Universidad Zaragoza

Trabajo Fin de Grado

Herramienta para la medición automática de cobertura de código mediante la técnica de caminos con profundidad de nivel 2 en Java

Tool for automatic code coverage measurement using the path coverage technique with depth level 2 in Java

Autor

Juan Catalán Bernal

Director

Miguel Ángel Latre Abadía

ESCUELA DE INGENIERÍA Y ARQUITECTURA

2024

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe acompañar al Trabajo Fin de Grado (TFG)/Trabajo Fin de Máster (TFM) cuando sea depositado para su evaluación).

D./Dª. Juan Catalán Bernal,

con nº de DNI 73009179A en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de (Grado/Máster) Grado en Ingeniería Informática, (Título del Trabajo) Herramienta para la medición automática de cobertura de código mediante la técnica de caminos de profundidad de nivel 2 en Java,

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 5 de Septiembre de 2024

Firmado por CATALAN BERNAL JUAN - ***0917** el día 05/09/2024 con un certificado emitido por AC FNMT Usuarios

Fdo: _____

AGRADECIMIENTOS

Primeramente quiero agradecer a Miguel Ángel Latre, tutor de este trabajo, por todo el tiempo dedicado, los ánimos recibidos y por haber ayudado a que la experiencia de trabajar en el TFG haya sido (a excepción de los últimos momentos de estrés) entretenida y satisfactoria.

Agradecer también a todos aquellos compañeros y amigos que he conocido en la carrera, que convierten los días de aburrimiento y de estrés, en días que, aunque sigan siendo estresantes y llenos de trabajos, vuelves a casa con una sonrisa y con (al menos unas pocas) ganas de volver al día siguiente. Entre ellos destacar a aquellos que conocí casi al comienzo de la carrera como Ernesto, Antonio, Jaime y Ainhoa, y a aquellos que aunque los he conocido el ultimo año han sido algo clave para haber llegado al día de hoy: Carlos, Senso, Raúl, Jaime, Carlota y Marina.

Por último agradecer a mis personas más cercanas, tanto a mi familia de sangre por haber puesto la confianza en mí para comenzar la carrera y seguir en ella a pesar de un mal comienzo, como a los amigos que considero parte de mi familia como Pablete, Miguel Ángel, Dieguito, Sara, Diego Del, Claudia y Adri (que me ha acompañado en la carrera un día tras otro) por haberme entendido y apoyado en los días malos de la carrera y por haber celebrado cuando las cosas han ido bien.

Herramienta para la medición automática de cobertura de código mediante la técnica de caminos con profundidad de nivel 2 en Java

RESUMEN

A la hora de realizar *testing* de sistemas informáticos es común utilizar diferentes *técnicas de diseño de pruebas*. Estas son métodos definidos para obtener, a partir de unos requisitos o especificaciones determinados, casos de prueba que proporcionan una determinada cobertura para los mismos.

Una de ellas es la técnica de *caminos con profundidad de nivel 2*, la cual es una técnica de *testing* basada en la estructura que pretende diseñar un conjunto de casos de prueba que asegure que todos los pares de aristas adyacentes del grafo de flujo asociado al código objeto de pruebas se ejecuten al menos una vez.

La técnica de caminos con profundidad 2 es una técnica muy exhaustiva para la que no existen herramientas de medición de cobertura que comprueben que las pruebas diseñadas están cubriendo todos los caminos deseados. Este TFG busca crear una herramienta para *Java* que automatice el proceso de medición de cobertura de pruebas diseñadas con esta técnica, ofreciendo un informe de cobertura de código tras la ejecución de las pruebas.

Para ello se va a desarrollar un agente en Java que mediante instrumentación de código Java, sea capaz de analizar un método generando su grafo de flujo asociado, identificando las situaciones de prueba y modificando el código para registrar de manera dinámica las situaciones de prueba que se vayan ejecutando.

Además se va a implementar un plugin para el IDE IntelliJ que integre el agente previamente mencionado, para facilitar el uso de la herramienta con una configuración simple por parte de los programadores.

Tool for automatic code coverage measurement using the path coverage technique with depth level 2 in Java

ABSTRACT

At the time of performing testing of computer systems, it is common to use different *test design techniques*. These are defined methods to obtain, from certain requirements or specifications, test cases that provide a certain coverage for them.

One of them is the *path coverage technique with level 2 depth*, which is a structure-based *testing* technique aimed at designing a set of test cases that ensures that all pairs of adjacent edges of the flow graph associated with the code under test are executed at least once.

The path coverage technique with level 2 depth technique is a very exhaustive technique for which there are no coverage measurement tools that check whether the designed tests are covering all the desired paths. This final project aims to create a tool for *Java* that automates the process of measuring the coverage of tests designed with this technique, offering a code coverage report after the tests are executed.

To achieve this, a Java agent will be developed that, through Java code instrumentation, is capable of analyzing a method, generating its associated flow graph, identifying test situations, and modifying the code to dynamically record the test situations that are being executed.

In addition, a plugin for the IntelliJ IDE will be implemented to integrate the previously mentioned agent, making it easier for programmers to use the tool with a simple configuration.

Índice

Capítulo 1

Introducción y objetivos

1.1. Contexto

1.1.1. Testing y técnicas de diseño de pruebas

A la hora de realizar *testing* de sistemas informáticos existen diferentes formas de definir que se va a probar, desde enfoques más informales como pruebas basadas en la experiencia (pruebas exploratorias, basadas en listas de comprobación, etc.) hasta la utilización de diferentes técnicas de diseño de prueba [1].

Las técnicas de diseño de pruebas son métodos definidos para obtener, a partir de unos requisitos o especificaciones determinados, casos de prueba de un software que proporcionan una determinada cobertura para el mismo. Cada técnica enfoca las pruebas a aspectos diferentes para adaptarse a la exigencia de la estrategia de pruebas elegida.

Las diferentes técnicas se pueden clasificar en dos grupos [2]:

- **Técnicas basadas en la especificación** (o técnicas de pruebas de caja negra): técnicas donde la *test basis* (requisitos, especificación, modelos) es usada como la principal fuente de información para diseñar casos de prueba.

  Un ejemplo de técnica basada en la especificación es la **técnica de particiones de equivalencia** donde, para un determinado objeto de pruebas (función, módulo, programa que se desea testear) se busca definir las diferentes particiones de equivalencia de los parámetros o entradas del objeto de pruebas (casos conceptualmente semejantes), y a partir de ellas diseñar pruebas que cubran el conjunto de las particiones de equivalencia identificadas.

- **Técnicas basadas en la estructura** (o técnicas de pruebas de caja blanca): técnicas donde la estructura del objeto de pruebas (generalmente el código fuente) es utilizada como la fuente primaria de información para diseñar los casos de prueba.

Algunos ejemplos de técnicas basadas en la estructura son:

- **Instrucciones *(Statement Testing)***: es una técnica que pretende diseñar un conjunto de casos de prueba que asegure que todas las instrucciones del código objeto de pruebas se ejecuten al menos una vez.
- **Ramas *(Branch Testing)***: es una técnica que pretende diseñar un conjunto de casos de prueba que asegure que todas las aristas del grafo de flujo asociado al código objeto de pruebas se ejecuten al menos una vez.
- **Caminos de profundidad 2 *(Edge-Pair Testing)***: esta técnica se puede entender como una extensión de la técnica de cobertura de ramificación donde, en vez de todas las aristas del grafo de flujo, se pretende asegurar que se ejecutan todos las pares de aristas adyacentes. Es la técnica en la que se centra este TFG y se describe con más detalle en la sección 1.1.4.

1.1.2. Cobertura de código

Cuando se realiza *testing* es conveniente tener medidas que nos permitan observar como de bien se esta realizando los test, y las partes del programa o el código que se están probando con mayor y menor exhaustividad. Una de las medidas más utilizadas para ello es la cobertura de código, que muestra el grado en el que código fuente de un programa es ejecutado durante la ejecución de un conjunto de pruebas en concreto [3].

Que un programa tenga una cobertura de código baja al ejecutar los test indica que solo una pequeña parte del programa se ha ejecutado durante los test, lo que sugiere que existe una mayor probabilidad de encontrar fallos inesperados respecto a un programa con alta cobertura de código.

Existen diferentes tipos de cobertura de código, los más comunes (y que múltiples herramientas son capaces de medir automáticamente) son:

- **Cobertura de líneas**: mide el número de líneas del código fuente que se han ejecutado.
- **Cobertura de instrucciones**: mide el número de instrucciones del código fuente que se han ejecutado.
- **Cobertura de clases**: mide el número de clases (en lenguajes orientados a objetos) que se han utilizado durante la ejecución de los tests.
- **Cobertura de funciones o métodos**: mide el número de funciones o métodos (en lenguajes orientados a objetos) que han sido invocados durante la ejecución de los tests.

– **Cobertura de ramas**: mide el número de ramas de cada estructura de control (como *if* o *while*) que han sido ejecutadas. Por ramas se entiende los diferentes caminos que puede tomar el flujo del programa al evaluarse las condiciones que los guardan como *true* o *false*.

– **Cobertura de condiciones**: mide el número de sub-expresiones booleanas que han sido evaluadas con sus diferentes posibles valores.

Un ejemplo de los diferentes tipos de cobertura y su medición se encuentra en el Anexo A.2.

Los IDE generalmente son capaces medir algunos de estos tipos de cobertura y ofrecerte un informe. En la figura 1.1 se puede observar un ejemplo de informe de cobertura del IDE IntelliJ, con los diferentes tipos de cobertura que este IDE es capaz de medir.

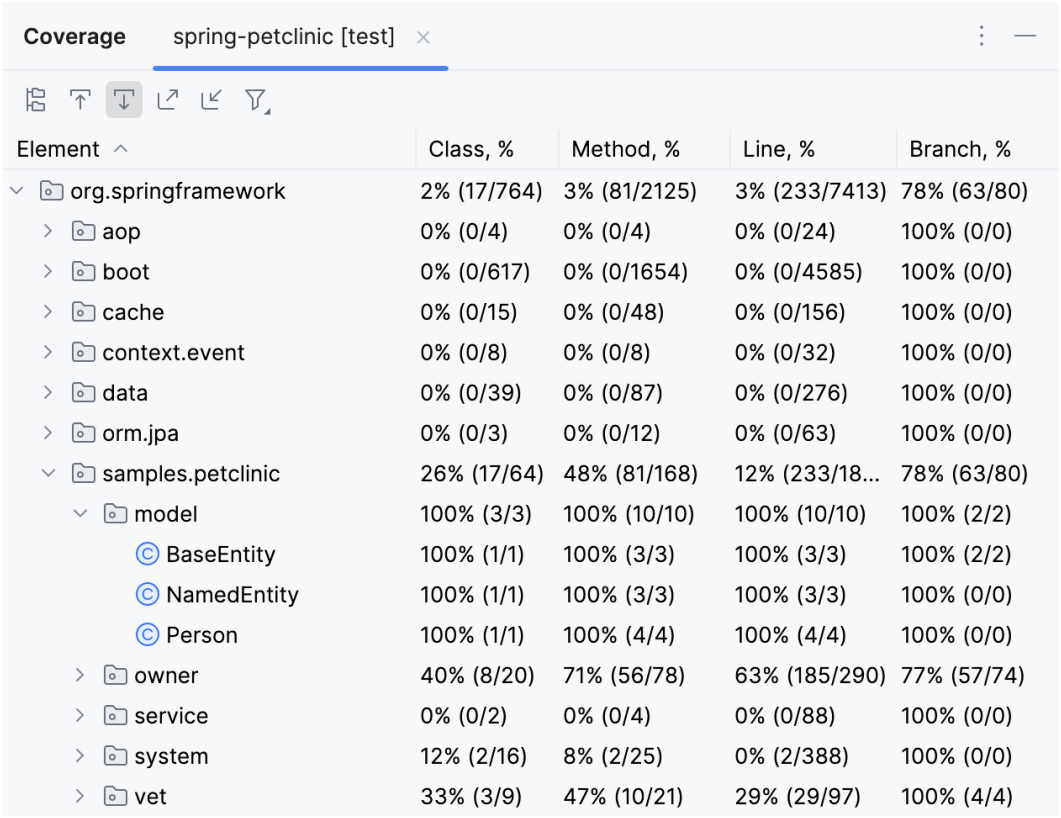

Figura 1.1: Ejemplo de informe de cobertura de IntelliJ

1.1.3. Automatización de las pruebas

La automatización de pruebas implica la escritura de código específico cuya única función es verificar que el código de producción (es decir, el código que realmente ejecutará el usuario final y que resuelve el problema que el software pretende abordar) se comporta como se espera en diversas situaciones. Este código de pruebas se ejecuta de forma automática, permitiendo la repetición de pruebas sin intervención manual, lo cual ahorra tiempo y minimiza el riesgo de errores humanos.

Para poder automatizar las pruebas es necesario un entorno de automatización de pruebas y *test harness*: esto es una colección de software que facilita la ejecución de pruebas, incluye el entorno general y los scripts específicos para un programa o proyecto. En el caso de Java, el entorno de automatización de pruebas predominante es JUnit[1].

1.1.4. Técnica de caminos con profundidad de nivel 2

Este trabajo esta centrado en la técnica de *caminos con profundidad de nivel 2*, la cual es una técnica de *testing* basada en la estructura que pretende diseñar un conjunto de casos de prueba que asegure que todos los pares de aristas adyacentes del grafo de flujo asociado al código objeto de pruebas se ejecuten al menos una vez.

Para aplicar la técnica se pueden seguir los siguientes pasos:

1. Generar el grafo de control asociado al código objeto de pruebas.

2. Determinar los nodos "relevantes" del grafo (nodo inicial, nodos finales y nodos predicado) y determinar las aristas que los conectan basándose en el grafo completo.

   Esta técnica es particularmente sensible a la definición de nodo predicado (nodos en los que el flujo del programa se puede dividir). Consideramos nodo predicado a cada condición simple que aparece en la guarda de una instrucción condicional o iterativa.

3. Incluir como situaciones de prueba todas las combinaciones de caminos de dos aristas que pasan por nodos predicado.

4. Generar caminos sobre el grafo hasta que se hayan incluido todas las situaciones de prueba.

5. Generar casos de prueba que satisfagan los caminos.

[1] `https://junit.org/junit5/`

Ejemplo de aplicación de la técnica

Objeto de pruebas Dado el siguiente método con nombre *buscar*, que va a ser nuestro objeto de pruebas:

```java
/**
 * Busca un dato determinado en un vector de enteros.
 *
 * @param v
 *             - el vector no nulo en el que se busca el entero.
 * @param datoBuscado
 *             - el dato que se quiere buscar en el vector «v».
 *
 * @return Si en el vector «v» hay un dato igual a «datoBuscado»,
 *         devuelve el índice de la componente en la que se encuentra.
 *         En caso contrario devuelve -1;
 */
public int buscar(int[] v, int datoBuscado) {
    int i = 0;

    while (i < v.length) {
        if (v[i] == datoBuscado) {
            return i;
        }
        i++;
    }

    return -1;
}
```

Figura 1.2: Código Java de la función *buscar*

Los pasos a seguir son:

Generar el grafo de control Se va a utilizar como identificador de vértices el número de linea de su representación en el código para facilitar la lectura del grafo.

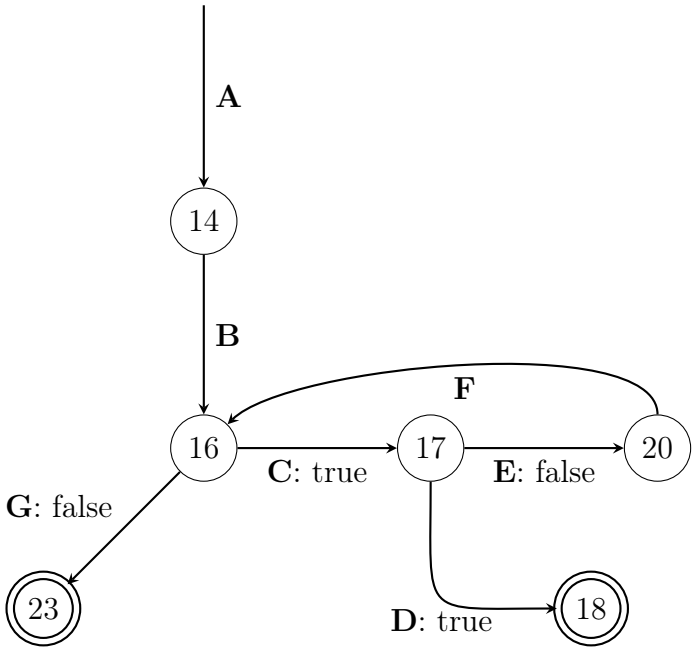

Figura 1.3: Grafo de flujo asociado al código Java de 1.2

Determinar los nodos "relevantes" del grafo Para ello, a partir del grafo completo, nos centramos únicamente en los nodos inicial, finales y predicado, y determinamos las aristas que los conectan.

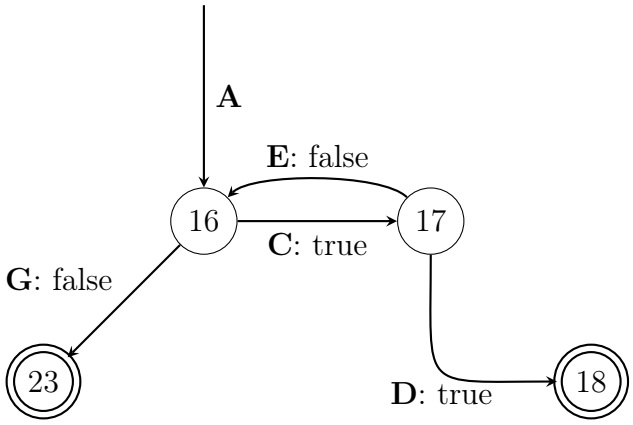

Figura 1.4: Grafo de flujo con nodos "relevantes" basado en el grafo de la figura 1.3

Generar situaciones de prueba Las situaciones de prueba serán todo aquel conjunto de pares de aristas adyacente, en este caso:

AC , AG , CD , CE , EC , EG

Generar caminos que incluyan todas las situaciones de prueba De manera iterativa e incremental, se definen caminos que cubran algunas de las situaciones de prueba restantes.

Por ejemplo, podríamos empezar con un camino que únicamente cubra **AG**.

Caminos	**Situaciones de prueba restantes**
AG	AC, ~~AG~~, CD, CE, EC, EG

Tras esto, podemos seguir con un camino que cubra **AC** y **CD**.

Caminos	**Situaciones de prueba restantes**
AG, ACD	~~AC~~, ~~AG~~, ~~CD~~, CE, EC, EG

Para finalizar, podemos buscar un camino que cubra las situaciones restantes: **CE**, **EC** y **EG**.

Caminos	**Situaciones de prueba restantes**
AG, ACD, ACECEG	~~AC~~, ~~AG~~, ~~CD~~, ~~CE~~, ~~EC~~, ~~EG~~

Generar casos de prueba que satisfagan los caminos En este caso, un ejemplo de casos de prueba que satisfacen los caminos previamente determinados se ven en la tabla 1.1.

Caso	**Parámetros**		**Resultado**	**Camino cubierto**
	int[] v	*int datoBuscado*		
1	[]	1	-1	AG
2	[1,2,3]	1	0	ACD
3	[1, 2]	3	-1	ACECEG

Tabla 1.1: Tabla de casos de prueba para el método de la figura 1.2

1.2. Objetivos

La técnica de caminos con profundidad 2 es una técnica muy exhaustiva para la que no existen herramientas de medición de cobertura que comprueben que las pruebas diseñadas están cubriendo todos los caminos deseados.

Este TFG busca crear una herramienta para Java que automatice el proceso de medición de cobertura de pruebas diseñadas con esta técnica, ofreciendo un informe de cobertura de código tras la ejecución de las pruebas.

Para ello se buscará un medidor de cobertura para Java, similar a los que ya existen para otros tipos de cobertura más fáciles, que realice los pasos de la técnica explicados en la sección 1.1.4 y sea capaz de reconocer que situaciones de prueba han sido ejecutadas y cuales no.

También se buscará que el uso de esta herramienta sea lo más sencillo posible para el programador y que el informe sea útil para complementar las pruebas en caso de que haya situaciones de prueba no ejecutadas.

Capítulo 2

Requisitos y casos de uso

Al tener este TFG un objetivo muy específico, los requisitos del sistema son concretos y dirigidos a cumplir el objetivo.

2.1. Requisitos

- **RF1**: La herramienta permitirá medir la cobertura de profundidad 2 alcanzada a partir de un código fuente (el objeto de pruebas) durante la ejecución de un conjunto de test que prueba el objeto de pruebas.
  - **RF1.1**: La herramienta identificará automáticamente las situaciones de prueba de profundidad 2 del objeto de pruebas.
  - **RF1.2**: La herramienta registrará las situaciones de prueba de profundidad 2 cubiertas del objeto de pruebas al ejecutar los test.
- **RF2**: La herramienta proporcionará un informe sobre porcentaje de cobertura, situaciones de prueba cubiertas y situaciones de prueba no cubiertas al finalizar la ejecución de los test.
- **RF3**: La herramienta de medición de cobertura está integrada como plugin en un *IDE* (entorno de desarrollo integrado).

2.1.1. Aspectos fuera de ámbito

Definir si una situación de prueba es posible o no

Hay que tener en cuenta que existen situaciones de prueba imposibles, ya que la ejecución de dos aristas consecutivas en el grafo puede ser imposible por estar guardadas por condiciones dependientes entre sí e incompatibles. Se puede ver un ejemplo de esto en el Anexo A.1.

Este tipo de restricciones de caminos imposibles se deben tener en cuenta cuando se realiza la técnica de manera manual pero la herramienta no es capaz de reconocerlos, ya que requeriría de un análisis contextual sobre el grafo y no es el objetivo de este trabajo. Además, es un problema no decidible según Durelli et al [4]. En cambio la herramienta ofrece la posibilidad de indicarle si existe algún camino imposible y que los tenga en cuenta a la hora de calcular la cobertura.

Generar los caminos y casos de prueba

Esta herramienta está enfocada como un medidor de cobertura para verificar que la técnica de caminos de profundidad 2 se ha aplicado correctamente, por lo que, como otros medidores de cobertura, su propósito no es diseñar las pruebas sino medir la cobertura cuando se ejecutan las pruebas.

2.2. Casos de uso

En la figura 2.1 se puede ver el diagrama de casos de uso de la herramienta.

2.2.1. Actores y descripción de los casos de uso asociados

Desarrollador

Es la persona que desea verificar que las pruebas que ha implementado utilizando la técnica de caminos de profundidad 2 cubren todas las situaciones de prueba.

- **Medir cobertura de profundidad 2**: El desarrollador ejecuta sus pruebas indicando que quiere medir la cobertura de caminos de profundidad 2 y ver un informe al finalizar la ejecución de los test.

- **Instalar herramienta**: El desarrollador instala y configura la herramienta para poder medir cobertura de caminos de profundidad 2. Puede instalarla de manera manual o como un plugin de un IDE para facilitar su uso.

2.2.2. Diagrama de casos de uso

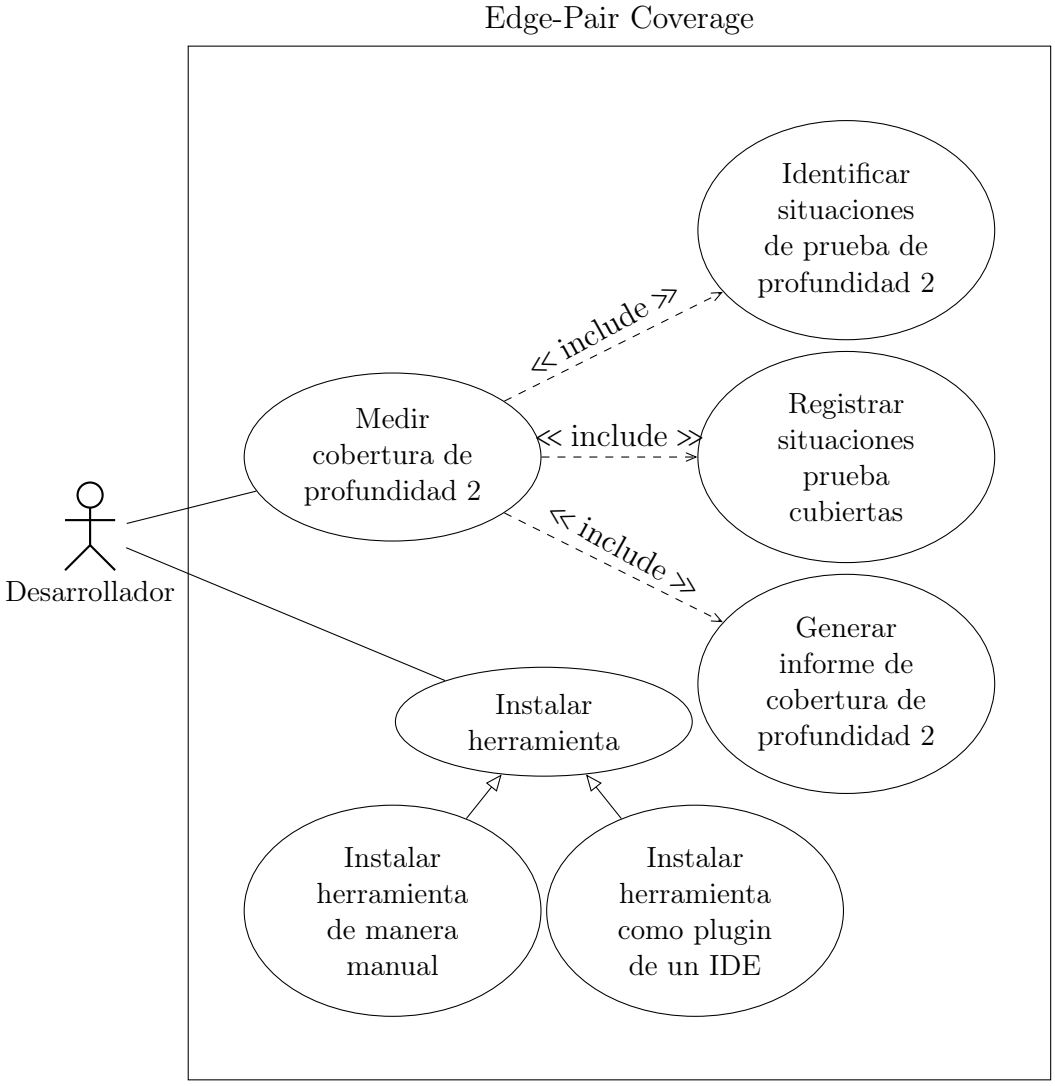

Figura 2.1: Diagrama de casos de uso de la herramienta Edge-Pair Coverage

Capítulo 3

Análisis

3.1. Estado de las herramientas de cobertura de código actuales

Se ha realizado un análisis de las herramientas de medición de cobertura de código para Java más populares [5] con varios objetivos:

- **Verificar que realmente no existe ninguna herramienta con este propósito**: A la técnica se le conoce por varios nombres (caminos de profundidad, *Edge-pair testing*, etc) y es posible que algunas herramientas de cobertura de código ya incluyan soporte para esta técnica pero con un nombre diferente.
- **Buscar una herramienta que pueda servir de punto de partida**: Implementar un medidor de cobertura puede no ser una tarea sencilla, y si existiese una herramienta de código abierto que permitiese la extensión de sus funcionalidades de manera sencilla podría suponer una reducción de riesgos a la hora de realizar el proyecto.
- **Conocer los mecanismos que utilizan**: Al no tener experiencia implementando medidores de cobertura puede ser útil conocer cuáles son los mecanismos y sus variaciones que utilizan las diferentes herramientas del mercado existentes.

En la tabla 3.1 se incluye un resumen de las diferentes herramientas: qué ofrecen, el mecanismo por el cual lo consiguen, si es similar a la herramienta que buscamos y si el código es accesible.

Herramienta	**Tipo cobertura**	**Mecanismo**	**¿Parecido?**	**¿Código accesible?**	**URL**
Cobertura	Básico	Instrumentación	No	GitHub público	`https://cobertura.github.io/cobertura/`
CodeCover	Básico y bucles	Instrumentación	No, pero ofrece bucles	Open source	`http://codecover.org/`
Emma	Básico	Instrumentación	No	Open source	`https://emma.sourceforge.net/`
Gretel	Líneas	Instrumentación	No	Sin actividad desde el 2002	`https://www.cs.uoregon.edu/research/perpetual/Software/Gretel`
Hansel	Gretel pero con JUnit 4				`https://hansel.sourceforge.net/`
JaCoCo	Básico	Instrumentación on-the-fly con agentes	No	GitHub público	`https://www.eclemma.org/jacoco/`
JCov	Básico	Instrumentación	No	GitHub público pero sin mucha actividad	`https://wiki.openjdk.org/display/CodeTools/jcov`
NoUnit	Cobertura estática				`https://nounit.sourceforge.net/`
Pitest	Mutation testing				`https://pitest.org/`
Quilt	Muy básico	Instrumentación	No	Sourceforge sin actividad desde 2003	`https://quilt.sourceforge.net/`
Serenity BDD	Test de aceptación y cobertura centrada en los test				`https://serenity-bdd.info/`
Atlassian Clover	Básico	Instrumentación sobre fuentes	No	Open source desde 2017	`https://www.atlassian.com/es/software/clover`
Qt Coco	Incluye *Modified Condition Decision Coverage* y *Multiple Condition Coverage*	Privado, de pago con licencia de uso			`https://www.qt.io/product/quality-assurance/coco`

Tabla 3.1: Comparación de herramientas de cobertura de código para Java

Algunos conceptos necesarios que se utilizan en la tabla 3.1 son:

- **Tipo cobertura**: se han definido dos grupos para simplificar la lectura:
  - **Muy básico**: líneas, instrucciones.
  - **Básico**: líneas, instrucciones, funciones, ramas.

  Además también se menciona *Modified Condition Decision Coverage* y *Multiple Condition Coverage*, dos tipos de cobertura más avanzada y exhaustiva [1] [2].

- **Instrumentación**: es el proceso de modificar software para poder realizar análisis sobre el, generalmente añadir instrucciones que permiten registrar su comportamiento dinámico (en tiempo de ejecución). Según el objeto de la instrumentación puede ser:
  - **Sobre fuentes**: se modifica el código fuente antes de compilarlo.
  - **Sobre ejecutables**: se modifica el ejecutable. Normalmente se utiliza instrumentación *on-the-fly* que permite modificar el ejecutable en tiempo de ejecución pero justo antes lanzar el programa. Por ejemplo, en Java se utilizan agentes que interceptan los *.class* antes de cargarlos en la máquina virtual de Java. Este aspecto se analiza con más detalle en la sección 4.2.1.

En este análisis no se ha encontrado ninguna herramienta existente y popular que ofrezca lo que se plantea en este TFG. En el contexto académico, lo más similar que se ha encontrado es la herramienta *BA-CF* que proponen y utilizan Matheus Silva y Marcos Chaim [6]. Sin embargo, esta herramienta tiene como entrada las situaciones de prueba, mientras que, los objetivos planteados en este TFG, se plantean determinar las situaciones de prueba a partir del código fuente.

3.2. Metodología

Se va a utilizar un ciclo de vida iterativo-incremental con las siguientes fases:

- **Primera versión funcional**: a modo de estudio de viabilidad, en ella se verificara que los siguientes procesos se pueden realizar:
  - Calcular un grafo de flujo a partir de un archivo fuente de Java.
  - Identificar todos los posibles caminos de profundidad 2 a partir de su grafo de flujo.

- Instrumentar un código Java identificando las expresiones condicionales (nodos predicado) y añadiendo código antes de las propias condiciones y al comienzo de las diferentes ramas.
- Utilizar la información generada sobre los posibles caminos de profundidad 2 para que la instrumentación genere información acerca de los caminos de profundidad 2 visitados.
- Generar un informe legible por un humano acerca de la cobertura de caminos de profundidad 2.

– **Diferentes iteraciones para mejorar la usabilidad**: en ellas se buscará:

- El informe muestra información basada en el **código fuente** de los caminos que no se han visitado, para poder identificarlos y mejorar los test.
- Se automatiza todo el proceso con una configuración de un ***runner*** (archivo de configuración que permite definir algunas opciones de ejecución, como opciones de la máquina virtual de Java, variables de entorno, etc.) de un IDE (se valora *IntelliJ*[2] como primera opción, este es un IDE para Java y otros lenguajes basados en la JVM de la empresa JetBrains).
- Toda la herramienta se transforma en un **plugin de un IDE** para poder ejecutarlo con una configuración muy simple.

[2] `https://www.jetbrains.com/es-es/idea/`

Capítulo 4

Diseño e implementación

Esta sección está organizada en:

- **Diseño algorítmico de la solución**: donde se define el diseño algorítmico de la solución a alto nivel.
- **Diseño e implementación de un agente en Java**: donde se explican las decisiones de diseño tomadas, pseudocódigo de los aspectos más importantes de la implementación y las dificultades encontradas.
- **Diseño e implementación de un plugin para IntelliJ**: donde se explica brevemente como se implementa un plugin para IntelliJ, se muestra el diseño de la interfaz y un breve resumen del funcionamiento.

4.1. Diseño algorítmico de la solución

De forma análoga a como se realiza la técnica de manera manual, la herramienta debe seguir los pasos de la técnica, como se ve en la figura 4.1, para verificar que durante la ejecución del programa objeto de pruebas o sus test se ejecutan todos los pares de aristas adyacentes:

- **Obtener el grafo de flujo asociado a ese programa**: Este proceso requiere de un análisis léxico-sintáctico del programa y se puede realizar a partir del fuente (la forma en la que se aplica la técnica de manera manual) o a partir del ejecutable. En este caso es más favorable realizar el análisis a partir del ejecutable ya que el bytecode de Java es de un nivel más bajo de abstracción y por lo tanto es más fácil analizar su estructura.
- **Identificar los pares de aristas (situaciones de prueba)**: A partir del grafo creado en el paso anterior, se debe identificar todos los conjuntos de pares de

aristas adyacentes, es decir, todas las situaciones de prueba a verificar que se ejecutan.

- **Instrumentar el ejecutable Java**: Se debe modificar el ejecutable Java para que, en tiempo de ejecución, se registre las aristas ejecutadas para poder mostrarlo en el informe.

- **Generar un informe al finalizar la ejecución**: Al terminar de ejecutarse el programa, se le mostrará al usuario un informe que indicará cuantos pares de aristas se han ejecutado y cuales faltan por ejecutarse.

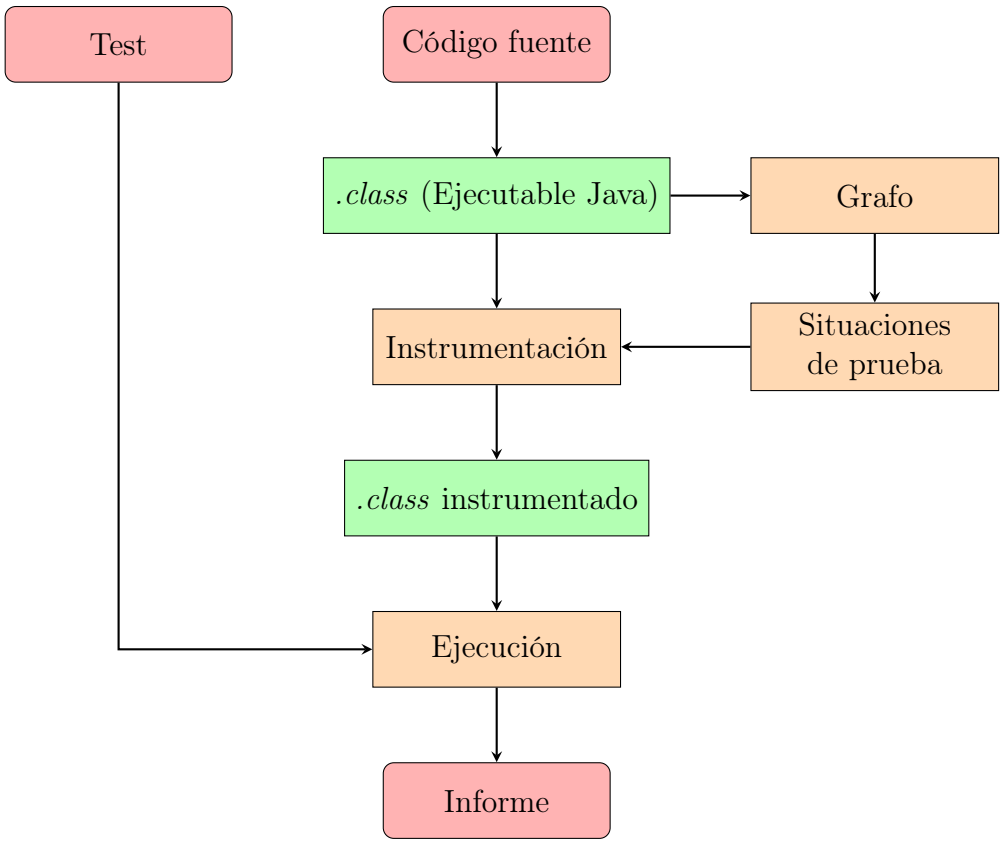

Figura 4.1: Diagrama de flujo de la solución propuesta

4.2. Diseño e implementación de un agente en Java

La implementación del agente se puede encontrar en el siguiente repositorio público de GitHub: `https://github.com/juan-catalan/tfg`

En el Anexo C.1 se puede ver una explicación de la arquitectura del agente.

4.2.1. Análisis de herramientas de instrumentación de código Java

Las dos herramientas más usadas para instrumentación de código Java son Javassist[3] y ASM[4].

Javassist es, de las dos, la más sencilla de utilizar y con mayor grado de abstracción, por lo que se realizaron unas pruebas de viabilidad para verificar si el proyecto era factible con esta herramienta. Los resultados no fueron favorables debido a:

- El **core** de su API esta en un grado de abstracción mayor al necesario: permite modificar y crear clases pero basado en Java y no en el bytecode.
- Posee una API para trabajar a nivel de bytecode (el nivel de abstracción necesario para este proyecto), pero la documentación y trabajos de la comunidad con esa API son escasos.

Tras estos insatisfactorios resultados se optó por realizar unas pruebas de viabilidad similares pero con la herramienta ASM. Estos resultados fueron completamente satisfactorios debido a:

- **Es la herramienta de instrumentalización utilizada por otros medidores de cobertura** como Cobertura o Jacoco: esto es un indicio positivo ya que implica que la herramienta ha sido utilizada para un propósito similar al nuestro.
- **API centrada en manipulación de bytecode**: este exactamente el grado de abstracción necesario para este proyecto.
- **Muy buena documentación**: la herramienta cuenta con una guía de usuario de 150 páginas donde explican todas las posibilidades de la herramienta a través de ejemplos y diagramas, de una manera muy organizada. Con esta guía de usuario se consiguió realizar las pruebas de viabilidad necesarias para verificar que se podía llevar a cabo el proyecto:
  - Identificar nodos predicado de un *.class*.

[3] `https://www.javassist.org/`
[4] `https://asm.ow2.io/`

- Manipular el bytecode de un *.class* añadiendo código antes y despues de cada nodo predicado.

4.2.2. Agentes en Java e instrumentación

Los agentes en Java son un tipo especial de clase, que utilizando la API de Java Instrumentation[5], pueden interceptar programas ejecutándose en la máquina virtual de Java (*JVM*) y modificar su bytecode. El bytecode es, de manera simplificada, el ensamblador de la JVM. En el Anexo B se puede ver una breve explicación de las operaciones bytecode.

Los agentes de Java implementan instrumentación sobre ejecutables *on-the-fly* (en tiempo de ejecución). En concreto, los agentes en Java pueden realizar la instrumentación:

- **Antes de cargar las clases en la máquina virtual de Java**: se añaden como parámetro al lanzar el programa Java utilizando la opción: `-javaagent: path-to-agent.jar`.
- **Durante la ejecución del programa en la máquina virtual de Java**: permiten enlazarse a una máquina virtual de Java durante la ejecución del programa.

Para nuestro objetivo la opción más conveniente es la primera: un agente que instrumente el código antes de cargarlo en la JVM, ya que al estar enfocado a utilizarse junto a los test interesa que la instrumentación este presente desde el comienzo para obtener toda la información posible.

4.2.3. Implementación de un agente en Java utilizando la librería ASM

Para implementar un agente de este tipo en Java hace falta, entre otras cosas, una clase que implemente la interfaz `ClassFileTransformer`: Esta interfaz posee un método `transform`, que es invocado por la JVM por cada clase a cargar en ella. Como parámetro este método recibe los bytes que definen todo el *.class* y devuelve los bytes resultado de haber modificado el *.class*.

Aquí es donde la librería ASM facilita esta tarea permitiendo la gestión del bytecode a través de una API: leer atributos de la clase, leer sus métodos: obtener sus instrucciones bytecode, añadir instrucciones bytecode en cualquier punto, etc.

[5] `https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/package-summary.html`

En la figura 4.1, los procesos indicados en color naranja (a excepción de la ejecución) son aquellos realizados por el agente:

1. Analizar el bytecode para obtener un grafo de control de los objetos de prueba.

   En la figura 4.2 se puede ver el pseudocódigo del algoritmo de análisis de bytecode para generar un grafo de control de flujo.

```
function ANALYZE(insn)        ▷ Siendo insn un conjunto de instrucciones bytecode
    for all i ∈ insn do
        if i is first instruction then
            Añadir i como vértice
            n ← nextPredicateOrEndNode(i)
            Añadir arista entre i y n
        else if i is predicate node then
            Añadir i como vértice
            nTrue ← nextPredicateOrEndNodeEvaluatingTrue(i)
            Añadir arista entre i y nTrue
            nFalse ← nextPredicateOrEndNodeEvaluatingFalse(i)
            Añadir arista entre i y nFalse
    return graph
```

Figura 4.2: Pseudocódigo del algoritmo de análisis de flujo de bytecode

2. Identificar las situaciones de prueba a partir del grafo.

   En la figura 4.3 se puede ver el pseudocódigo del algoritmo de obtención de situaciones de prueba a partir de un grafo de control de flujo.

```
function OBTAINTESTSITUATIONS(graph)
    testSituations ← ∅
    for all v ∈ graph.vertices do
        for all vIn ∈ v.verticesEntradas do
            for all vOut ∈ v.verticesSalida do
                new ← TestSituation(vIn, v, vOut)
                testSituations ← testSituations ∪ new
    return testSituations
```

Figura 4.3: Pseudocódigo del algoritmo de obtención de situaciones de prueba

3. Instrumentar los diferentes objetos de prueba para registrar la situaciones de prueba cubiertas.

   En la figura 4.4 se puede ver el el código fuente de un método muy sencillo llamado *esPar*. Al realizar los pasos anteriores sobre él se han identificado los nodos relevantes, en este caso el nodo 1 es un nodo predicado, el nodo 2 es un

nodo final que se alcanza si se evalúa como verdadero el nodo 1, y el nodo 3 es un nodo final que se alcanza si se evalúa como falso el nodo 1.

La idea es, que tras identificar los nodos y las situaciones de prueba, modificar el objeto de pruebas añadiendo instrucciones para registrar su ejecución. Un ejemplo del resultado de esta modificación (simplificada) se puede ver en la figura 4.5.

```
boolean esPar(Integer i){
    // Nodo 1
    if  (i%2 == 0){
        // Nodo 2
        return true;
    }
    else {
        // Nodo 3
        return false;
    }
}
```

Figura 4.4: Código fuente del método *esPar* antes de la instrumentación

```
boolean esParInstrumentado(Integer i){
    registrarNodo(1);
    if  (i%2 == 0){
        registrarArista(
            EdgeType.TRUE);

        registrarNodo(2);
        return true;
    }
    else {
        registrarArista(
            EdgeType.FALSE);

        registrarNodo(3);
        return false;
    }
}
```

Figura 4.5: Ejemplificación como código Java del método *esPar* después de la instrumentación

4.2.4. Dificultades encontradas

Al estar trabajando a nivel de bytecode, en vez de código fuente en Java, ocurren algunas discrepancias o aspectos a tener en cuenta:

Asignaciones booleanas

Las asignaciones booleanas de Java como:

```
boolean esPar = (n%2 == 0)
```

Figura 4.6: Ejemplo de asignación booleana en Java

se compilan al bytecode con una estructura más similar a esta:

```
boolean esPar;
if (n%2 == 0) esPar = true;
else esPar = false;
```

Figura 4.7: Ejemplificación como código Java de una asignación booleana en bytecode

La diferencia es que en la versión bytecode se genera un nodo predicado (que no existía en la versión Java) por cada asignación booleana.

Esto puede ser un problema, ya que al generar el grafo de manera manual no se tiene en cuenta estas asignaciones booleanas como nodos predicado, lo que provoca que las situaciones de prueba que se producen al aplicar la técnica sean menos que los que genera la herramienta automáticamente y por lo tanto la cobertura no sea la esperada.

Para solucionar esto se ha implementado una función que identifique en el bytecode la estructura de las asignaciones booleanas. No obstante, existe el pequeño inconveniente que el bytecode generado por el compilador es indistinguible en los dos casos (figuras 4.6 y 4.7), por lo que al final la herramienta de medición de cobertura no puede determinar si el código original contenía o no una instrucción condicional (y por lo tanto, debe tener en cuenta la existencia de un nodo predicado adicional).

Por este motivo y por si alguien quisiera aplicar la técnica entendiendo las asignaciones booleanas como nodos predicado (lo cual hace la técnica aún mas exhaustiva) la herramienta permite configurar si debe entender las asignaciones booleanas como nodos predicado o no.

Implementación de las estructuras condicionales y los operadores lógicos *and* y *or* en bytecode

A nivel de bytecode las estructuras condicionales (`if`) se implementan con saltos condicionales entre etiquetas, pero estos saltos pueden ser implementados por el compilador de diferentes maneras. Por ejemplo, en una estructura (`if {...} else {...}`), dos posibilidades (no exhaustivas) de como implementarla en bytecode serían:

1. Evaluar la condición del `if` y si resulta ser *false* saltar a la etiqueta del código del `else`.

2. Evaluar la negación de la condición del `if` y si resulta ser *true* saltar a la etiqueta del código del `else`.

Esto puede generar discrepancias en la representación del grafo respecto a como se generaría el grafo a partir del código fuente. En el caso 2 de los ejemplos anteriores:

- Según código fuente: desde el nodo `if` se va a la instrucción `else` al evaluarse como *false*.

- Según bytecode: desde el nodo `if` se va a la instrucción `else` al evaluarse (la condición inversa) como *true*.

En el caso de *javac*[6], el compilador estándar de Oracle y el utilizado por defecto por IntelliJ, se ha identificado como implementan las estructuras de control básicas para evitar discrepancias respecto al código fuente.

Estas discrepancias aumentan al utilizarse operados lógicos *and* y *or* en las estructuras condicionales, ya que las diferentes maneras de implementarlas aumentan y no se ha conseguido descubrir una manera de identificar, a partir de bytecode, el tipo de operadores lógicos que existían en el código fuente; y por lo tanto algunas aristas (resultado de operadores lógicos) pueden tener una etiqueta discrepante respecto al código fuente. Un ejemplo se puede ver en `https://github.com/juan-catalan/tfg/issues/1`.

Como la discrepancia es meramente visual (ya que el cálculo de cobertura se realiza correctamente), se ha decidido no tener en cuenta este problema por el momento.

4.3. Implementación de un plugin de integración del agente para el IDE IntelliJ

La implementación del plugin para IntelliJ se puede encontrar en el siguiente repositorio público de GitHub: `https://github.com/juan-catalan/tfg_intellij_plugin`

En el Anexo C.1 se puede ver una explicación de la arquitectura del plugin.

4.3.1. Contexto sobre *IntelliJ Platform Plugin SDK*

IntelliJ Platform[7] es la plataforma de código abierto utilizada y creada por JetBrains para desarrollar IDEs, también es la plataforma usada por Google para el desarrollo de Android Studio.

Es un entorno de aplicaciones basadas en la JVM, basada en componentes, con un conjunto de herramientas de interfaz de usuario de alto nivel para crear paneles de herramientas, vistas de árbol y listas, así como menús emergentes y cuadros de diálogo.

Dentro de la IntelliJ Platform se encuentra *Intellij Platform Plugin SDK* que facilita la creación de plugins para IDEs basados en la IntelliJ Platform.

Entre las funcionalidades y componentes que ofrece este SDK, los que se han utilizado en el proyecto son: los componentes UI basados en *Swing*[8] para mantener la consistencia entre la UI del IDE, el sistema de *actions* que permite añadir botones en diferentes secciones del IDE y asociarles funcionalidad (como se ve en la figura 4.9), el

[6] `https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javac.html`

[7] `https://plugins.jetbrains.com/docs/intellij/welcome.html`

[8] Librería gráfica para crear GUI en Java: `https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html`)

sistema para añadir una ventana de ajustes personalizable (como se ve en la figura 4.8), el sistema para acceder al sistema de ficheros, el sistema para acceder al analizador sintáctico y semántico del código fuente y el sistema de control de procesos del IDE.

4.3.2. Diseño de la interfaz y funcionamiento

Menú de ajustes del plugin

En la figura 4.8 se puede ver el menú de ajustes del plugin, este te permite decidir el tipo de informe (nativo como en la figura 4.12 o HTML como en la figura 4.13) y decidir si la herramienta debe entender las asignaciones booleanas como nodos predicado.

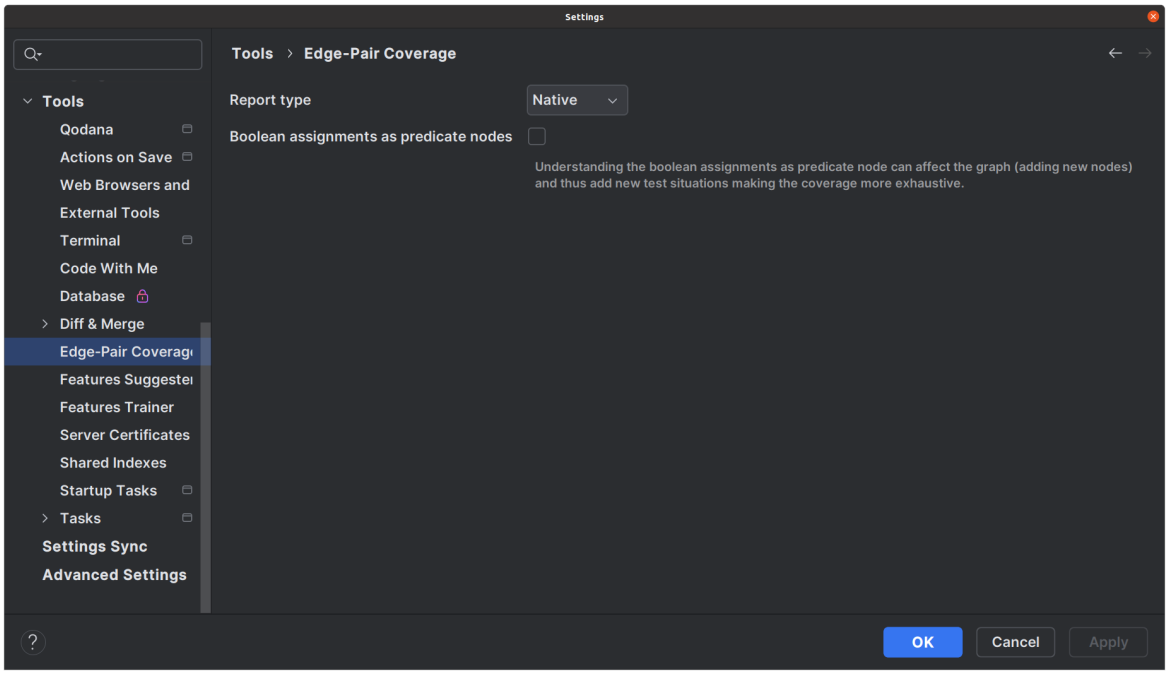

Figura 4.8: Menú de ajustes del plugin

Botón de ejecución con cobertura de profundidad 2

En la figura 4.9 se puede ver el botón para ejecutar un conjunto de pruebas midiendo la cobertura de profundidad 2, este botón se encuentra en el menú "Run" del IDE. Cuando el usuario pulsa en él se muestran las ventanas de selección de métodos y situaciones imposibles (figuras 4.10 y 4.11) y tras finalizar la selección se ejecuta el conjunto de pruebas añadiendo el agente (y sus parámetros) como parámetro de la JVM.

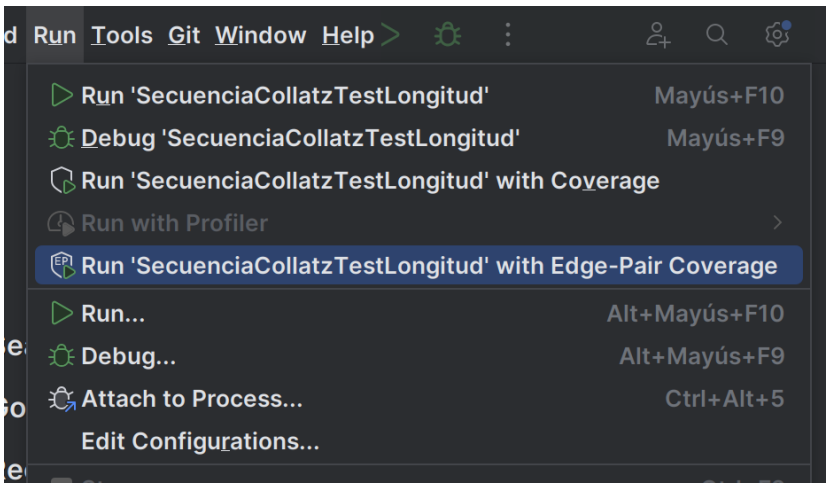

Figura 4.9: Botón "Run with Edge-Pair Coverage" en el menú "Run"

Ventana de selección de métodos y situaciones imposibles

En la figura 4.10 se puede ver la ventana de selección de métodos a medir la cobertura. Para obtener las clases y métodos a mostrar se utiliza el sistema de acceso a ficheros para buscar los ficheros Java del proyecto y luego utilizando el analizador sintáctico del IDE se obtienen las diferentes clases y métodos de los ficheros.

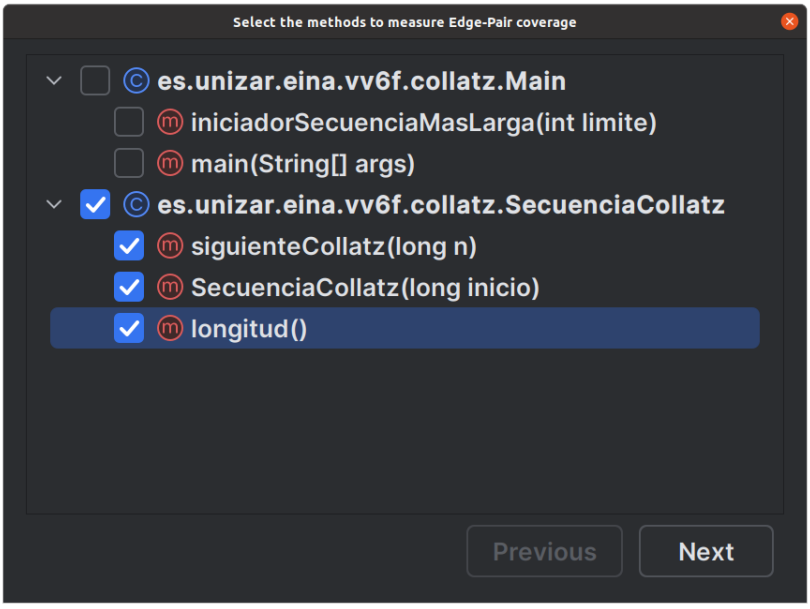

Figura 4.10: Ventana de selección de métodos a medir la cobertura

En la figura 4.11 se puede ver la ventana de selección de situaciones imposibles. Recibe los métodos seleccionados en la figura 4.10 y permite al usuario indicar el número de situaciones imposibles para cada método.

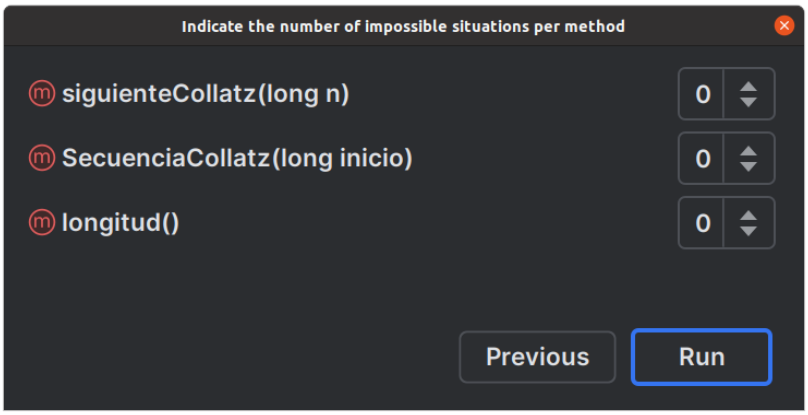

Figura 4.11: Ventana de selección de situaciones imposibles de los métodos seleccionados

Informes de cobertura de profundidad 2

Como se ha indicado en los ajustes del plugin, existen dos tipos de informes: uno nativo y otro HTML. Al finalizar la ejecución de un conjunto de pruebas (si se ha seleccionado medir la cobertura de profundidad 2) se abre automáticamente el tipo de informe seleccionado en los ajustes.

En la figura 4.12 se puede ver el informe nativo que busca integrarse mejor con la estética del IDE.

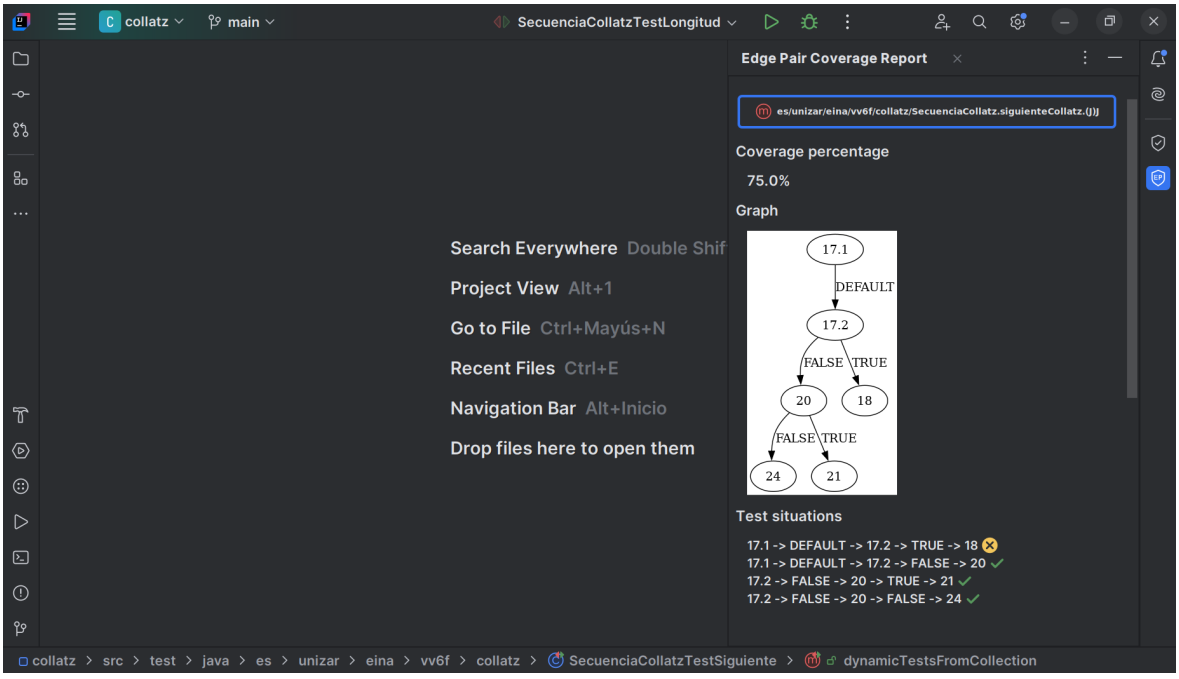

Figura 4.12: Informe de cobertura de profundidad 2 basado en Swing

En la figura 4.13 se puede ver el informe HTML que busca ser más visual y atractivo.

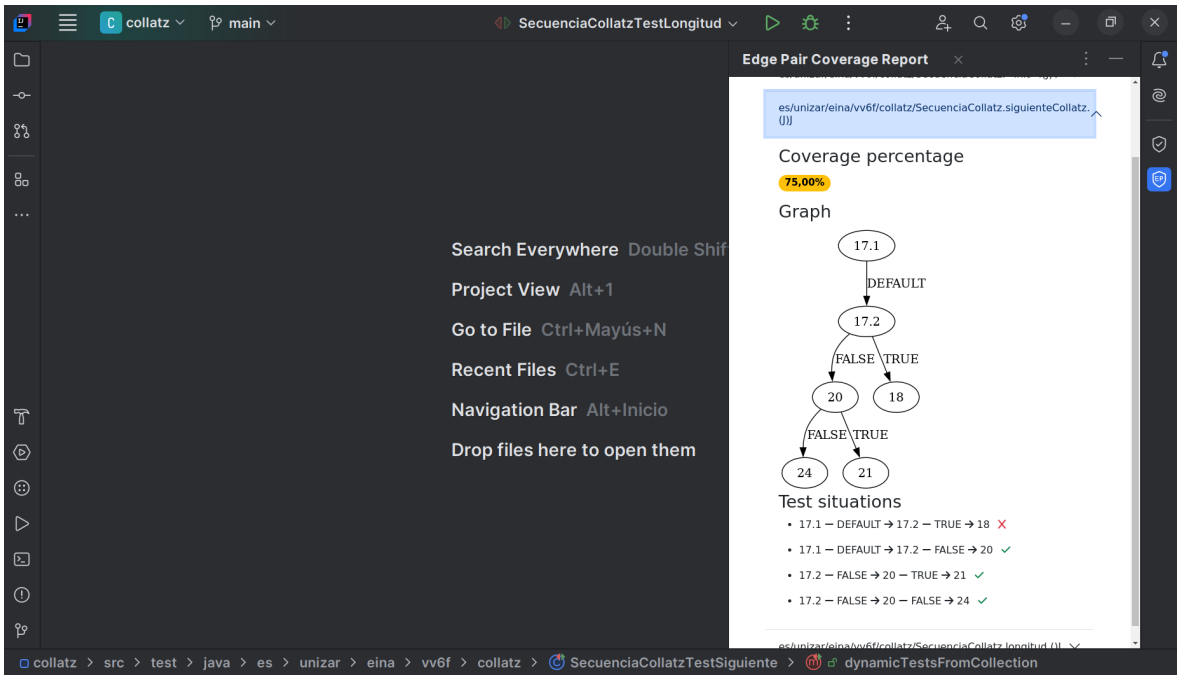

Figura 4.13: Informe de cobertura de profundidad 2 utilizando HTML

Capítulo 5

Validación

Para validar la corrección de la herramienta nos hemos centrado en tres aspectos:

- La corrección de los grafos generados a partir de un *.class*.
- La identificación de las situaciones de prueba a partir de un *grafo*.
- El correcto registro por parte de la instrumentación de los nodos y caminos recorridos.

Para facilitar la validación se cuenta con el material y exámenes resueltos de la asignatura *Verificación y Validación*, que propone ejercicios donde aplicar la técnica y sus soluciones. De esta manera se puede verificar que la herramienta es correcta utilizando casos con una solución externa a este desarrollo.

5.1. Corrección de los grafos

5.1.1. Diseño

Como parte del material existen problemas de examen donde, a partir de un código en Java, se debe obtener el grafo de flujo. La idea es comprobar que el grafo que genera la herramienta es isomorfo del grafo esperado (la solución del examen, calculada a mano).

A continuación se indican los diferentes casos de prueba con alguna información sobre el objeto de pruebas en el problema de examen, como el número de líneas (LOC por sus siglas en inglés) o sobre el grafo de flujo asociado, como el número de vértices y aristas:

Curso	Convocatoria	Nombre del método	LOC	Nº vértices	Nº aristas
2014-2015	2ª	*imagen*	17	6	7
2015-2016	1ª	*esPrimo*	20	7	11
2016-2017	1ª	*buscar*	11	6	7
2018-2019	1ª	*buscar*	21	8	9
2019-2020	1ª	*minimo*	15	8	9
2021-2022	1ª	*replaceDigits*	12	5	7
2022-2023	1ª	*calcularPuntuacion*	15	5	7

Tabla 5.1: Casos de prueba para validar la corrección de los grafos

5.1.2. Implementación

Implementación de los casos de prueba

`https://github.com/juan-catalan/tfg/blob/master/src/test/java/org/juancatalan/edgepaircoverage/GetControlFlowGraphTest.java`

Implementación de mecanismo de verificación de isomorfismos

Para comprobar isomorfismos entre grafos existe un módulo en la biblioteca utilizada para la representación de los grafos de flujo JGraphT[9]. El problema con este módulo es que únicamente funciona para grafos simples, es decir, grafos no dirigidos que no contengan ciclos o múltiples aristas entre dos vértices. Los grafos utilizados en el proyecto no son grafos simples sino pseudografos dirigidos ya que para expresar un grafo de flujo es necesario que:

- Sean grafos dirigidos.
- Permitan múltiples aristas entre dos vértices (ya que puedes ir de un nodo predicado a otro por dos caminos distintos).
- Permitan ciclos para implementar los bucles.

Como utilizar este módulo no era viable y tampoco podemos utilizar otro tipo de grafo, se decidió implementar un algoritmo de verificación de isomorfismo para pseudografos dirigidos basado en matrices de adyacencia.

[9] `https://jgrapht.org/`

Hace falta una adaptación del concepto de matriz de adyacencia para este tipo de grafos, en concreto ha hecho falta:

- Expresar la dirección de las aristas
- Expresar múltiples aristas entre nodos
- Expresar los diferentes tipos de aristas

Para conseguir esto en las filas están representados los vértices origen, en las columnas los vértices destino y en las intersecciones se define un conjunto de valores (los diferentes tipos de aristas) que indica la posible conexión entre el vértice origen y el vértice destino.

Ejemplo de matriz de adyacencia Dado el siguiente fragmento de código en Java donde se especifica una función de nombre *paresHasta*:

```
1     public static List<Integer> paresHasta(int max){
2         List<Integer> listaPares = new ArrayList<>();
3         for (int i = 0; i<=max; i++){
4             if (i % 2 == 0){
5                 listaPares.add(i);
6             }
7         }
8         return listaPares;
9     }
```

Figura 5.1: Código Java de la función *paresHasta*

Su grafo de flujo asociado, donde cada vértice es un nodo que se encuentra en la línea del código que indica su etiqueta, sería:

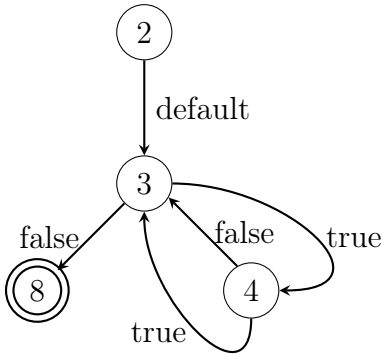

Figura 5.2: Grafo de flujo asociado al fragmento de código Java 5.1

Este grafo se puede representar como la matriz de adyacencia de la figura 5.3. En esta matriz se definen los vértices y las aristas entre ellos, por ejemplo la celda marcada en azul indica que desde el **vértice 4** puedes alcanzar el **vértice 3** desde 2 aristas diferentes:

- Una al evaluarse la condición como **true**
- Otra al evaluarse la condición como **false**

Vértices origen \ Vértices destino	2	3	4	8
2	[]	[DEFAULT]	[]	[]
3	[]	[]	[TRUE]	[FALSE]
4	[]	$[TRUE, FALSE]$	[]	[]
8	[]	[]	[]	[]

Figura 5.3: Matriz de adyacencia asociada al grafo de la figura 5.2

Mecanismo de verificación de isomorfismo entre matrices de adyacencia

Con esta representación como matriz de adyacencia se puede calcular si un grafo es isomorfo de otro de manera sencilla (conceptualmente, no a nivel computacional) ya que un grafo x es isomorfo de otro grafo y si el grafo x es una permutación del grafo y.

Por ejemplo, el grafo de la figura 5.4 se puede ver visualmente que es isomorfo del grafo de la figura 5.2 pues es el resultado de "intercambiar" los nodos 2 y 8.

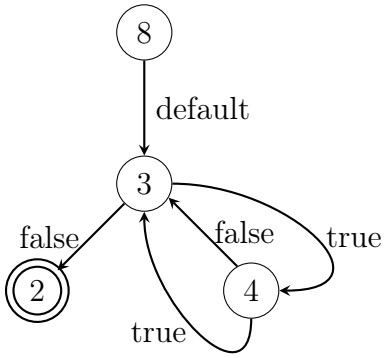

Figura 5.4: Grafo de flujo isomorfo del grafo de flujo de la figura 5.2

En la figura 5.5 podemos ver la matriz de adyacencia asociada al grafo 5.4. En ella se puede observar el isomorfismo con la matriz de adyacencia de la figura 5.3 ya que es el resultado de permutar los vértices 2 y 8.

Vértices origen	Vértices destino: 2	3	4	8
2	[]	[]	[]	[]
3	[FALSE]	[]	[TRUE]	[]
4	[]	[TRUE, FALSE]	[]	[]
8	[]	[DEFAULT]	[]	[]

Figura 5.5: Matriz de adyacencia isomorfa a la matriz de adyacencia de la figura 5.3

La implementación de este mecanismo de verificación de isomorfismos se puede encontrar en: `https://github.com/juan-catalan/tfg/blob/master/src/test/java/org/juancatalan/edgepaircoverage/utils/AdjacencyMatrix.java`

5.1.3. Resultados

Se han encontrado 4 defectos en la clase `ControlFlowAnalyser`:

- Si en el código bytecode aparecían 2 `goto` seguidos, el grafo generado tenía un nodo de más (el asociado al segundo `goto`).
- No se identificaban las instrucciones bytecode `ifnull` y `ifnonull` como nodos predicado.
- Si el código fuente tenía una asignación booleana el grafo generado tenía nodos de más (relacionado con lo descrito en la sección 4.2.4).
- Si el código fuente tenía una asignación booleana con puertas lógicas anidadas el grafo generado tenía nodos de más (problema relacionado con la solución del defecto anterior).

Finalmente todos los test se ejecutan y pasan sin ningún error.

5.2. Identificación de las situaciones de prueba a partir de un grafo

5.2.1. Diseño

Como cuando se aplica de forma manual la técnica, se debe obtener las situaciones de prueba de profundidad 2 a partir del grafo. Este es un proceso más trivial que los otros pero se ha decidido diseñar pruebas para ello, enfocado más como herramienta de regresión que como forma de encontrar defectos actuales.

Se han utilizado los problemas de examen donde un apartado es obtener las situaciones de prueba. En concreto se han implementado los siguientes test:

Curso	**Convocatoria**	**Nombre del método**	**Número de situaciones de prueba**
2014-2015	2ª	*imagen*	10
2015-2016	1ª	*esPrimo*	12
2016-2017	1ª	*buscar*	10
2018-2019	1ª	*buscar*	12
2019-2020	1ª	*minimo*	11
2021-2022	1ª	*replaceDigits*	12
2022-2023	1ª	*calcularPuntuacion*	12

Tabla 5.2: Casos de prueba para validar la identificación de situaciones de prueba

5.2.2. Implementación

`https://github.com/juan-catalan/tfg/blob/master/src/test/java/org/juancatalan/edgepaircoverage/ObtenerSituacionesPruebaTest.java`

5.2.3. Resultados

No se ha encontrado ningún defecto.

5.3. Correcto registro de los nodos y caminos recorridos.

5.3.1. Diseño

Como parte del material existen ejercicios de examen donde tras obtener el grafo de flujo de un código en Java se definen los diferentes caminos de profundidad 2 que cubren todas las situaciones de prueba que existen y los diferentes casos de prueba que cubren todos estos caminos.

Con estos casos de prueba (que técnicamente cubren todas las situaciones de prueba posible) debemos verificar que la cobertura de caminos que ofrece la herramienta al ejecutarlos también es del 100 % en cada uno de ellos.

Los casos a probar son:

Curso	Convocatoria	Nombre del método	Número de casos de prueba necesarios para cobertura completa
2014-2015	2ª	*imagen*	3
2015-2016	1ª	*esPrimo*	5
2016-2017	1ª	*buscar*	3
2018-2019	1ª	*buscar*	4
2019-2020	1ª	*minimo*	5
2021-2022	1ª	*replaceDigits*	3
2022-2023	1ª	*calcularPuntuacion*	5

Tabla 5.3: Casos de prueba para validar el correcto registro de nodos y caminos recorridos

5.3.2. Implementación

`https://github.com/juan-catalan/tfg/blob/master/src/test/java/org/juancatalan/edgepaircoverage/CoverageMeasurementTest.java`

5.3.3. Resultados

Se han encontrado 2 defectos:

- Si el código bytecode comenzaba con una instrucción condicional que implementaba un bucle, al volver al comienzo del bucle se registraba una visita al nodo inicial en vez de al primer nodo predicado.
- No se volvía a inicializar el camino recorrido actual si se lanzaba una excepción.

Finalmente todos los test se ejecutan y pasan sin ningún error.

Capítulo 6

Conclusiones

Este trabajo tenía como objetivo crear una herramienta para Java, fácil de instalar y utilizar, que automatice el proceso de medición de cobertura de pruebas diseñadas con la técnica de caminos de profundidad 2, ofreciendo un informe de cobertura de código tras la ejecución de las pruebas, que permita al desarrollador complementar las pruebas en caso de que haya situaciones de prueba no ejecutadas.

Para conseguir este objetivo se ha tenido que: establecer unos requisitos y casos de uso de la herramienta solicitada, realizar un análisis de las herramientas de medición de cobertura ya existentes, definir una metodología de trabajo adecuada para el tipo de proyecto, realizar un análisis de herramientas de instrumentación y manipulación de bytecode para Java, diseñar e implementar una solución (un agente Java y un plugin para el IDE IntelliJ que lo integre en él), diseñar una estrategia de pruebas e implementar las pruebas (incluyendo la implementación de un mecanismo de verificación de isomorfismo basado en matrices de adyacencia).

Finalmente el resultado, un plugin para IntelliJ, se puede descargar o instalar en IntelliJ o Android Studio desde el siguiente enlace: `https://plugins.jetbrains.com/plugin/25103-edge-pair-coverage`

6.1. Gestión del proyecto

Como se ha indicado en la sección 3.2 se ha utilizado un ciclo de vida iterativo-incremental buscando como objetivo final convertir la herramienta en un plugin para un IDE.

Durante el proyecto se han ido abordando las diferentes fases y problemas de manera concisa y directa, y finalmente se ha conseguido completar todas las iteraciones propuestas para mejorar la usabilidad, incluyendo el desarrollo de un plugin para IntelliJ.

En la figura 6.1 se puede observar las horas dedicadas y su agrupación por tareas.

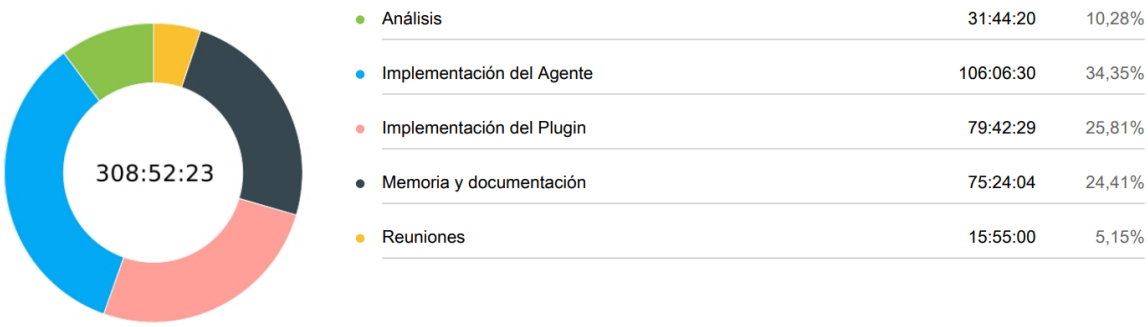

Figura 6.1: Informe con las horas dedicadas al proyecto dividido por tareas

6.2. Trabajo futuro

6.2.1. Extensión de la herramienta a otros lenguajes basados en la JVM

Uno de los posibles rumbos de trabajo futuro podría ser extender la herramienta de medición de cobertura de profundidad 2 a otros lenguajes basados en la JVM, ya que el agente que se encarga de todo el proceso de medición de cobertura trabaja a nivel de bytecode de la JVM y, sin haber realizado aún pruebas, se cree que funcionaría sin apenas modificaciones para otros lenguajes como Kotlin, Scala o Groovy.

6.2.2. Herramientas de soporte a la aplicación de la técnica

Otro posible rumbo sería, utilizando toda la lógica desarrollada para la herramienta, cambiar el enfoque y desarrollar otra herramienta para el soporte a la aplicación de la técnica, es decir que facilite poder aplicarla ofreciendo, por ejemplo, generación automática del grafo de control de flujo a partir de un archivo Java, generación de situaciones de prueba, generación de posibles caminos que cubran todas las situaciones de prueba, etc.

6.3. Reflexiones sobre la técnica de caminos de profundidad 2

Al estar trabajando y profundizando en la técnica de caminos de profundidad 2 han surgido principalmente dos reflexiones sobre la técnica:

La importancia de la definición de las situaciones de prueba En muchas de las diferentes descripciones de aplicación de la técnica no se especifica que las situaciones de prueba son únicamente aquel conjunto de pares de aristas que pasan por un nodo

predicado, sino todo el conjunto de pares de aristas. Esto provoca una gran cantidad de situaciones que no ofrecen ninguna información relevante y por tanto la cobertura resultada no ofrece realmente resultados concluyentes.

Por ejemplo, en una estructura como la de la derecha, si implementásemos un test que cubriera únicamente el caso de evaluar el `if` como verdadero, al tener mucha instrucciones y por tanto muchos nodos, daría una cobertura de caminos de profundidad 2 muy elevada a pesar de haber evaluado únicamente una rama del nodo predicado.

```
if (condition){
    // Muchas nodos no predicado
    // ...
    // Un nodo final
}
else {
    // Un nodo final
}
```

Creemos que las descripciones de la técnica deberían especificar de manera más clara el concepto de situación de prueba, ya que de la otra manera se pierde el enfoque de la técnica al incluirse muchas situaciones de prueba que no aportan información relevante.

La importancia de la definición de nodo predicado Incluso en las descripciones de aplicación de la técnica donde sí que definen las situaciones de prueba como el conjunto de pares de aristas que pasan por un nodo predicado, no especifican que se entiende por nodo predicado.

Como se ha mencionado en la sección 1.1.4 esta técnica es sensible a la definición de nodo predicado. Uno de los casos que más nos ha llamado la atención son las asignaciones booleanas, mencionado previamente en la sección 4.2.4, ya que dependiendo de si se interpretan como nodos predicado o no, pueden hacer aún más exhaustiva la técnica o se puede utilizar para falsear la técnica reduciendo significativamente el número de situaciones de prueba en presencia de código con muchos operadores lógicos *and* y *or*.

Por ejemplo en la figura 6.2 se puede ver un código que incluye un `if` con varios operadores lógicos y a su lado el grafo de control de flujo asociado, que tiene un total de 8 situaciones de prueba de caminos de profundidad 2.

```
// ini
if ((cond1 || cond2) &&
 ↪  cond3){
    // nf1
}
else {
    // nf2
}
```

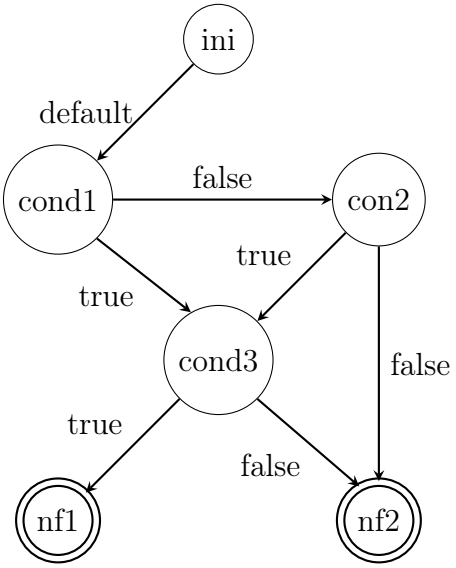

Figura 6.2: Ejemplo de código con múltiples operadores lógicos y su grafo de control

Si no entendemos las asignaciones booleanas como nodos predicado, podríamos modificar el código como se ha hecho en la figura 6.3 de manera que simplificamos el grafo, reduciendo el numero de situaciones de prueba a 2.

```
// ini
boolean cond = (cond1 ||
 ↪  cond2) && cond3;
if (cond){
    // nf1
}
else {
    // nf2
}
```

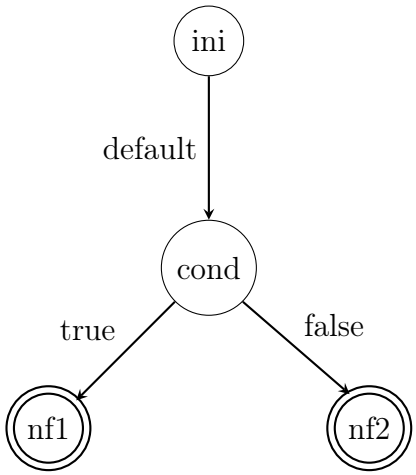

Figura 6.3: Código equivalente al de la figura 6.2 unificando múltiples operadores lógicos y su grafo de control

Este es un detalle que puede cambiar la exhaustividad de la técnica pero que fácilmente puede pasar desapercibido a quienes la van a aplicar. Por este motivo creemos que se debe tener en cuenta antes de aplicarla, decidiendo si, por el contexto del problema, vale la pena buscar ese extra de exhaustividad o si por el contrario puede aumentar el número de casos sin aportar beneficios.

A raíz de esto también se plantea como posible trabajo futuro, más relacionado con la investigación académica, un estudio para demostrar de manera empírica si existen beneficios notables (en este caso número de defectos encontrados) al entender las asignaciones booleanas como nodos predicados, aumentando de esta manera las situaciones de prueba.

6.4. Valoración personal

Este TFG ha sido una experiencia divertida y enriquecedora, ya que se trataba de un tipo de proyecto fuera de lo que estaba acostumbrado a realizar en la carrera (jamás me habría imaginado implementando un medidor de cobertura) y que me ha hecho adentrarme un poco en el mundo de la instrumentación de código Java y la manipulación de código bytecode.

Como reto ha sido entretenido y enriquecedor puesto que no había una solución directa y sencilla por lo que he tenido que leer mucha documentación y plantear muchas ideas para descubrir como se podía implementar lo que teníamos en mente.

El resultado final es, al menos para mí, muy satisfactorio ya que se ha conseguido implementar todo lo que se propuso al comienzo sin saber muy bien hasta donde podríamos llegar.

Bibliografía

[1] Tim Koomen, Leo van der Aalst, Bart Broekman, and Michiel Vroon. TMap Next, for result-driven testing. UTN Publishers, 2006.

[2] ISO/IEC/IEEE International Standard. Software and systems engineering: Software testing: Part 4: Test techniques. ISO/IEC/IEEE 29119-4:2015, pages 1–149, December 2015.

[3] Glenford J. Myers, Corey Sandler, Tom Badgett, and Todd M. Thomas. The Art of Software Testing, Second Edition. Wiley, June 2004.

[4] Vinicius H.S. Durelli, Marcio E. Delamaro, and Jeff Offutt. An experimental comparison of edge, edge-pair, and prime path criteria. Science of Computer Programming, 152:99–115, January 2018.

[5] Alexandra Altvater. Code Coverage Tools: 25 Tools for Testing in C, C++, Java — stackify.com. `https://stackify.com/code-coverage-tools/`, April 2024.

[6] Matheus Silva and Marcos Chaim. Bit-wise all edge-pairs coverage. In Proceedings of the XXXVII Brazilian Symposium on Software Engineering, SBES '23, page 267–276, New York, NY, USA, 2023. Association for Computing Machinery.

Lista de Figuras

Lista de Tablas

Anexos

Anexo A

Ejemplos de conceptos basados en código fuente

A.1. Ejemplo de situaciones imposibles en la técnica de caminos de profundidad 2

Dado el siguiente fragmento de código en Java donde se especifica una función de nombre *imagen*:

```
1     public static int imagen(int n) {
2         boolean negativo = n < 0;
3         if (negativo) {
4             n = -n;
5         }
6
7         int imagenEspecular = 0;
8         while (n != 0) {
9             imagenEspecular = 10 * imagenEspecular + n % 10;
10            n = n / 10;
11        }
12
13        if (negativo) {
14            return -imagenEspecular;
15        }
16        else {
17            return imagenEspecular;
18        }
19    }
```

Figura A.1: Código Java de la función *imagen*

Jamás se va a poder evaluar como *true* la condición de la linea 3: `if (negativo){...}` y como *false* la condición de la linea 8: `while (n != 0){...}` ya que n no puede ser menor estricto que 0 e igual que 0 al mismo tiempo.

A.2. Ejemplo de medición de cobertura de código

A.2.1. Objeto de pruebas

Dado el siguiente método con nombre *buscar*, que va a ser nuestro objeto de pruebas:

```java
/**
 * Busca un dato determinado en un vector de enteros.
 *
 * @param vector
 *             - el vector en el que se busca el dato «datoBuscado». Debe
 *             ser distinto de null y tener al menos una componente.
 * @param datoBuscado
 *             - el dato que se quiere buscar en el vector «vector».
 *
 * @return Si en el vector «vector» hay un dato igual a «datoBuscado»,
 *         devuelve el índice de la componente en la que se encuentra.
 *         En caso contrario, devuelve -1;
 */
public static int buscar(int[] vector, int datoBuscado) {
    int i = 0;

    while (i < vector.length - 1 && vector[i] != datoBuscado) {
        i++;
    }

    if (vector[i] == datoBuscado) {
        return i;
    } else {
        return -1;
    }
}
```

Figura A.2: Código Java de la función *buscar*

Los diferentes tipos de cobertura que vamos a medir son **instrucciones**, **ramas** y **condiciones**. Vamos a identificar todas ellas en el código:

Instrucciones

Podemos encontrar 7 instrucciones:

1. `int i = 0;`
2. `while (i < vector.length - 1 && vector[i] != datoBuscado)`
3. `i++;`
4. `if (vector[i] == datoBuscado)`

5. `return i;`
6. `else`
7. `return -1;`

Ramas

Las ramas se entienden mejor si representamos el grafo de control de la función, cada arista de salida de los nodos que representan estructuras de control son una rama (marcadas en azul en el grafo):

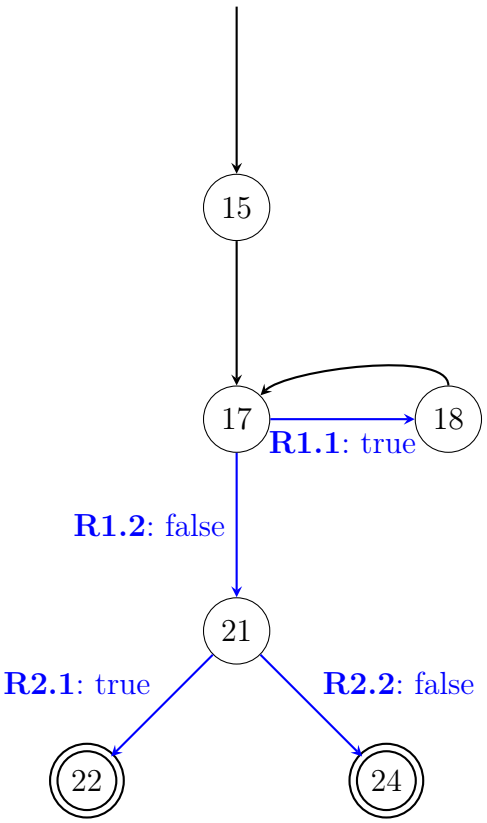

Figura A.3: Grafo de flujo asociado al código Java de 1.2

En el grafo podemos identificar 4 ramas.

Condiciones

Podemos encontrar 3 condiciones:

1. `i < vector.length - 1`
2. `vector[i] != datoBuscado`
3. `vector[i] == datoBuscado`

A.2.2. Ejecución y cobertura

Si ejecutamos el método con los siguiente parámetros: `buscar(new int[]{1,2}, 1)` podemos calcular los resultados de los diferentes tipos de cobertura:

Cobertura de sentencias

Se ejecutaran la sentencias: 1,2,4,5.

Teniendo en cuenta que hay un total de 7 sentencias, tenemos una cobertura de 4/7, es decir **57.14 %**.

Cobertura de ramas

De las 4 ramas, se ejecutarán las ramas **R1.2** y **R2.1**, un total de 2/4, es decir, una cobertura del **50 %**.

Cobertura de condiciones

De las 3 condiciones que tenemos, podemos evaluar cada una como verdadero o falso, lo que da un total de 6 posibilidades (si analizamos cada condición de manera independiente al resto).

En este caso las condiciones definidas en A.2.1 se evaluan como:

1. `i < vector.length - 1`:

   Verdadero

2. `vector[i] != datoBuscado`:

   Falso

3. `vector[i] == datoBuscado`:

   Verdadero

En total tenemos 6 posibilidades, y hemos cubierto 3, lo que nos da una cobertura de 3/6, es decir, del **50 %**.

A.2.3. Cobertura completa

El objetivo de la medición de cobertura de código es conocer que falta por probar para llegar a una cobertura del 100 % añadiendo casos de prueba.

Un ejemplo de casos de prueba adicionales para alcanzar una cobertura completa para los diferentes tipos de cobertura serían:

Cobertura completa de instrucciones

Si ejecutamos el método con los siguiente parámetros:

- `buscar(new int[]{1,2}, 1)`: se ejecutarían las instrucciones 1, 2, 4 y 5.
- `buscar(new int[]{1,2}, 3)`: se ejecutarían las instrucciones 1, 2, 3, 4, 6 y 7.

Por lo tanto tendríamos una cobertura de 7/7 instrucciones, es decir **100 %**.

Cobertura completa de ramas

Si ejecutamos el método con los siguiente parámetros:

- `buscar(new int[]{1,2}, 1)`: se ejecutarían las ramas **R1.2** y **R2.1**.
- `buscar(new int[]{1,2}, 3)`: se ejecutarían las ramas **R1.1**, **R1.2** y **R2.2**.

  Por lo tanto tendríamos una cobertura de 4/4 ramas, es decir **100 %**.

Cobertura completa de condiciones

Si ejecutamos el método con los siguientes parámetros:

- `buscar(new int[]{1,2}, 1)`: las condiciones definidas en A.2.1 se evaluan como:

  1. `i < vector.length - 1`:
     Verdadero
  2. `vector[i] != datoBuscado`:
     Falso
  3. `vector[i] == datoBuscado`:
     Verdadero

- `buscar(new int[]{1}, 3)`: las condiciones definidas en A.2.1 se evaluan como:

  1. `i < vector.length - 1`:
     Verdadero y **Falso**
  2. `vector[i] != datoBuscado`:
     Verdadero
  3. `vector[i] == datoBuscado`:
     Falso

  Por lo tanto tendríamos una cobertura de 6/6 valores en condiciones, es decir **100 %**.

Anexo B

Breve resumen de bytecode Java y sus instrucciones

B.1. Instrucciones

Los principales tipos de instrucciones bytecode de la JVM son:

B.1.1. Instrucciones de *load* y *store*

Estas instrucciones transfieren valores entre las variables locales y la pila de operandos del marco de la VM. Existen diferentes variantes para los diferentes tipos de datos de la JVM. Ejemplos: iload, fload, dload, aload, bipush, sipush, etc.

B.1.2. Instrucciones aritméticas

Estas instrucciones permiten computar el resultado de una operación aritmética entre dos valores de la pila de operandos, realizando *push* de vuelta del resultado a la pila de operandos. Existen varios tipos de operaciones:

- Aritmética entera o de coma flotante: suma, resta, multiplicación, división, resto, etc.

- Aritmética de bits: bit-shifting, OR, AND, XOR, etc.

B.1.3. Instrucciones de conversión de tipos

Estas instrucciones permiten conversion entre los diferentes tipos de la JVM. Existen conversiones sin perdidas (de un tipo de dato con menor numero de bits que el destino) y con perdidas (de un tipo de dato a otro con menor numero de bits). Ejemplos: i2l, i2f, i2b, i2c.

B.1.4. Instrucciones de creación y acceso a objetos

Estas instrucciones permiten crear una nueva instancia de una clase o acceder a los campos de un objeto. Ejemplos: new, newarray, getstatic, arraylength, instanceof.

B.1.5. Instrucciones de transferencia de control

Estas instrucciones permiten modificar el flujo de ejecución de un programa. Existen varios tipos de operaciones:

- Salto condicional: modifican la dirección de salto dependiendo del resultado de la comparación. Ejemplos: ifeq. ifgt, ifnull, etc.

- Salto incondicional: modifican la dirección de salto de manera incondicional. Ejemplos: goto, jsr, etc.

B.1.6. Instrucciones de invocación de métodos

Estas instrucciones permiten invocar los métodos de alguna clase u objeto. Ejemplos: invokevirtual, invokeinterface, invokstatic, invokedynamic.

B.1.7. Instrucciones de devolución de valores

Estas instrucciones permiten devolver el control en un método a su invocador, devolviendo un dato si es necesario. Ejemplos: ireturn (para devolver un entero), freturn (para devolver un float), return (para métodos *void*).

B.2. Ejemplo de programa en bytecode

B.2.1. Método en Java

```
37 public static int longitudCollatz(long inicio) {
38     int longitud = 1;
39     long siguienteProbar = inicio;
40     while (siguienteProbar!=1){
41         siguienteProbar = siguienteCollatz(siguienteProbar);
42         longitud++;
43     }
44     return longitud;
45 }
```

Figura B.1: Código Java de la función *longitudCollatz*

B.2.2. Método en bytecode

```
public static longitudCollatz(J)I
    L0:
        linenumber 38 L0
        iconst_1
        istore 2
    L1:
        linenumber 39 L1
        lload 0
        lstore 3
    L2:
        linenumber 40 L2
        frame append [I J]
        lload 3
        lconst_1
        lcmp
        ifeq L3
    L4:
        linenumber 41 L4
        lload 3
        invokestatic es/unizar/eina/vv6f/collatz/SecuenciaCollatz.siguienteCollatz (J)J
        lstore 3
    L5:
        linenumber 42 L5
        iinc 2 1
        goto L2
    L3:
        linenumber 44 L3
        frame same
        iload 2
        ireturn
    L6
        localvariable inicio J L0 L6 0
        localvariable longitud I L1 L6 2
        localvariable siguienteProbar J L2 L6 3
        maxstack = 4
        maxlocals = 5
```

Figura B.2: Código bytecode de la función *longitudCollatz*

Anexo C

Diagramas arquitecturales

C.1. Arquitectura del agente Java

Al ser una herramienta local para el soporte al testing con la cual se interactúa únicamente cuando se añade como agente a una ejecución Java, no es relevante ni su vista de despliegue, ni su vista de componente-conector, por lo que la explicación de la arquitectura va a estar centrada en la vista estática, especialmente en explicar los diferentes paquetes, sus propósitos, las clases incluidas y sus responsabilidades.

C.1.1. Diagrama de paquetes

Como se ve en la figura C.1 las clases están estructuradas en diferentes paquetes según su propósito. Los diferentes paquetes son:

- **controlFlow**: contiene aquellas clases relacionadas con el análisis de control de flujo de código bytecode:
  - *ControlFlowAnalyser*: clase que contiene toda la lógica para obtener el grafo de control de flujo a partir de unas instrucciones bytecode.

    Posee métodos públicos para analizar una lista de instrucciones bytecode y para obtener los resultados del análisis de diferentes maneras: como un grafo con `AbstractInsnNode`, con identificadores numéricos "artificiales", o con el número de línea del código fuente del nodo correspondiente.

    Además posee múltiples métodos privados que facilitan el análisis de las instrucciones bytecode.
  - *EdgePair*: clase que representa una situación de prueba (un par de aristas) de la técnica de caminos de profundidad 2.
- **graph**: contiene aquellas clases relacionadas con la definición de los grafos utilizados, y utilidades de transformación de grafos:

- *EdgeType*: tipo enumerado que representa los diferentes tipos de arista que puede haber en el grafo.
- *BooleanEdge*: clase que extiende la arista por defecto de la librería JGraphT: `DefaultEdge`[10], añadiendo información acerca del tipo de arista.
- *GraphToDotTransfomer*: clase de utilidad que permite exportar un objeto `DirectedPseudograph` a su correspondiente notación DOT[11].

– **DTO** (Data Transfer Object): contiene aquellas clases utilizadas como objetos de transferencia de datos a la hora de exportar los resultados de la medición de cobertura:

- *MethodReportDTO*: clase que representa la información necesaria de un informe de cobertura de caminos de profundidad 2 para un método en específico.
- *EdgePairDTO*: clase que representa la información necesaria para identificar una situación de prueba en concreto.

Algunas clases que no pertenecen a ningún paquete de propósito específico son:

– **Agent**: es la clase que actúa como punto de entrada a la instrumentación, la JVM invoca su método `premain` y este se encarga de añadir un `ClassFileTransformer` a través de la Instrumentation API. Además se encarga de realizar el *parse* de los argumentos que recibe el agente.

– **AgentTransformer**: clase que implementa `ClassFileTransformer` y que la JVM invoca su método `transform` por cada clase a cargar en ella. Para cada clase de la que se desea medir su cobertura, se obtiene su grafo de control de flujo, sus situaciones de prueba, y se modifica el método añadiendo instrucciones que permiten registrar de manera dinámica las aristas se vayan ejecutando.

Además añade un `ShutdownHook` (un hilo de Java que se ejecuta justo antes de apagar la JVM) al `Runtime` de la aplicación para imprimir el informe al finalizar su ejecución.

[10] `https://jgrapht.org/javadoc/org.jgrapht.core/org/jgrapht/graph/DefaultEdge.html`
[11] `https://graphviz.org/doc/info/lang.html`

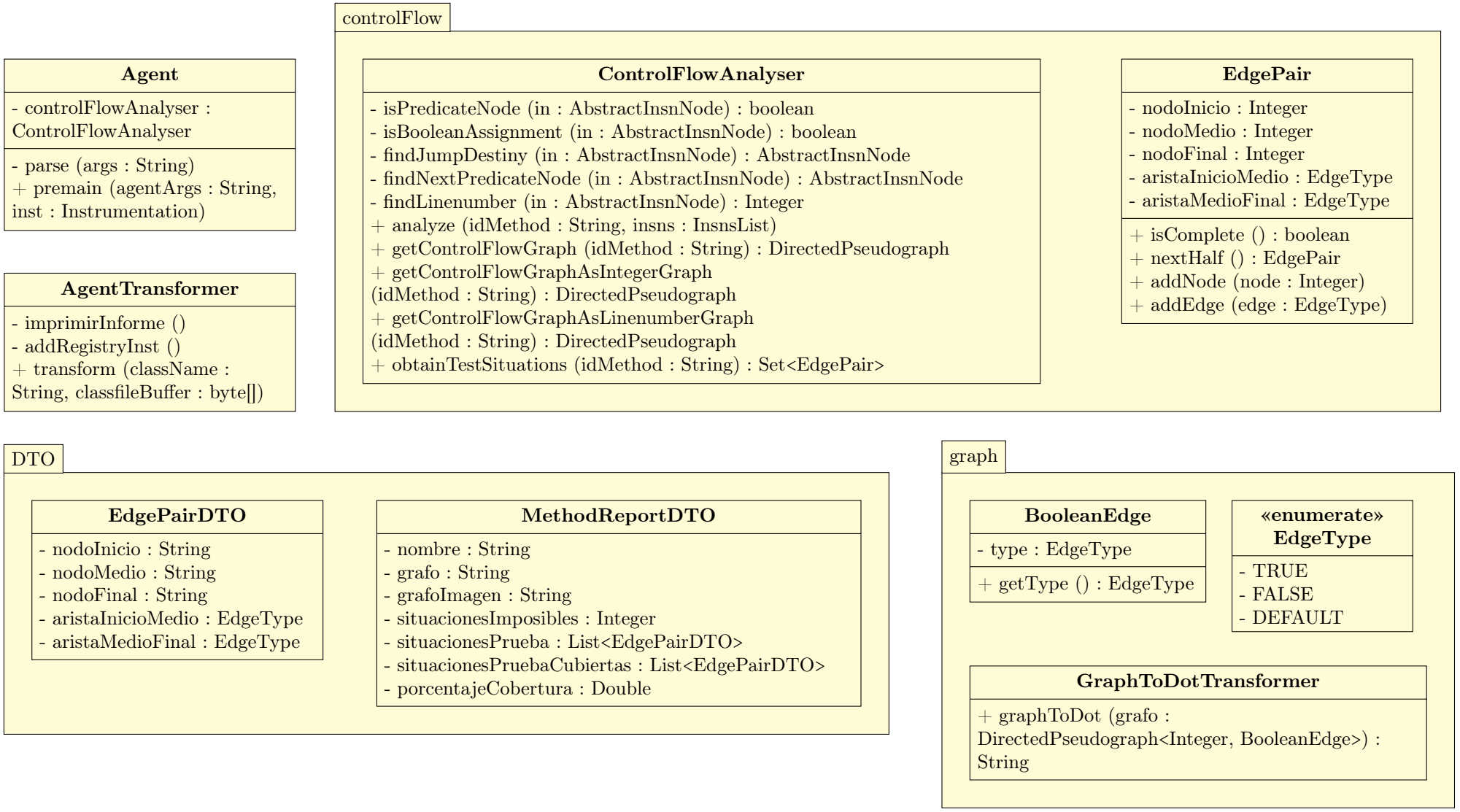

Figura C.1: Diagrama de paquetes del agente

C.2. Arquitectura del plugin de IntelliJ

Al ser un plugin local para un IDE, lo más relevante acerca de su arquitectura es:

- **Vista estática**: se van a explicar los diferentes paquetes, sus propósitos, las clases incluidas y sus responsabilidades.

- **Vista dinámica**: únicamente se va a mostrar como interactúan los diferentes objetos del sistema cuando un usuario hace click en el botón de "Run with 'Edge-Pair' coverage".

C.2.1. Contexto sobre la *IntelliJ Platform Plugin SDK*

Para entender algunos aspectos de la arquitectura del plugin, hay que explicar antes los sistemas y componentes que ofrece la IntelliJ Platform y que han sido utilizados en este proyecto:

- **User Interface Components**: conjunto de componentes *Swing*[12] personalizados para mantener la consistencia entre la UI del IDE.

- **Actions**: sistema que permite añadir acciones personalizadas a botones que puedes incluir diferentes secciones del IDE.

- **Settings**: sistema que permite definir y persistir ajustes personalizados por el usuario.

- **Virtual File System**: sistema que encapsula el acceso a ficheros, permitiendo subscribirte a los diferentes eventos de un ficheros (modificación, eliminación, etc).

- **PSI**: el *Program Structure Interface* es el sistema responsable del análisis sintáctico y semántico del código fuente, permitiendo acceder a su modelo, obtener y modificar atributos, métodos, etc.

- **Execution**: sistema que controla la ejecución de procesos dentro del IDE. En este proyecto únicamente se han utilizado los siguientes conceptos:

    - **RunConfiguration**: permite acceder y persistir las diferentes opciones de ejecución, así como variables de entorno y argumentos.

[12]Librería gráfica para crear GUI en Java: `https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html`)

- **Execution events**: permite subscribirte a diferentes eventos relacionados con la ejecución de un proceso (comienzo, finalización, etc) con la implementación de un `ExecutionListener`.

C.2.2. Diagrama de paquetes

Como se ve en la figura C.2 las clases están estructuradas en diferentes paquetes según su propósito. Los diferentes paquetes son:

- **actions**: contiene aquellas clases relacionadas con las *actions* del plugin:
  - *RunWithEdgePairCoverage*: clase que extiende `AnAction` que contiene toda la lógica para abrir el selector de métodos y ejecutar la configuración actual añadiendo el agente (y sus paremetros) a la ejecución.
- **toolWindows**: contiene aquellas clases relacionadas con las *Tool windows* del plugin:
  - *EdgePairCoverageReportNativeWindow*: clase que devuelve un JPanel que permite visualizar el informe con elementos nativos del IDE.
  - *EdgePairCoverageReportNativeWindowFactory*: clase que implementa `ToolWindowFactory` y es encargada de instanciar *EdgePairCoverageReportNativeWindow*.
  - *EdgePairCoverageReportHTMLWindow*: clase que devuelve un JPanel que permite visualizar el informe como un HTML.
  - *EdgePairCoverageReportHTMLWindowFactory*: clase que implementa `ToolWindowFactory` y es encargada de instanciar *EdgePairCoverageReportHTMLWindow*.
- **settings**: contiene aquellas clases relacionadas con los ajustes del plugin:
  - *AppSettings*: clase que representa la información a persistir como ajustes del plugin.
- **dialogs**: contiene aquellas clases relaciones con los *dialogs*:
  - *SeleccionarMetodosDialog*: clase que extiende `DialogWrapper` e implementa el dialog para seleccionar métodos a medir la cobertura y sus situaciones imposibles.
- **panels**: contiene aquellas clases relaciones con los *panels*:

- *MethodReportPanel*: clase que extiende `JPanel` e implementa el panel para mostrar el informe de cobertura de un método.
- *MethodSelectorPanel*: clase que extiende `JPanel` e implementa el panel para seleccionar los métodos a medir la cobertura.
- *ReportPanel*: clase que extiende `JPanel` e implementa el panel para mostrar el informe de cobertura completo.
- *SituacionesImposiblesSelectorPanel*: clase que extiende `JPanel` e implementa el panel para indicar las situaciones imposibles de los métodos seleccionados.

Algunas clases que no pertenecen a ningún paquete de propósito especifico son:

– **MyExecutionListener**: clase que implementa `ExecutionListener` y se subscribe al evento de finalización de ejecución para mostrar el informe en una *ToolWindow*.

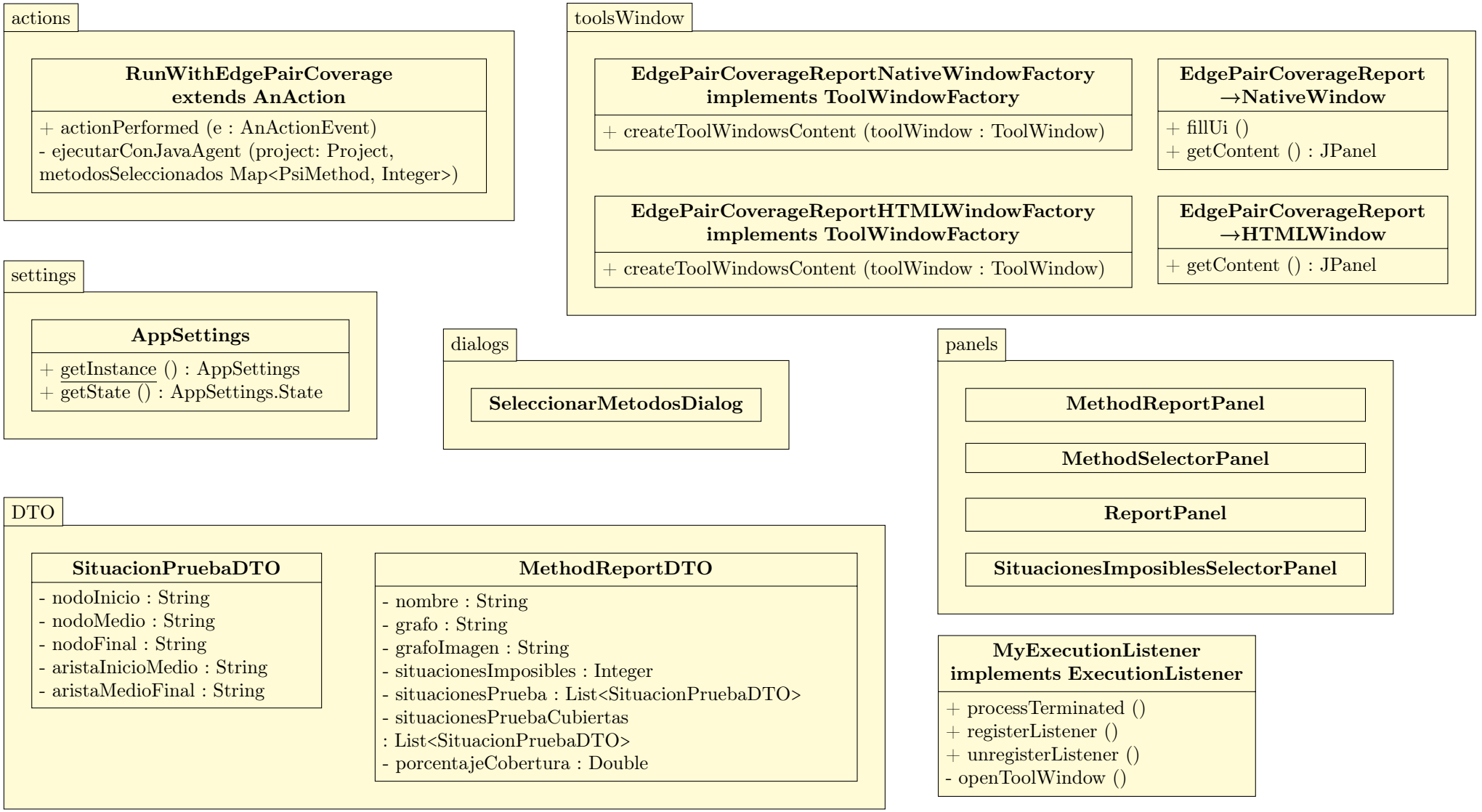

Figura C.2: Diagrama de paquetes del plugin para IntelliJ

C.2.3. Diagrama de secuencia

En la figura C.3 se puede ver el diagrama de secuencia del caso de uso "Medir cobertura de profundidad 2" desde la perspectiva del plugin.

Este diagrama de secuencia comienza con:

1. El actor Desarrollador hace *click* en el botón de "Run with 'Edge-Pair' coverage".

   Este botón tiene asociado la *action* RunWithEdgePairCoverage que inicialmente muestra el *dialog* SeleccionarMetodoDialog .

2. El Desarrollador hace *click* en el botón de "Next" y se muestra el *dialog* SeleccionarMetodoDialog.

3. El Desarrollador hace *click* en el botón de "Run" y se ejecuta la configuración existente añadiendo el agente y los parámetros.

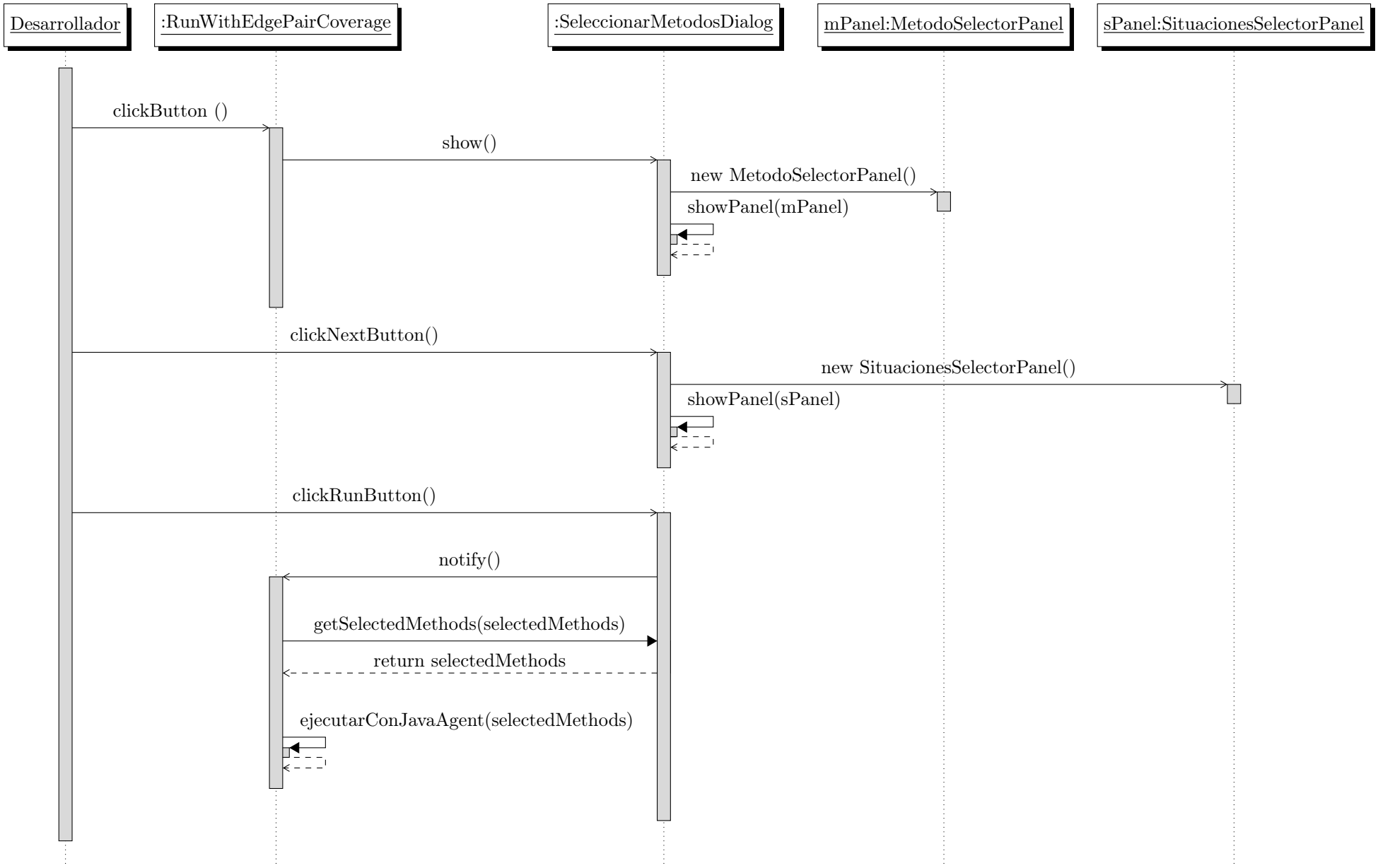

Figura C.3: Diagrama de secuencia del caso de uso “Medir cobertura de profundidad 2” desde la perspectiva del plugin para IntelliJ