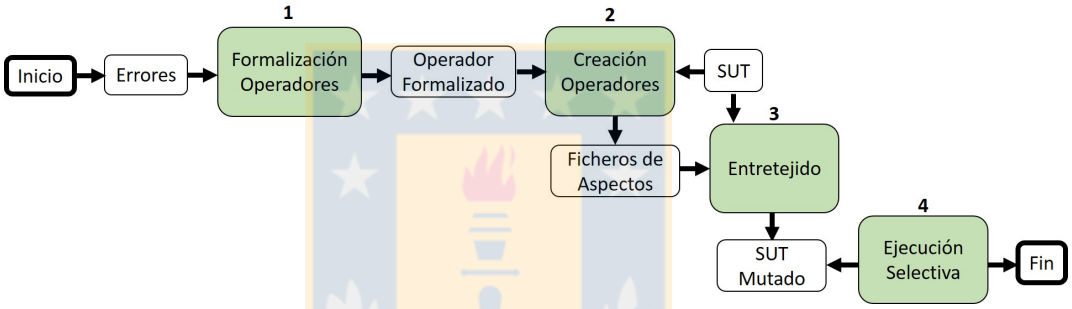

Figura 3.2: Esquema general del enfoque propuesto. Fuente: Elaboración propia

(i) **Formalización de operadores de mutación:** El primer paso para aplicar la mutación es seleccionar los operadores de mutación que se desean usar para generar los errores. Estos operadores surgen a partir de errores comunes que cometen los programadores. Entonces, la especificación de un operador partirá de la formalización de los elementos que lo componen a partir de los errores identificados.

(ii) **Creación de operadores:** Mientras que en la primera etapa solo se formalizaban los operadores de mutación a partir de los errores identificados, en esta etapa se realiza su creación física a través de la generación de los aspectos. En esta etapa, a partir de técnicas de introspección se obtiene información de los métodos a mutar y se seleccionan los operadores de mutación que ya han sido formalizados y que se desea aplicar a esos métodos. De esta selección se genera el aspecto correspondiente al operador de mutación.

(iii) **Entretejido:** En la segunda etapa se obtienen los aspectos correspondientes a los operadores de mutación previamente formalizados. En esta etapa se entreteje la versión del SUT original con los aspectos para obtener el SUT mutado que incluirá la versión del código original junto con los errores inyectados.

(iv) **Ejecución selectiva de mutantes:** Después de obtener el SUT entretejido con el código original y los errores inyectados, en esta etapa es posible ejecutar de forma independiente cada una de las mutaciones activándolas o desactivándolas mediante parámetros especificados en tiempo de ejecución.

3.2.2. Formalización de operadores de mutación

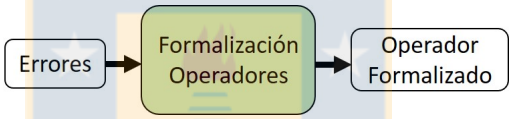

Figura 3.3: Primera Etapa. Formalización de operadores de mutación. Fuente: Elaboración propia

En la Figura 3.3 se grafica el proceso que se realiza en la primera etapa, que tiene como entrada los errores que se han identificado. Partiendo de esta identificación se definen a través de un modelo los operadores de mutación. Este proceso tiene como salida los operadores de mutación formalizados que serán incorporados al catálogo para ser utilizados con posterioridad.

Un Operador de Mutación puede representarse de forma abstracta como un conjunto de características que lo definen como el contexto en el que se ejecutará, elementos descriptivos que lo identifican, cambios que implementa en el código fuente, condiciones que debe cumplir, momento en el que debe realizarse la ejecución, entre otras características mostradas en la Figura 3.4. Definir un modelo para describir operadores de mutación a más alto nivel especificando cada una de sus características permitirá que para su generación solo se deban definir los diferentes elementos que lo componen de forma independiente, sin la necesidad de codificarlo directamente dentro de la herramienta de análisis mutación.

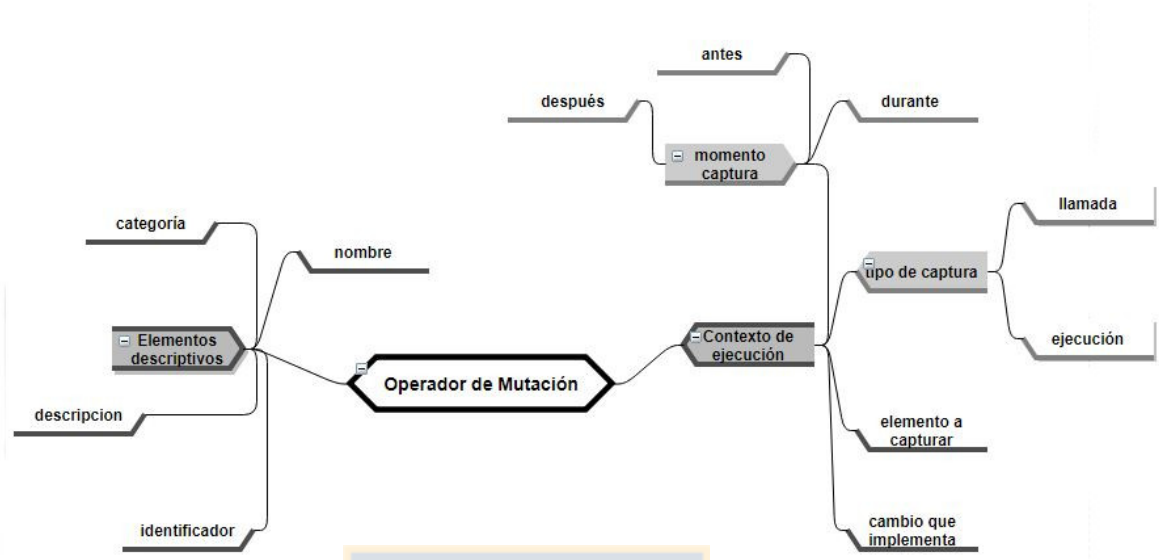

Figura 3.4: Mapa de conceptos que definen un operador de mutación. Fuente: Elaboración propia

Los conceptos identificados se han formalizado para representar un operador de mutación, en el que se definen los elementos que lo componen como atributos de una entidad que pueden ser instanciables, reutilizables y modificables. Estos elementos se han descrito como Tokens, detallados en la Tabla 3.5. Los Tokens representan la sintaxis de un aspecto AspectJ y tienen un significado en el proceso de generación de código para ser sustituidos por los fragmentos de código fuente que correspondan. A los elementos descriptivos no se les ha asociado Tokens, su uso se limita solo a la identificación del operador de mutación.

- **Elementos descriptivos**
  - Categoría: Categoría a la que pertenece el operador de mutación.
  - Acrónimo: Identificador del operador de mutación.
  - Nombre: Nombre del operador de mutación.
  - Descripción: Descripción textual del error que genera.
- **Tokens de elementos a sustituir**

Elemento	**Token**
Categoría	-
Acrónimo	-
Nombre	-
Descripción	-
Nombre del aspecto	ASPECT_NAME
Importaciones	IMPORT
Tipo de retorno	RETURN_TYPE
Momento de la captura	EXECUTION_MOMENT
Nombre de la clase, Nombre del objeto	CLASS_OBJECT
Tipo de parámetros, Nombre de parámetros	TYPE_PARAMETER_NAME
Método a capturar	CAPTURE_METHOD
Tipo de captura	TYPE_OF_CAPTURE
Firma del método	METHOD_SIGNATURE
Nombre del objeto	OBJECT_NAME
Nombre de argumentos	PARAMETER_NAME
Sentencia de cambio	CHANGE
Sentencia original	ORIGINAL

Tabla 3.5: Definición de elementos de un operador de mutación. Fuente: Elaboración Propia

- IMPORT: Importaciones necesarias para que el aspecto compile.
- RETURN_TYPE: Tipo de dato que retorna el método.
- EXECUTION_MOMENT: Palabras clave *before*, *after*, *around*.
- CLASS_OBJECT: Nombre de la clase y nombre del objeto para el que se esté creando el aspecto.
- TYPE_PARAMETER_NAME: Tipos de datos y nombres de los argumentos del método a capturar.
- CAPTURE_METHOD: Firma del método principal que se selecciona mediante introspección.
- TYPE_OF_CAPTURE: Palabras clave *call*, *execution*.
- METHOD_SIGNATURE: Firma del método a capturar.

- OBJECT_NAME: Nombre del objeto a capturar.
- PARAMETER_NAME: Nombre de los parámetros del método a capturar.
- CHANGE: Implementación del cambio sobre el código original del método sobre el que interviene el *pointcut*
- ORIGINAL: Codificación original del método sobre el que interviene el *pointcut*.

3.2.3. Creación de Operadores de Mutación

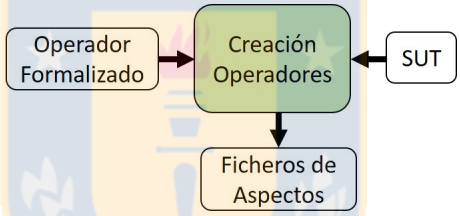

Figura 3.5: Segunda Etapa. Creación de operadores de mutación. Fuente: Elaboración propia

Como se describe en la primera etapa, Sección 3.2.2, una vez que han sido formalizados los operadores de mutación a partir de los errores identificados, el segundo paso es crear físicamente dichos operadores a partir de la generación de los aspectos. Como se muestra en la Figura 3.5 se toma como entrada (i) información del SUT, en este caso, sobre la clase y método al cual se le va a aplicar el operador de mutación y (ii) el operador de mutación.

En base a esta información de entrada, se realiza un proceso de transformación que genera los aspectos utilizados para mutar el SUT. En este tipo de transformaciones, el código que se genera procede de una plantilla, mostrada en el Fragmento 3.3, que contiene tokens cuyo significado se detalla en la Tabla 3.5. La generación de código sustituye esos tokens por los fragmentos de código que correspondan tanto a la especificación del operador de mutación como al código a mutar del SUT.

```
1
2  import java.util.Properties;
3  import java.io.FileReader;
4  IMPORT;
5
6  public privileged aspect ASPECT_NAME
7  {
8      RETURN_TYPE EXECUTION_MOMENT(CLASS_OBJECT TYPE_PARAMETER_NAMES):
9      withincode(CAPTURE_METHOD)
10     && TYPE_OF_CAPTURE (METHOD_SIGNATURE)
11     && target(OBJECT_NAME)
12     && args(PARAMETER_NAMES)
13     {
14         try
15         {
16             Properties p=new Properties();
17             p.load(new FileReader("conf.properties"));
18             boolean active = Boolean.parseBoolean
19             (p.getProperty("ASPECT_NAME"));
20
21             if(active)
22             {
23                 CHANGE;
24             }
25             else
26             {
27                 ORIGINAL;
28             }
29         }
30         catch(Exception e){
31             ORIGINAL;
32         }
33     }
34 }
```

Fragmento 3.3: Plantilla para la creación de un aspecto. Fuente: Elaboración propia

La plantilla para la generación de aspectos, Fragmento 3.3, está dividida en varias partes que permiten crear el código completo del aspecto. Cada parte está formada por tokens que se sustituirán dependiendo de la información contenida dentro del método a mutar y la información definida en el operador de mutación. Esta plantilla indica también las importaciones necesarias para la compilación del aspecto, el nombre del aspecto, los pointcuts y los errores a especificar. A continuación, se describe con mayor detalle esta plantilla.

(i) **Pointcuts**: De la línea 8 a la 12 se representa la sintaxis del pointcut encargada de atrapar las llamadas a métodos. Se compone de los siguientes tokens: RETURN_TYPE, EXECUTION_MOMENT, CLASS_OBJECT, TYPE_PARAMETER_NAMES, CAPTURE_METHOD, TYPE_OF_CAPTURE, METHOD_SIGNATURE, OBJECT_NAME, PARAMETER_NAMES.

(ii) **Errores**: A partir de la línea 14, dentro del pointcut se incluyen los errores a inyectar en un método. Cada error se genera a partir de un operador de mutación. Los tokens que forman esta parte de la plantilla son CHANGE, ORIGINAL.

Esta formalización tiene como objetivo especificar el operador de mutación de manera tal que su generación no dependa de la modificación del código fuente de la herramienta de análisis de mutación. Para describir con mayor detalle lo que se pretende, mostramos un ejemplo práctico donde se ha caracterizado como se crean los operadores de mutación en la herramienta Mujava (hasta el momento la única que dispone de su código fuente publicado). En la Figura 3.6 se muestra el diseño de la arquitectura de operadores de mutación que esta herramienta presenta y en el Fragmento 3.4 se describe parte de la implementación del Operador AOR (Reemplazo de Operador Aritmético). Este operador, como se muestra, para su implementación fue necesaria la adición de una nueva clase dentro de la estructura definida, así como la modificación otras partes de la arquitectura para su funcionamiento. Esto hace que crear un nuevo operador sea directamente dependiente de modificar código fuente dentro de la herramienta de análisis de mutación. La definición en esta propuesta no supone estas modificaciones, solo se necesita completar la especificación del operador con los elementos que lo componen y este quedará listo para ser generado. En la Figura 3.7 se muestra un prototipo donde es posible especificar el operador sin tener conocimientos de detalles de implementación de la herramienta de análisis de mutación.

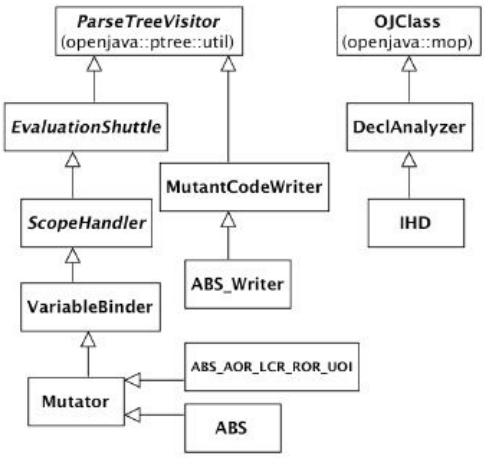

Figura 3.6: Arquitectura de operadores de mutación, herramienta Mujava. Fuente: MuJava [6]

```
1  public class AORB extends Arithmetic_OP
2  {
3    public void visit( BinaryExpression p ) throws ParseTreeException
4      {
5           Expression left = p.getLeft();
6           left.accept(this);
7           Expression right = p.getRight();
8           right.accept(this);
9           if (isArithmeticType(p))
10          {
11           int op_type = p.getOperator();
12           switch (op_type)
13           {
14            case BinaryExpression.TIMES :
15                    aorMutantGen(p, BinaryExpression.TIMES);
16                    break;
17            case BinaryExpression.DIVIDE :
18                    aorMutantGen(p, BinaryExpression.DIVIDE);
19                    break;
20            case BinaryExpression.MOD :
21                    aorMutantGen(p, BinaryExpression.MOD);
22                    break;
23            case BinaryExpression.PLUS :
24                    aorMutantGen(p, BinaryExpression.PLUS);
25                    break;
26            case BinaryExpression.MINUS :
27                    aorMutantGen(p, BinaryExpression.MINUS);
28                    break;
29           }
30          }
31     }
32 }
```

Fragmento 3.4: Implementación del Operador AOR por la herramienta Mujava. Fuente: Mujava [6]

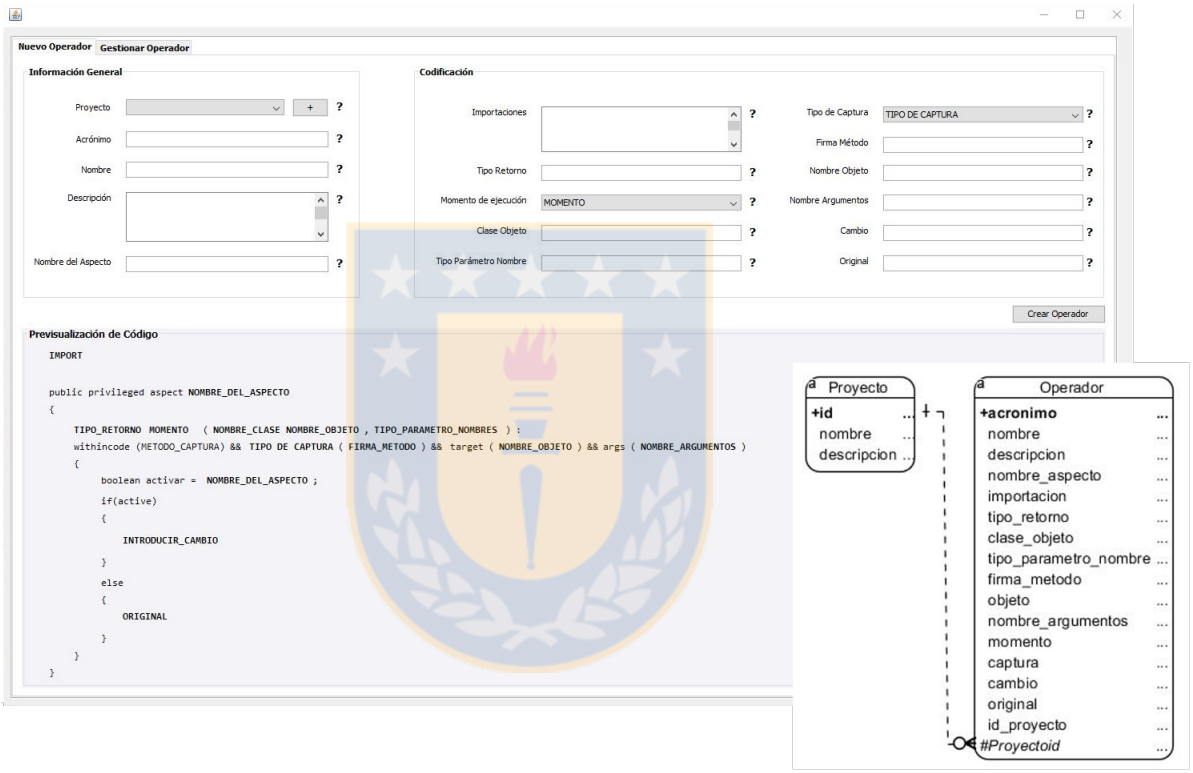

Figura 3.7: Prototipo para la definición de operadores de mutación en la solución propuesta. Fuente: Elaboración propia

La configuración mostrada en el prototipo 3.7, permite especificar los elementos que definen el operador de mutación (estos elementos han sido detallados en la Tabla 3.5). El objetivo de estos elementos es generar una plantilla de aspectos (detallado en la Sección 3.2.3), la cual representa el operador de mutación, en este prototipo hemos considerado la visualización de los elementos que se van sustituyendo dentro de la plantilla.

3.2.4. Entretejido

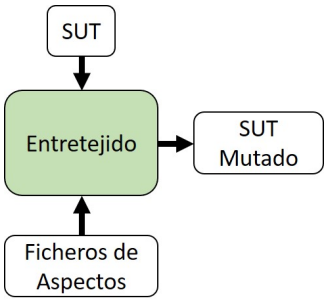

Figura 3.8: Tercera Etapa. Entretejido del SUT con los de aspectos. Fuente: Elaboración propia

La segunda etapa, Sección 3.2.3, produce como salida un conjunto de aspectos creados a partir de la información de los operadores de mutación y del SUT a mutar. En esta etapa, se genera la versión mutada del SUT. Como se muestra en la Figura 3.8 este proceso toma como entrada (i) los aspectos que han sido generados y (ii) el SUT, produciendo como salida una única versión del SUT con los errores inyectados.

Este proceso de entretejido es el mecanismo que permite combinar la definición de las clases (joinpoints), con el código que implementa el cambio que se desea introducir (advices), de manera que, cuando el sistema esté en funcionamiento, el advice y el joinpoint se ejecuten conjuntamente. Este proceso se efectúa en tiempo de compilación (variante estática) donde se compila el SUT con los aspectos dando lugar al SUT empaquetado con los aspectos entrelazados. Para realizar este proceso de entretejido se hace uso de librerías AspectJ [26] [28], este es un framework para la plataforma Java muy utilizado para la POA.

Este enfoque no produce un fichero físico por cada mutación del código original, ya que el entretejido del SUT con los aspectos genera una única versión del SUT entrelazada con todos los errores inyectados. Dicho de otro modo, en los aspectos se implementa el código de los errores que se desean entretejer en el código fuente del SUT. Cuando se entreteje el SUT con los aspectos se genera una nueva versión del SUT donde para cada método en el que hay que introducir un error, el SUT entrelazado tendrá la versión del código original más los errores. Cuando un mutante incluye varios errores, es necesario un mecanismo que permita controlar qué mutación

se activará al ejecutar el programa.

3.2.5. Ejecución selectiva de mutantes

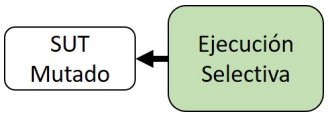

Figura 3.9: Cuarta Etapa. Ejecución selectiva de mutantes. Fuente: Elaboración propia

En la tercera etapa 3.2.4 se obtiene una versión mutada del SUT con todos los aspectos entrelazados. En esta etapa es posible realizar la ejecución de los mutantes. Para ello, a través de ciertos parámetros de configuración es posible activar o desactivar cada uno de los errores introducidos en el SUT, dando lugar a la ejecución del mutante de interés que se desee analizar, es decir, su ejecución selectiva. A continuación, se detalla su funcionamiento:

- **Plantilla Aspecto:** A partir de la línea 11 del Fragmento 3.5, que muestra el código del aspecto, en la implementación del advice se hace uso de sentencias condicionales. Estas sentencias permiten que al tejer el código original del SUT con los aspectos y generar el código del SUT entretejido, el código del pointcut de un aspecto ejecute el código original o código mutado, dependiendo de si se activa o no el error.

- **Configuración de la Ejecución Selectiva:** En el Fragmento 3.6 se muestra un ejemplo de configuración de propiedades, donde se especifican los aspectos que se entretejen con el SUT. En la línea 3 el token ASPECT_NAME se sustituye por el nombre del aspecto y este tomará el valor booleano especificado en la propiedad *value* que activará o desactivará la ejecución de esa mutación dentro del SUT.

```
1
2 import java.util.Properties;
3 import java.io.FileReader;
4 IMPORT;
```

```
5
6  public privileged aspect ASPECT_NAME
7  {
8          // POINTCUT CODE
9
10     {
11         // ADVICE CODE
12         try
13         {
14             Properties p=new Properties();
15             p.load(new FileReader("conf.properties"));
16             boolean active = Boolean.parseBoolean
17             (p.getProperty("ASPECT_NAME"));
18
19             if(active)
20             {
21                 CHANGE;
22             }
23             else
24             {
25                 ORIGINAL;
26             }
27         }
28         catch(Exception e){
29             ORIGINAL;
30         }
31     }
32 }
```

Fragmento 3.5: Activación de mutaciones en plantilla de aspecto. Fuente: Elaboración propia

```
1  #CONFIGURACION EJECUCION SELECTIVA DE MUTACIONES
2
3  ASPECT_NAME = true|false
```

Fragmento 3.6: Plantilla de configuración de propiedades para la ejecución selectiva de mutantes. Fuente: Elaboración propia

3.3. Conclusión

En este Capítulo se ha desarrollado un Análisis Conceptual, donde a partir de un conjunto de conceptos y sus relaciones hemos propuesto una solución para la creación de un marco conceptual que de soporte a la especificación y generación de nuevos operadores de mutación. El marco propuesto cubre los siguientes elementos:

1. Se han analizado un conjunto de conceptos referentes al contexto de la investigación con el objetivo de profundizar e investigar las posibles relaciones que pueden ocurrir entre estos conceptos y así extraer nuevas soluciones.

2. Se ha desarrollado un marco conceptual donde se expone una nueva perspectiva sobre el concepto operador de mutación, los elementos que lo componen y sus relaciones. Este enfoque tomando elementos del concepto POA para su especificación.

3. A partir del marco conceptual definido se ha creado una solución que engloba un proceso de cuatro etapas. Estas etapas están compuestas por actividades de entrada/salida y abarcan: un modelo para formalizar y crear nuevos operadores de mutación, una forma de entretejer los operadores con el SUT y un mecanismo para ejecutarlos selectivamente.

4. El proceso de la solución propuesta se ha definido a través de la creación de marcas y plantillas de texto que contienen los elementos representativos del operador. Estas plantillas guían gran parte de las etapas del proceso.

Capítulo 4

Prueba de Concepto

En este Capítulo se realiza una Prueba de Conceptos, con el objetivo de demostrar la viabilidad de la solución propuesta. En esta prueba se aplican cada una de las etapas de la solución propuesta a un caso de ejemplo. Para este caso se usó como SUT una aplicación de tecnología móvil, donde primero, se identifican 12 errores, a partir de estos se especificaron y generaron los operadores de mutación con el lenguaje de aspectos AspectJ, se generaron los mutantes y se implementó la mutación selectiva sobre los mutantes generados.

4.1. Aplicación SUT

Un aspecto fundamental en el desarrollo de aplicaciones móviles es la capacidad de la aplicación para geolocalizar la posición del usuario a través de los recursos del teléfono. Esta tecnología permite que las aplicaciones conozcan la posición del usuario en todo momento, adaptando su comportamiento en consecuencia. Buscar sitios de interés o usuarios cercanos son ejemplos muy comunes del uso de la tecnología de geolocalización.

Actualmente existen dos aproximaciones diferentes para la geolocalización: el GPS y la localización en base a antenas de telecomunicaciones o redes Wifi. Si nos centramos en la forma de acceder a estos recursos desde nuestras aplicaciones, Android proporciona dos formas de hacerlo: utilizar las clases del Android Framework Location API, o hacerlo a través de Google Play Services Location API, siendo esta última la recomendada por Android. La ventaja de la segunda opción es que proporciona una capa de software que simplifica el uso de las herramientas de geolocalización y que, además, permite hacerlo de forma más eficiente tanto en precisión de las localizaciones obtenidas como en el consumo de batería en el dispositivo.

Para este ejemplo se usó una aplicación de tecnología móvil que permite registrar

lugares de interés para el usuario, hacer un seguimiento de sus puntos de entrada y salida, y a partir de esta información, registrar las rutas por donde ha transitado. La aplicación usa el servicio de localización de Google Play Services Location API.

Esta aplicación tiene un diseño por capas. La capa de presentación está compuesta por el paquete Activities. Este paquete contiene todas las clases y componentes gráficos que forman parte de las interfaces gráficas de usuario. Estas actividades están formadas por la parte lógica que refiere a clases Java, encargadas de manipular todo el procesamiento de la actividad y la parte gráfica formada por ficheros XML que contienen todos los elementos gráficos visuales que se despliegan al usuario. En la Figura 4.2 se muestra el diagrama de clases de la Capa de presentación. La capa Lógica de Negocio está formada por los paquetes Model, Network y Service. Estos paquetes contienen las clases que implementan funcionalidades relacionadas con la lógica de la aplicación, los servicios y la gestión del modelo de dominio. En la Figura 4.1 se muestra el diagrama de clases de la capa de lógica de negocio.

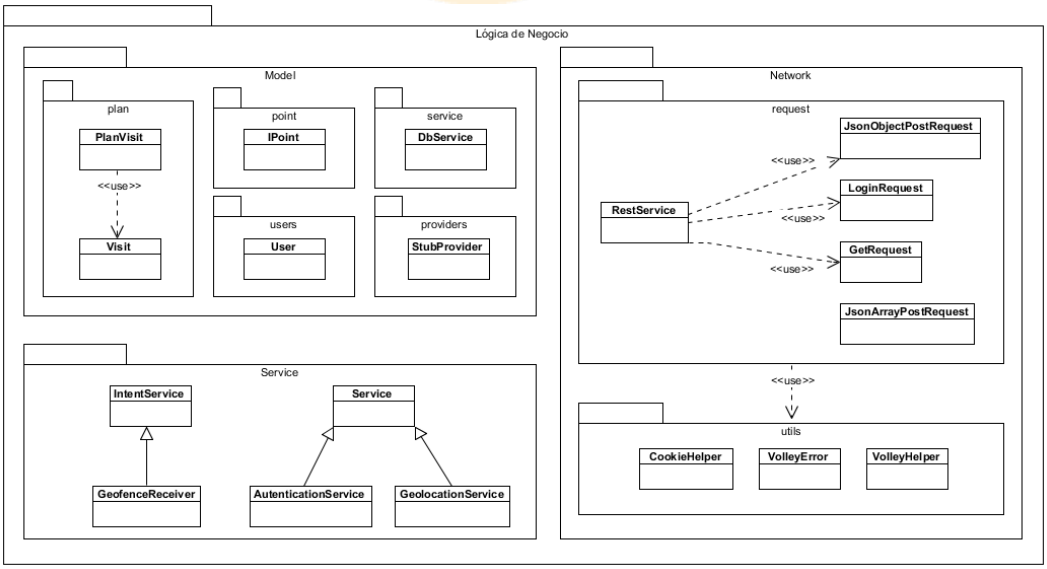

Figura 4.1: Componentes, Capa Lógica de Negocios. Fuente: Elaboración propia

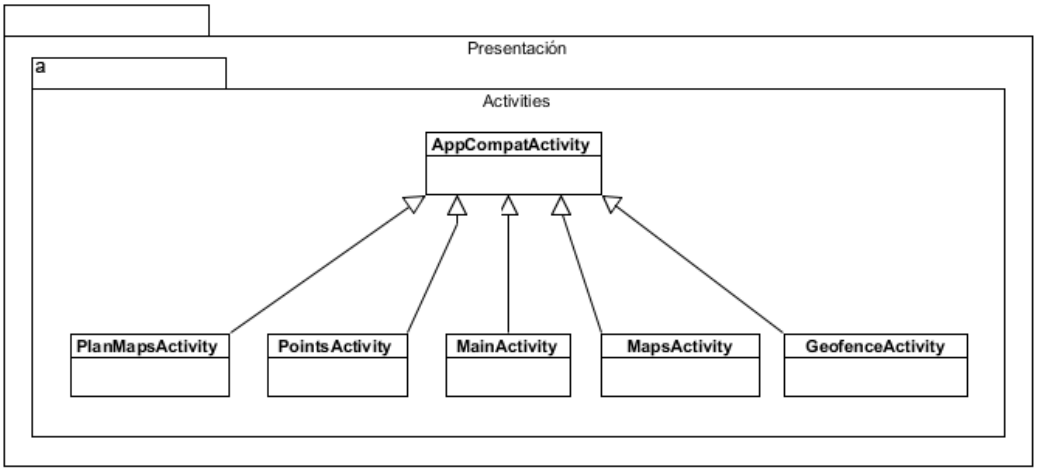

Figura 4.2: Componentes, capa de presentación. Fuente: Elaboración propia

4.2. Catálogo de Errores

En este epígrafe se describe un catálogo de 12 errores, relacionados en la Tabla 4.1. Se toma para este caso el Error 09 para describir todas las etapas de la solución propuesta (para detalles sobre el resto consultar los Anexos A, B, C, D,E, F, G, H, I, J, K). A partir de estos errores se definen los operadores de mutación que se aplicarán a los métodos de las clases que forman parte de la lógica de presentación y la lógica de negocios de la aplicación que se ha definido como SUT.

Identificador	Descripción	Categoría
ERR-01	No se especifica que se utilizará el API de geolocalización al crear el cliente del API de Google Services	Sensores
ERR-02	No se inicia el cliente del API de Google Services en la actividad que solicita la recepción de actualizaciones de posición	Sensores
ERR-03	No se comprueba que la última posición conocida por el dispositivo es null	Sensores
ERR-04	Se establecen valores de interval y fastestInterval contradictorios	Sensores
ERR-05	Se establece el valor 0 para fastestInterval.	Sensores
ERR-06	No se comprueba antes de solicitar el inicio/fin de la recepción de actualizaciones de posición si el cliente del API de Google Services sigue activo	Sensores
ERR-07	Al transformar una posición en una dirección, se confunden los campos de latitud y longitud	Sensores
ERR-08	Al transformar una posición en una dirección, se confunden los campos de latitud y longitud	Sensores
ERR-09	Al crear un Geofence se confunden los parámetros de latitud y longitud	Interacción interna
ERR-10	Al crear un Geofence se establece un valor 0 para el radio que lo delimita	Interacción interna
ERR-11	Al crear un Geofence se establece una duración negativa para el mismo	Interacción interna
ERR-12	Al tratar un evento de entrada/salida de un Geofence, se confunde el tipo de evento	Interacción interna

Tabla 4.1: Errores identificados clasificados en categorías.Fuente: Elaboración Propia

4.2.1. ERR-09: Errores al crear una zona de Geofence

- **Contexto:** El sistema operativo Android permite a los desarrolladores de aplicaciones obtener notificaciones de cuando un usuario entra o sale de una zona de interés. Estas zonas de interés se denominan *Geofences*. Un *Geofence* está determinado por una localización geográfica expresada en términos de latitud y longitud, y un radio alrededor de dicha localización. Así, cualquier aplicación con los permisos necesarios (es decir, poder acceder a funcionalidades de

geolocalización del dispositivo) puede solicitar al sistema operativo que le notifique cuando la persona entra o sale de una de esas zonas de interés. Estas funcionalidades se utilizan principalmente para control de presencia (por ejemplo, controlar cuándo un trabajador entra en las instalaciones de la empresa, cuándo un cliente entra de nuevo en nuestra tienda, o cuando un niño sale de la zona de alcance fijada por sus padres).

Pese a que crear Geofences y recibir las notificaciones es relativamente sencillo, se trata también de un código propenso a errores. El primer detalle que puede provocar un error es que el centro del Geofence no se especifica con un objeto de la clase *Location*, sino que se especifica a partir de los valores de latitud y longitud, lo que puede dar lugar a una confusión. Por una parte, crear mal el *Geofence* a partir de su localización central provocará que la aplicación móvil reciba y traslade al usuario notificaciones de entrada/salida de zona erróneas.

- **Fallo que inserta:** Se intercambia la latitud por la longitud.
- **Error que provoca:** Al crear un *Geofence* con una ubicación errónea éste realizará notificaciones de localización incorrectas al usuario.

4.3. Formalización de operadores de mutación

En este epígrafe se aplica la primera etapa de la solución propuesta 3.2.2 donde se formalizan los operadores de mutación a partir del catálogo de errores 4.2 que han sido identificados.

4.3.1. Operador ERR-09

- **Forma de reproducirlo**

```
1 CircleOptions MapFragment.createCircleOptions(double, double,
     float)
2 -->
3 createCircleOptions(geofence, p1, p0, p2)
```

- **Métodos que afecta**

```
1 void MapFragment.displayGeofences()
```

- **Formalización del Operador**

Token	Elemento
ASPECT_NAME	Aspecto _OpLoc09_MapFragmentDisplayGeofences
RETURN_TYPE	CircleOptions
EXECUTION_MOMENT	around
CLASS_OBJECT	MapFragment mapfragment
TYPE_PARAMETER_NAME	double p0, double p1, float p2
CAPTURE_METHOD	void MapFragment. displayGeofences()
TYPE_OF_CAPTURE	call
METHOD_SIGNATURE	CircleOptions MapFragment.createCircleOptions(double, double, float)
OBJECT_NAME	mapfragment
PARAMETER_NAME	p0, p1, p2
CHANGE	return proceed (mapfragment, p1, p0, p2)

Tabla 4.2: Formalización Operador ERR-09. Fuente: Elaboración Propia

4.4. Creación de Operadores de Mutación

A partir de los operadores formalizados en la Sección 4.3 se aplica la segunda etapa de la solución propuesta, Epígrafe 3.2.3, donde se crean físicamente los aspectos que corresponden a cada uno de los operadores de mutación.

4.4.1. Aspecto Operador ERR-09

```
1  public privileged aspect Aspecto_OpLoc09_MapFragmentDisplayGeofences
2  {
3      CircleOptions around(MapFragment mapfragment, double p0, double
           p1, float p2):
4      withincode(void MapFragment.displayGeofences())
5
6      && call (CircleOptions MapFragment.createCircleOptions(double,
           double, float))
7      && target(mapfragment)
8      && args(p0, p1, p2)
9      {
10         return proceed(mapfragment, p1, p0, p2);
```

```
11      }
12  }
```

Fragmento 4.1: Aspecto Operador ERR-09. Fuente: Elaboración propia

4.5. Entretejido

En este epígrafe se detalla cómo se realiza el proceso de entretejido del SUT con los operadores de mutación. Para este proceso se aplica la tercera etapa de la solución propuesta, referida en la Sección 3.2.4, que toma como entrada: (i) los aspectos que se han abordado en la Sección 4.4 correspondientes a cada uno de los operadores de mutación y (ii) el SUT, detallado en la Sección 4.1 en su versión original. El resultado de este proceso es la generación del SUT entretejido con los aspectos, en este caso mediante entretejido estático a través de librerías AspectJ donde se inyectan los errores en tiempo de compilación. En el entretejido estático el compilador produce, por cada clase entretejida, un fichero objeto que incluye el código compilado de la clase más el código compilado del advice.

En los fragmentos 4.2 y 4.3 se muestra un ejemplo de entretejido entre el Aspecto que implementa el Operador ERR-09 4.2.1 y la clase MapFragment a la que se le inyectan los errores.

```
1  public privileged aspect Aspecto_OpLoc09_MapFragmentDisplayGeofences
2  {
3      CircleOptions around(MapFragment mapfragment, double p0, double
           p1, float p2):
4      withincode(void MapFragment.displayGeofences())
5
6      && call (CircleOptions MapFragment.createCircleOptions(double,
           double, float))
7      && target(mapfragment)
8      && args(p0, p1, p2)
9      {
10         return proceed(mapfragment, p1, p0, p2);
11     }
12 }
```

Fragmento 4.2: Aspecto Operador ERR-09. Fuente: Elaboración propia

```
1  public class MapFragment
2  {
3    protected void displayGeofences()
```

```
4   {
5     HashMap<String, SimpleGeofence> geofences =
          SimpleGeofenceFactory.getInstance().getSimpleGeofences();
6     for (Map.Entry<String, SimpleGeofence> item : geofences.entrySet
          ())
7     {
8       SimpleGeofence sg = item.getValue();
9       map.addCircle(createCircleOptions(sg.getLatitude(),
10                                        sg.getLongitude(),
11                                        sg.getRadius()));
12    }
13  }
14 }
```

Fragmento 4.3: Clase MapFragment modificada por el Aspecto Operador ERR-09. Fuente: Elaboración propia

Como resultado del proceso de entretejido el compilador introduce a nivel de bytecode las instrucciones necesarias para ejecutar el advice adecuado al alcanzarse el punto de enlace especificado. En este caso se intercepta el método *createCircleOptions(..)*. En el Fragmento 4.4 se muestra el método *displayGeofences()* descompilado; a partir de la línea 15 se muestra la intercepción del aspecto *Aspecto_OpLoc09_MapFragmentDisplayGeofences* sobre el método *createCircleOptions*, donde en vez de ejecutar el método original se ejecuta la mutación que contiene el error.

```
1  protected void displayGeofences()
2  {
3      java.util.HashMap geofences =
4      app.utils.SimpleGeofenceFactory.getInstance().getSimpleGeofences
           ();
5
6      float f;
7      double d1;
8      double d2;
9      MapFragment localMapFragment;
10
11     for (java.util.Iterator localIterator =
12     geofences.entrySet().iterator();
13     localIterator.hasNext();
14
15     map.addCircle(createCircleOptions_aroundBody1advice (
16                                                 this,
17                                                 localMapFragment,
18                                                 d2,
19                                                 d1,
20                                                 f,
```

```
21      app.aspect.Aspecto_OpLoc09_MapFragmentDisplayGeofences.aspectOf
           (),
22      localMapFragment,
23      d2,
24      d1,
25      f,
26      null)))
27
28      {
29       java.util.Map.Entry item=(java.util.Map.Entry)localIterator.
            next();
30       SimpleGeofence sg = (SimpleGeofence)item.getValue();
31
32       f = sg.getRadius();
33       d1 = sg.getLongitude();
34       d2 = sg.getLatitude();
35       localMapFragment = this;
36      }
37  }
38
39   private static final CircleOptions createCircleOptions_aroundBody0(
40                                                MapFragment
                                                    paramMapFragment1,
41                                                MapFragment
                                                    paramMapFragment2,
42                                                double paramDouble1,
43                                                double paramDouble2,
44                                                float paramFloat)
45   {
46      return paramMapFragment2.createCircleOptions(paramDouble1,
47                                                   paramDouble2,
48                                                   paramFloat);
49
50   }
```

Fragmento 4.4: Método displayGeofences(..) entretejido con el Aspecto Operador ERR-09. Fuente: Elaboración propia

4.6. Ejecución selectiva de mutantes

En este epígrafe se detalla cómo se implementa la cuarta etapa de la solución propuesta, referida en la Sección 3.2.5, donde se implementa la ejecución selectiva de mutantes. La Ejecución selectiva es un mecanismo que se ha creado para permitir seleccionar qué mutaciones de las entretejidas en SUT deben ejecutarse, a través de parámetros que se obtienen en tiempo de ejecución, ver Fragmento 3.6, permitiendo

su activación o desactivación de forma independiente.

En los fragmentos 4.5 y 4.6 se muestra un ejemplo que especifica la activación del Aspecto Operador ERR-09 4.2.1 entrelazado con la clase *MapFragment*, Fragmento 4.3, y generado como *Aspecto_OpLoc09_MapFragmentDisplayGeofences*. En este caso el aspecto se establece en *true* afectando al SUT con la ejecución del error inyectado. Es posible utilizar este mecanismo sin la necesidad de recompilar la aplicación para cambiar el modo de ejecución, solamente modificar los parámetros con los valores de *true* o *false* para activar o no la ejecución de los errores que se han inyectado.

```
1  public privileged aspect Aspecto_OpLoc09_MapFragmentDisplayGeofences
2  {
3
4      CircleOptions around(MapFragment mapfragment, double p0, double
           p1, float p2):
5      withincode(void MapFragment.displayGeofences())
6      && call (CircleOptions MapFragment.createCircleOptions(double,
           double, float))
7      && target(mapfragment)
8      && args(p0, p1, p2)
9      {
10         try
11         {
12              Properties p=new Properties();
13              File pp = new File(Environment.
                    getExternalStorageDirectory().getPath() + "/conf.
                    properties");
14              p.load(new FileReader(pp));
15              boolean active = Boolean.parseBoolean(p.getProperty("
                    Aspecto_OpLoc09_MapFragmentDisplayGeofences"));
16
17              if(active)
18              {
19                 return proceed(mapfragment, p1, p0, p2);
20              }
21              else
22              {
23                 return proceed(mapfragment, p0, p1, p2);
24              }
25          }
26          catch(Exception e)
27          {
28              return proceed(mapfragment, p0, p1, p2);
29          }
30     }
31 }
```

Fragmento 4.5: Aspecto ERR-09 con mecanismo de ejecución selectiva. Fuente: Elaboración propia

```
1 #CONFIGURACION EJECUCION SELECTIVA DE MUTACIONES
2
3 Aspecto_OpLoc09_MapFragmentDisplayGeofences = true
```

Fragmento 4.6: Ejecución selectiva del Operador ERR-09. Fuente: Elaboración propia

4.7. Conclusión

En este Capítulo se desarrolló una Prueba de Concepto con el objetivo de demostrar la viabilidad de la solución que se propuso en el capítulo 3. Esta prueba se realizó sobre una aplicación de tecnología móvil donde para ejecutar cada una de las etapas del proceso se especificaron y generaron 12 operadores de mutación para un ambiente Java a través del lenguaje AspectJ. Esta prueba demuestra que es posible generar operadores de mutación a partir de su especificación conceptual haciendo uso de la tecnología existente. Con la realización de esta prueba de concepto se ha cumplido el objetivo general de nuestra investigación.

Capítulo 5

Conclusiones y Trabajo Futuro

5.1. Conclusiones

La Técnica de Mutación está bien establecida en la investigación de ingeniería de software, sin embargo, en la práctica aún existen líneas en las que pueden aportarse nuevos conocimientos. Una Revisión Sistemática de la Literatura constató que una limitante común en las propuestas actuales es la inexistencia de un marco para representar operadores de mutación, si bien, existen investigaciones y herramientas de análisis de mutación que proponen e implementan operadores, se carece de una forma genérica que permita especificar y generar nuevos. Este trabajo ha puesto su atención en esta limitante. Para dar respuesta a este problema se han analizado un conjunto de conceptos referentes al contexto de la mutación, con el objetivo de entender y construir nuevas relaciones que proporcionen una solución a este inconveniente, para ello se utilizó el método investigativo Análisis Conceptual, específicamente se realizó un Análisis Constructivo, donde se combinaron un conjunto de conceptos existentes con el fin de definir un marco conceptual que permita especificar y generar nuevos operadores de mutación. El marco conceptual definido representa el concepto de operador de mutación como un conjunto de elementos que lo componen y las relaciones entre ellos, estos elementos se especifican a través de Programación de Orientada a Aspectos, y a partir de su especificación es posible generar nuevos. A partir de este marco conceptual se ha desarrollado una solución que se compone de cuatro etapas, donde se detalla un modelo para especificar y generar nuevos operadores, así como una forma de entretejerlos con el SUT y ejecutarlos selectivamente. Con el objetivo de demostrar la viabilidad de la solución conceptual definida, se desarrolló una Prueba de Concepto donde se especificaron y generaron 12 operadores de mutación sobre una aplicación Java a través del lenguaje AspectJ y se ejecutaron cada una de las etapas del proceso. En esta investigación se han cumplido los objetivos específicos y

en consecuencia el objetivo general, tomando esto en cuenta puede argumentarse que esta investigación ha aportado un enfoque diferente al actual en el campo de pruebas de mutación.

5.2. Trabajo Futuro

El enfoque que hemos propuesto para especificar y generar operadores de mutación es dependiente de un lenguaje de aspectos relacionado al lenguaje del SUT. Como trabajo futuro se plantea realizar una especificación genérica que pueda ser independiente de la tecnología.

Bibliografía

[1] Judy. `http://www.mutationtesting.org/judy2/download.html`, visited May 2018.

[2] Jumble. `http://jumble.sourceforge.net/`, visited May 2018.

[3] Jumble, mutation operators. `http://jumble.sourceforge.net/mutations.html`, visited May 2018.

[4] Major. `http://mutation-testing.org/downloads/`, visited May 2018.

[5] Major, mutation operators. `http://mutation-testing.org/doc/`, visited May 2018.

[6] Mujava. `https://github.com/jeffoffutt/muJava`, visited May 2018.

[7] Agrawal, H., DeMillo, R., Hathaway, R., Hsu, W., Hsu, W., Krauser, E., Martin, R. J., Mathur, A., and Spafford, E. Design of mutant operators for the C programming language. Tech. rep., Technical Report SERC-TR-41-P, Software Engineering Research Center, Department of Computer Science, Purdue University, Indiana, 1989.

[8] Ammann, P., and Offutt, J. *Introduction to software testing*. Cambridge University Press, 2016.

[9] Bogacki, B., and Walter, B. Aspect-oriented Response Injection: an Alternative to Classical Mutation Testing. In *Software Engineering Techniques: Design for Quality*. Springer, Boston, MA, 2006, pp. 273–282. DOI: 10.1007/978-0-387-39388-9_26.

[10] Bogacki, B., and Walter, B. Evaluation of Test Code Quality with Aspect-Oriented Mutations. In *Extreme Programming and Agile Processes in Software Engineering* (June 2006), Springer, Berlin, Heidelberg, pp. 202–204.

[11] Budd, T. A. *Mutation Analysis of Program Test Data*. PhD thesis, Yale University, New Haven, CT, USA, 1980.

[12] Delamaro, M. E., Maidonado, J. C., and Mathur, A. P. Interface mutation: An approach for integration testing. vol. 27, pp. 228–247.

[13] Delgado-Pérez, P., Medina-Bulo, I., Domínguez-Jiménez, J. J., García-Domínguez, A., and Palomo-Lozano, F. Class mutation operators for C++ object-oriented systems. vol. 70, pp. 137–148.

[14] DeMillo, R. A., Lipton, R. J., and Sayward, F. G. Hints on Test Data Selection: Help for the Practicing Programmer. vol. 11, pp. 34–41.

[15] Deng, L., Offutt, J., Ammann, P., and Mirzaei, N. Mutation operators for testing Android apps.

[16] Deng, L., Offutt, J., Ammann, P., and Mirzaei, N. Mutation operators for testing Android apps. vol. 81, pp. 154–168.

[17] Derezińska, A. Advanced mutation operators applicable in C# programs. In *Software Engineering Techniques: Design for Quality*. Springer, 2006, pp. 283–288.

[18] Derezińska, A., and Ha\las, K. Analysis of mutation operators for the python language. In *Proceedings of the Ninth International Conference on Dependability and Complex Systems DepCoS-RELCOMEX. June 30–July 4, 2014, Brunów, Poland* (2014), Springer, pp. 155–164.

[19] Ghosh, S., and Mathur, A. P. Interface mutation. vol. 11, pp. 227–247.

[20] Glass, R. L., Vessey, I., and Ramesh, V. Research in software engineering: an analysis of the literature. vol. 44, pp. 491–506.

[21] Irvine, S. A., Pavlinic, T., Trigg, L., Cleary, J. G., Inglis, S., and Utting, M. Jumble java byte code to measure the effectiveness of unit tests. In *Testing: Academic and industrial conference practice and research techniques-MUTATION, 2007. TAICPART-MUTATION 2007* (2007), IEEE, pp. 169–175.

[22] Jia, Y., and Harman, M. An analysis and survey of the development of mutation testing. vol. 37, pp. 649–678.

[23] Just, R. The Major Mutation Framework: Efficient and Scalable Mutation Analysis for Java. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2014), ISSTA 2014, ACM, pp. 433–436.

[24] Just, R., Schweiggert, F., and Kapfhammer, G. M. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering* (2011), IEEE Computer Society, pp. 612–615.

[25] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. Aspect-oriented programming. In *ECOOP'97 — Object-Oriented Programming* (June 1997), Springer, Berlin, Heidelberg, pp. 220–242.

[26] Kiselev, I. *Aspect-Oriented Programming with AspectJ.* Sams, Indianapolis, IN, USA, 2002.

[27] Kosterec, M. METHODS OF CONCEPTUAL ANALYSIS. *Filozofia 71*, 3 (2016).

[28] Laddad, R. *AspectJ in Action: Enterprise AOP with Spring Applications*, 2nd ed. Manning Publications Co., Greenwich, CT, USA, 2009.

[29] Lipton, R. Fault diagnosis of computer programs. student report, 1971.

[30] Ma, Y.-S., Kwon, Y.-R., and Offutt, J. Inter-class mutation operators for Java. In *Software Reliability Engineering, 2002. ISSRE 2003. Proceedings. 13th International Symposium on* (2002), IEEE, pp. 352–363.

[31] Ma, Y.-S., and Offutt, J. Description of class mutation operators for java. *Electronics and Telecommunications Research Institute, Korea, Tech. Rep.* (2005).

[32] Ma, Y.-S., and Offutt, J. Description of method-level mutation operators for java. *Electronics and Telecommunications Research Institute, Korea, Tech. Rep.* (2005).

[33] Ma, Y.-S., Offutt, J., and Kwon, Y. R. MuJava: An automated class mutation system. vol. 15, pp. 97–133.

[34] Ma, Y.-S., Offutt, J., and Kwon, Y.-R. MuJava: A Mutation System for Java. In *Proceedings of the 28th International Conference on Software Engineering* (New York, NY, USA, 2006), ICSE '06, ACM, pp. 827–830.

[35] Madeyski, L., and Radyk, N. Judy - a mutation testing tool for java. vol. 4, pp. 32–42.

[36] Mateo, P. R., and Usaola, M. P. Bacterio: Java mutation testing tool: A framework to evaluate quality of tests cases. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on* (2012), IEEE, pp. 646–649.

[37] Mateo, P. R., and Usaola, M. P. Bacterio: Java mutation testing tool: A framework to evaluate quality of tests cases. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on* (2012), IEEE, pp. 646–649.

[38] Mateo, P. R., Usaola, M. P., and Offutt, J. Mutation at the multi-class and system levels. vol. 78, pp. 364–387.

[39] Myers, G. J., Sandler, C., and Badgett, T. *The Art of Software Testing.* John Wiley & Sons, Sept. 2011. Google-Books-ID: GjyEFPkMCwcC.

[40] Offutt, A. J., Lee, A., Rothermel, G., Untch, R. H., and Zapf, C. An experimental determination of sufficient mutant operators. vol. 5, pp. 99–118.

[41] Offutt, A. J., and Untch, R. H. Mutation 2000: Uniting the Orthogonal. In *Mutation Testing for the New Century*, W. E. Wong, Ed., no. 24 in The Springer International Series on Advances in Database Systems. Springer US, 2001, pp. 34–44. DOI: 10.1007/978-1-4757-5939-6_7.

[42] Petersen, K., Feldt, R., Mujtaba, S., and Mattsson, M. Systematic Mapping Studies in Software Engineering. In *EASE* (2008), vol. 8, pp. 68–77.

[43] Polo, M. Using aspect-oriented programming for mutation testing of third-party components. In *CIBSE 2014: Proceedings of the 17th Ibero-American Conference Software Engineering* (01 2014), pp. 247–260.

[44] Polo, M., Tendero, S., and Piattini, M. Integrating techniques and tools for testing automation. vol. 17, pp. 3–40.

[45] Shahriar, H., and Zulkernine, M. Mutec: Mutation-based testing of cross site scripting. In *Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems* (2009), IEEE Computer Society, pp. 47–53.

[46] Untch, R. H., Offutt, A. J., and Harrold, M. J. Mutation analysis using mutant schemata. In *ACM SIGSOFT Software Engineering Notes* (1993), vol. 18, ACM, pp. 139–148.

[47] Usaola, M. P., Rojas, G., Rodríguez, I., and Hernández, S. An Architecture for the Development of Mutation Operators. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)* (Mar. 2017), pp. 143–148.

[48] Woodward, M. R. Mutation testing—its origin and evolution. vol. 35, pp. 163–169.

Anexos

Apéndice A

Anexo I: Error-01

A.1. ERR-01: No inclusión del API de localización al crear el cliente de Google Play Services

- **Contexto:** El primer paso para acceder a la geolocalización a través de Google Play Services es crear una instancia del cliente del API. En el momento de crear este cliente, hay que especificar qué APIs en concreto se van a utilizar con ese cliente. Es sencillo olvidar este paso y que, cuando la aplicación vaya a utilizar ese cliente para acceder a las funciones de geolocalización falle.

  Tanto el impacto como la influencia de este error son altos, ya que provocaría que, aunque el uso de geolocalización esté autorizado por el usuario en el dispositivo y el cliente de acceso a Google Play Services se cree correctamente, no se podrían utilizar las funciones de geolocalización. Esto provocaría a su vez que la aplicación tuviera que detener su ejecución, o que siguiera ejecutándose sin poder registrar las posiciones del dispositivo.

- **Fallo que inserta:** No inclusión del API de localización al crear el cliente de Google Play Services.

- **Error que provoca:** La aplicación detendrá su ejecución, o seguirá ejecutándose sin poder registrar las posiciones del dispositivo (con la correspondiente pérdida de datos).

A.1.1. Operador ERR-01

Forma de reproducirlo

```
1 GoogleApiClient.Builder GoogleApiClient.Builder.addApi(Api<? extends
      NotRequiredOptions>)
2  -->
3 addApi(null)
```

Método que afecta

```
1 void GeolocationService.buildGoogleApiClient()
```

Formalización del Operador

Token	Elemento
ASPECT_NAME	Aspecto _OpLoc01 _GeolocationServiceBuildGoogleApiClient
RETURN_TYPE	GoogleApiClient.Builder
EXECUTION_MOMENT	around
CLASS_OBJECT	GoogleApiClient.Buildergoogleapiclient
TYPE_PARAMETER_NAME	Api<? extends NotRequiredOptions>p0
CAPTURE_METHOD	void GeolocationService.buildGoogleApiClient()
TYPE_OF_CAPTURE	call
METHOD_SIGNATURE	GoogleApiClient.Builder GoogleApiClient.Builder.addApi (Api extends NotRequiredOptions)
OBJECT_NAME	googleapiclient
PARAMETER_NAME	p0
CHANGE	return proceed (googleapiclient, null)

Tabla A.1: Formalización Operador ERR-01. Fuente: Elaboración Propia

A.1.2. Aspecto Operador ERR-01

```
13 public privileged aspect
       Aspecto_OpLoc01_GeolocationServiceBuildGoogleApiClient
14 {
15     GoogleApiClient.Builder around(GoogleApiClient.Builder
           googleapiclient, Api<? extends NotRequiredOptions> p0):
16     withincode(void GeolocationService.buildGoogleApiClient())
17
18     && call (GoogleApiClient.Builder GoogleApiClient.Builder.addApi(
           Api<? extends NotRequiredOptions> extends NotRequiredOptions
           >))
19     && target(googleapiclient)
20     && args(p0)
21     {
22         return proceed(googleapiclient, null);
23     }
24 }
```

Fragmento A.1: Aspecto Operador ERR-01. Fuente: Elaboración propia

Apéndice B

Anexo II: Error-02

B.1. ERR-02: Error al crear la conexión del cliente de Google Play Services API

- **Contexto:** Una vez creada la instancia de *GoogleApiClient*, al arrancar la *Activity* que la va a usar, debe hacer que se cree la conexión al servicio de Google Play Services. Es posible que el programador omita los pasos de conexión y desconexión por considerar que es suficiente con crear el cliente de Google Services.

  Al igual que en el Error ERR-01, este error tendría un impacto e influencia altos, ya que provocaría una excepción al intentar acceder a las funcionalidades de geolicalización, ocasionando que la aplicación se detenga.

- **Fallo que inserta:** Desconecta el servicio de geolocalización.

- **Error que provoca:** No muestra la geolocalización.

B.1.1. Operador ERR-02

Forma de reproducirlo

```
1 void GoogleApiClient.connect()
2  -->
3 GoogleApiClient.disconnect()
```

Método que afecta

```
1 void MapsActivity.onCreate(Bundle)
```

Formalización del Operador

Token	Elemento
ASPECT_NAME	Aspecto_OpLoc02_MapsActivityOnCreate
RETURN_TYPE	void
EXECUTION_MOMENT	around
CLASS_OBJECT	GoogleApiClient googleapiclient
TYPE_PARAMETER_NAME	
CAPTURE_METHOD	void MapsActivity.onCreate (Bundle)
TYPE_OF_CAPTURE	call
METHOD_SIGNATURE	void GoogleApiClient.connect()
OBJECT_NAME	googleapiclient
PARAMETER_NAME	
CHANGE	googleapiclient.disconnect()

Tabla B.1: Formalización Operador ERR-02. Fuente: Elaboración Propia

B.1.2. Aspecto Operador ERR-02

```
26 public privileged aspect Aspecto_OpLoc02_MapsActivityOnCreate
27 {
28     void around(GoogleApiClient googleapiclient ):
29     withincode(void MapsActivity.onCreate(Bundle))
30
31     && call (void GoogleApiClient.connect())
32     && target(googleapiclient)
33     && args()
34     {
35         googleapiclient.disconnect();
36     }
37 }
```

Fragmento B.1: Aspecto Operador ERR-02. Fuente: Elaboración propia

Apéndice C

Anexo III: Error-03

C.1. ERR-03: Obtención de la última posición del usuario

- **Contexto:** Un escenario particular en el uso de geolocalización es la obtención de la última posición conocida del usuario. El teléfono va guardando estas posiciones y, para aquellas aplicaciones que no necesitan conocer esa posición con precisión ni actualizarla periódicamente, es una opción válida y que no consume muchos recursos. El problema de esta operación es que puede que esa aplicación no se conozca debido a que el teléfono nunca ha tenido activados ninguno de los mecanismos de geolocalización, porque el teléfono se ha reiniciado, etc. La aplicación móvil que haga uso de esta funcionalidad tiene que contemplar esta situación.

  El impacto de ignorar esta comprobación sería alto para la propia aplicación. Es un error que, si bien es frecuente, es difícil de detectar. Si el dispositivo ha registrado alguna posición en algún momento desde que se ha encendido, lo normal es que la última posición conocida no sea *null*. Al probar la aplicación de forma manual, este es el caso más común. Sin embargo, el error ocurriría si el usuario ejecuta la aplicación en un terminal nuevo en el que el GPS no se ha activado nunca, o si tras apagar y volver a encender el dispositivo esta última posición conocida se ha eliminado.

- **Fallo que inserta:** Anula la última posición conocida por el dispositivo.

- **Error que provoca:** La no detección del último punto de localización registrado.

C.1.1. Operador ERR-03

Forma de reproducirlo

```
1 Location FusedLocationProviderApi.getLastLocation(googleApiClient)
2 -->
3 getLastLocation(null)
```

Método que afecta

```
1 MapsActivity.getLastKnownLocation()
```

Formalización del Operador

Token	Elemento
ASPECT_NAME	Aspecto _OpLoc03 _MapsActivityGetLastKnownLocation
RETURN_TYPE	Location
EXECUTION_MOMENT	around
CLASS_OBJECT	FusedLocationProviderApi fusedlocationproviderapi
TYPE_PARAMETER_NAME	GoogleApiClient p0
CAPTURE_METHOD	Location MapsActivity . getLastKnownLocation()
TYPE_OF_CAPTURE	call
METHOD_SIGNATURE	Location FusedLocationProviderApi.getLastLocation (GoogleApiClient)
OBJECT_NAME	fusedlocationproviderapi
PARAMETER_NAME	p0
CHANGE	return proceed (fusedlocationproviderapi, null)

Tabla C.1: Formalización Operador ERR-03. Fuente: Elaboración Propia

C.1.2. Aspecto Operador ERR-03

```
38 public privileged aspect
       Aspecto_OpLoc03_MapsActivityGetLastKnownLocation
39 {
40     Location around(FusedLocationProviderApi
           fusedlocationproviderapi, GoogleApiClient p0):
41     withincode(Location MapsActivity.getLastKnownLocation())
42
43     && call (Location FusedLocationProviderApi.getLastLocation(
           GoogleApiClient))
44     && target(fusedlocationproviderapi)
```

```
45      && args(p0)
46      {
47          return proceed(fusedlocationproviderapi, null);
48      }
49  }
```

Fragmento C.1: Aspecto Operador ERR-03. Fuente: Elaboración propia

Apéndice D

Anexo IV: Error-04

D.1. ERR-04: Error en los intervalos de recepción de actualizaciones de posición

- **Contexto:** Cuando la aplicación está interesada en recibir de forma continua actualizaciones sobre la posición del usuario, el servicio de localización de Google Play Services comunica dichas actualizaciones de forma periódica, siguiendo los intervalos especificados por el cliente. El cliente debe especificar dos intervalos en concreto: *setInterval* que especifica la periodicidad con la que se recibirán las actualizaciones de posición y *setFastestInterval* que establece la periodicidad máxima que es capaz de soportar la aplicación (cada vez que se recibe una actualización de la posición puede ser necesario ejecutar tareas como realizar consultas a servicios externos, lo que puede llevar un tiempo significativo). Al establecer estos intervalos para solicitar actualizaciones de posición, se pueden confundir los valores de los dos parámetros.

  Este error especifica cómo el tiempo *fastestInterval* es el doble del tiempo de *interval*, con lo que la aplicación está pidiendo que se le envíe una actualización de la posición cada segundo, mientras que también está diciendo que el tiempo mínimo que necesita para procesarla es de 2 segundos. Este error puede tener un impacto alto en la aplicación, ya que podría saturarla por un envío excesivo de posiciones del dispositivo, lo que haría que no pudiera dedicar recursos a ejecutar otras funciones.

- **Fallo que inserta:** Se especifica el valor de *fastestInterval* más alto que el de *interval*

- **Error que provoca:** Satura la aplicación debido a un envío excesivo de posiciones del dispositivo, lo que haría que no pudiera dedicar recursos a ejecutar

otras funciones.

D.1.1. Operador ERR-04

Forma de reproducirlo

```
1 LocationRequest Locationrequest.setFastestInterval(long)
2 -->
3 Locationrequest.setFastestInterval(p0*2)
```

Métodos que afecta

```
1 void MapsActivity.createLocationRequest()
2 void GeolocationService.createLocationRequest()
```

Formalización del Operador

Token	Elemento
ASPECT_NAME	Aspecto _OpLoc04 _GeolocationServiceCreateLocRequesthline
RETURN_TYPE	LocationRequest
EXECUTION_MOMENT	around
CLASS_OBJECT	LocationRequest locationrequest
TYPE_PARAMETER_NAME	long p0
CAPTURE_METHOD	void GeolocationService . createLocationRequest()
TYPE_OF_CAPTURE	call
METHOD_SIGNATURE	LocationRequest Locationrequest.setFastestInterval(long)
OBJECT_NAME	locationrequest
PARAMETER_NAME	p0
CHANGE	return proceed (locationrequest, p02)

Tabla D.1: Formalización Operador ERR-04. Fuente: Elaboración Propia

D.1.2. Aspecto Operador ERR-04

```
50 public privileged aspect
      Aspecto_OpLoc04_GeolocationServiceCreateLocationRequest
51 {
52     LocationRequest around(LocationRequest locationrequest, long p0)
          :
53     withincode(void GeolocationService.createLocationRequest())
```

```
54
55      && call (LocationRequest Locationrequest.setFastestInterval(long
            ))
56      && target(locationrequest)
57      && args(p0)
58      {
59          return proceed(locationrequest, p0*2);
60      }
61  }
```

Fragmento D.1: Aspecto Operador ERR-04. Fuente: Elaboración propia

Apéndice E

Anexo V: Error-05

E.1. ERR-05: Error en los intervalos de recepción de actualizaciones de posición

- **Contexto:** Este error implementa una situación similar al Error ERR-04, aunque con el efecto contrario. El sistema operativo Android comunica actualizaciones de posición a las aplicaciones al cumplirse el intervalo más pequeño que alguna de ellas ha establecido con *setInterval*. Así, si una aplicación pone un *fastestInterval* de 0 (un programador podría pensar en establecer este valor, pues ya indica con *setInterval* el intervalo de tiempo en que desea recibir las actualizaciones), la aplicación se puede ver desbordada al recibir actualizaciones demasiado rápidas (pero que no son demasiado rápidas para otra aplicación que ha establecido un *setInterval* menor).
- **Fallo que inserta:** Aumenta el parámetro de *fastestInterval*.
- **Error que provoca:** La aplicación se puede ver desbordada al recibir actualizaciones demasiado frecuentes.

E.1.1. Operador ERR-05

Forma de reproducirlo

```
1 LocationRequest locationrequest.setFastestInterval(long p0)
2 -->
3 setFastestInterval(0)
```

Métodos que afecta

```
1 void MapsActivity.createLocationRequest()
```

Formalización del Operador

Token	Elemento
ASPECT_NAME	OpLoc05 _GeolocationServiceCreateLocationRequest
RETURN_TYPE	LocationRequest
EXECUTION_MOMENT	around
CLASS_OBJECT	LocationRequest locationrequest
TYPE_PARAMETER_NAME	long p0
CAPTURE_METHOD	void GeolocationService.createLocationRequest()
TYPE_OF_CAPTURE	call
METHOD_SIGNATURE	LocationRequest Locationrequest.setFastestInterval(long)
OBJECT_NAME	locationrequest
PARAMETER_NAME	p0
CHANGE	return proceed (locationrequest, 0)

Tabla E.1: Formalización Operador ERR-05. Fuente: Elaboración Propia

E.1.2. Aspecto Operador ERR-05

```
62 public privileged aspect
       Aspecto_OpLoc05_GeolocationServiceCreateLocationRequest
63 {
64     LocationRequest around(LocationRequest locationrequest, long p0)
           :
65     withincode(void GeolocationService.createLocationRequest())
66
67     && call (LocationRequest Locationrequest.setFastestInterval(long
           ))
68     && target(locationrequest)
69     && args(p0)
70     {
71         return proceed(locationrequest, 0);
72     }
73 }
```

Fragmento E.1: Aspecto Operador ERR-05. Fuente: Elaboración propia

Apéndice F

Anexo VI: Error-06

F.1. ERR-06: Detener la recepción de posiciones sin comprobar el cliente de Google Play Services

- **Contexto:** El cliente del servicio de Google Play Services es el objeto que nos permite acceder a las funcionalidades de geolocalización. Este servicio es externo a la aplicación y por tanto, se puede detener sin que se sepa. Así, cada vez que le solicitamos que se inicien o detengan las actualizaciones de localización del dispositivo debemos comprobar que la conexión del cliente con el servicio sigue activa.

  Este error es difícil de detectar en pruebas manuales si no se provoca forzosamente, ya que el servicio de Google Play Services no suele detenerse. Sin embargo, puede ocurrir y provocar un fallo en nuestra aplicación. Es muy habitual que el desarrollador omita esta comprobación (ya que no es una situación que se suela dar), por lo que su frecuencia es Alta. Tanto el impacto en la aplicación como la influencia en otros componentes del sistema son Altos, ya que este error puede provocar que se detenga la ejecución de la aplicación móvil.

- **Fallo que inserta:** Se detiene el servicio de Google Play Services antes de solicitar la recepción de actualizaciones de posición del API.

- **Error que provoca:** Se detiene la ejecución de la aplicación.

F.1.1. Operador ERR-06

Forma de reproducirlo

```
1 PendingResult<Status> FusedLocationProviderApi.
     requestLocationUpdates
2 (GoogleApiClient, LocationRequest, LocationListener)
3 -->
```

```
4 googleapiclient.disconnect()
5 requestLocationUpdates(GoogleApiClient, LocationRequest,
    LocationListener)
```

Método que afecta

```
1 void MapsActivity.startLocationUpdates()
```

Formalización del Operador

Token	Elemento
ASPECT_NAME	Aspecto_OpLoc06 _MapsActivityStartLocationUpdates
RETURN_TYPE	PendingResult<Status>
EXECUTION_MOMENT	around
CLASS_OBJECT	FusedLocationProviderApi fusedlocationproviderapi
TYPE_PARAMETER_NAME	GoogleApiClient p0, LocationRequest p1, LocationListener p2
CAPTURE_METHOD	void MapsActivity.startLocationUpdates()
TYPE_OF_CAPTURE	call
METHOD_SIGNATURE	PendingResult FusedLocationProviderApi. requestLocationUpdates (GoogleApiClient, LocationRequest, LocationListener)
OBJECT_NAME	fusedlocationproviderapi
PARAMETER_NAME	p0, p1, p2
CHANGE	p0.disconnect() ; return proceed (fusedlocationproviderapi, p0, p1, p2);

Tabla F.1: Formalización Operador ERR-06. Fuente: Elaboración Propia

F.1.2. Aspecto Operador ERR-06

```
74 public privileged aspect
     Aspecto_OpLoc06_MapsActivityStartLocationUpdates
75 {
76     PendingResult<Status> around
77     (FusedLocationProviderApi fusedlocationproviderapi,
          GoogleApiClient p0, LocationRequest p1, LocationListener p2):
78     withincode(void MapsActivity.startLocationUpdates())
79
```

```
80      && call (PendingResult<Status> FusedLocationProviderApi.
           requestLocationUpdates(GoogleApiClient, LocationRequest,
           LocationListener))
81      && target(fusedlocationproviderapi)
82      && args(p0, p1, p2)
83      {
84          p0.disconnect();
85          return proceed(fusedlocationproviderapi, p0, p1, p2);
86      }
87  }
```

Fragmento F.1: Aspecto Operador ERR-06. Fuente: Elaboración propia

Apéndice G

Anexo VII: Error-07

G.1. ERR-07: Error en la obtención de una dirección a partir de una localización

- **Contexto:** Una funcionalidad muy típica en las aplicaciones móviles que utilizan geolocalización es transformar una localización en una dirección postal. En la gran mayoría de los casos el usuario final no requiere las coordenadas de latitud y longitud, sino una dirección que sea entendible. La clase *Geocoder* permite transformar una posición en una dirección, y una dirección en una localización. El método *getFromLocation* nos dará la dirección que corresponde a una posición determinada. Sin embargo, esa posición no se especifica mediante un objeto de la clase *Location*, sino que se especifica a partir de la latitud y la longitud.

  Un error que puede cometer el programador en esta operación es confundir los parámetros de latitud y longitud. Este error se puede reproducir forzando esa confusión. La frecuencia con la que se puede dar este error es media. El impacto para la propia aplicación móvil es alto, ya que estaría proporcionando al usuario una dirección que no tiene ningún sentido.

- **Fallo que inserta:** Se confunden los parámetros de latitud y longitud.

- **Error que provoca:** La aplicación provee al usuario una dirección incorrecta.

G.1.1. Operador ERR-07

Forma de reproducirlo

```
1 List<Address> Geocoder.getFromLocation(double, double, int)
2 -->
3 getFromLocation(geocoder, p1, p0, p2)
```

Método que afecta

```
1 Geocoder.getFromLocation()
```

Formalización del Operador

Token	Elemento
ASPECT_NAME	Aspecto _OpLoc07 _MapsActivityCreateLocationRequest
RETURN_TYPE	List<Address>
EXECUTION_MOMENT	around
CLASS_OBJECT	Geocoder geocoder
TYPE_PARAMETER_NAME	double p0, double p1, int p2
CAPTURE_METHOD	void MapsActivity.createLocationRequest()
TYPE_OF_CAPTURE	call
METHOD_SIGNATURE	List <Address>Geocoder.getFromLocation (double, double, int)
OBJECT_NAME	geocoder
PARAMETER_NAME	p0, p1, p2
CHANGE	return proceed (geocoder, p1, p0, p2) ;

Tabla G.1: Formalización Operador ERR-07. Fuente: Elaboración Propia

G.1.2. Aspecto Operador ERR-07

```
1  public privileged aspect
       Aspecto_OpLoc07_MapsActivityCreateLocationRequest
2  {
3      List<Address> around(Geocoder geocoder, double p0, double p1,
           int p2):
4      withincode(void MapsActivity.createLocationRequest())
5
6      && call (List<Address> Geocoder.getFromLocation(double, double,
           int))
7      && target(geocoder)
8      && args(p0, p1, p2)
9      {
10         return proceed(geocoder, p1, p0, p2);
11     }
12 }
```

Fragmento G.1: Aspecto Operador ERR-07. Fuente: Elaboración propia

Apéndice H

Anexo VIII: Error-08

H.1. ERR-08: Error al abrir la aplicación de Google Maps desde otra aplicación

- **Contexto:** Algo muy típico en una aplicación Android que maneja entidades georeferenciadas es abrir automáticamente *Google Maps* o *Google Navigation* para que el usuario vea en un mapa dónde se encuentra esa entidad, o incluso para que el navegador del sistema le guie hasta esa posición. Un ejemplo típico sería una aplicación turística que cuenta con puntos de interés como monumentos, restaurantes, etc. En lugar de incorporar en la propia aplicación turística la navegación, se suele optar por realizar esas funciones con *Google Maps* y *Google Navigation*, invocándolas desde la propia aplicación (por ejemplo, mediante el típico botón Cómo llegar a).

  Invocar a estas aplicaciones es una acción muy sencilla implementada mediante un *Intent*, que nos permite invocarlas mediante una *URI*. Sin embargo, estas *URIs* admiten distintos tipos de parámetros, y es fácil equivocarse al construir la llamada. Por ejemplo, en el caso de *Google Maps*, la *URI* en el caso más básico sería *geo:latitud, longitud*. Sin embargo, también admite *URIs* más complejas. También se puede especificar el nivel de zoom con z=zoom, utilizar una etiqueta en la posición mostrada con lng(label), etc. Un error típico consistiría en especificar mal estos parámetros.

  De forma similar al caso del error anterior, el impacto de este error para la propia aplicación móvil es Alto, ya que impediría al usuario utilizar la funcionalidad de invocar al navegador para poder desplazarse a una localización determinada.

- **Fallo que inserta:** Al transformar una posición en una dirección, se confunden los campos de latitud y longitud.

- **Error que provoca:** Impediría al usuario utilizar la funcionalidad de invocar al navegador para poder desplazarse a una localización determinada.

H.1.1. Operador ERR-08

Forma de reproducirlo

```
showMap(View view, Double lat, Double lon)
-->
showMap(View view, Double Long, Double Lat)
```

Métodos que afecta

```
mLocationRequest.setInterval(..)
mLocationRequest.setFastestInterval(..)
```

Formalización del Operador

Token	**Elemento**
ASPECT_NAME	Aspecto _OpLoc08 _MapsActivityCreateLocationRequest
RETURN_TYPE	void
EXECUTION_MOMENT	around
CLASS_OBJECT	Object objec
TYPE_PARAMETER_NAME	View p0, double p1, double p2
CAPTURE_METHOD	void MapsActivity . createLocationRequest()
TYPE_OF_CAPTURE	call
METHOD_SIGNATURE	void Object.showMap(View, double, double)
OBJECT_NAME	object
PARAMETER_NAME	p0, p1, p2
CHANGE	return proceed (object, p0, p2, p1) ;

Tabla H.1: Formalización Operador ERR-08. Fuente: Elaboración Propia

H.1.2. Aspecto Operador ERR-08

```
88 public privileged aspect
       Aspecto_OpLoc08_MapsActivityCreateLocationRequest
89 {
90     void around(Object object, View p0, double p1, double p2):
91     withincode(void MapsActivity.createLocationRequest())
```

```
92
93      && call (void Object.showMap(View, double, double))
94      && target(object)
95      && args(p0, p1, p2)
96      {
97          return proceed(object, p0, p2, p1);
98      }
99  }
```

Fragmento H.1: Aspecto Operador ERR-08. Fuente: Elaboración propia

Apéndice I

Anexo X: Error-10

I.1. ERR-10: Errores al establecer el radio de un Geofence

- **Contexto:** Un error similar puede ocurrir al establecer el radio que delimita el *Geofence*, interpretando que un valor 0 resulta en un radio por defecto que establece el sistema. La valoración de frecuencia es mayor en este caso, ya que se trata de un error más fácil de cometer.

- **Fallo que inserta:** Establece el radio del *Geofence* en el valor 0.

- **Error que provoca:** No se hace visible el radio que ocupa del *Geofence* en el mapa.

I.1.1. Operador ERR-10

Forma de reproducirlo

```
1 CircleOptions MapFragment.createCircleOptions(double, double, float)
2 -->
3 createCircleOptions(geofence, p0, p1, 0)
```

Método que afecta

```
1 void MapFragment.displayGeofences()
```

Formalización del Operador

Token	Elemento
ASPECT_NAME	Aspecto _OpLoc10_MapFragmentDisplayGeofences
RETURN_TYPE	CircleOptions
EXECUTION_MOMENT	around
CLASS_OBJECT	MapFragment mapfragment
TYPE_PARAMETER_NAME	double p0, double p1, float p2
CAPTURE_METHOD	void MapFragment. displayGeofences()
TYPE_OF_CAPTURE	call
METHOD_SIGNATURE	CircleOptions MapFragment.createCircleOptions(double, double, float)
OBJECT_NAME	mapfragment
PARAMETER_NAME	p0, p1, p2
CHANGE	return proceed (mapfragment, p0, p1, 0)

Tabla I.1: Formalización Operador ERR-10. Fuente: Elaboración Propia

I.1.2. Aspecto Operador ERR-10

```
100 public privileged aspect Aspecto_OpLoc10_MapFragmentDisplayGeofences
101 {
102     CircleOptions around(MapFragment mapfragment, double p0, double
          p1, float p2):
103     withincode(void MapFragment.displayGeofences())
104
105     && call (CircleOptions MapFragment.createCircleOptions(double,
          double, float))
106     && target(mapfragment)
107     && args(p0, p1, p2)
108     {
109         return proceed(mapfragment, p0, p1, 0);
110     }
111 }
```

Fragmento I.1: Aspecto Operador ERR-10. Fuente: Elaboración propia

Apéndice J

Anexo XI: Error-11

J.1. ERR-11: Error al establecer el tiempo de duración de un Geofence

- **Contexto:** Otro de los parámetros que hay que establecer al crear un *Geofence* es el tiempo durante el que estará activo. Si la duración es un número positivo, se eliminará el *Geofence* pasado ese número de milisegundos. Si es un número negativo, el sistema entiende que ese *Geofence* ya ha expirado, y lo elimina o no lo crea.

  En este caso, de producirse el error, la aplicación móvil directamente dejaría de emitir las notificaciones de entrada/salida en zonas de interés.

- **Fallo que inserta:** Establece una duración negativa.

- **Error que provoca:** Al crear un *Geofence*, como se establece una duración negativa, el sistema entiende que ese *Geofence* ya ha expirado, y lo elimina.

J.1.1. Operador ERR-11

Forma de reproducirlo

```
1 Geofence.Builder Geofence.Builder.setExpirationDuration(long p0)
2 -->
3 setExpirationDuration(p0*(-1))
```

Método que afecta

```
1 Geofence SimpleGeofence.toGeofence()
```

Formalización del Operador

Token	Elemento
ASPECT_NAME	Aspecto _OpLoc11_SimpleGeofenceToGeofence
RETURN_TYPE	Geofence.Builder
EXECUTION_MOMENT	around
CLASS_OBJECT	Geofence.Builder geofence
TYPE_PARAMETER_NAME	long p0
CAPTURE_METHOD	Geofence SimpleGeofence.toGeofence()
TYPE_OF_CAPTURE	call
METHOD_SIGNATURE	Geofence.Builder Geofence.Builder. setExpirationDuration(long)
OBJECT_NAME	geofence
PARAMETER_NAME	p0
CHANGE	return proceed (geofence, p0 (-1))

Tabla J.1: Formalización Operador ERR-11. Fuente: Elaboración Propia

J.1.2. Aspecto Operador ERR-11

```
113 public privileged aspect Aspecto_OpLoc11_SimpleGeofenceToGeofence
114 {
115     Geofence.Builder around(Geofence.Builder geofence, long p0):
116     withincode(Geofence SimpleGeofence.toGeofence())
117
118     && call (Geofence.Builder Geofence.Builder.setExpirationDuration
            (long))
119     && target(geofence)
120     && args(p0)
121     {
122         return proceed(geofence, p0 * (-1));
123     }
124 }
```

Fragmento J.1: Aspecto Operador ERR-11. Fuente: Elaboración propia

Apéndice K

Anexo XII: Error-12

K.1. ERR-12: Error al comprobar el tipo de evento notificado de un Geofence

- **Contexto:** Cuando se recibe una notificación de un evento relacionado con un *Geofence*, dicho evento puede señalar una acción de entrada o una acción de salida de ese *Geofence*. Los códigos de cada tipo de evento están definidos como constantes en la clase *Geofence(Geofence.GEOFENCE_TRANSITION_ENTER*, y *Geofence.GEOFENCE_TRANSITION_EXIT)*. Al tratar cada evento, el programador puede confundir los dos eventos, produciendo un error en la aplicación.

  En este caso impacto e influencia tienen una valoración de alto, ya que de producirse este error se podría producir un caos en la notificación y envío de eventos de entrada/salida (por ejemplo, notificando dos veces seguidas que el usuario ha llegado a un lugar).

- **Fallo que inserta:** Error al confundir el tipo de evento notificado de un *Geofence*.

- **Error que provoca:** Se produce un error en la aplicación sobre la notificación y envío de eventos de entrada/salida.

K.1.1. Operador ERR-12

Forma de reproducirlo

```
1 int GeofencingEvent.getGeofenceTransition()
2 -->
3 Geofence.GEOFENCE\_TRANSITION\_EXIT
```

Método que afecta

```
1 void GeofenceReceiver.onHandleIntent(Intent)
```

Formalización del Operador

Token	Elemento
ASPECT_NAME	Aspecto _OpLoc12_GeofenceReceiverOnHandleIntent
RETURN_TYPE	int
EXECUTION_MOMENT	around
CLASS_OBJECT	GeofencingEvent geofencingevent
TYPE_PARAMETER_NAME	
CAPTURE_METHOD	void GeofenceReceiver.onHandleIntent(Intent)
TYPE_OF_CAPTURE	call
METHOD_SIGNATURE	int GeofencingEvent.getGeofenceTransition()
OBJECT_NAME	geofencingevent
PARAMETER_NAME	
CHANGE	return Geofence.GEOFENCE_TRANSITION_EXIT

Tabla K.1: Formalización Operador ERR-12. Fuente: Elaboración Propia

K.1.2. Aspecto Operador ERR-12

```
125 public privileged aspect
        Aspecto_OpLoc12_GeofenceReceiverOnHandleIntent
126 {
127     int around(GeofencingEvent geofencingevent ):
128     withincode(void GeofenceReceiver.onHandleIntent(Intent))
129
130     && call (int GeofencingEvent.getGeofenceTransition())
131     && target(geofencingevent)
132     && args()
133     {
134         return Geofence.GEOFENCE_TRANSITION_EXIT;
135     }
136 }
```

Fragmento K.1: Aspecto Operador ERR-12. Fuente: Elaboración propia