UNIVERSIDAD
NACIONAL
DE COLOMBIA

ANÁLISIS Y COMPARACIÓN DE ALGORITMOS DE IDENTIFICACIÓN DE CARACTERÍSTICAS APLICADOS A UNA FAMILIA DE PRODUCTOS DE SOFTWARE.

Nicolás Ordóñez Chala

Universidad Nacional de Colombia
Facultad de Ingeniería
Departamento de Ingeniería de Sistemas e Industrial
Bogotá D.C., Colombia
2018

Análisis y comparación de algoritmos de identificación de características aplicados a una familia de productos de software.

Nicolás Ordóñez Chala

Tesis presentada como requisito parcial para optar al título de:
Magister en Ingeniería de Sistemas y Computación

Director (a):
Jabier Martínez Perdiguero
Ph.D. Ingeniería de Sistemas y computación

Codirector (a):
Jairo Hernán Aponte Melo
Ph.D. Ingeniería de Sistemas y computación

Línea de Investigación:
Evolución y mantenimiento de Software
Grupo de Investigación:
COLSWE

Universidad Nacional de Colombia
Facultad de Ingeniería
Departamento de Ingeniería de Sistemas e Industrial
Bogotá D.C., Colombia
2018

A mis dos madres, a mi papá y a mi novia. Gracias por el apoyo incondicional.

No basta con proponerse a hacer algo, si solo se queda pensamientos.

La disciplina tarde o temprano vencerá a la inteligencia. - Yokoi Kenji

Agradecimientos

Agradezco a mi familia: a mis dos madres, a mi papá y mi novia. Por estar conmigo durante todo este recorrido, darme aliento y brindarme siempre la energía para seguir adelante. Su compañía, amor y comprensión fueron fundamentales para que en mi ser estuviera presente la voluntad y el deseo de pensar en grande.

Agradezco todo mi estudio a la Universidad Nacional de Colombia, entidad donde me forme como ingeniero y me brindó la oportunidad de desarrollar mi pensamiento crítico y entusiasmo por la investigación.

Agradezco a Jabier Martinez, mi director de tesis, el cual fue una guía y apoyo durante todo el proceso, me brindo opciones y sin él no hubiera sido posible este trabajo.

Agradezco a Tecnalia, empresa de investigación, la cual me recibió cordialmente durante mi estancia en España para el desarrollo de mi práctica.

Resumen

El desarrollo de software es una tendencia hoy en día por lo que la cantidad de código que se ha generado en los últimos años ha aumentado significativamente. Adicionalmente, el aumento de demanda y la necesidad de reducir los tiempos de desarrollo generan problemas a corto y largo plazo que dificultan la mantenibilidad de cualquier proyecto. La reutilización de elementos de software (código fuente, modelos, entre otros), ha sido una práctica que se ha realizado a través de técnicas que no tienen en cuenta su soporte. Las líneas de productos de software (SPL) promueven la construcción de elementos que se puedan reutilizar y actualizar de acuerdo a las necesidades de manera organizada. En busca de la reutilización, existen técnicas extractivas que buscan componentes en proyectos legados o monolíticos, software que se creó teniendo en cuenta un objetivo específico. Existen varios casos de estudio, en donde se ha construido una SPL a partir de elementos que no buscaban ser reutilizados originalmente. Particularmente, ArgoUML SPL fue un proyecto creado en Java, el cual ha sido usado para verificar la efectividad de los algoritmos automáticos de análisis y extracción de características. Sin embargo, debido a la cantidad de variantes que existen en este caso de estudio, se hace evidente mostrar una comparación que sintetice y establezca un punto de referencia para la construcción de algoritmos de recuperación de características. Por tal motivo en este trabajo se realizó la construcción de una medida de comparación y se implementó una técnica que sirva como punto de referencia para futuras investigaciones utilizando dicha métrica unificada.

Palabras clave: Líneas de productos de software, trazabilidad, recuperación de características, ArgoUML SPL.

Abstract

Nowadays, Software development is a trend, so the amount of code that has been generated in recent years has increased significantly. Additionally, the demand of software is also increasing, and the need to reduce the development time generates short and long-term problems that make difficult the maintenance of any project. The reuse of software elements (source code, models, among others), has been a practice that has been carried out through techniques that do not pay enough attention to the maintenance of the multiple projects. Software product lines (SPL) promote the construction of assets that can be reused and updated according to needs in a systematic manner. In search of reuse, there are instructive techniques that look for components in legacy or monolithic projects, software that was created with a specific objective in mind. There are several cases of study, where an SPL has been built from assets that did not seek to be reused originally. In particular, ArgoUML SPL was a project created in Java, which has been extensively used to test automatic techniques of identification and extraction of features. However, due to the number of variants that exist in this case study, it is evident to show a comparison that synthesizes and establishes a reference point for the construction of feature recovery algorithms. The main objective of this work is build a groundtruth to compare feature location algorithms and implement a technique which can be used as a reference to compare future researches.

Keywords: Software product line, traceability, feature location, ArgoUML SPL.

Contenido

Lista de figuras

Lista de ecuaciones

Lista de tablas

Lista de Símbolos y abreviaturas

Abreviaturas

Abreviatura	Término
SPL	Línea de productos de software (Origen inglés: Software Product Line)
LSI	Indexación semántica latente. (Origen inglés: Latent Semantic Indexing)
IR	Recuperación de información (Origen inglés: Information Retrieval)
FCA	Análisis formal de conceptos (Origen inglés: Formal Concept Analysis)
KLOC	Mil líneas de código (KLOC – Origen inglés: Kilo Lines of Code)
LPS	Líneas de productos de software

Introducción

Con el fin de automatizar la construcción de software, se ha desarrollado una metodología denominada líneas de productos de software (SPL) la cual busca crear una variedad de productos de software a partir de un solo proceso de desarrollo [1]. Es decir, partiendo de un conjunto de requerimientos, se establece un alcance con el cual se construye un grupo de features (características) los cuales derivan productos para un grupo de clientes con necesidades particulares [3]. Lo anterior, se puede expresar a través de un diagrama de variabilidad que muestra los features que pueden o no estar presentes dependiendo de cómo se defina durante su construcción [6]. Para la elaboración de una SPL se puede partir de un proyecto existente, esto debido a que existe gran cantidad de código con el potencial de ser reutilizado, pero su dificultad se resume en que, difícilmente un proyecto tiene componentes independientes; es decir, los posibles features que se pueden encontrar están altamente acoplados, lo que dificulta que se puedan extraer de manera manual para su futura utilización en otro proyecto [2]. A esto se le llama estrategia extractiva, su investigación busca eliminar las barreras que dificultan la reutilización de componentes altamente acoplados.

Paralelamente, la reutilización de software es una de las prácticas más usadas por programadores que consiste en la utilización de tareas realizadas previamente que sirvan para un nuevo desarrollo que necesita la misma funcionalidad, o que se puede adaptar fácilmente para la construcción de una característica, de manera qué se puede ahorrar tiempo y esfuerzo para terminar un nuevo proyecto. Esto es conocido como técnicas de clone-and-own o copy-paste-modify [15]. Si no se organiza desde un principio toda esta cantidad de información, es muy difícil que se pueda reutilizar de manera óptima [2]. La selección inicial de frameworks y herramientas de desarrollo teniendo en cuenta buenas prácticas, son algunas formas de facilitar la organización de código desde el principio de del proyecto para que la reutilización se pueda hacer de manera más sencilla, aun así, puede llegar a ser una tarea complicada. Tal afirmación, implica que cada vez que

comienza un proyecto nuevo por muy claros que se tengan los conceptos, es difícil cumplir con el plan inicial [2].

Para eliminar las barreras mencionadas, se hizo notorio la necesidad crear escenarios de pruebas que permitieran validar la eficacia de las metodologías de identificación y análisis de features [26]. Como resultado, se construyó la herramienta ArgoUML SPL, la cual por medio de notaciones ifdef realizó la separación de features, de manera que a través de herramientas de preprocesamiento se pudiera construir un producto completamente funcional a partir de la selección de features [16]. Lo anterior serviría para evidencia, de que la elaboración de una familia a través de un proyecto cómo ArgoUML era posible, y además, serviría para que se pudieran poner a prueba los algoritmos que buscarán features de manera automática. Lamentablemente, debido a la diversidad de posibilidades que existe en una SPL, las investigaciones posteriores utilizaron diferentes combinaciones de features, por lo que no se podía evidenciar las diferencias, fortalezas y debilidades entre cada técnica de localización de features.

El siguiente trabajo se encarga de contrarrestar las debilidades encontradas en las investigaciones realizadas hasta el momento en ArgoUML SPL. Los capítulos presentados a continuación están organizados de la siguiente forma: Capitulo uno, se describe de manera general la motivación y objetivos del presente trabajo; capítulo dos, descripción general del campo de SPL y una explicación de los temas que están orientados a la dificultad que existe para determinar un buen algoritmo de identificación de características; capítulo tres, revisión sistemática del caso de estudio seleccionado para mostrar las principales dificultades; capítulo cuatro, utilización de las dificultades encontradas para mostrar el diseño de una solución que facilite la forma de comparar algoritmos de identificación de características; capítulo cinco, explica la implementación del conjunto de algoritmos más utilizados según lo encontrado en el capítulo tres y su utilización en la solución diseñada en el capítulo cuatro; finalmente el capítulo seis describe un conjunto de conclusiones y trabajos futuros a partir del trabajo realizado.

1. Visión general

1.1 Motivación

Durante el desarrollo de software es usual que se deban programar múltiples veces las mismas funcionalidades en diferentes proyectos para satisfacer necesidades similares. Debido al aumento de demanda de software se ha necesitado satisfacer los proyectos en tiempos cada vez más cortos, obligando a reutilizar componentes de manera desorganizada [2]. Con el fin de mejorar la forma de implementar estas prácticas es importante clasificar los elementos que conforman un proyecto para que se puedan utilizar más fácilmente. Existen muchos repositorios públicos y privados que terminan siendo utilizados como librerías, pero no como fuentes de componentes que sirvan para agilizar el desarrollo de software. Para lograr lo anterior se deben implementar técnicas que organicen toda la información de forma automática. Sin embargo, se debe tener previamente un grupo de casos de prueba que permitan evaluar fácilmente el rendimiento de los algoritmos que ejecuten está clasificación.

1.2 Antecedentes y justificación

Con el fin de automatizar la construcción de software, se ha desarrollado una metodología denominada líneas de productos de software (SPL) la cual busca crear una variedad de productos de software a partir de un solo proceso de desarrollo [1]. Es decir, partiendo de un conjunto de necesidades, construir un grupo de características (features) que sean seleccionables para el desarrollo de un producto final. Dependiendo del cliente que lo consuma se decide utilizar o no una característica, esto es una familia de productos de software, la cual se puede expresar a través de un diagrama de variabilidad que muestra los features que deben o no estar presentes dependiendo de cómo se defina durante su construcción. Para la elaboración de dicho desarrollo se puede partir de un proyecto

existente. Esto debido a que existe gran cantidad de código que puede ser reutilizado, pero su dificultad se resume en que existen problemas de calidad de código, es decir, los posibles features que se pueden encontrar están altamente acoplados entre sí, lo que dificulta que se puedan extraer de manera independiente para su futura utilización en otro proyecto [2]. A esto se le llama estrategia extractiva, su investigación busca eliminar las barreras que dificultan la reutilización de componentes altamente acoplados [1].

¿Qué hacer para contrarrestar estas dificultades? La utilización de etiquetas a lo largo del código para diferenciar componentes dio como resultado la herramienta ArgoUML SPL, de esta forma, a través de herramientas de preprocesamiento se hace la lectura automática para construir productos completamente funcionales a partir de la selección de features [16]. La construcción de ArgoUML SPL sirvió como evidencia para demostrar que la creación de una familia a partir de un proyecto ya existente era posible. ArgoUML se convierte entonces en un caso de estudio que se utiliza para evaluar técnicas que extraigan componentes de software automáticamente [10]. Sin embargo, a causa de la diversidad de posibilidades de una SPL, cada investigación presentó sus resultados pero sin un estándar que comparará estudios [26].

1.3 **Identificación del problema**

Una línea de productos de software (SPL) es un conjunto de sistemas que comparten un conjunto de componentes en común. Su estudio surgió desde hace veinte años cuando un grupo de investigadores propuso que se construyeran productos de software a través de componentes que fueran comunes a varios proyectos, y fue Pohl el que busco propagar está idea [1]. Para construir una SPL existen tres formas, entre ellas, la estrategia extractiva, que busca rescatar componentes de los proyectos de software que existen actualmente. Debido a que cuando se construye un proyecto no está planeado para reutilizarse, es difícil identificar features exitosamente en segmentos de código altamente acoplados [2]. Para poner a prueba los métodos propuestos por diferentes investigadores se crearon pruebas como ArgoUML SPL, cuyo factor especial es que se desarrolló a partir de un proyecto que no tenía la intención final de ser convertido en una LPS.

Inicialmente, autores en el área propusieron algoritmos manuales en donde se hacía una revisión exhaustiva para encontrar componentes. Con el paso el tiempo y el desarrollo de la tecnología, se probaron diferentes algoritmos para que se hiciera de manera automática.

Fue hasta 2011, que con la salida de ArgoUML SPL, se empezó a probar la eficacia de los algoritmos. Es más, existen diferentes personas que han utilizado este proyecto para validar sus algoritmos de extracción de características de manera automática. Sin embargo, cada uno realiza métodos de validación diferentes, esto incluye diferentes combinaciones de componentes de ArgoUML SPL. Es importante que exista una única forma de medir su desempeño así como de definir el mejor algoritmo de extracción de características.

Al no existir una forma establecida para comparar técnicas, se encuentra que no es posible comparar las investigaciones, debido a que diferentes datos de entrada dificultan conocer bajo que contexto es mejor una técnica que otra. De la misma forma, las diferencias en los formatos de salida son un obstáculo para medir el desempeño, lo que hace que no sea correcto organizar todos los resultados y comparar gráficamente cual presenta mejores indicadores. Adicionalmente, los niveles de granularidad de cada investigación dificultan conocer cual tuvo mejor nivel de detalle, teniendo en cuenta los diferentes escenarios evaluados. Ayudar a establecer un único ambiente que permita borrar las brechas mencionadas anteriormente es el objetivo de esta investigación.

1.4 Objetivo general y objetivos específicos

El propósito del presente trabajo es realizar un estudio comparativo de los algoritmos de identificación de características que se han puesto a prueba con ArgoUML-SPL. Para lograr esto se proponen los siguientes objetivos específicos:

- **Diferenciar y comparar las investigaciones que han evaluado sus algoritmos utilizando ArgoUML SPL.** Para poder proponer una metodología que permita comparar las investigaciones relacionadas con la identificación de características, es importante conocer el contexto que se encuentra en ArgoUML SPL. El desarrollo de este objetivo se hizo por medio de una revisión de literatura (Ver capítulo 3).
- **Elaborar una medida de desempeño para las técnicas de identificación de características.** Debido a la variabilidad en familias de productos de software es difícil comparar las técnicas de identificación de características entre sí. Para lograr determinar bajo qué condiciones es mejor cada técnica, se utilizará la comparación

realizada en el objetivo anterior, con el fin de diseñar pruebas que utilicen los aspectos más relevantes encontrados.

- **Diseñar un técnica que sirva como punto referencia para la medida de desempeño propuesta.** Se extiende la funcionalidad de herramientas libres para implementar una técnica que sea capaz de utilizar los algoritmos más utilizados encontrados en el primer objetivo. De esta manera se asegura establecer una base que tenga en cuenta las prácticas utilizadas hasta el momento en la literatura. Su construcción busca ayudar a la comunidad académica a: Entender el funcionamiento de la medida de desempeño del objetivo anterior y promover la utilización de medidas que permitan comparar correctamente las diferentes técnicas.

1.5 Metodología y alcance

El desarrollo del actual trabajo se realizó haciendo una analogía a la metodología de desarrollo de software teniendo en cuenta la siguiente estructura:

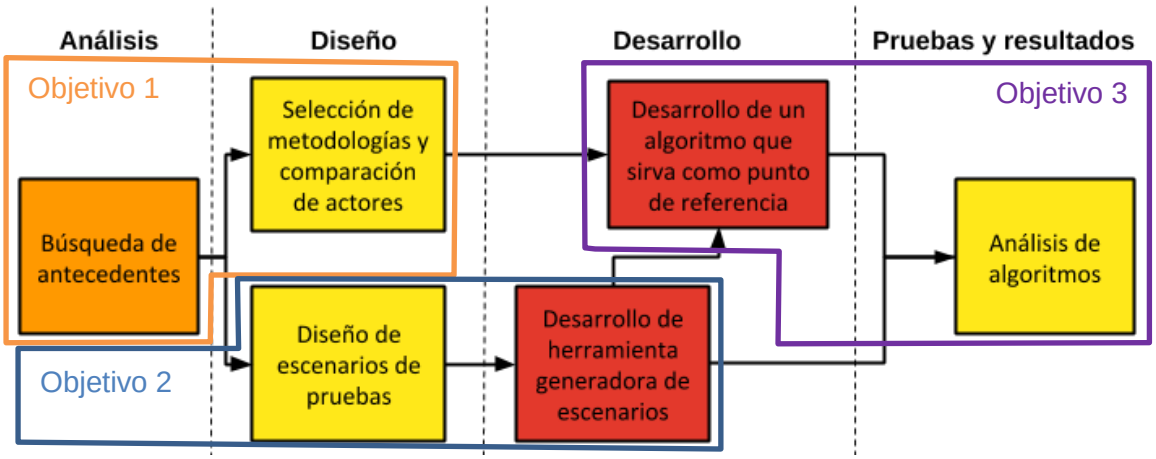

Ilustración 1: Metodología desarrollada, presentada haciendo analogía con el ciclo de desarrollo de software.

- **Etapa de análisis: Búsqueda de antecedentes**

  Se realizó un estudio exploratorio alrededor del campo seleccionado, enfocándose en el tema que motivó al desarrollo de este trabajo: recuperación de características de forma automática. Esta primera fase muestra aspectos generales que ayudan a entender el contexto en el cual se desarrolló el trabajo. Adicionalmente, evidencia

la existencia de casos de estudio que se utilizan para evaluar investigaciones. Entre estos, ArgoUML SPL sobresale al tener aspectos que proponen un reto.

- **Etapa de diseño: Selección de metodologías y comparación de actores**

  Acorde al objetivo uno, se realizó una búsqueda sistemática siguiendo la metodología de Kitchenham, utilizando criterios de inclusión y exclusión se obtuvo información relacionada con ArgoUML SPL. Entre los resultados principales encontrados estuvieron: escenarios de prueba utilizados, métricas definidas, niveles de granularidad utilizados, entre otras. Todo lo anterior sirve como evidencia para mostrar la importancia de realizar el objetivo dos y tres, ya que se encuentra que no existe forma de comparar los estudios, debido a la diversidad de escenarios que existen para el caso de estudio. El resultado final de la etapa de diseño da como resultado el Anexo A, que muestra, entre varios aspectos, las técnicas más utilizadas que servirán como justificación para ser implementadas en un único algoritmo como se realizó en la etapa de desarrollo.

- **Etapa de diseño: Diseño de escenarios de pruebas**

  Aprovechando la recopilación de información relacionada al caso de estudio, se comprendió el funcionamiento del caso de estudio que se encuentra expuesto en un repositorio público. Se plantearon escenarios que están directamente relacionados con la búsqueda sistemática realizada, agregando escenarios que promueven dificultades al estar cercanas a un caso industrial.

- **Etapa de desarrollo: Desarrollo de herramienta generadora de escenarios**

  Directamente relacionada con el objetivo dos, se utilizó lo desarrollado en la etapa de diseño para empezar a estudiar e implementar el diseño de un caso de prueba unificado para contrarrestar las debilidades y dificultades evidenciadas en la etapa anterior, en otras palabras, la definición de un GroundTruth. De forma tal, que se establece una forma única de realizar la comparación de estudios que se encargan de hacer recuperación de características de manera automática.

- **Etapa de desarrollo: Desarrollo de un algoritmo que sirva como punto de referencia**

  En relación al objetivo tres, se aprovecha la definición del GroundTruth y el análisis desarrollado en la etapa anterior para realiza un algoritmo que tome las técnicas más utilizadas en la literatura, encontradas en el objetivo uno. Esto con el fin establecer una línea base que sirva como eje de comparación para futuras investigaciones. Adicionalmente, se realiza la construcción de un ejemplo que sirva para probar su desempeño de manera manual.

- **Etapa de pruebas y resultados: Análisis de algoritmos**

  Finalmente, se realiza la evaluación de rendimiento del algoritmo implementado y como valor adicional, se realiza la comparación con otros algoritmos. Además, se resume todo el desarrollo del trabajo con: Conclusiones relacionadas con los escenarios desarrollados en el objetivo dos y posibles trabajos futuros que complementen la investigación.

2. Revisión de literatura

2.1 Motivación

Con el fin de entender el campo de SPL, es importante mostrar un contexto de la investigación realizada. De esta manera se puede entender más fácilmente cuales son los aspectos que llevaron a comprender cuales es la dificultad que busca confrontar esta tesis. En este capítulo se aclaran conceptos generales que se encuentran alrededor de SPL, el entendimiento de definiciones básicas y la investigación realizada a lo largo de la historia facilitan en gran medida el desarrollo del trabajo. A continuación se presentan algunas preguntas que ayudan a comprender la importancia de esta línea de investigación.

2.2 Líneas de productos de software: Estrategia extractiva

2.2.1 ¿Qué es una SPL?

Se entiende como línea de productos de software (SPL) al paradigma de desarrollo de software utilizando la personalización en masa [1], lo que consiste en la construcción de un conjunto de productos de software que puede satisfacer las necesidades de varios clientes [5]. La idea de automatizar la creación de software surgió en 1968 [7] y posteriormente se formalizó el concepto en términos de familias de productos de software [8][9]. Cómo lo muestra la ilustración 1, para su construcción se establecen dos fases, la ingeniería de dominio [6], la cual consiste en el análisis y desarrollo de los componentes de software que participarán en la línea para la siguiente fase; y la ingeniería de aplicación, que busca realizar la configuración del producto, en otras palabras, la selección de componentes asociados a features que mejor satisfacen a un cliente en particular.

Si bien la reutilización siempre ha sido importante en el campo del desarrollo de software, los desarrolladores se dieron cuenta que optimizar la manera en que se reutilizan componentes de software más elaborados puede favorecer a la construcción de proyectos [11]. Un feature es un conjunto de elementos de software que brindan una funcionalidad

completa al cliente que se puede traducir en diferentes formas, como arquitecturas, modelos, requerimientos, código fuente, entre otros. Una variante es la integración de un conjunto determinado de features determinados dentro de una SPL. Por lo que se puede decir que SPL busca la reutilización óptima de componentes a través de features (características).

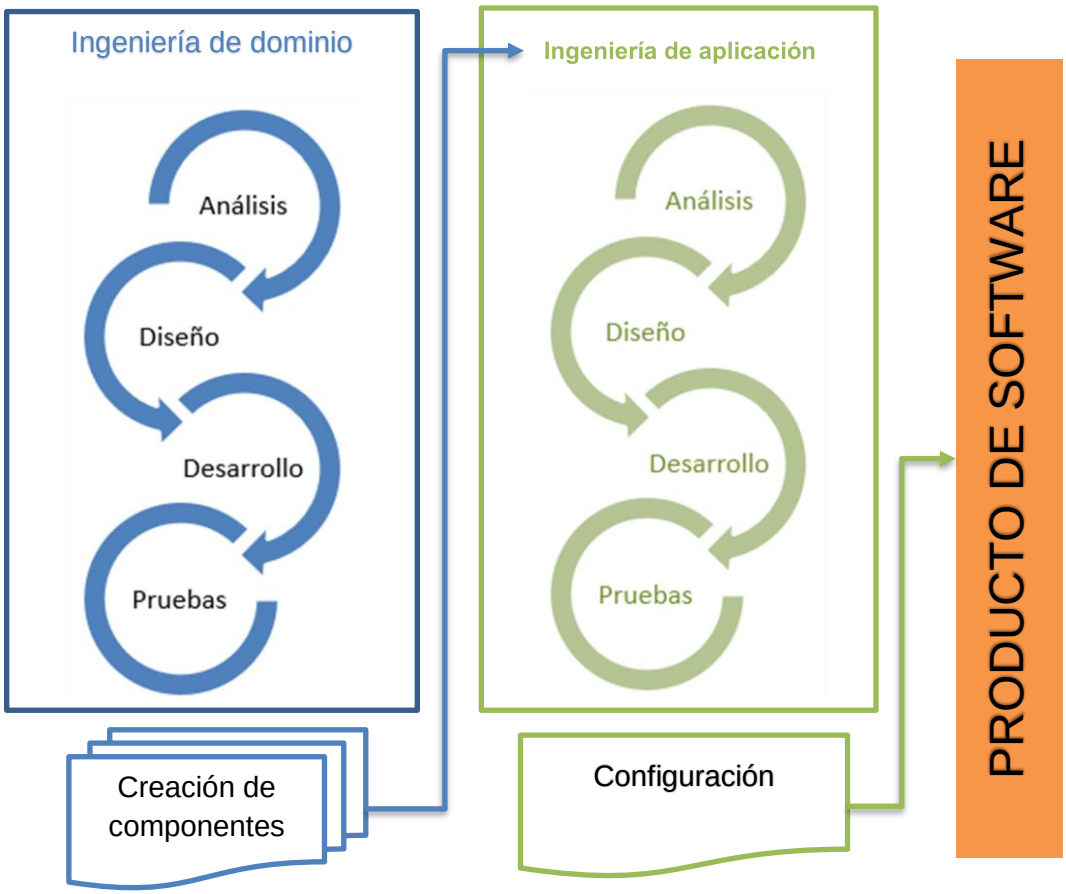

Ilustración 2: Proceso de desarrollo de una SPL.

A partir de todo lo anterior, se evidencia que, a gran escala, SPL tiene los siguientes campos [12]:

- **Múltiples líneas de productos de software:** Consiste en el estudio de la creación de procesos que faciliten la construcción de más de una SPL.
- **Desarrollo dirigido por modelos:** Se encarga de realizar el estudio de transformaciones de modelos para acercar la fase de diseño a fase de programación.

- **Herramientas de soporte:** Enfocado en la creación de software que facilite la elaboración de una SPL en cualquiera de sus fases.
- **Economía y modelos de costos:** Estudio en la manera de justificar la viabilidad de la aplicación de SPL en el sector industrial.
- **Aseguramiento de calidad:** Estudiar las necesidades del cliente para cumplir con los objetivos planteados en base al alcance definido.
- **Especialización de dominio:** Fortalecer la fase de construcción y diseño de componentes para facilitar la configuración de dichos componentes.
- **Mejora y evaluación de los procesos:** Se necesita realizar diferentes propuestas y más elaboradas que permitan llegar a un momento de estabilidad para que se pueda implementar en todos los ámbitos con mayor de nivel de seguridad sobre sus procesos.

La división presentada anteriormente fue construida desde una perspectiva de alto nivel, por lo que fácilmente pueden existir investigaciones en la literatura que abordan dos o más campos mencionados. De todas maneras, existen otros estudios, basados en análisis más cuantitativos [13] que muestran que a lo largo de los años la reutilización de software sigue presente como campo de investigación debido a su dificultad para gestionarse y su importancia para empresas con familias de productos.

2.2.2 ¿Cuáles son las estrategias de creación de SPL?

Si bien la reutilización es una práctica que se ha abarcado en diferentes campos, desde la reutilización de requerimientos resumidos a través de historias de usuario, hasta la modificación de modelos altamente complejos con el fin de no empezar desde cero. El desarrollo de una SPL busca reducir los tiempos de entrega de software generados por desarrollos de un solo uso, trabajo en bajos niveles de abstracción, inmadurez en los procesos de negocio, el aumento de la demanda e incremento de construcción de sistemas monolíticos complejos [2]. Teniendo en cuenta lo anterior, se pueden definir tres maneras de construir una SPL: proactiva, es el desarrollo desde cero; reactiva, construir componentes reutilizables a medida que avanza el tiempo; y extractiva, reutilizar elementos de proyectos de software ya existentes [3]. En particular, la reutilización se puede realizar de manera organizada a través del desarrollo de la ingeniería de dominio [1] o

desorganizada, por medio de técnicas cotidianas como clone-and-own o copy-paste-modify [4]. En cualquiera de los casos, si se tienen en cuenta que la mayoría de los sistemas son altamente acoplados, aún es un reto reutilizar componentes de manera óptima [10].

2.2.3 ¿Qué es la estrategia extractiva?

Es un proceso de reingeniería (Ilustración 3) que busca extraer elementos de software en proyectos de software existentes, de manera que queden independientes y funcionales para que sirvan nuevamente en otro desarrollo o bien, que quede funcional para una nueva línea de productos.

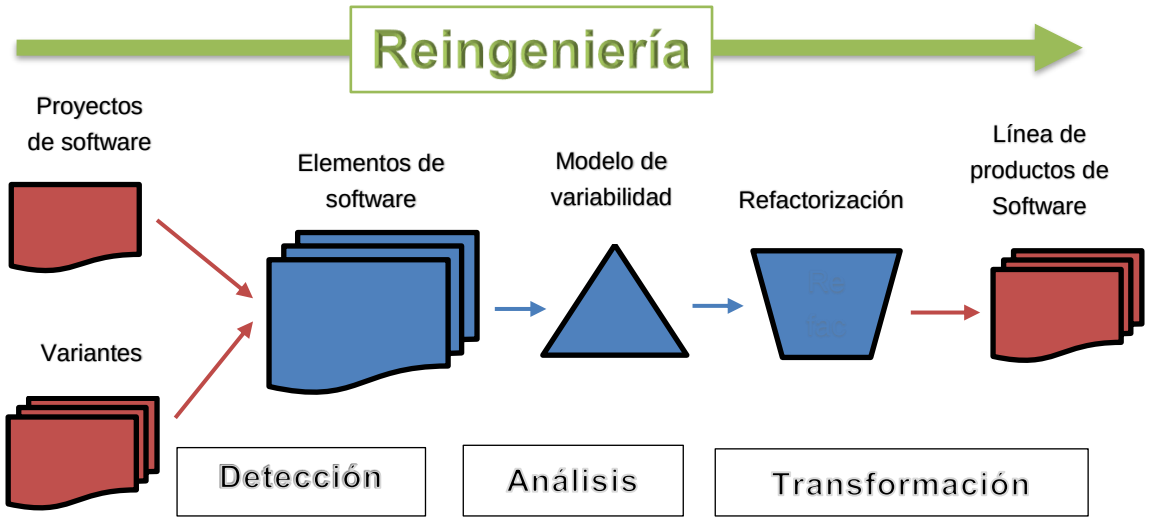

Ilustración 3 Proceso de extracción de componentes para el desarrollo de una Línea de Productos de Software (LPS)

De manera general, el procedimiento consta de varias fases [15]:

- **Detección de características (feature location):** Búsqueda de features que se encuentran dentro del proyecto de software representados de alguna manera: métodos, variables, clases, entre otros.
- **Análisis (feature analysis):** Representación de un diagrama de características o un modelo de variabilidad capaz de representar las características encontradas.

- **Transformación o derivación de productos (product derivation):** Cambios en el código por medio de técnicas de refactorización para que cada elemento sea manipulable y permita su adaptación a una línea de productos de software.

2.3 Identificación de features: Contexto y técnicas.

Es el proceso por el cual, se hace la recolección y lectura de todos los elementos de software de interés. Con lo anterior, se realiza la construcción de las relaciones que existan entre las funcionalidades, los features existentes, y la estructura en sí [29]. La ilustración 4, muestra los tres tipos de estrategias que se pueden apreciar en la literatura [30]:

- **Análisis estático:** Se realiza la representación de diferentes formas de los elementos de software ya existentes para facilitar los procesos de localización y realización de trazabilidad.
- **Análisis dinámico:** Busca la identificación de relaciones por medio de pruebas de ejecución guiadas.
- **Recuperación de información (Information retrieval - IR):** Es la utilización de técnicas de similitud de texto, aunque estás técnicas pueden ser aplicadas tanto para análisis estático como dinámico.

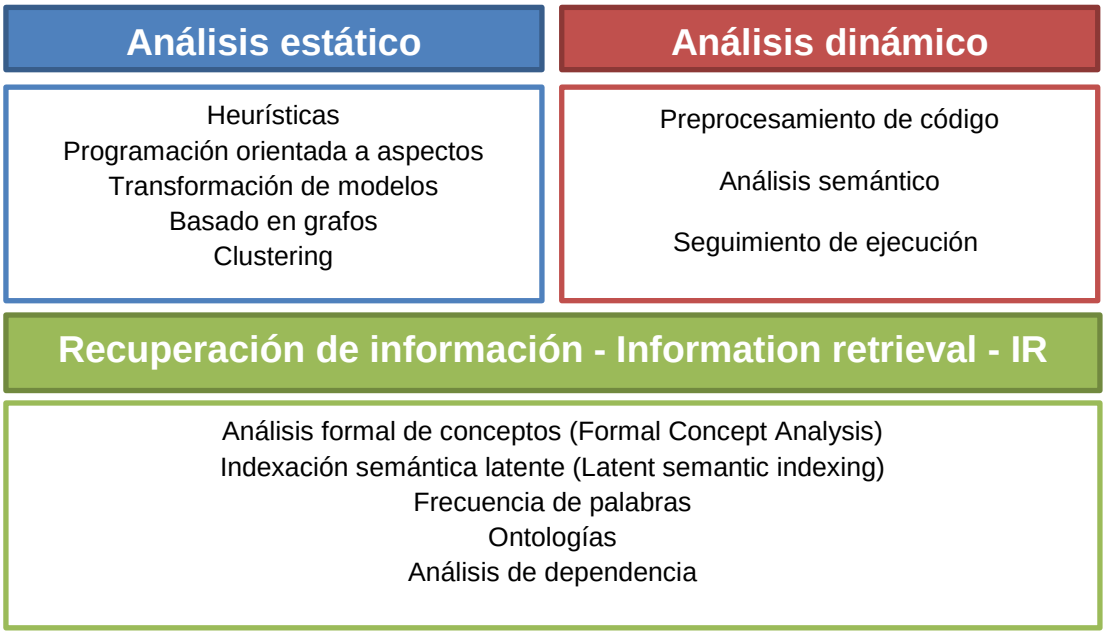

Ilustración 4 Técnicas de feature location clasificadas por el tipo de estrategia.

En algunos casos, se tiene en cuenta la participación del usuario para la fabricación de resultados finales, a lo que se le llama salidas guiadas, o bien, salidas planas haciendo referencia a una lista de artefactos de manera no organizada [29]. En la ilustración 4 se puede ver un listado de técnicas más comunes para cada tipo de estrategia de identificación de features.

2.4 Tipos de enfoques: Manual versus automático

Adicional a las divisiones planteadas previamente, se puede estructurar una organización diferente enfocándose en la forma de implementación de la técnica de identificación de features. Aunque existen muchas técnicas, a continuación, se muestran las que se han considerado más relevantes.

2.4.1 Enfoques manuales

Para la extracción de características existen tres tendencias principales:

- **Por medio de herramientas de preprocesamiento de código [17]:** Marcación por medio de notaciones que le permitan diferenciar al compilador que segmentos deben empaquetarse.
- **Programación orientada a aspectos [18]:** Utilización del paradigma de programación orientada a aspectos para que le permitan seleccionar los elementos de software que se compilarán y harán parte del entregable por medio de estructuras definidas.
- **Programación orientada a características [19]:** Por medio de la refactorización de código se hace la separación de features teniendo en cuenta sus interacciones e influencias dentro del código, por medio de estudio de estructuras de código para que queden independientes.

2.4.2 Enfoques automáticos

Se han encontrado distintos algoritmos que buscan recuperar la relación entre cada elemento de software y el proyecto de manera automatizada; entre los más comunes se encuentran:

- **Formal Concept Analysis [20]:** Es una técnica que utiliza la teoría de grafos para recuperar la relación que existe entre cada uno de los elementos de software y las características descritas dentro del proyecto.
- **Information retrieval [21]:** Son técnicas usadas para la recuperación de información que tiene base matemática y buscan recuperar la relación entre una cadena de entrada y un documento
- **Vector Space Model [22][23]:** Es un modelo que realiza la representación de documentos por medio de vectores que identifican el texto con una posición en el espacio, lo que permite realizar trabajar con texto de manera más directa.

2.5 Clone-and-own

Como se muestra en la ilustración 5, es una técnica de reutilización de elementos de software que consisten en la utilización de fragmentos de código fuente y modificarlo acorde a nuevas necesidades [14]. Su utilización nace de la reutilización de software ya existente para la construcción de nuevos proyectos y adaptarlo a los nuevos

requerimientos [32]. Lamentablemente, dicha actividad tiene varias debilidades, ya que se basa en gestionar los componentes de manera independiente siguiendo el ciclo de desarrollo de software de cualquier producto, por lo que en términos técnicos y financieros se convierten en elementos que deben mantenerse [31].

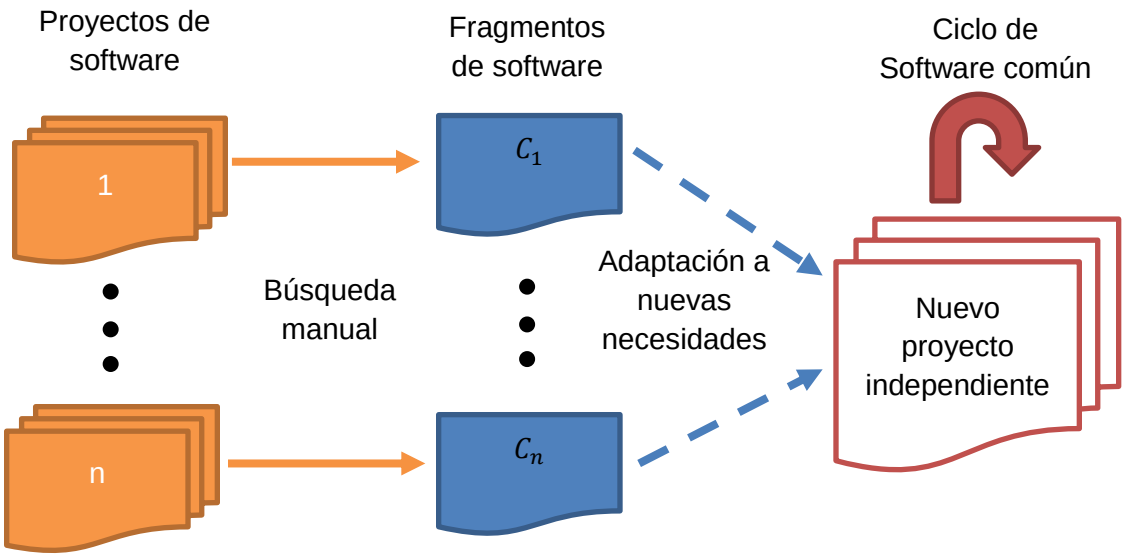

Ilustración 5 Proceso de copiar, pegar y modificar elementos separados de otros proyectos.

2.6 Casos de estudio

A lo largo de los años se han creado múltiples casos de estudio en torno a la creación de SPL [24]. Entre los casos más usados y significativos, basados en la literatura se encuentran:

2.6.1 Mobile Media [35]

Desarrollado en Java y aspecJ, originado de extender un feature ya estructurado, mobilephoto, y construyendo features obligatorios y opcionales adicionales (Ilustración 5). Donde por medio del paradigma de la programación orientada a aspectos se construyó una SPL. Con un tamaño de cerca de 3 KLOC. (Lines of code – Líneas de código) por escenario y con la particularidad de ser un caso industrial, utilizado en Nokia; Siemens y RIM [36].

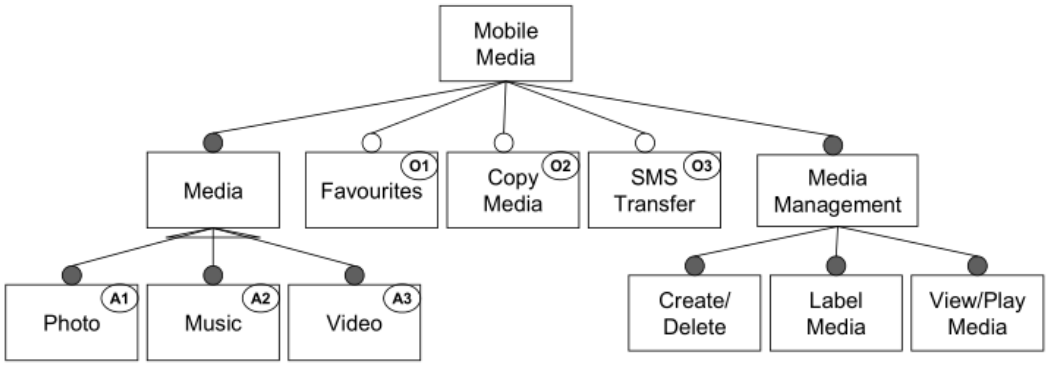

Ilustración 6 Diagrama de variabilidad simplificado del caso de estudio Mobile Media [36].

2.6.2 ArgoUML SPL

Desarrollado en Java y originado del proyecto ArgoUML, fue construido por medio de notaciones de código que se encargan de diferenciar que segmentos de código hacen referencia a cada feature [16]. Particularmente, fue diseñado con la ayuda de una herramienta de procesamiento externa javapp [1], la cual acepta directivas similares a las de C o C++ cómo #ifdef #ifndef y #else. Es la evidencia más sólida, por tamaño y relevancia, de la construcción de una SPL a partir de un proyecto monolítico.

2.7 Resumen

En este capítulo se muestran conceptos e historia de SPL, así como las estrategias que existen para su implementación. En el camino se fue mostrando cómo la detección de features presenta una dificultad mayor a la esperada. A continuación se aborda dicha problemática centrándonos en un caso de estudio: ArgoUML SPL.

[1] Disponible en http://www.slashdev.ca/javapp

3. Estudios realizados de ArgoUML SPL en sus pruebas

3.1 Motivación

Aprovechando el entendimiento del campo de SPL, y dimensionando la dificultad que existe para detectar features, se realizó este capítulo. Se presenta el desarrollo como un estudio sistemático, ya que esto sirve como base para recopilar información existente y relevante en torno al caso de estudio seleccionado. El planteamiento de preguntas que buscan identificar los aspectos más relevantes relacionados a la comparación de algoritmos sirven como eje fundamental para realizar en los siguientes capítulos. Para eso se tomó como base el procedimiento que se sigue en la literatura [34]. Los resultados presentados en este capítulo se basan en las tablas que se presentan en el Anexo A.

3.2 ArgoUML SPL: Descripción general

Construido en 2011, con la principal idea de brindar a la comunidad un caso de estudio con un tamaño mayor a los existentes hasta la fecha. Cada una de las variantes está constituida por un mínimo de 110KLOCs y un máximo de 148KLOCs. En la ilustración 6 se muestra que el proyecto tiene 8 features opcionales: LOGGING, COGNITIVE SUPPORT, STATE, ACTIVITY, USE CASE, COLLABORATION y DEPLOYMENT; y 2 obligatorios: DIAGRAMS, CLASS (Su descripción se puede detallar en la tabla 1); lo que hace que pueda tener 256 variantes. (Este cálculo se hace elevando dos al número total de features opcionales[6], ya que por cada feature opcional existen dos posibilidades de configuración)

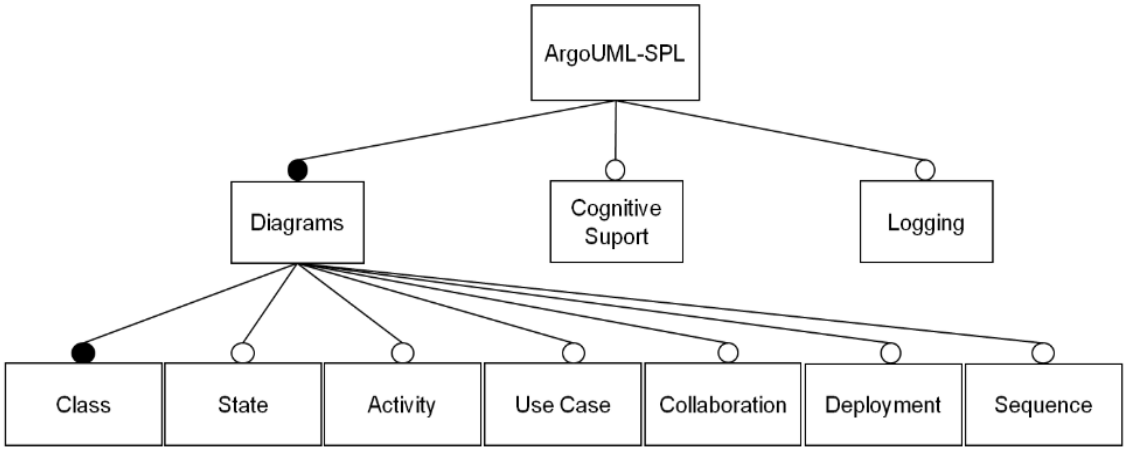

Ilustración 7 Diagrama de variabilidad de ArgoUML-SPL [24].

Feature	Descripción	LOC
State	State diagrams are used to describe the behavior of a system. State diagrams describe all of the possible states of an object as events occur. Each diagram usually represents objects of a single class and track the different states of its objects through the system.	3917
Activity	Activity diagrams describe the workflow behavior of a system. Activity diagrams are similar to state diagrams because activities are the state of doing something. The diagrams describe the state of activities by showing the sequence of activities performed.	2282
Use Case	A use case is a set of scenarios that describing an interaction between a user and a system. A use case diagram displays the relationship among actors and use cases. The two main components of a use case diagram are use cases and actors.	2712
Collaboration	Collaboration diagrams are used to show how objects interact to perform the behavior of a particular use case, or a part of a use case. Along with sequence diagrams, collaborations are used by designers to define and clarify the roles of the objects that perform a particular flow of events of	1579

	a use case. They are the primary source of information used to determining class responsibilities and interfaces.	
Deployment	A deployment diagram models the run-time architecture of a system. It shows the configuration of the hardware elements (nodes) and shows how software elements and artifacts are mapped onto those nodes.	3147
Sequence	Interaction diagrams model the behavior of use cases by describing the way groups of objects interact to complete the task. Sequence Diagram displays the time sequence of the objects participating in the interaction. This consists of the vertical dimension (time) and horizontal dimension (different objects).	5379
Cognitive	Simple agents that continuously execute in a background thread of control. They analyze the design as the designer is working and suggest possible improvements. These suggestions range from indications of syntax errors, to reminders to return to parts of the design that need finishing, to style guidelines, to the advice of expert designers.	16319
Logging	The purpose of debug log and trace messages is to provide a mechanism that allows the developer to enable output of minor events focused on a specific problem area and to follow what is going on inside ArgoUML.	2159

Tabla 1 Tabla con la descripción de cada Feature de ArgoUML-SPL y sus líneas de código [16].

A diferencia de los anteriores casos de uso, este presenta un mayor reto para la implementación de técnicas de identificación de características por su gran tamaño, ya que se asemeja más a un escenario real [26].

3.3 Preguntas de investigación

A continuación, con el propósito de conocer el estado actual de las investigaciones alrededor del caso de estudio seleccionado y para poder desarrollar los siguientes capítulos se plantean las siguientes preguntas de investigación:

- **¿Cuántas investigaciones han utilizado ArgoUML SPL cómo caso de prueba para evaluar técnicas de identificación de características?** Debido a que ArgoUML[2] es una herramienta que se utilizaba comúnmente para modelamiento de proyectos, existe la dificultad de saber la relevancia de dicho caso de estudio; adicionalmente, la cantidad de investigaciones que han utilizado el caso dentro de sus pruebas.
- **¿Cuáles son las técnicas más comunes?** Basado en los resultados de la pregunta anterior, es importante obtener las técnicas más utilizadas para identificación de características.
- **¿Cuáles han sido las técnicas de medición de resultados?** Así como las técnicas son importantes, es indispensable saber cuál fue la forma en que se realizó la manera de evaluar el rendimiento del enfoque utilizado.
- **¿Cuáles han sido las variantes utilizadas?** Considerando solo aquellos que utilizaron el caso de estudio, es importante conocer cuáles fueron las variantes más comunes, con el fin de saber si existe una forma de comparación de algún tipo.

3.4 Proceso de selección

3.4.1 ¿Cuál es el foco de la investigación?

La metodología PICO (Acrónimo en inglés – Population, Intervention, Comparison, OutCome) es el proceso por el cual se realiza la selección de los elementos de interés para la revisión basado en cuatro aspectos [48]:

- **Población:** Personas en el área de líneas de productos de software

[2] Disponible aquí http://argouml-downloads.tigris.org/

- **Intervención:** Debe estar presente el uso del caso de estudio de ArgoUML-SPL.
- **Comparación:** Se desea conocer las principales técnicas usadas, las variantes utilizadas en las pruebas y la forma de medición.
- **Salidas:** El mayor interés es saber cuántos investigadores han utilizado ArgoUML-SPL en pruebas relacionadas con SPL

3.4.2 Ecuación de búsqueda y motores de búsqueda

Basado en los datos de interés definidos se establecieron varias ecuaciones de búsqueda (ilustración 8) las cuales no incluyeran resultados que no interesarán en la búsqueda, cómo la utilización de ArgoUML para graficar diagramas UML; adicionalmente, que no excluyeran estudios que pudieran darle valor a la investigación cómo estudios que utilizaran el proyecto de interés dentro de sus pruebas. Por lo que finalmente se escogió una ecuación sencilla que acogiera los términos principales (ArgoUML y SPL) y un filtro adicional de tiempo (fecha inicial es el año en el cuál fue publicado el artículo de ArgoUML SPL)

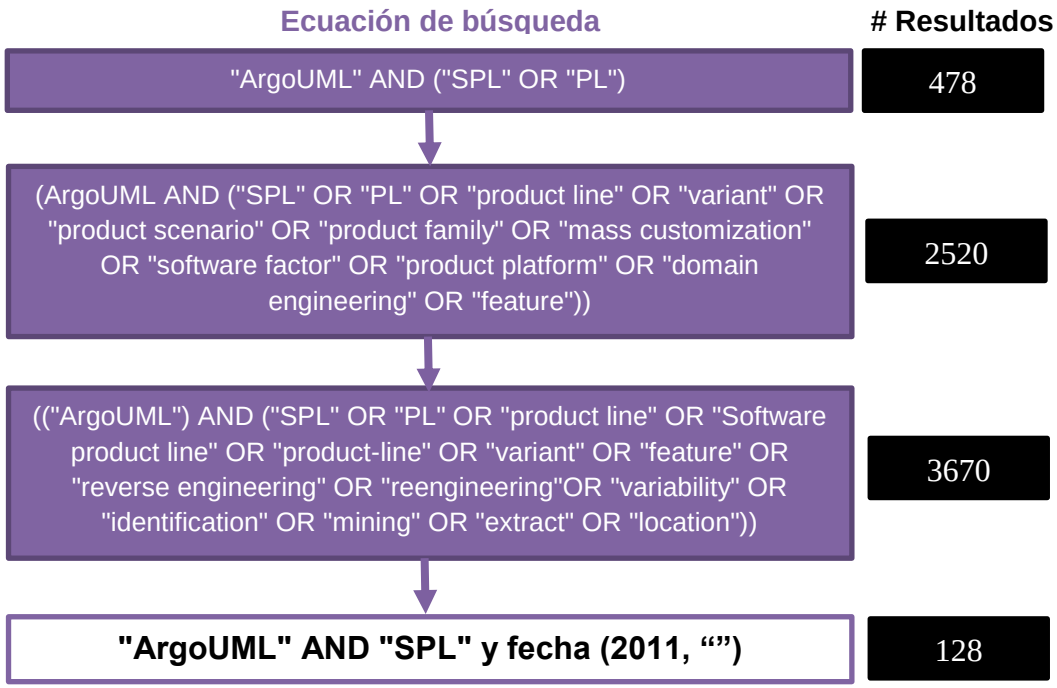

Ilustración 8 Proceso de refinación de la ecuación de búsqueda. Cantidad de resultados basado en el motor Google Scholar.

Para el desarrollo del estudio se tuvieron en cuenta los siguientes motores de búsqueda: Google Académico, IEEE, Scopus y Web of Science. En todas los anteriores se realizó la búsqueda por medio de la ecuación y adicionalmente por medio del filtro que permite buscar los artículos que han citado un artículo en particular (forward snowballing [49]), como lo muestra la tabla 2.

	Por Ecuación	**Por referencia**
Google Scholar	128	80
IEEE Explore	5	21
Scopus	9	42
Web of Science	5	22
Springer	11	0
Total	158	165

Tabla 2 Número de investigaciones encontradas en cada motor de búsqueda filtrados por ecuación y por referencia.

3.4.3 Criterios de inclusión y exclusión

Existieron varios pasos de exclusión que fueron llevados a cabo (Ilustración 10):

- **Duplicidad de documentos:** Debido a la cantidad de artículos de las diferentes bases de datos, se encontró bastante duplicidad de investigaciones (Ilustración 8).

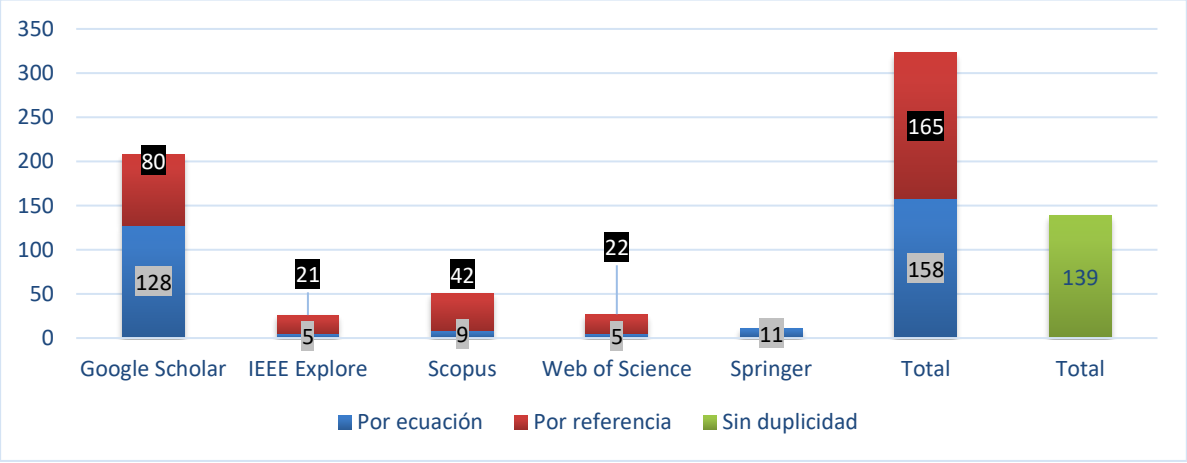

Ilustración 9 Distribución de artículos académicos.

- **Dificultad de idioma:** Se tuvieron en cuenta artículos que estén escritos en inglés por la universalidad que tiene.
- **Relación con la investigación de interés:** Debido a como se planteó inicialmente la investigación, se hizo una lectura de los artículos restantes, y se revisó principalmente la introducción y la sección de resultados en donde se pudiera encontrar el motivo por el que fue referenciado o utilizado ArgoUML SPL. Algunos de estos, mencionaban la herramienta en la sección de trabajo futuro, o utilizaban ArgoUML para el modelamiento de un proyecto y hacían su aplicación en un campo de SPL diferente a la identificación de features, estos se omitieron ya que no hacen parte del foco de está investigación.
- **Repetición de investigaciones:** Es entendible que entre los motores de búsqueda se encuentren las mismas investigaciones, por lo que unificar esto redujo el número de investigaciones encontradas. Además, no se tuvo en cuenta cualquier documento encontrado en los motores de búsqueda que fueran presentaciones o short papers. Es decir, solo artículos completos que se hayan presentado formalmente en conferencias o revistas.

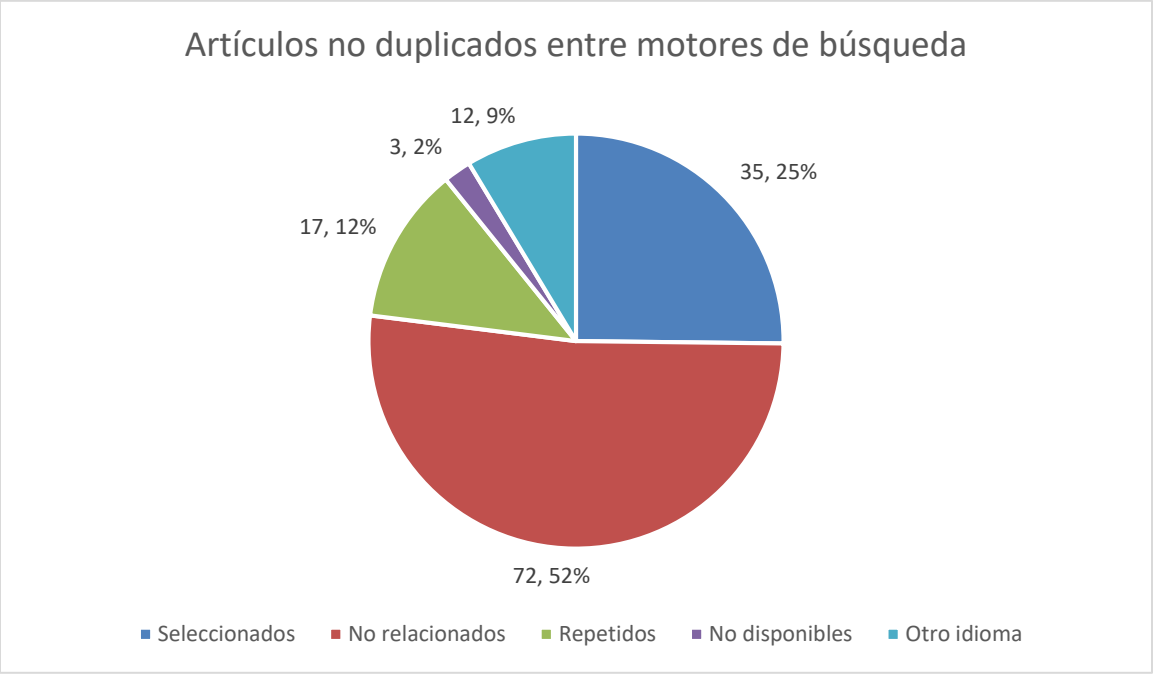

Ilustración 10 Cantidad de artículos por filtro de exclusión.

3.5 Resultados

Se analizaron 35 artículos que pasaron los filtros previamente mencionados.

3.5.1 ¿Cuántos han utilizado ArgoUML SPL en sus pruebas de identificación de features?

Como se muestra en la ilustración 10, a pesar de que existen múltiples artículos que utilizan ArgoUML SPL, no todos son utilizados para evaluar el rendimiento de algoritmos de identificación de features. Algunos lo utilizaban como herramienta de soporte para obtener el impacto en la refactorización de código [39] o para sugerencias de nombres de features basado en el contenido [40], entre otros. Por lo que aquellos que utilizan ArgoUML SPL como caso de prueba para la identificación de características son 24.

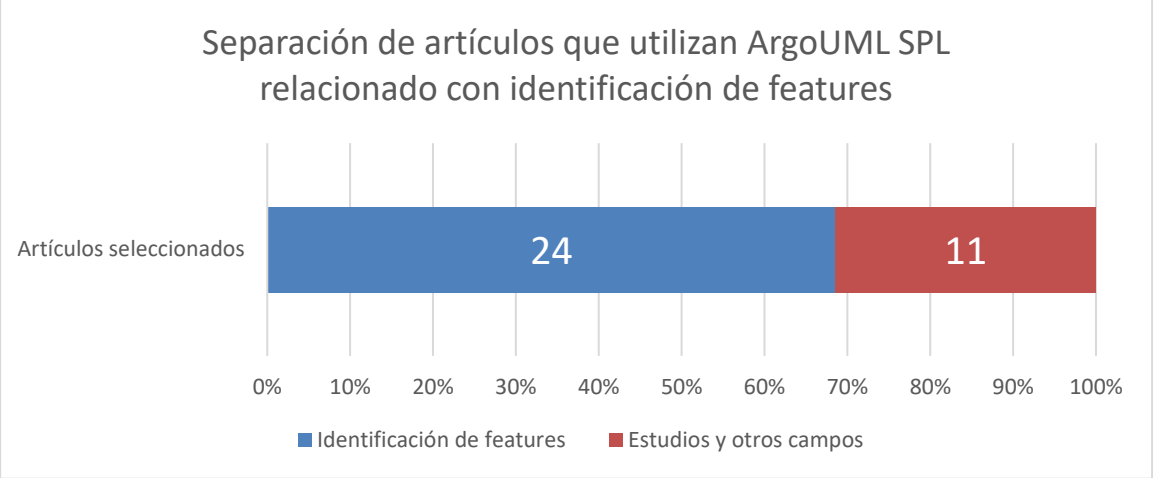

Ilustración 11 Clasificación de artículos que utilizan ArgoUML SPL

3.5.2 ¿Cuáles son las técnicas más comunes?

Durante la revisión de artículos se evidencia que 13 investigaciones hacen uso de indexación semántica latente (LSI – Latent Semantic Indexing), y de igual forma 13 hacen uso de análisis formal de conceptos (FCA – Formal Concept analysis). Los demás, se dividen en hacer uso de algoritmos genéticos, dependencias de grafos, utilización de árboles sintácticos, heurísticas, diferentes variaciones de recuperación de información (LSA, VSM – Vector Space Model).

3.5.3 ¿Cuáles son las formas de medición de resultados?

La mayor técnica de medición usada fue Precision and Recall (Ilustración 12). Aunque esta métrica tiene una base teórica, existen diferencias entre las investigaciones que no permiten comparar los resultados encontrados, ya que la forma en que recuperan la información la expresan por medio de diferentes formas y niveles de granularidad: Clases, número de puntos de variación, números de clases relacionadas separadas por features opcionales y obligatorios, entre otros.

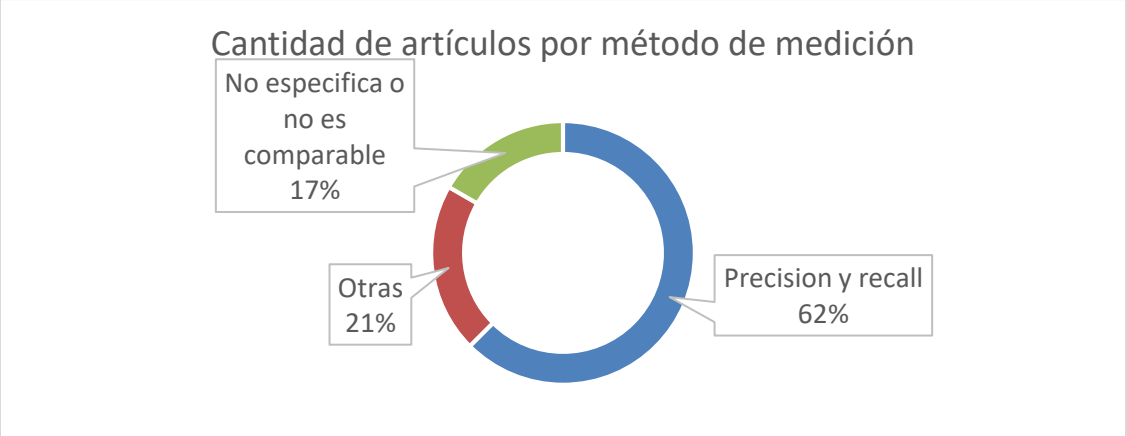

Ilustración 12 Cantidad de artículos divididos por el método de medición utilizado.

3.5.4 ¿Cuáles han sido las variantes utilizadas?

Se puede ver una distribución porcentual de los estudios evaluados en la ilustración 13, los cuales poseen los siguientes datos: 2 no dicen que variantes se utilizaron, 3 mencionan las features de interés para la investigación, 5 mencionan la cantidad de variantes utilizadas pero no describen el conjunto de variantes, 4 mencionan el conjunto de variantes, 1 generó las variantes de manera aleatoria, 2 mencionan que utilizaron todas las variantes posibles, y 7 utilizaron el conjunto de variantes establecido en el artículo inicial [16].

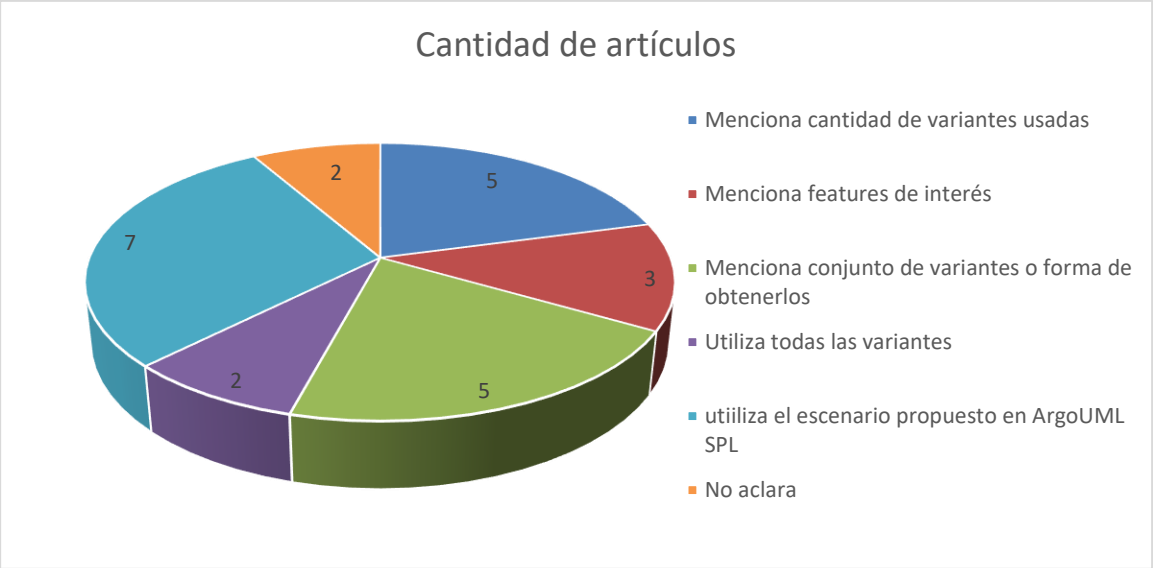

Ilustración 13 Cantidad de artículos separados por la forma en que describe las variantes utilizadas.

3.6 Análisis complementarios

3.6.1 Investigación a lo largo del tiempo

Dentro de los artículos encontrados, se evidencia que ha sido constante la investigación que se ha hecho utilizando dicho caso de estudio desde su surgimiento hasta hoy.

Ilustración 14 Investigaciones que utilizan ArgoUML SPL como caso de estudio a lo largo de los años.

3.6.2 Técnicas de identificación de features

En relación al numeral 2.4.2, se puede ver una tendencia al uso de FCA y LSI; ya que más del 50% de los artículos que han desarrollado técnicas de identificación de features han utilizado estos algoritmos, esto se puede ver resumido en la ilustración 15.

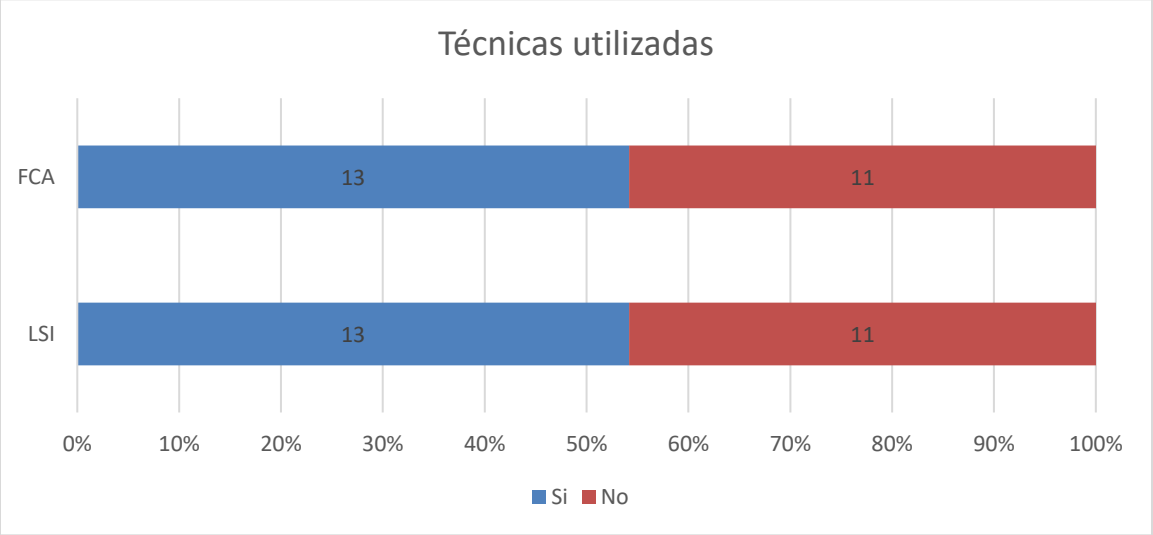

Ilustración 15 Porcentaje de artículos que han utilizado FCA y LSI como método de identificación de features.

3.6.3 Entradas y salidas

Es importante saber cuáles eran las entradas que se necesitaban en cada investigación y la salida esperada.

- **Entradas:** En 17 de 24 investigaciones se utilizó como recurso el código fuente.
- **Salidas:** En este caso, existió una mayor diversidad de salidas esperadas: listas de features con conjuntos de clases asociados, modelos de variabilidad, grupos de clases separados por pertenecer a features opcionales u obligatorios, entre otros.

3.6.4 Nivel de granularidad

Dentro de las posibilidades que existían la mayoría de las investigaciones desarrollaban un nivel de profundidad al nivel de la definición de nombre de clases, alrededor de 18 de

24, aunque también algunos complementaban dicha profundidad acompañado de declaraciones de métodos, o de atributos.

3.7 **Resumen**

En al anterior capítulo se mostró un estudio sistemático. A continuación se presenta cuál fue la importancia de cada una de las preguntas:

- **¿Cuántas investigaciones han utilizado ArgoUML SPL como caso de prueba para evaluar técnicas de identificación de características?** Se encontró que existen veinticuatro estudios que se focalizan en identificación de características utilizando el caso de estudio seleccionado. Encontrar está respuesta ayudó a identificar cuáles eran las investigaciones de interés para establecer una verdadera comparación entre estudios.
- **¿Cuáles son las técnicas más comunes?** Al revisar detalladamente las veinticuatro investigaciones se encontró que las técnicas más usadas son: LSI y FCA. Dicho resultado sirvió para saber que algoritmos seleccionar para proponer una técnica que sirva como línea base.
- **¿Cuáles han sido las técnicas de medición de resultados?** Si bien la mayoría de los estudios utilizaron el estándar (Precission and Recall), la forma de calcular estas métricas variaba entre cada investigación. De esta forma se evidenció la dificultar para comparar los resultados de las técnicas. Lo que lleva también a que sea difícil concluir cuál es la mejor forma de recuperar características hasta el momento.
- **¿Cuáles han sido las variantes utilizadas?** Teniendo en cuenta la diversidad que existen entre las familias de productos, cada estudio utilizo diferentes variantes que se adecuaban a su propia investigación. Lo anterior sirve como justificación para saber que aunque existieran hipotéticamente métricas unificadas, la diversidad de escenarios genera un obstáculo adicional para comparar los estudios encontrados en la primera pregunta.

Así es como todas las preguntas planteadas sirven como razones de peso para promover la realización del siguiente capítulo.

4. Unificando ArgoUMLSPL: Benchmark

4.1 Motivación

Gran parte de este capítulo se basa en el trabajo publicado en SPLC 2018 (Software product line conference) [26]. Teniendo en cuenta todo el trabajo realizado durante los anteriores capítulos es importante aclarar que: Existen varios casos de uso, sin embargo, el más usado es ArgoUML SPL [24]. Este fue desarrollado por medio de herramientas de preprocesamiento de código que permiten realizar anotaciones a lo largo del proyecto para marcar segmentos de código que pertenecen a un feature en particular [25]. El uso de ingeniería inversa para la construcción de SPL basado en un caso ya existente, es algo que se ha intentado hacer de diferentes formas [27]. Su aporte a la comunidad académica sirvió como inicio para el desarrollo de pruebas enfocadas en localización de features, ayudando a la construcción de escenarios de prueba más cercanos a la realidad [26].

A continuación, utilizando las respuestas obtenidas del capítulo anterior, se realiza la construcción de una forma única de comparar los estudios de recuperación de características: Un groundtruth. Su construcción tiene el principal objetivo de aprovechar todos los estudios realizados en un área para establecer una única medida de forma que estandarice la manera en que se comparan los resultados de las investigaciones.

4.2 Problemas actuales

A pesar de la importancia de ArgoUML SPL, existen varios factores a tener en cuenta. Ya que el contexto en el que se desarrollaron las pruebas con dicho caso de estudio tiene diferentes variables evidenciadas en la ilustración 16.

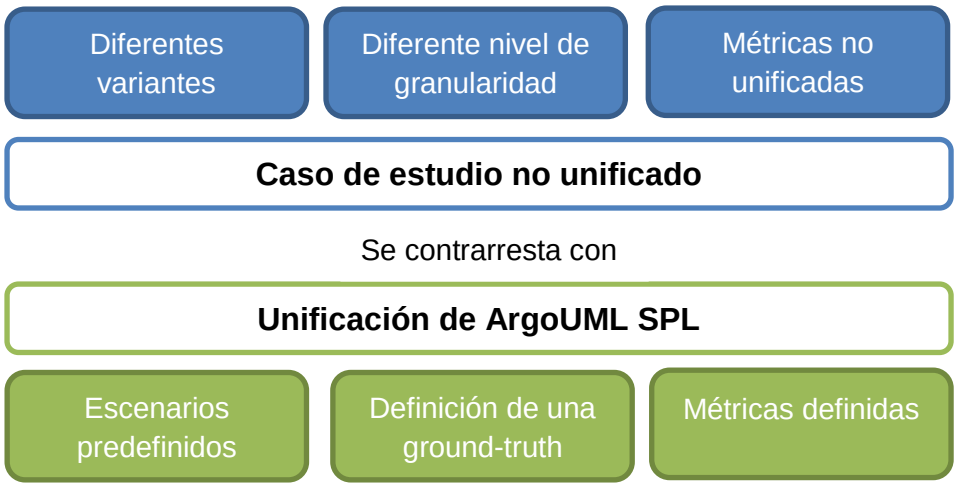

Ilustración 16 Espejo de problemáticas y soluciones alrededor de identificación de features.

- **Diferencias en los escenarios utilizados:** El estudio realizado en el capítulo 2, muestra las diferentes variantes. Debido al gran tamaño que tiene dicho caso de estudio [16] es normal que cada investigación haya seleccionado por su cuenta un grupo aleatorio de variantes. Además, aun dejando a un lado este factor, existe una dificultad por encontrar los elementos de código asociados a las interacciones entre features [28].
- **Diferentes niveles de granularidad:** El caso de estudio utiliza anotaciones de código en diferentes tipos según los de segmentos de código: declaraciones de variables, declaraciones de métodos, declaraciones de clases, entre otros.
- **Métricas no unificadas:** Cómo consecuencia, de los dos factores anteriores y a la variabilidad de salidas esperadas en cada investigación, así mismo existe diversidad en las métricas definidas en cada estudio, lo que dificulta la comparación para la evaluación del desempeño.

4.3 Utilización de la prueba

Se desarrolló un proyecto capaz de tener en cuenta los aspectos necesarios para contrarrestar los efectos de los problemas encontrados.

4.3.1 Generación de escenarios predefinidos

La creación de escenarios predefinidos permite establecer un set de pruebas que sea usado para comparar resultados independientemente del enfoque que posea cada investigación. Cada escenario fue desarrollado con un objetivo particular [26]:

- **Escenario original:** Una variante que contiene todos los features opcionales presentes tal y como se presentaba en el proyecto del cual se basaron para la construcción del caso de estudio [16].
- **Escenario tradicional:** Posee diez diferentes variantes que están contenidas dentro de las investigaciones que han utilizado el caso de estudio dentro de sus pruebas como se refleja en la ilustración 12.
- **Escenario de interacciones de parejas:** Un conjunto de nueve variantes que se generan utilizando el algoritmo T-Wise de featureIDE[3] los cuales representan una diversidad diferente a la que se encuentra en un escenario aleatorio.
- **Escenarios aleatorios:** Representado a través de 11 escenarios con 2, 3, 4, 5, 6, 7, 8, 9, 10, 50 y 100 variantes respectivamente. Su objetivo principal es determinar qué tan sensible es la técnica de identificación de features según el número de variantes [38].
- **Escenario con todas las variantes:** Se presenta un escenario que contiene las 256 variantes posibles en el caso de estudio. Si bien, no es un caso real, la principal intención de este escenario es medir la escalabilidad de la técnica.

Con el fin de acercar los escenarios generados a partir del caso de estudio a casos reales, se procede a eliminar pistas en el código que muestren el proceso desarrollado para la construcción de la línea de la siguiente forma: Se eliminan todos los comentarios de código

[3] Disponible en http://www.featureide.com/

que están directamente relacionados con las declaraciones que ayudan a la construcción de variantes dentro del caso de estudio. Sin embargo, esto no significa que se eliminan comentarios que no están relacionados con la construcción de la variante, por el contrario, aportan una explicación al código que se encuentra desarrollado. Por ejemplo, en la ilustración 17 dentro de un recuadro naranja se ven los comentarios relacionados con la construcción de la variante los cuales se eliminan, pero el código marcado en verde no será eliminado en la variante dentro de los escenarios ya que estos no están relacionados directamente con ningún feature. Con lo anterior se ilustra que la funcionalidad del proyecto original no se ve afectada.

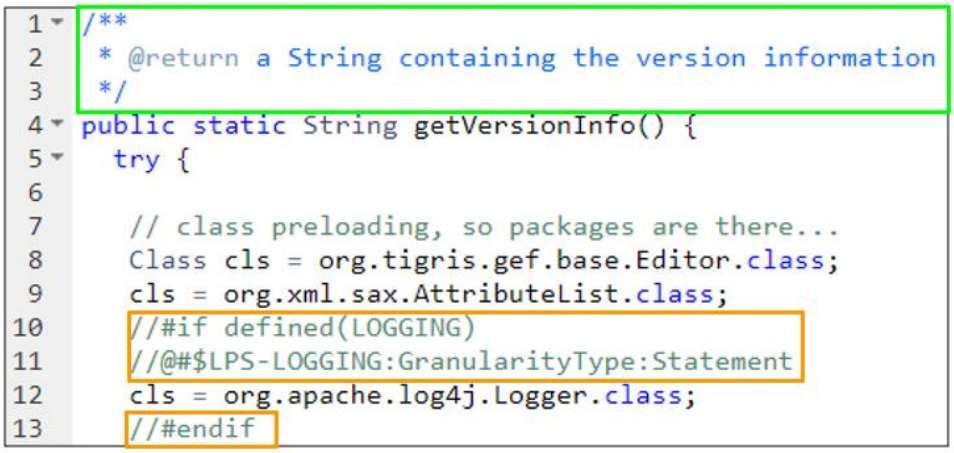

Ilustración 17 Segmento de código que ilustra los dos tipos de comentarios que se encuentran dentro del proyecto ArgoUML-SPL [16].

4.3.2 Formato de entrada

Una vez definidos los escenarios de prueba se hizo evidente la necesidad de la creación de un formato único de entrada que tuviera en cuenta los niveles de granularidad que maneja el caso de estudio. De esta forma establecer una forma de comparar resultados para el desarrollo de este trabajo. Por dicho motivo, se define la manera en que deberían mostrarse los elementos de software que estén asociados a un feature por medio de diferentes tipos de relación:

- **Declaración de una clase:** Se escribe la declaración de la clase si está presente en su totalidad cuando un feature está presente dentro de una variante (Ilustración 18).

```
1   //#if defined(COGNITIVE)
2   //@#$LPS-COGNITIVE:GranularityType:Package
3
4   /*⟺*/
9
10  package org.argouml.uml.cognitive;
11  import org.argouml.cognitive.Critic;...
12
13  /*⟺*/
23  public class UMLToDoItem extends ToDoItem {⟺}
166 //#endif
```

Ilustración 18 Segmento de código de ArgoUMLSPL con la declaración ifdef marcada en color rojo. Código tomado del proyecto ArgoUML-SPL. [16].

- **Declaración de una clase adicionando la etiqueta Refinement:** Se escribe si existen segmentos de código dentro de una clase como declaraciones de variables o librerías que se agregan o eliminan por la presencia de un feature (Ilustración 19).

```
1   package org.argouml.uml.diagram.ui;
2
3   import javax.swing.Action;
4   import javax.swing.Icon;
5   import javax.swing.JSeparator;
6   import javax.swing.SwingUtilities;
7
8   //#if defined(COGNITIVE)
9   //@#$LPS-COGNITIVE:GranularityType:Import
10  import org.argouml.cognitive.Designer;
11  import org.argouml.cognitive.Highlightable;
12  import org.argouml.cognitive.ToDoItem;
13  import org.argouml.cognitive.ToDoList;
14  import org.argouml.cognitive.ui.ActionGoToCritique;
15  //#endif
16
17  /*⟺*/
28  public abstract class FigEdgeModelElement
29      extends FigEdgePoly
30      implements
31          VetoableChangeListener,... {⟺}
```

Ilustración 19 Segmento de código que muestra en rojo las líneas que variarán dependiendo de la feature seleccionada. Código tomado del proyecto ArgoUML-SPL. [16].

Es decir, si existe declaraciones ifdef que afectan la integridad de la clase, estos pueden verse reflejados también en las clases que extiende o implementa. Para la ilustración 19 se deberá mostrar de la siguiente forma: org.argouml.uml.diagram.ui.FigEdgeModelElement Refinement

- **Declaración de un método:** Si existe un método que en su totalidad está presente cuando una feature se encuentra seleccionada en la variante. Dicho comportamiento no afecta la declaración de la clase, por lo que son dos relaciones diferentes. Por ejemplo, si un método se encuentra entre una etiqueta ifdef en su totalidad se deberá mostrar, basándose en la ilustración 20, de la siguiente forma: org.argouml.util.Tools logVersionInfo()

```
1   //#if defined(LOGGING)
2       //@#$LPS-LOGGING:GranularityType:Method
3       //@#$LPS-LOGGING:Localization:EntireMethod
4       /**
5        * Print out some version info for debugging.
6        */
7       public static void logVersionInfo() {⟷}
21      //#endif
```

Ilustración 20 Método marcado en rojo para mostrar las líneas de código que se ven afectadas por la declaración ifdef. Código tomado del proyecto ArgoUML-SPL. [16].

- **Declaración de un método adicionando la etiqueta refinement:** Análogamente a como sucede con la etiqueta adicionada en la declaración de clase (Ilustración 21), si existe segmentos de código marcado como declaraciones de variables o desarrollo lógica dentro de un método se deberá mencionar el método acompañado de la etiqueta Refinement.

```
1  /**
2   * @return a String containing the version information
3   */
4  public static String getVersionInfo() {
5    try {
6
7     // class preloading, so packages are there...
8     Class cls = org.tigris.gef.base.Editor.class;
9     cls = org.xml.sax.AttributeList.class;
10    //#if defined(LOGGING)
11    //@#$LPS-LOGGING:GranularityType:Statement
12    cls = org.apache.log4j.Logger.class;
13    //#endif
```

Ilustración 21 Asignación de variable marcado en rojo dentro de un método entre clausulas ifdef. Código tomado del proyecto ArgoUML-SPL. [16].

Sumando todos los tipos de relación entre los features y los elementos de software existentes en el caso de estudio, se pueden encontrar: 439 Clases completas, 388 partes de clases, 44 métodos completos y 871 partes de métodos para un total de 1742 diferentes elementos de software (Ilustración 22).

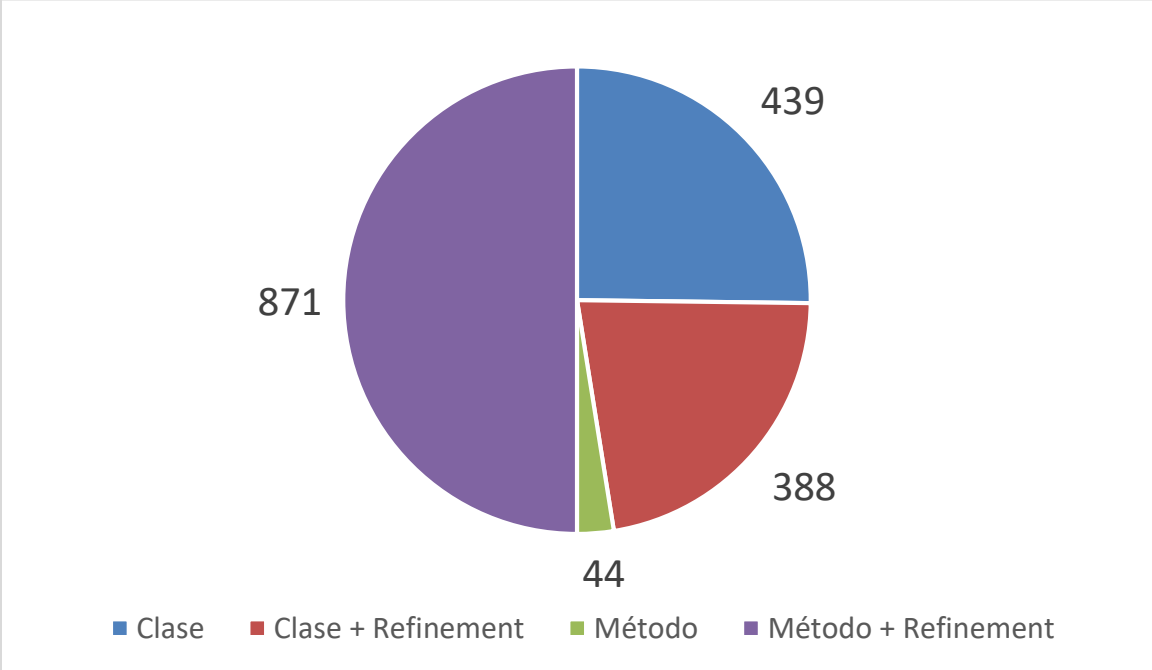

Ilustración 22 Cantidad de elementos de software diferenciados por el tipo de relación.

4.4 Estructura del proyecto ArgoUML SPL Benchmark

La explicación de la estructura del proyecto puede ayudar a entender más fácilmente el funcionamiento del benchmark. Estando conformado principalmente por las carpetas contenidas dentro de src: featuresInfo, groundTruth, scenarios, src, yourResults y yourResultsMetrics (Ver ilustración 23). Cada una de las anteriores tiene un propósito específico que le brindan valor al benchmark.

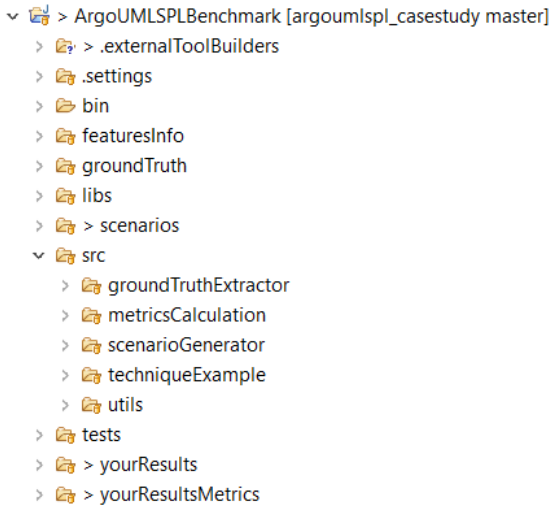

Ilustración 23 Estructura del proyecto ArgoUML SPL Benchmark

- **featureInfo:** Contiene el nombre y descripción oficial de cada uno de los features descritos en el numeral 2.1 ArgoUML SPL: Descripción general.

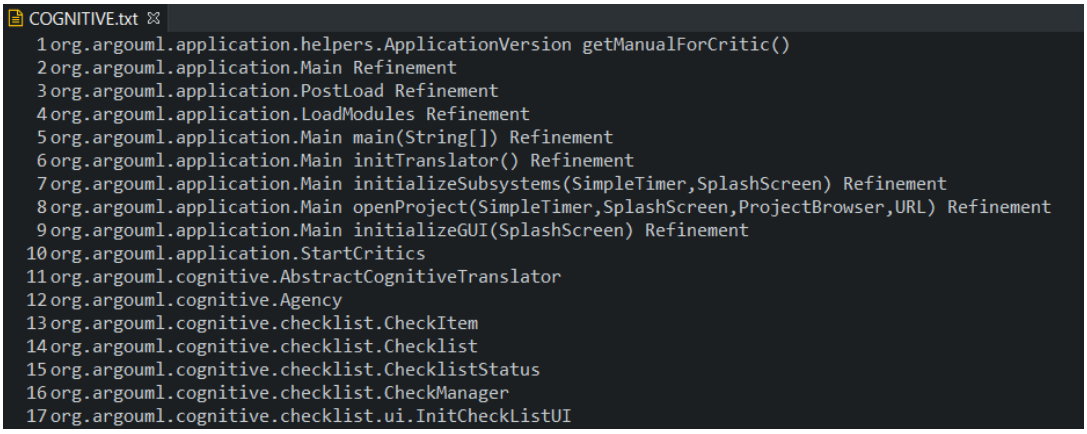

Ilustración 24 Ejemplo de un archivo de texto con el formato del GrountTruth

- **groundTruth:** Un listado de archivos de texto plano que contiene el listado de elementos de software asignados a un feature, a una interacción de dos features, a una interacción de tres features o a una negación de un feature, o en otras palabras, elementos de software que se eliminan al no estar presente un feature. En total son 24 archivos diferentes, como lo muestra la ilustración 24. La construcción de estos archivos es esencial para saber de antemano cuales son los fragmentos de código que están asociados a las diferentes features y sus relaciones; de esta manera se puede comparar línea a línea la eficiencia de las técnicas de identificación de características y obtener una métrica de desempeño.

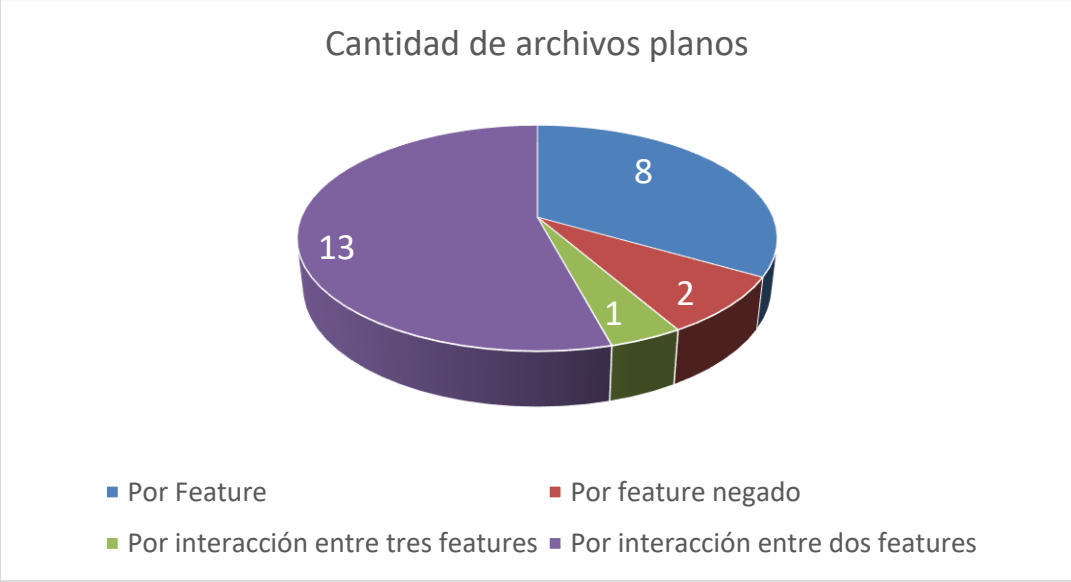

Ilustración 25 Cantidad de archivos planos con las respuestas de los elementos de software asociados por feature y sus interacciones.

- **Scenarios:** Posee cada uno de los escenarios que se explicaron en la sección 4.2.1 de este documento.
- **Src:** Contiene las clases Java que sirvieron para la creación del benchmark. Su explicación se verá más a fondo en la sección 4.5
- **yourResults:** Aquí deben ir los resultados que encuentre la técnica que se ponga a prueba en este benchmark.
- **yourResultsMetrics:** Aquí se generarán archivos en formato .csv que mostrarán el valor numérico en términos de métricas basado en los resultados que se encuentran en la carpeta yourResults.

4.5 Desarrollo de la prueba

Para la construcción de la prueba fue necesario la implementación en Java de varias clases (Ilustración 26) que tienen una labor específica en el proceso del desarrollo de la estructura del benchmark.

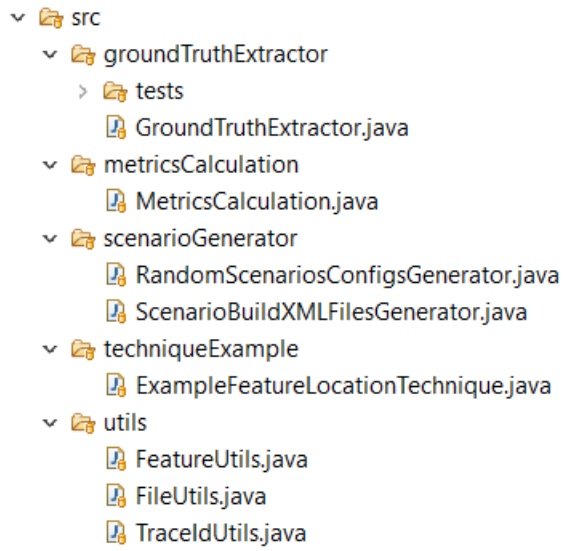

Ilustración 26 Estructura de la carpeta src dentro del proyecto ArgoUML-SPL Benchmark

4.5.1 Generación respuestas

¿Cómo se realizó la construcción de los archivos GroundTruth? La explicación está basada en el código que se encuentra en la carpeta groundTruthExtractor. Su construcción se basó en el entendimiento de las cláusulas ifdef usadas en el proyecto original [16]. El procedimiento para obtener las respuestas esperadas consiste en varios pasos (Ilustración 27) que se pueden resumir de la siguiente forma: Primero, se realiza la lectura de la estructura de las carpetas para posteriormente hacer una lectura de todas las clases Java que se encuentran en cada una de las carpetas que poseen código fuente de ArgoUML SPL que incluye las anotaciones; segundo, por medio de árboles sintácticos se realiza la lectura del código; tercero, se crea un archivo de texto por cada una de las relaciones ya descritas. Todo lo anterior aprovecha las notaciones ya realizadas [16] que muestran cuales métodos y clases están asignados a cada feature.

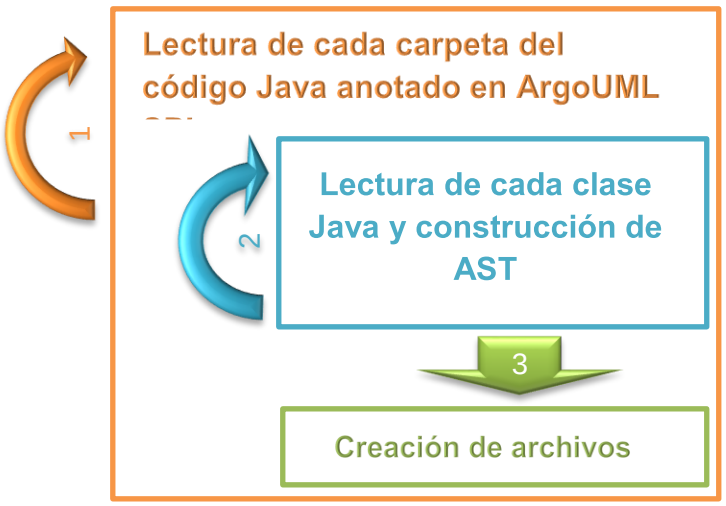

Ilustración 27 Procedimiento de creación de los archivos con las respuestas.

En otras palabras, al estar el proyecto original junto con las cláusulas utilizadas, se sabía previamente cuales segmentos de código hacían referencia a un feature, por lo que utilizando está información se realizó la construcción de árboles sintácticos que leyeran la estructura de la LPS y así escribir los archivos que obtuvieran cada una de las trazas.

4.5.2 Desarrollo de lógica de preprocesamiento de código

Para la construcción de los escenarios se realizó la implementación de clases que se encargan de automatizar el proceso, de esta manera al ejecutar cada clase se asegura que los escenarios se generen de forma aleatorias. Por lo que basado en dicha estructura se realizó la implementación en dos partes:

- **Construcción de archivos config:** Contiene la lista de los features que estarán presentes en la variante. Consta de la selección de manera aleatoria de los features opcionales del caso de estudio.
- **Construcción de los archivos xml:** Cada archivo xml contiene un procedimiento para ser ejecutado en ANT[4] y generar cada uno de los escenarios ya mencionados. Por lo que por medio de la estructura definida por SPLevo[5] (Aquí se describe la

[4] Documentación disponible en https://www.eclipse.org/eclipse/ant/index.php
[5] Documentación disponible en https://sdqweb.ipd.kit.edu/wiki/SPLevo/Case_Studies/ArgoUML-SPL

forma en la que se pueden generar las variantes), se hace la escritura de un archivo que contiene los features que se encuentran descritos en los archivos config.

4.5.3 Construcción de métricas automáticas

Se realizó la construcción de una clase que se encarga de las métricas para la evaluación de algoritmos [37]:

- **Precision:** Equivale al número de resultados correctos dividido sobre el número total de resultados que se obtuvieron (Ecuación 1). En otras palabras, se califica la cantidad de resultados correctos dentro de las trazas obtenidas por la técnica. Responde a la pregunta: ¿Qué tanta proporción de resultados correctos se encontraron? Por lo que adaptado al caso de estudio sería:

$$\boldsymbol{Precision} = \frac{\boldsymbol{número\ de\ trazas\ correctas}}{\boldsymbol{número\ de\ trazas\ correctas} + \boldsymbol{número\ de\ trazas\ incorrectas}}$$

Ecuación 1 Cálculo de Precision para el benchmark ArgoUML-SPL.

- **Recall:** Equivale al número de resultados correctos dividido por el número total de resultados correctos sumado al número de resultados encontrados que no corresponden a un resultado correcto (Ecuación 2). Responde a la pregunta: ¿Cuál es la proporción de resultados incorrectos? En otras palabras, se califica la cantidad de resultados correctos.

$$\boldsymbol{Recall} = \frac{\boldsymbol{número\ de\ trazas\ correctos}}{\boldsymbol{número\ de\ trazas\ correctas} + \boldsymbol{número\ de\ trazas\ encontrados\ incorrectos}}$$

Ecuación 2 Cálculo de Recall para el benchmark ArgoUML-SPL.

- **F Measure:** Es un cálculo basado en precisión and recall que se encarga de obtener la relación entre estos a través del promedio armónico (Ecuación 3). Y se realiza de esta forma ya que es la forma de medir el promedio y la frecuencia

general de los resultados, por lo que se puede decir que responde a la pregunta: ¿Cuál es la efectividad?

$$\boldsymbol{F\ Measure = 2 * \frac{precision * recall}{precision + recall}}$$

Ecuación 3 Cálculo de F Measure basado en precision y recall.

4.6 **Resumen**

En el capítulo anterior se encontraron dificultades en la comparación de estudios que utilizaban ArgoUML SPL. Esto se utilizó para construir una forma de comparación que tomó dichas debilidades y las utilizó como puntos a favor del GroundTruth planteado. Se explico la forma en que se generaron las respuestas esperadas, se estableció un formato de entrada y de salida para le utilización de la unificación del caso de estudio, además se mostró la estructura del proyecto la funcionalidad de cada uno de los elementos que allí se encuentran. Con la explicación realizada de la forma en que se construyó esta forma de comparación, se adapta la técnica que utiliza los algoritmos más utilizados según la literatura basado en el resultado de la revisión sistemática.

5. Técnica de línea base, estructura y pruebas realizadas

5.1 Motivación

Ahora que se construyó una forma de comparación para técnicas de recuperación de características, es importante utilizar los algoritmos más utilizados que se encontraron en la revisión sistemática para ponerlos a prueba. De esta forma, proponer un algoritmo que sirva como punto de referencia para comparar futuros estudios. A continuación se realiza una explicación de la técnica de línea base propuesta, la cual es la concatenación de tres algoritmos utilizados en la literatura según lo que se evidencia en la sección 3.6.2.

5.2 Técnicas y métodos aplicados para la recuperación de información

Dentro de los que se han analizado, existen un conjunto selecto que ha sido utilizado tanto en la literatura como en el caso de estudio seleccionado. A continuación, se presentan los algoritmos más usados.

5.2.1 Análisis de concepto formal (Formal Concept Analysis FCA)

Existen muchos ejemplos clásicos sobre el funcionamiento de este algoritmo en términos matemáticos.[6] Utilizando los conceptos aplicados, se puede entender FCA como la utilización del contexto de las entidades para obtener los elementos compartidos y diferentes de un grupo de elementos [40]. Aplicado al presente trabajo, cada subconjunto de líneas de código está asociado a un único grupo que puede estar presente en la

[6] Ejemplos disponibles aquí http://www.upriss.org.uk/fca/examples.html

intersección o la exclusión de dos o más features. Para facilitar su comprensión se realiza la siguiente explicación:

En la ilustración 28, se puede apreciar que existen 21 elementos que corresponden a 3 diferentes tipos de elementos que han sido asociados a tres diferentes conjuntos. Cada elemento representa un tipo de segmento de código: Clases, métodos y atributos donde la cantidad de elementos es variable y el tipo de elementos varía según la granularidad que maneje cada enfoque; y cada conjunto representa una variante en SPL. Por lo que FCA se basa en la intersección de elementos de software encontrados en un conjunto de variantes, de manera que cada grupo posee elementos diferentes, a estos grupos se le denominan bloques (ilustración 28). Los cuales contienen elementos de software donde existen elementos que pertenecen a una o varias variantes (Tabla 3).

Bloque	*Variante 1*	*Variante 2*	*Variante 3*
1	∈	∉	∉
2	∈	∈	∉
3	∈	∈	∈
4	∈	∉	∈
5	∉	∈	∉
6	∉	∈	∈
7	∉	∉	∈

Tabla 3 Pertenencias de elementos asociados a bloques.

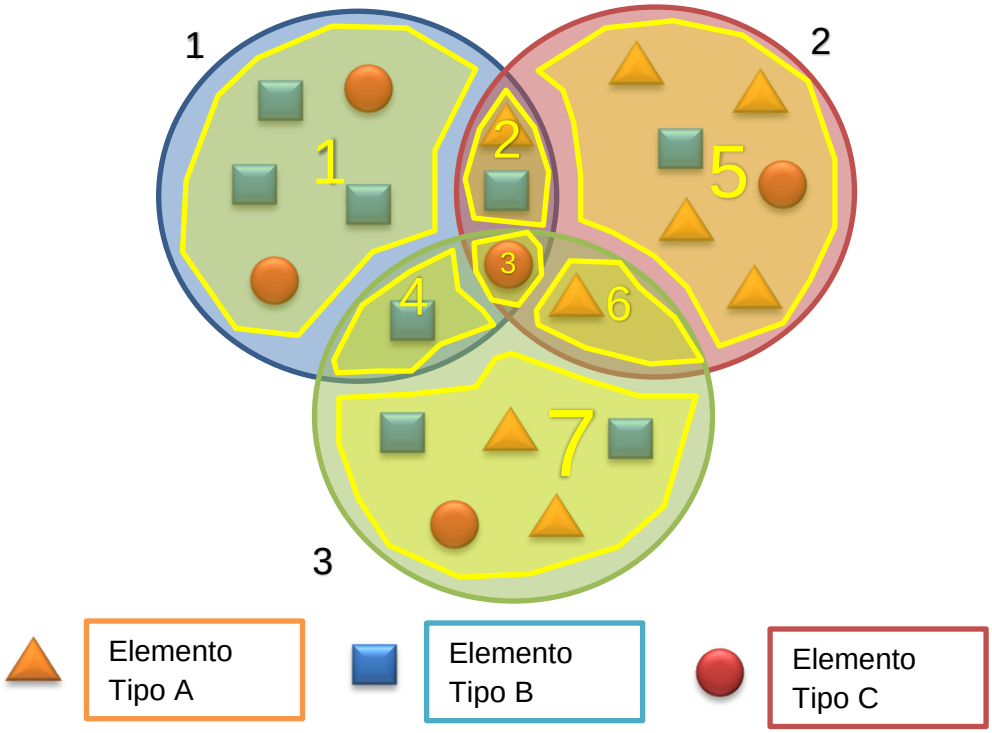

Ilustración 28 Diagrama de Venn explicando el resultado de FCA que corresponde con los conjuntos de las diferentes intersecciones de las variantes 1, 2 y 3 según la tabla 3.

5.2.2 Feature estricto específico (Strict Feature Specific SFS)

Es un algoritmo que utiliza el concepto de bloques obtenido de FCA y se basa en dos principios: Un bloque siempre está en los artefactos que implementan un feature, y un bloque nunca aparece en un artefacto que no implementa este feature [40]. En otras palabras, utiliza los agrupamientos de segmentos de código realizados por bloques para determinar que: un feature está presente en un bloque que lo implementa, y por otro lado, un feature no se encontrará en un bloque que no implemente dicha feature. En el estudio realizado en el capítulo 2 a la agrupación de elementos asociados por el artefacto, se les conoce como clusters de elementos [42], o en algunos casos son denominados temas de código [43]; aunque ambos términos hacen referencia al concepto de bloque.

5.2.3 Recuperación de información (IR Information retrieval) por medio de indexación semántica latente (Latent semantic indexing LSI)

Es un algoritmo que se encarga de recuperar la similitud entre una oración o petición (llamada query) y un conjunto de documentos. Es generalmente usado para el desarrollo de buscadores. Por medio de algebra lineal se hacen transformaciones de matrices para calcular la similitud entre estas dos entidades: la petición y los documentos. Dichas matrices se crean contando el número de veces que aparece cada término, palabras únicas, en los documentos [44]. Para ejemplificar, si tenemos los siguientes datos, extraídos de un conocido ejemplo [45], se podría ver la matriz como se ve en ilustración 29.

- **Petición:** *gold silver truck*
- **Documento 1:** *Shipment of gold damaged in a fire*
- **Documento 2:** *Delivery of silver arrived in a silver truck*
- **Documento 3:** *Shipment of gold arrived in a truck*

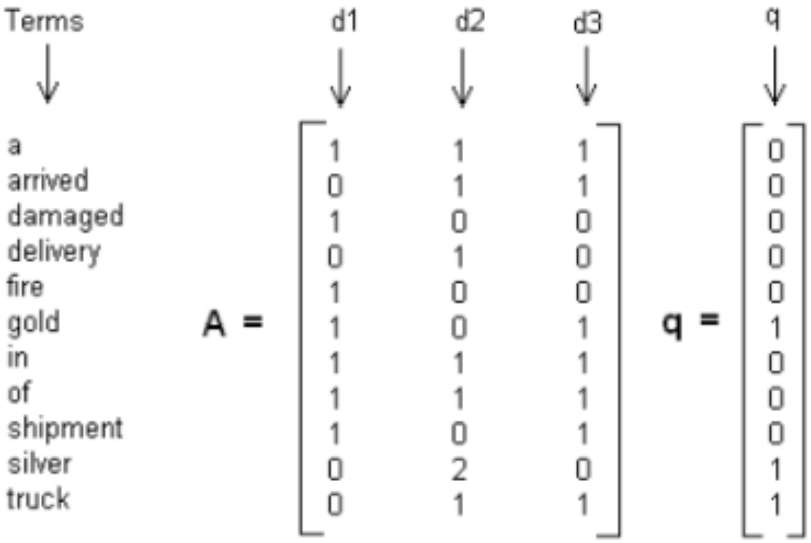

Ilustración 29 Tomado del ejemplo clásico Golden Truck [45]

La construcción de esta matriz permitirá que se realicen un conjunto de operaciones lineales que permiten obtener la similitud de la petición para cada uno de los documentos, como la aproximación bajo rango que hace una reducción K dimensional y otras adicionales que se pueden ver en las ecuaciones 4, 5 y 6.

$$A = USV^T$$

Ecuación 4 Descomposición de la matriz A en U, S y la traspuesta de V.

$$q = q^T U_k S_k^{-1}$$

Ecuación 5 Cálculo del vector q para determinar la cercanía entre la matriz de documentos y el vector de petición.

$$sim(q,d) = \frac{q \cdot d}{|q||d|}$$

Ecuación 6 Calculo de la similitud del coseno aplicado a una petición y un conjunto de documentos.

5.3 Caso de prueba: Casa domótica.

Una vez creados los archivos mostrados en el numeral anterior, se realiza la aplicación de las técnicas mencionadas en la sección 4.1., por lo que solo se deben modificar las preferencias de BUT4Reuse [7], para que se ejecute la técnica, que será el conjunto de los tres algoritmos descritos.

Pero adicional a esto, se hizo evidente la creación de un caso de prueba básico que tuviera en cuenta los elementos de trazabilidad que tiene en cuenta el punto de referencia: nombres de clases, nombres de métodos, nombres de clases que solo se crean si lo

[7] Disponible en https://but4reuse.github.io/

requiere otro feature y nombres de métodos que solo están presentes si otro feature lo requiere [26].

Para esto se diseñó un ejemplo que considera los elementos mencionados anteriormente (Ilustración 30).

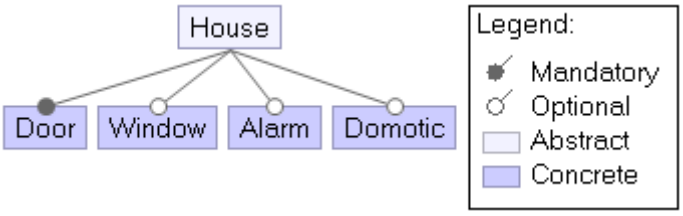

Ilustración 30 Diagrama de variabilidad del ejemplo del caso de prueba presentado. Diseñado en FeatureIDE[8].

5.3.1 Descripción general

Se hará una breve explicación al caso creado, y posteriormente se entrarán en detalles técnicos, como la estructura del proyecto y su importancia.

Es necesario suponer, que se desea hacer la construcción de casas que sean capaces de tener cierto nivel de variabilidad. Para ello, se posee como premisa que toda casa tiene puertas, pero no siempre tiene ventanas. Adicionalmente, se ofrece que la casa tenga alarmas las cuales están acopladas a las puertas y a las ventanas, en caso de que las tenga. Finalmente, se requiere que la casa tenga la facilidad de ser automática, o domótica, lo que consiste en que tanto las ventanas como las puertas se pueden abrir de manera automática con reconocimiento de usuario. En caso de que la casa tenga alarmas, estás solo se ejecutarán si se abre la puerta de manera manual.

Adicionalmente, se dará una descripción básica en inglés de cada feature para la construcción de la featureList:

- House: The house is the representation of a building.
- Door: The entrance of a house that can be either open or close.

[8] Disponible en http://www.featureide.com/

- Window: The window of a house that can be either open or close.
- Alarm: An alarm is a device to control the access to the house. If the access is not allowed the police will be called.
- Domotic: A domotic house allows to automatically open or close doors or windows.

5.3.2 Estructura del proyecto

Ahora que se tiene un contexto básico del ejemplo, se puede ver en la estructura del proyecto que cada *feature* se ve representado a través de una clase (Ilustración 31). El principal motivo de que la clase Domotic se encuentre en otro paquete es para que su ausencia o presencia involucre aspectos adicionales como la creación de *import*.

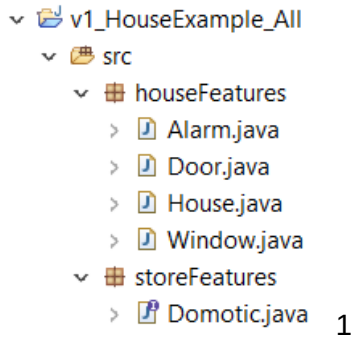

Ilustración 31 Estructura del proyecto de prueba para el algoritmo.

Ahora bien, el caso base, es que se encuentren todos los features presentes, pero al tener tres features opcionales se sabe que se puede tener un total de 8 variantes (Ilustración 32): Core (House y Door); Core y Window; Core y Alarm; Core y Domotic; Core, Window y Alarm; Core, Window y Domotic; Core, alarm y domotic; y Core, alarma, window y domotic.

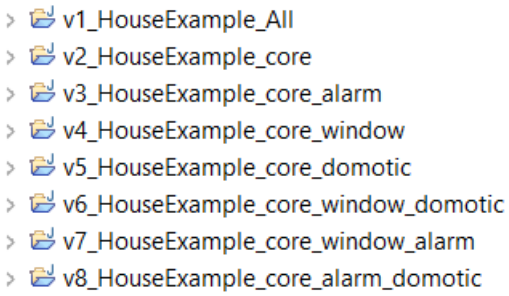

Ilustración 32 Escenarios posibles para el ejemplo de la casa domótica.

5.3.3 Explicación de clase y su representación

Para entender la importancia de la facilidad de poder hacer seguimiento a los elementos que se implementan, se muestra a continuación el contenido que hace único a cada una de las clases.

- **House.java:** Contiene una lista de puertas y ventanas (Ilustración 33). En caso de que la feature window, no se encuentre presente en la variante, se elimina todo lo que esté relacionado con la ventana.

```java
package houseFeatures;

import java.util.List;

/**
 * THE HOUSE IS THE REPRESENTATION OF A 1-FLOOR BUILDING WITH 1-ROOM THAT COMES
 * WITH N-DOORS AND N-WINDOWS
 *
 * @author Nicolas Ordoñez Chaña
 *
 */
public class House {

    // THE HOUSE COMES WITH A LIST OF DOORS
    public List<Door> doors;

    // THE HOUSE COMES WITH A LIST OF WINDOWS
    // NOT ALWAYS THE HOUSES COME WITH WINDOWS
    public List<Window> windows;

}
```

Ilustración 33 Contenido de la clase House.java para la variante que tiene todos los features.

- **Door.java**: Es la representación de una puerta de la casa (Ilustración 34 y 35). La variante que solo tiene Core, el código se verá modificado de manera que solo lo esencial de la puerta se encuentra, es decir, los métodos open y close. Además, el método open, no tendría las líneas 22 y 23 que hacen referencia a la alarma. También, la línea 4 se eliminaría, ya que no sería una puerta automática por lo que la definición de la clase no contendría la implementación de Domotic.

```
1  package houseFeatures;
2
3  //JUST IF THE DOOR IS DOMOTIC
4  import storeFeatures.Domotic;
5
7  * REPRESENTATION OF THE ASSET DOOR IN A HOUSE IMPLEMENTS ONLY IF DOMOTIC
12 public class Door implements Domotic {
13
14     // JUST IF THE WINDOW COME WITH ALARM
15     private Alarm alarm;
16
17     /**
18      * IT ALLOWS THE USER TO OPEN THE DOOR IF THE USER IS NOT ALLOWED ACCORDING TO
19      * THE ALARM THE ALARM WILL CALL THE POLICE
20      */
21     public void open() {
22         if (alarm.userAllowed())
23             alarm.callPolice();
24         // OPEN DOOR
25     }
26
28     * IT ALLOWS THE USER CLOSE THE DOOR
30     public void close() {
33
34     /**
35      * IF THE HOUSE IS DOMOTIC IT RECOGNIZE THE USER AND IT WILL OPEN THE DOOR
36      * AUTOMATICALLY
37      */
38     public void automaticOpen() {
39         // MAGICALLY OPEN DOOR
40     }
41
43     * IF THE HOUSE IS DOMOTIC IT WILL CLOSE THE DOOR AUTOMATICALLY
45     public void automaticClose() {
48 }
```

Ilustración 34 Clase Door con todos los features opcionales presentes.

```java
package houseFeatures;

/* REPRESENTATION OF THE ASSET DOOR IN A HOUSE
public class Door {

    /* IT ALLOWS THE USER TO OPEN THE DOOR
    public void open() {

    /* IT ALLOWS THE USER CLOSE THE DOOR
    public void close() {
        // CLOSE DOOR
    }

}
```

Ilustración 35 Door puerta con solo los features obligatorios presentes.

- **Alarm.java:** Es la representación de una alarma. Es necesaria para que se pueda implementar en las demás clases, pero dentro de su contenido no hay nada que dependa de otros features.
- **Domotic.java:** Es una interfaz que provee los métodos para que se pueda abrir automáticamente todos los componentes de la casa que se encuentran presentes en la variante (Ilustración 36).

```java
//IT IS NOT PROPERLY OF THE HOUSE
package storeFeatures;

/* IT ALLOWS TO IMPLEMENT AUTOMATIC FUNCTIONS OF THE DOOR OR THE WINDOW
public interface Domotic {

    // MAGICALLY OPEN THE HOUSE ARTEFACT
    void automaticOpen();

    // MAGICALLY CLOSE THE HOUSE ARTEFACT
    void automaticClose();

}
```

Ilustración 36 Interfaz domotic con los métodos que expone para las clases window y door.

5.4 Refactorización de código

Se revisó el procedimiento para la implementación del algoritmo LSI. Se extendió y refactorizó el código existente en BUT4Reuse, de manera que quedara general y así poder utilizarse para otras aplicaciones, tal y como se describe en la sección 4.1.3. (Ilustración 37).

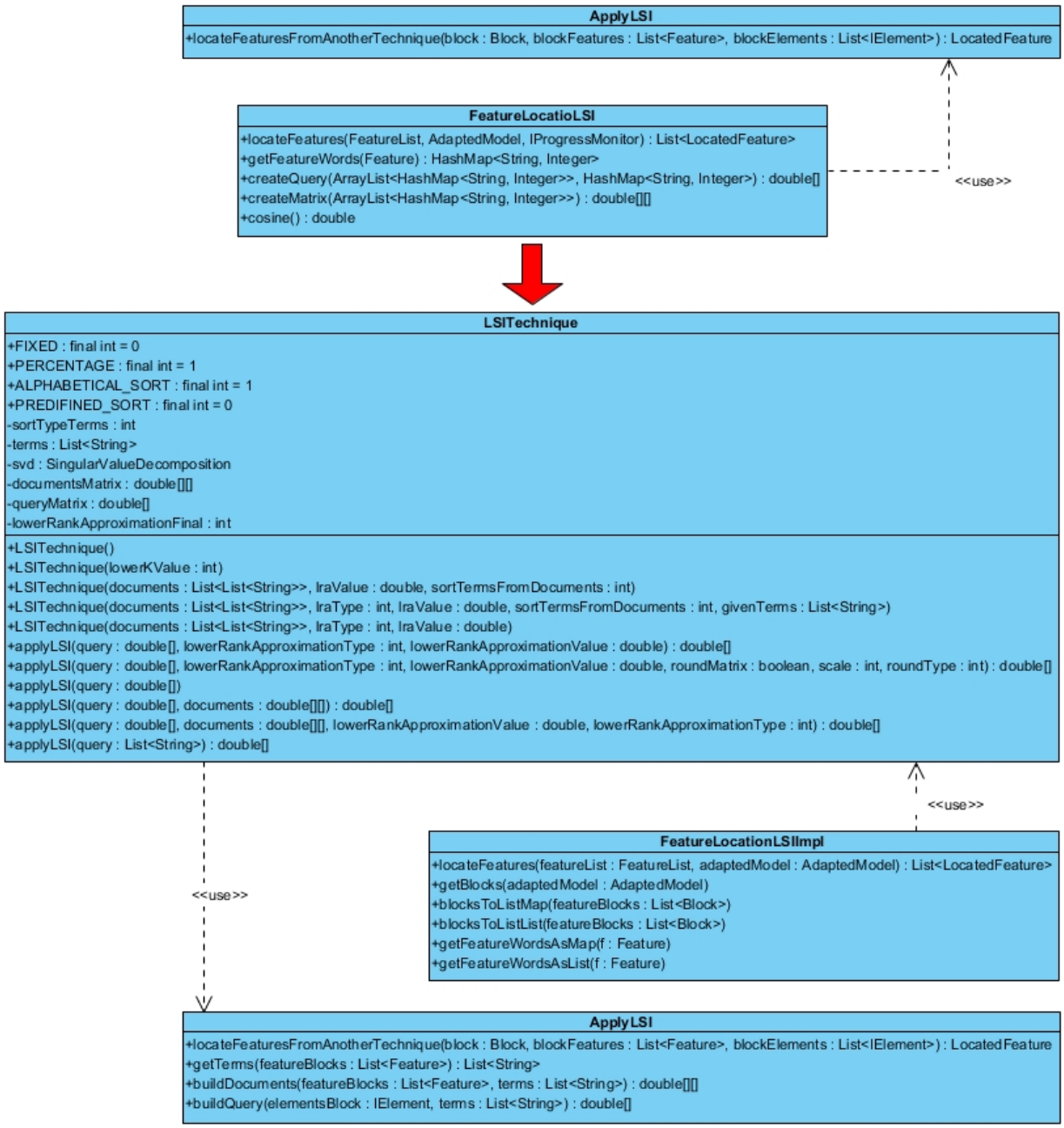

Ilustración 37 Refactorización del diagrama de clases para la técnica LSI.

5.4.1 Justificación de la refactorización

Existen varios factores que sustentan el desarrollo de la clase refactorizada LSITechnique:

- **Optimización del tiempo de ejecución en el cálculo de la descomposición singular de matrices:** A nivel computacional la complejidad de realizar la descomposición singular es $O(n^3)$ por lo que la nueva clase, se encarga de realizar este cálculo una vez por todo el conjunto de documentos.
- **Flexibilidad para el ordenamiento de filas en la matriz:** Para la construcción de la matriz se permite cambiar el orden de las filas basado en un orden determinado por el usuario o por medio de una búsqueda de todas las palabras diferentes dentro de los documentos que puede estar o no ordenado alfabéticamente. Ambas opciones anteriores pueden o no ordenarse alfabéticamente por medio de $Collection.sort()$[9] el cual es $O(n^{\log n})$. Es preciso decir que en caso de que se omita este paso, se aumentará el rendimiento.
- **Facilidad en el cambio de la variable que afecta la reducción de dimensiones:** Existe un valor *K* que debe estar entre 1 y la cantidad de términos, y funciona como una aproximación, eliminando un número K de filas. Esto, afecta directamente el rendimiento, ya que su complejidad es $O(nk^2)$, siendo n el número de documentos y K el número de filas que se deben calcular. Por lo que entre más pequeño sea K, más rápido será, pero debe ser lo suficientemente grande para que no se pierda el valor original de los documentos
- **Redondeo de los valores de la matriz:** Si bien, realizar el redondeo de números, de manera general, no es óptimo para la generación de números exactos, se realizó uno, para evitar afectar el resultado final de las operaciones, el cual consta así:

  $$0 \geq x \geq -1^{-4} \rightarrow \times = 0$$

  Existen números en el proceso del cálculo de la matriz mostrado en la sección 5.1.3. que son significativamente pequeños que, si bien afectan al resultado final,

[9] API disponible en https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html

tienen a confundir precisamente porque el valor de dicho número cambia el signo del resultado. (Ejemplo: -5.4^{-17}). Para contrarrestar esto, se realizó una asignación a cero sólo para valores mayores a -0.0001 y menores a cero. Este signo negativo para números excesivamente pequeños puede significar que puede haber similitud entre una petición y un documento, pero por las palabras que ambos no tienen, y el interés de esta técnica es buscar similitud por las palabras que se encuentran presentes.

- **Aseguramiento del funcionamiento de la técnica por medio de pruebas unitarias.** Adicional a la refactorización, se desarrollaron pruebas basadas en la literatura [46, 47, 48], las cuales constan con datos de entrada, petición y documentos y datos de salida, vector de similitudes entre la petición y cada uno de los documentos (Ilustración 38).

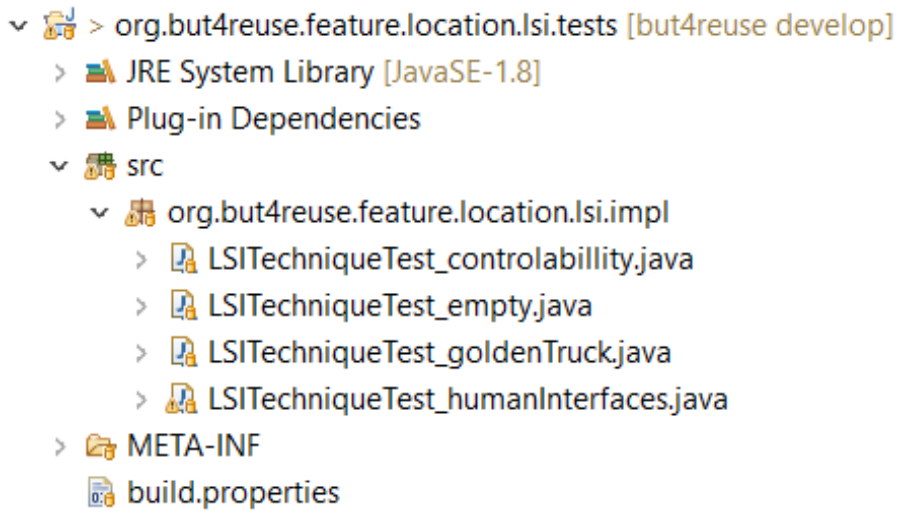

Ilustración 38 Estructura del proyecto de LSI vista desde Eclipse Photon[10].

5.5 Plan de trabajo y aplicación de la técnica para el desarrollo de la prueba.

Para la implementación de la técnica, se debe tener en cuenta las entradas y salidas que ofrece el estándar de referencia desarrollado en capítulos anteriores. Por lo cual se

[10] Disponible en https://www.eclipse.org/photon/

desarrolló un procedimiento que tuviera en cuenta todo lo anterior que se ve reflejado en la ilustración 39.

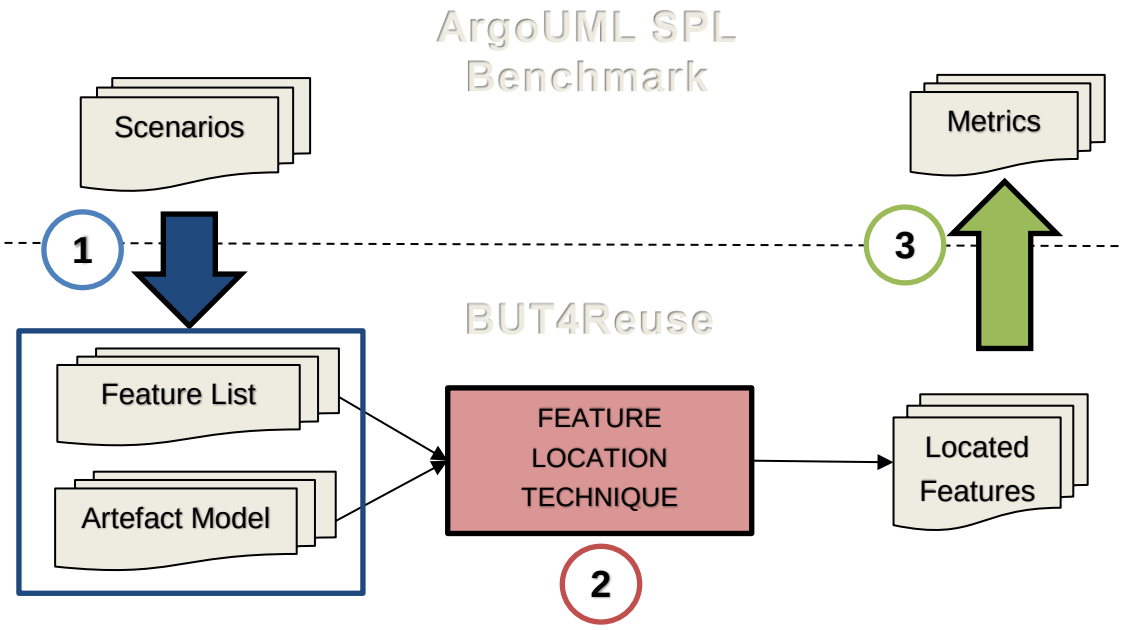

Ilustración 39 Procedimiento para el desarrollo de una técnica aplicada al punto de referencia.

5.5.1 Lectura y transformación de escenarios en objetos

Hace referencia al paso uno de la ilustración 39. Se necesita realizar la lectura de archivos para la transformación de datos que servirán como dato de entrada para le técnica de localización de features, estos son:

- **featuresInfo/features.txt:** Contiene el nombre y descripción de cada uno de los *features* involucrados en el caso de estudio como se muestra en la tabla 1 de la sección 2. 1..
- **Scenarios/*:** (Ilustración 38) Contiene las carpetas con cada uno de los escenarios propuestos en el *benchmark* en donde se realiza la lectura de sus subcarpetas:
    - *Configs*: Es la representación de una variante por medio de un archivo de texto plano que contiene la lista de *features* presentes.

- *Variants*: Contiene uno varios proyectos del caso de estudio que solo contiene las *features* presentes en el archivo *config*, es decir, el código fuente que está asociado a la variante.

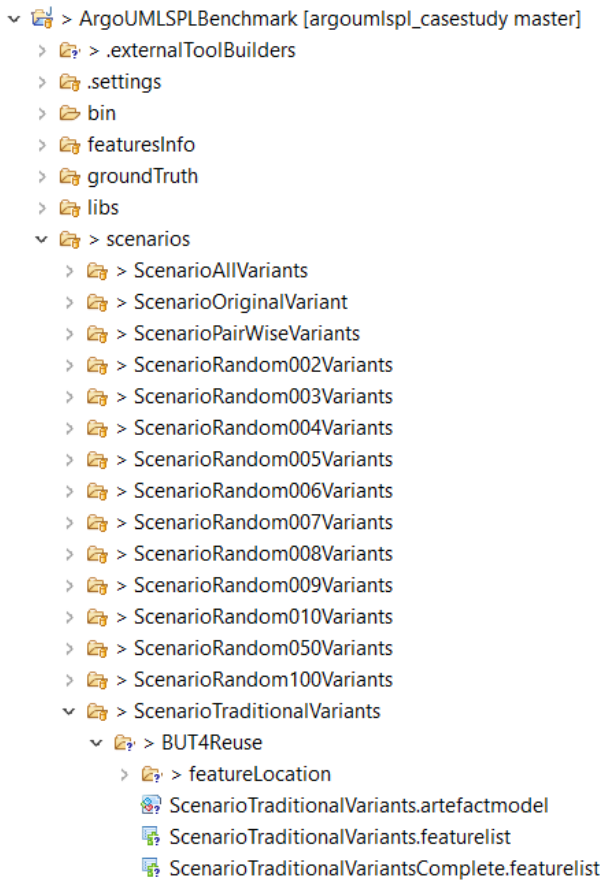

Ilustración 40 Puntos de interés del proyecto ArgoUML SPL Benchmark [16].

Por otra parte, la transformación incluye, por medio de los archivos *configs*, la construcción de dos tipos de datos para la herramienta BUT4Reuse:

- *artefactModel*: Archivo que representa cada una de las variantes del escenario (Ilustración 41).

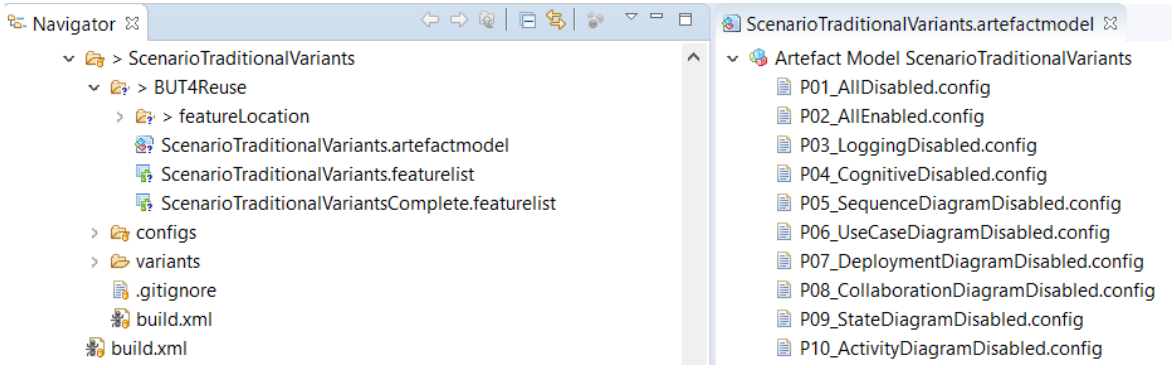

Ilustración 41 Captura de Pantalla de un artefactModel para un escenario del punto de referencia. Tomado de Eclipse Photon.

- *featureList*: Contiene la lista de todas las *features* con sus propiedades para realizar el seguimiento. Principalmente, una lista de artefactos implementados, que hace referencia en que variantes está presente.

A través de la interfaz, lo que se hizo fue realizar la construcción de un plugin que reconociera la carpeta *Scenarios* para la construcción de los objetos descritos anteriormente (Ilustración 42).

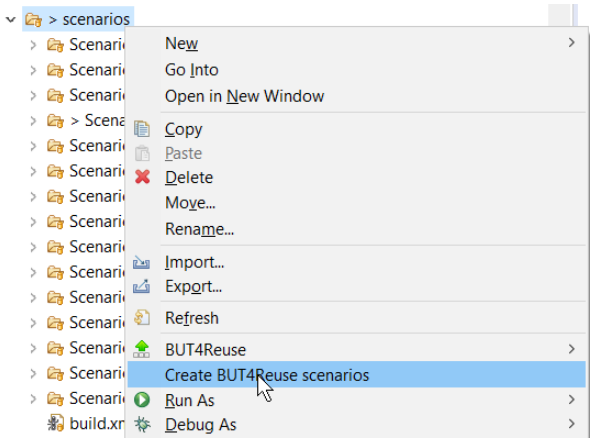

Ilustración 42 Eclipse funcionando con el nuevo botón habilitado para la creación automática de escenarios para la herramienta BUT4Reuse.

5.5.2 FCA + SFS + LSI: La técnica de la línea base

Hace referencia al paso dos de la ilustración 39 al inicio del capítulo 5.4. Basado en las técnicas más utilizadas para identificación de features y tomando como base el desarrollo realizado en BUT4Reuse, se implementó una técnica que combine tres enfoques: FCA + SFS + LSI.

Su procedimiento se basa en cuatro pasos principales: primero, obtener los bloques (FCA); segundo, asignar los features a los bloques a los que pertenece (SFS); tercero, construir la matriz de documentos basado en los bloques asociados a cada feature; y cuarto, aplicar LSI por cada feature asignado al elemento del bloque. Para calcular si un feature está o no asociado a un elemento de un bloque se obtiene aquel con la mayor similitud, esto se hace internamente en la clase ApplyLSI (Ilustración 43 y 44). Es decir, segmenta el código en subconjuntos y asocia una línea de código a un feature basado en la similitud entre un feature junto con su descripción y un bloque. Como premisa se tiene que se conoce en qué conjunto de variantes está implementado cada feature.

```
public class SFS_LSI implements IFeatureLocation {
    @Override
    public List<LocatedFeature> locateFeatures(FeatureList featureList, AdaptedModel adaptedModel,
            IProgressMonitor monitor) {

1       // Get SFS results, all located feature are 1 confidence
        StrictFeatureSpecificFeatureLocation sfs = new StrictFeatureSpecificFeatureLocation();
        List<LocatedFeature> sfsLocatedBlocks = sfs.locateFeatures(featureList, adaptedModel, monitor);

        List<LocatedFeature> locatedFeatures = new ArrayList<LocatedFeature>();

        // Get all the features of a given block and all its elements
        for (Block block : adaptedModel.getOwnedBlocks()) {
            // user cancel
            if (monitor.isCanceled()) {
                return locatedFeatures;
            }

2           monitor.subTask("Feature location FCA SFS and LSI. Features competing for Elements at " + block.getName()
                    + " /" + adaptedModel.getOwnedBlocks().size());
            List<Feature> blockFeatures = LocatedFeaturesUtils.getFeaturesOfBlock(sfsLocatedBlocks, block);
            List<IElement> blockElements = AdaptedModelHelper.getElementsOfBlock(block);

3           // Calculate LSI in each block
            ApplyLSI flsi = new ApplyLSI();
            List<LocatedFeature> lfs = flsi.locateFeaturesFromAnotherTechnique(block, blockFeatures, blockElements);
            if(lfs != null && !lfs.isEmpty()) {
                locatedFeatures.addAll(lfs);
            }

        }
        return locatedFeatures;
    }
}
```

Ilustración 43 Segmento de código que se encarga de realizar la implementación de la técnica de línea base dentro de la clase SFS_LSI.

```
// build documents
double[][] documents = buildDocuments(blockFeatures, terms);
if (documents != null) {
    for (IElement blockElement : blockElements) {
        // builder query from each blockElement
        double[] query = buildQuery(blockElement, terms);
        if (query != null) {
            // Get similarity from LSI Technique
            double vecSimilarity[] = lsiTechnique.applyLSI(terms, query, documents, dim, fixed,
                    LSITechnique.NO_SORT);

            //Get the position with the highest value
            int higherSimilarity = getHigherSimilarity(vecSimilarity);

            //if there exist one highest value it means that it found a feature
            if (higherSimilarity != -1) {
                double confidence = (vecSimilarity[higherSimilarity] + 1) / 2;
                if(confidence > 0) {
                    locatedFeatures.add(new LocatedFeature(blockFeatures.get(higherSimilarity), blockElement, confidence));
                }
            }
        }
    }
}
```

4

Ilustración 44 Segmento de código que se encarga de validar que un elemento de bloque se asocie a un feature en la clase ApplyLSI.

5.5.3 Generación de resultados en el formato del benchmark

Referente al paso tres de la ilustración 39, el objetivo principal es convertir los resultados obtenidos en BUT4Reuse en el formato establecido en el benchmark. El resultado de la técnica de feature location en BUT4Reuse es una serie de trazas entre features y elementos de código. El código que realiza esta transformación está disponible[11]. Básicamente se trata de una visualización que extiende BUT4Reuse con este propósito, pasar de las trazas en memoria a documentos de texto plano que espera el benchmark para calcular las métricas de manera automática.

5.5.4 Aplicación del caso de prueba: Casa domótica

Hace referencia al paso uno de la ilustración 39 al inicio del del capítulo 5.4. Al importar todas las variantes al artefact model, por medio de featureIdentification, se hace reconocimiento de todos los bloques que se encuentran en el conjunto de variantes.

[11] https://github.com/but4reuse/argouml-spl-benchmark_but4reuse-helper

Del ejemplo básico se puede explicar qué factores son los que permiten decir si una clase o un método tendrá la etiqueta “Refinement”.

- **Clase con etiqueta:** Si aparece un import, un modificador de clase (extends o implements) o una declaración de variable, sin estar presente la declaración de la clase.
- **Clase:** Cuando aparece el elemento declaración de clase.
- **Método con etiqueta:** Si aparece el cuerpo de un método o una variable dentro de un método sin estar presente la declaración del método. Adicionalmente, cuando aparece la declaración de un método sin la declaración de un cuerpo del mismo método.
- **Método:** Si aparece el elemento declaración de método acompañado de una declaración de cuerpo de método.

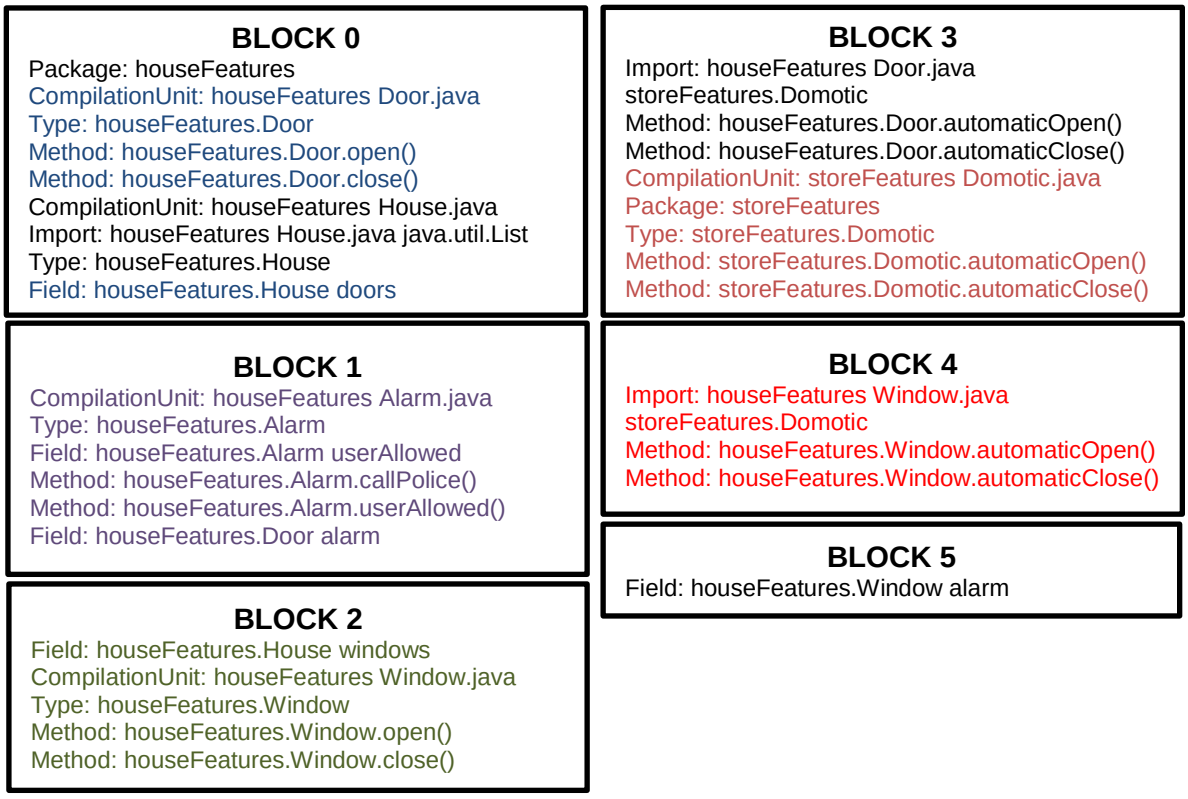

Ilustración 45 Bloques asociados al ejemplo de la casa domótica.

5.6 Resultados aplicados al caso de estudio: ArgoUML-SPL

El desarrollo de las pruebas se hizo en un computador con las siguientes especificaciones: Procesador Intel core i7-7700HQ 2.80 GHz (8 CPUs) y memoria RAM 8GB.

Dentro de la herramienta de BUT4Reuse se establecen una serie de filtros que son aceptados en la literatura para remover segmentos de texto no relevantes. De manera que se eliminan diferenciadores que pueden afectar el rendimiento del algoritmo [39], tales como: separación de palabras con Camel-case, ignorar mayúsculas, eliminación de conectores lógicos y tamizado de palabras (Por ejemplo, derivados de verbos), palabras generalmente usadas en programación (get y set) [39] (Ilustración 46).

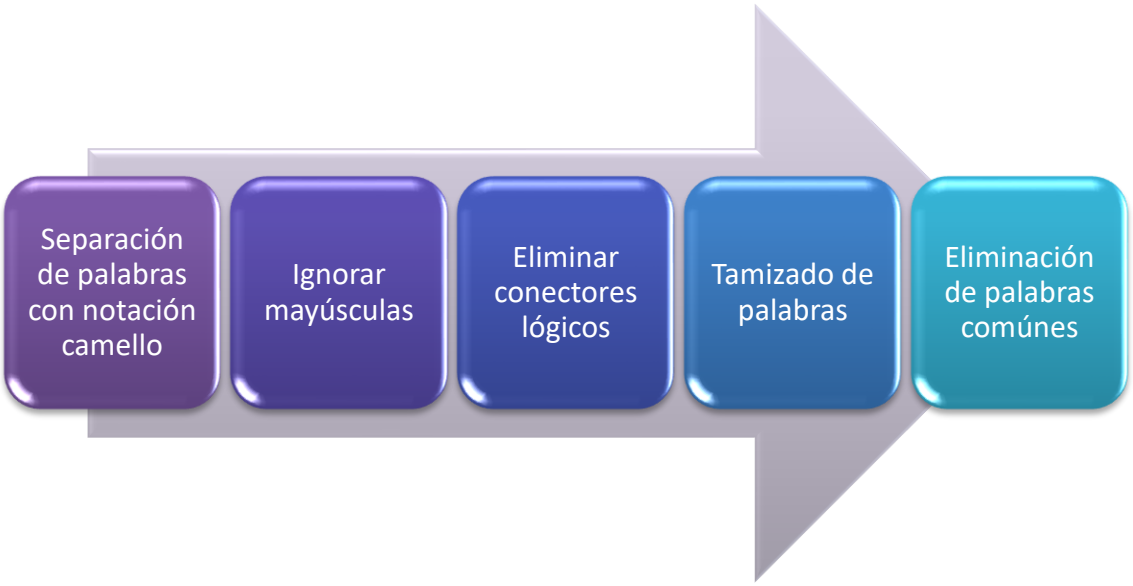

Ilustración 46 Preferencias utilizadas en las pruebas para el criterio de selección de términos aceptados en la técnica.

5.6.1 Ejecución de la prueba: Explicación de los resultados

Para cada uno de los escenarios se mostrará el tiempo de ejecución y los valores de las métricas desarrolladas en el capítulo 4.

- Tres variantes aleatorias (Tabla 4).

Variante	1	2	3
Features	LOGGING COLLABORATIONDIAGRAM	COGNITIVE ACTIVITYDIAGRAM STATEDIAGRAM USECASEDIAGRAM COLLABORATIONDIAGRAM	STATEDIAGRAM SEQUENCEDIAGRAM DEPLOYMENTDIAGRAM

Proceso	Tiempo (ms)
Adaptar código java (JDT)	30141
Identificación de bloques (FCA)	4478
Identificación de features (SFS + LSI)	3812

NAME	PRECISION	RECALL	FSCORE
ACTIVITYDIAGRAM	0,1111	0,0536	0,0723
COGNITIVE	0,0923	0,1020	0,0969
COLLABORATIONDIAGRAM	0,8000	0,4375	0,5657
DEPLOYMENTDIAGRAM	0,8500	0,4048	0,5484
LOGGING	0,9538	0,2902	0,4450
NOT_COGNITIVE	0,0108	1,0000	0,0213
SEQUENCEDIAGRAM	0,3151	0,2473	0,2771
STATEDIAGRAM	0,9123	0,4333	0,5876
USECASEDIAGRAM	0,0865	0,3788	0,1408
PROMEDIO	0,1722	0,1395	0,1148

Tabla 4 Ejecución de la línea base aplicada al escenario de tres variantes aleatorias.

- Cuatro variantes aleatorias (Tabla 5).

Variante	1	2	3
Features	ACTIVITYDIAGRAM STATEDIAGRAM SEQUENCEDIAGRAM COLLABORATIONDIAGRAM DEPLOYMENTDIAGRAM	ACTIVITYDIAGRAM STATEDIAGRAM COLLABORATIONDIAGRAM	LOGGING ACTIVITYDIAGRAM USECASEDIAGRAM COLLABORATIONDIAGRAM DEPLOYMENTDIAGRAM
Variante	4		
Features	COGNITIVE ACTIVITYDIAGRAM SEQUENCEDIAGRAM		

Proceso	Tiempo (ms)
Adaptar código java (JDT)	43338
Identificación de bloques (FCA)	5381
Identificación de features (SFS + LSI)	434

NAME	PRECISION	RECALL	FSCORE
ACTIVITYDIAGRAM	0,0204	0,2946	0,0382
COGNITIVE	0,025	0,0102	0,0145
COGNITIVE_AND_SEQUENCEDIAGRAM	0,0054	0,25	0,0106
COLLABORATIONDIAGRAM	0,2917	0,4375	0,35
DEPLOYMENTDIAGRAM	0,9677	0,7143	0,8219
LOGGING	0,747	0,2902	0,418
SEQUENCEDIAGRAM	0,95	0,6129	0,7451
STATEDIAGRAM	0,8723	0,3417	0,491
USECASEDIAGRAM	0,6579	0,3788	0,4808
PROMEDIO	0,1891	0,1388	0,1404

Tabla 5 Ejecución de la línea base aplicada al escenario de cuatro variantes aleatorias.

- Cinco variantes aleatorias (Tabla 6).

Variante	1	2	3
Features	ACTIVITYDIAGRAM STATEDIAGRAM SEQUENCEDIAGRAM COLLABORATIONDIAGRAM DEPLOYMENTDIAGRAM DEPLOYMENTDIAGRAM STATEDIAGRAM	LOGGING STATEDIAGRAM SEQUENCEDIAGRAM USECASEDIAGRAM DEPLOYMENTDIAGRAM	LOGGING STATEDIAGRAM COLLABORATIONDIAGRAM DEPLOYMENTDIAGRAM
Variante	**4**	**5**	
Features	COGNITIVE LOGGING USECASEDIAGRAM COLLABORATIONDIAGRAM DEPLOYMENTDIAGRAM	COGNITIVE LOGGING ACTIVITYDIAGRAM SEQUENCEDIAGRAM COLLABORATIONDIAGRAM DEPLOYMENTDIAGRAM	

Proceso	Tiempo (ms)
Adaptar código java (JDT)	45723
Identificación de bloques (FCA)	7571
Identificación de features (SFS + LSI)	3151

NAME	PRECISION	RECALL	FSCORE
ACTIVITYDIAGRAM	0,3000	0,0536	0,0909
COGNITIVE	0,0250	0,0102	0,0145
COGNITIVE_AND_DEPLOYMENTDIAGRAM	0,0051	0,0588	0,0094
COLLABORATIONDIAGRAM	0,9032	0,4375	0,5895
COLLABORATIONDIAGRAM_AND_LOGGING	1,0000	1,0000	1,0000
DEPLOYMENTDIAGRAM	0,0106	0,4048	0,0207
LOGGING	0,9490	0,2902	0,4444
LOGGING_AND_SEQUENCEDIAGRAM	1,0000	0,9500	0,9744
LOGGING_AND_STATEDIAGRAM	0,2857	0,3333	0,3077
SEQUENCEDIAGRAM	0,9492	0,6022	0,7368
STATEDIAGRAM	0,9762	0,3417	0,5062
USECASEDIAGRAM	0,5208	0,3788	0,4386
PROMEDIO	0,2885	0,2025	0,2139

Tabla 6 Ejecución de la línea base aplicada al escenario de cinco variantes aleatorias.

- Seis variantes aleatorias (Tabla 7).

Variante	1	2	3
Features	LOGGING STATEDIAGRAM DEPLOYMENTDIAGRAM	ACTIVITYDIAGRAM STATEDIAGRAM USECASEDIAGRAM COLLABORATIONDIAGRAM	ACTIVITYDIAGRAM USECASEDIAGRAM DEPLOYMENTDIAGRAM
Variante	**4**	**5**	**6**
Features	USECASEDIAGRAM COLLABORATIONDIAGRAM	STATEDIAGRAM SEQUENCEDIAGRAM USECASEDIAGRAM COLLABORATIONDIAGRAM DEPLOYMENTDIAGRAM	COGNITIVE ACTIVITYDIAGRAM COLLABORATIONDIAGRAM

Proceso	Tiempo (ms)
Adaptar código java (JDT)	36359
Identificación de bloques (FCA)	8431
Identificación de features (SFS + LSI)	1216

NAME	PRECISION	RECALL	FSCORE
ACTIVITYDIAGRAM	0,8367	0,3661	0,5093
ACTIVITYDIAGRAM_AND_STATEDIAGRAM	0,3077	0,6667	0,4211
COGNITIVE	0,7615	0,9014	0,8255
COLLABORATIONDIAGRAM	0,9074	0,7656	0,8305
DEPLOYMENTDIAGRAM	0,9677	0,7143	0,8219
DEPLOYMENTDIAGRAM_AND_USECASEDIAGRAM	0,2308	1,0000	0,3750
LOGGING	0,9300	0,2902	0,4423
NOT_COGNITIVE	0,0154	1,0000	0,0303
NOT_LOGGING	0,0144	1,0000	0,0284
SEQUENCEDIAGRAM	0,4600	0,2473	0,3217
STATEDIAGRAM	0,7931	0,3833	0,5169
USECASEDIAGRAM	0,7297	0,8182	0,7714
PROMEDIO	0,2898	0,3397	0,2456

Tabla 7 Ejecución de la línea base al escenario de seis variantes aleatorias.

- Siete variantes aleatorias (Tabla 8).

Variante	1	2	2
Features	COGNITIVE STATEDIAGRAM SEQUENCEDIAGRAM DEPLOYMENTDIAGRAM	COGNITIVE ACTIVITYDIAGRAM STATEDIAGRAM SEQUENCEDIAGRAM USECASEDIAGRAM	LOGGING STATEDIAGRAM COLLABORATIONDIAGRAM
Variante	**4**	**5**	**6**
Features	ACTIVITYDIAGRAM	USECASEDIAGRAM COLLABORATIONDIAGRAM DEPLOYMENTDIAGRAM	COGNITIVE LOGGING SEQUENCEDIAGRAM COLLABORATIONDIAGRAM DEPLOYMENTDIAGRAM
Variante	**7**		
Features	LOGGING ACTIVITYDIAGRAM STATEDIAGRAM SEQUENCEDIAGRAM USECASEDIAGRAM DEPLOYMENTDIAGRAM		

Proceso	Tiempo (ms)
Adaptar código java (JDT)	37057
Identificación de bloques (FCA)	15485
Identificación de features (SFS + LSI)	1788

NAME	PRECISION	RECALL	FSCORE
ACTIVITYDIAGRAM	0,8333	0,3571	0,5000
ACTIVITYDIAGRAM_AND_STATEDIAGRAM	0,3448	0,5556	0,4255
COGNITIVE	0,7733	0,8469	0,8084
COGNITIVE_AND_DEPLOYMENTDIAGRAM	0,8947	1,0000	0,9444
COLLABORATIONDIAGRAM	0,9032	0,4375	0,5895
DEPLOYMENTDIAGRAM	0,9677	0,7143	0,8219
DEPLOYMENTDIAGRAM_AND_USECASEDIAGRAM	0,6667	0,6667	0,6667
LOGGING	0,9839	0,9532	0,9683
LOGGING_AND_STATEDIAGRAM	0,5517	0,8889	0,6809
NOT_COGNITIVE	0,0154	1,0000	0,0303
NOT_LOGGING	0,0144	1,0000	0,0284
SEQUENCEDIAGRAM	0,9492	0,6022	0,7368
STATEDIAGRAM	0,7966	0,3917	0,5251
USECASEDIAGRAM	0,8689	0,8030	0,8346
PROMEDIO	0,3985	0,4257	0,3567

Tabla 8 Ejecución de la línea base al escenario de siete variantes aleatorias.

- Ocho variantes aleatorias (Tabla 9).

Variante	1	2	2
Features	ACTIVITYDIAGRAM USECASEDIAGRAM DEPLOYMENTDIAGRAM	COGNITIVE LOGGING ACTIVITYDIAGRAM COLLABORATIONDIAGRAM	LOGGING STATEDIAGRAM SEQUENCEDIAGRAM COLLABORATIONDIAGRAM
	4	**5**	**6**
	LOGGING STATEDIAGRAM SEQUENCEDIAGRAM DEPLOYMENTDIAGRAM	COGNITIVE LOGGING ACTIVITYDIAGRAM USECASEDIAGRAM	COGNITIVE STATEDIAGRAM SEQUENCEDIAGRAM COLLABORATIONDIAGRAM DEPLOYMENTDIAGRAM
	7	**8**	
	COGNITIVE ACTIVITYDIAGRAM STATEDIAGRAM SEQUENCEDIAGRAM COLLABORATIONDIAGRAM	SEQUENCEDIAGRAM DEPLOYMENTDIAGRAM	

Proceso	Tiempo (ms)
Adaptar código java (JDT)	55693
Identificación de bloques (FCA)	19851
Identificación de features (SFS + LSI)	1876

NAME	PRECISION	RECALL	FSCORE
ACTIVITYDIAGRAM	0,8333	0,3571	0,5000
ACTIVITYDIAGRAM_AND_STATEDIAGRAM	0,3077	0,6667	0,4211
COGNITIVE	0,7757	0,8469	0,8098
COGNITIVE_AND_DEPLOYMENTDIAGRAM	0,6800	1,0000	0,8095
COLLABORATIONDIAGRAM	0,9032	0,4375	0,5895
COLLABORATIONDIAGRAM_AND_LOGGING	0,9231	1,0000	0,9600
COLLABORATIONDIAGRAM_AND_LOGGING_AND_SEQUENCEDIAGRAM	0,2308	1,0000	0,3750
COLLABORATIONDIAGRAM_AND_SEQUENCEDIAGRAM	0,5000	0,5385	0,5185
DEPLOYMENTDIAGRAM	0,9677	0,7143	0,8219
DEPLOYMENTDIAGRAM_AND_LOGGING	0,5385	1,0000	0,7000
DEPLOYMENTDIAGRAM_AND_USECASEDIAGRAM	0,1200	1,0000	0,2143
LOGGING	0,9652	0,9532	0,9592
LOGGING_AND_USECASEDIAGRAM	0,5833	1,0000	0,7368
NOT_COGNITIVE	0,0154	1,0000	0,0303
NOT_LOGGING	0,0144	1,0000	0,0284
SEQUENCEDIAGRAM	0,9492	0,6022	0,7368
STATEDIAGRAM	0,9762	0,3417	0,5062
USECASEDIAGRAM	0,6429	0,8182	0,7200
PROMEDIO	0,4553	0,5948	0,4349

Tabla 9 Ejecución de la línea base aplicada al escenario de 8 variantes aleatorias.

- Nueve variantes aleatorias (Tabla 12).

Variante	1	2	3
Features	COGNITIVE ACTIVITYDIAGRAM STATEDIAGRAM SEQUENCEDIAGRAM DEPLOYMENTDIAGRAM	COGNITIVE LOGGING ACTIVITYDIAGRAM SEQUENCEDIAGRAM DEPLOYMENTDIAGRAM	COGNITIVE LOGGING ACTIVITYDIAGRAM
Variante	**4**	**5**	**6**
Features	STATEDIAGRAM COLLABORATIONDIAGRAM	LOGGING ACTIVITYDIAGRAM DEPLOYMENTDIAGRAM	LOGGING SEQUENCEDIAGRAM
	7	**8**	**9**
	LOGGING COLLABORATIONDIAGRAM	COGNITIVE LOGGING ACTIVITYDIAGRAM COLLABORATIONDIAGRAM	LOGGING ACTIVITYDIAGRAM STATEDIAGRAM USECASEDIAGRAM DEPLOYMENTDIAGRAM

Proceso	Tiempo (ms)
Adaptar código java (JDT)	89332
Identificación de bloques (FCA)	37284
Identificación de features (SFS + LSI)	1706

NAME	PRECISION	RECALL	FSCORE
ACTIVITYDIAGRAM	0,8333	0,3571	0,5000
ACTIVITYDIAGRAM_AND_LOGGING	0,9333	0,8750	0,9032
ACTIVITYDIAGRAM_AND_STATEDIAGRAM	0,3077	0,4444	0,3636
COGNITIVE	0,7764	0,8503	0,8117
COGNITIVE_AND_DEPLOYMENTDIAGRAM	0,8947	1,0000	0,9444
COGNITIVE_AND_LOGGING	0,9206	1,0000	0,9587
COLLABORATIONDIAGRAM	0,9032	0,4375	0,5895
COLLABORATIONDIAGRAM_AND_LOGGING	0,9231	1,0000	0,9600
DEPLOYMENTDIAGRAM	0,9677	0,7143	0,8219
DEPLOYMENTDIAGRAM_AND_LOGGING	1,0000	0,9286	0,9630
LOGGING	0,9839	0,9532	0,9683
LOGGING_AND_SEQUENCEDIAGRAM	1,0000	1,0000	1,0000
NOT_COGNITIVE	0,0154	1,0000	0,0303
NOT_LOGGING	0,0140	1,0000	0,0275
SEQUENCEDIAGRAM	0,9492	0,6022	0,7368
SEQUENCEDIAGRAM_AND_STATEDIAGRAM	0,0072	1,0000	0,0144
STATEDIAGRAM	0,7931	0,3833	0,5169
USECASEDIAGRAM	0,5208	0,3788	0,4386
PROMEDIO	0,5310	0,5802	0,4812

Tabla 10 Resultados de la línea base aplicados al escenario de 9 variantes aleatorias.

- Diez variantes aleatorias (Tabla 11).

Variante	1	7	2
Features	LOGGING SEQUENCEDIAGRAM COLLABORATIONDIAGRAM DEPLOYMENTDIAGRAM	COGNITIVE LOGGING STATEDIAGRAM SEQUENCEDIAGRAM USECASEDIAGRAM COLLABORATIONDIAGRAM	COGNITIVE LOGGING ACTIVITYDIAGRAM STATEDIAGRAM SEQUENCEDIAGRAM
Variante	**4**	**5**	**6**
Features	COGNITIVE LOGGING ACTIVITYDIAGRAM USECASEDIAGRAM DEPLOYMENTDIAGRAM	COGNITIVE ACTIVITYDIAGRAM STATEDIAGRAM SEQUENCEDIAGRAM DEPLOYMENTDIAGRAM	COGNITIVE ACTIVITYDIAGRAM SEQUENCEDIAGRAM USECASEDIAGRAM DEPLOYMENTDIAGRAM
2	**8**	**9**	10
LOGGING DEPLOYMENTDIAGRAM	ACTIVITYDIAGRAM COLLABORATIONDIAGRAM	COGNITIVE STATEDIAGRAM SEQUENCEDIAGRAM	SEQUENCEDIAGRAM

Proceso	Tiempo (ms)
Adaptar código java (JDT)	110034
Identificación de bloques (FCA)	26952
Identificación de features (SFS + LSI)	2864

NAME	PRECISION	RECALL	FSCORE
ACTIVITYDIAGRAM	0,8333	0,3571	0,5000
ACTIVITYDIAGRAM_AND_LOGGING	0,6667	0,1250	0,2105
ACTIVITYDIAGRAM_AND_STATEDIAGRAM	0,3571	0,5556	0,4348
COGNITIVE	0,7757	0,8469	0,8098
COGNITIVE_AND_DEPLOYMENTDIAGRAM	1,0000	0,9412	0,9697
COGNITIVE_AND_LOGGING	0,8406	1,0000	0,9134
COLLABORATIONDIAGRAM	0,9032	0,4375	0,5895
DEPLOYMENTDIAGRAM	0,9677	0,7143	0,8219
DEPLOYMENTDIAGRAM_AND_LOGGING	0,9286	0,9286	0,9286
DEPLOYMENTDIAGRAM_AND_USECASEDIAGRAM	0,5000	0,6667	0,5714
LOGGING	0,9839	0,9532	0,9683
LOGGING_AND_SEQUENCEDIAGRAM	1,0000	0,9500	0,9744
LOGGING_AND_STATEDIAGRAM	0,8000	0,2222	0,3478
LOGGING_AND_USECASEDIAGRAM	1,0000	1,0000	1,0000
NOT_COGNITIVE	0,0152	1,0000	0,0299
NOT_LOGGING	0,0144	1,0000	0,0284
SEQUENCEDIAGRAM	0,9492	0,6022	0,7368
STATEDIAGRAM	0,9762	0,3417	0,5062
USECASEDIAGRAM	0,8710	0,8182	0,8438
GLOBAL RESULTS	0,5993	0,5608	0,5077

Tabla 11 Ejecución de la línea base aplicado al escenario de 10 variantes aleatorias.

- Diez variantes tradicionales (Tabla 12).

Variante	1	2	3
Features		COGNITIVE LOGGING ACTIVITYDIAGRAM STATEDIAGRAM SEQUENCEDIAGRAM USECASEDIAGRAM COLLABORATIONDIAGRAM DEPLOYMENTDIAGRAM	COGNITIVE ACTIVITYDIAGRAM STATEDIAGRAM SEQUENCEDIAGRAM USECASEDIAGRAM COLLABORATIONDIAGRAM DEPLOYMENTDIAGRAM
Variante	**4**	**5**	**6**
Features	LOGGING ACTIVITYDIAGRAM STATEDIAGRAM SEQUENCEDIAGRAM USECASEDIAGRAM COLLABORATIONDIAGRAM DEPLOYMENTDIAGRAM	COGNITIVE LOGGING ACTIVITYDIAGRAM STATEDIAGRAM USECASEDIAGRAM COLLABORATIONDIAGRAM DEPLOYMENTDIAGRAM	COGNITIVE LOGGING ACTIVITYDIAGRAM STATEDIAGRAM SEQUENCEDIAGRAM COLLABORATIONDIAGRAM DEPLOYMENTDIAGRAM
7	**8**	**9**	**10**
COGNITIVE LOGGING ACTIVITYDIAGRAM STATEDIAGRAM SEQUENCEDIAGRAM USECASEDIAGRAM COLLABORATIONDIAGRAM	COGNITIVE LOGGING ACTIVITYDIAGRAM STATEDIAGRAM SEQUENCEDIAGRAM USECASEDIAGRAM DEPLOYMENTDIAGRAM	COGNITIVE LOGGING ACTIVITYDIAGRAM SEQUENCEDIAGRAM USECASEDIAGRAM COLLABORATIONDIAGRAM DEPLOYMENTDIAGRAM	COGNITIVE LOGGING STATEDIAGRAM SEQUENCEDIAGRAM USECASEDIAGRAM COLLABORATIONDIAGRAM DEPLOYMENTDIAGRAM

Proceso	Tiempo (ms)
Adaptar código java (JDT)	155676
Identificación de bloques (FCA)	27605
Identificación de features (SFS + LSI)	974

NAME	PRECISION	RECALL	FSCORE
ACTIVITYDIAGRAM	0,8333	0,3571	0,5000
ACTIVITYDIAGRAM_AND_LOGGING	0,6667	0,1250	0,2105
ACTIVITYDIAGRAM_AND_STATEDIAGRAM	0,3077	0,4444	0,3636
COGNITIVE	0,7757	0,8469	0,8098
COGNITIVE_AND_DEPLOYMENTDIAGRAM	1,0000	0,9412	0,9697
COGNITIVE_AND_LOGGING	0,9206	1,0000	0,9587
COGNITIVE_AND_SEQUENCEDIAGRAM	1,0000	0,7500	0,8571
COLLABORATIONDIAGRAM	0,9032	0,4375	0,5895
COLLABORATIONDIAGRAM_AND_LOGGING	1,0000	1,0000	1,0000
COLLABORATIONDIAGRAM_AND_LOGGING_AND_SEQUENCEDIAGRAM	0,6000	1,0000	0,7500
COLLABORATIONDIAGRAM_AND_SEQUENCEDIAGRAM	0,5385	0,5385	0,5385
DEPLOYMENTDIAGRAM	0,9677	0,7143	0,8219
DEPLOYMENTDIAGRAM_AND_LOGGING	1,0000	0,9286	0,9630
DEPLOYMENTDIAGRAM_AND_USECASEDIAGRAM	1,0000	0,6667	0,8000
LOGGING	0,9652	0,9532	0,9592
LOGGING_AND_SEQUENCEDIAGRAM	1,0000	0,9500	0,9744
LOGGING_AND_STATEDIAGRAM	0,8000	0,2222	0,3478
LOGGING_AND_USECASEDIAGRAM	1,0000	1,0000	1,0000
NOT_COGNITIVE	0,0154	1,0000	0,0303
NOT_LOGGING	0,0144	1,0000	0,0284
SEQUENCEDIAGRAM	0,9492	0,6022	0,7368
STATEDIAGRAM	0,7931	0,3833	0,5169
USECASEDIAGRAM	0,8689	0,8030	0,8346
GLOBAL RESULTS	0,7466	0,6943	0,6484

Tabla 12 Línea base aplicado al escenario tradicional de 10 variantes.

5.6.2 Análisis de resultados

A continuación, se presentan algunas gráficas comparativas que evidencian el desempeño de algoritmo implementado teniendo en cuenta el tiempo de ejecución y las métricas establecidas. Las gráficas de dichos resultados se pueden apreciar en las ilustraciones 47 y 48. Dónde el eje X presenta cada uno de los escenarios puestos a prueba, y el eje Y presenta una escala de 0 a 1 para graficar los resultados de las métricas (Precission, recall y F Measure)

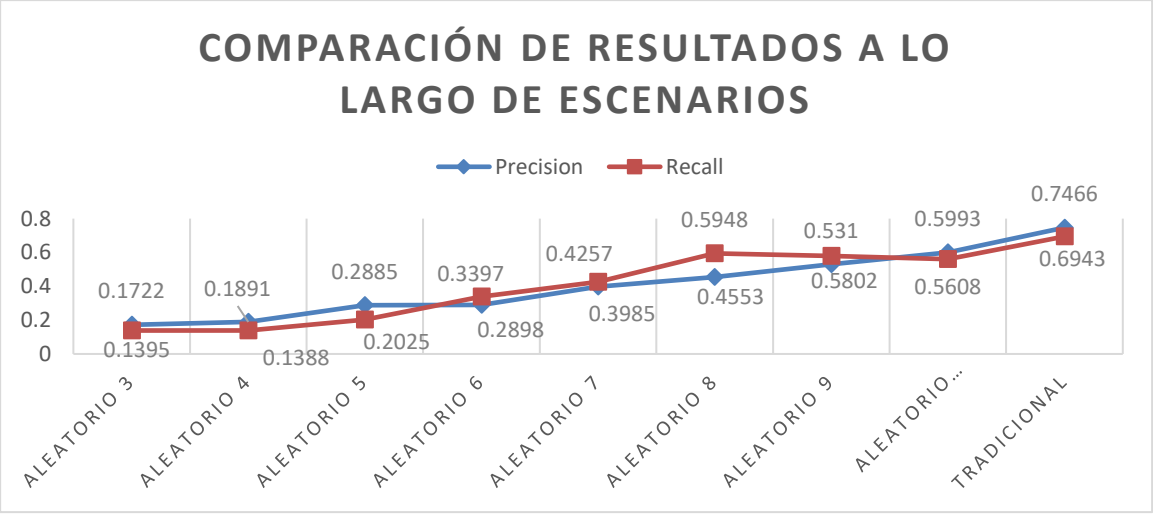

Ilustración 47 Comparación de métricas para los diferentes escenarios aplicando FCA + SFS + LSI.

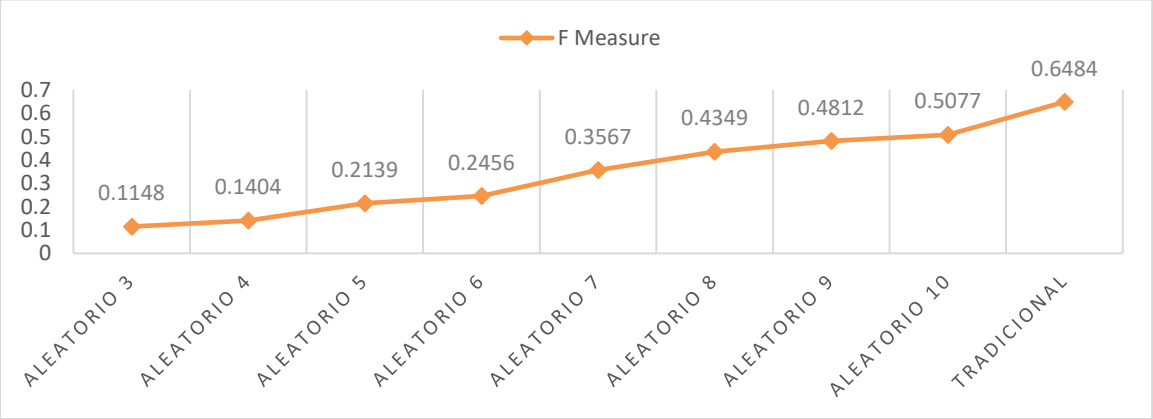

Ilustración 48 Comparación de F measure para los diferentes escenarios aplicando FCA + SFS + LSI.

En primer lugar, revisando las métricas establecidas en el capítulo 5, se evidencia que el aumento de variantes aumenta levemente el desempeño por cada uno de los escenarios (Ilustración 47 y 48). Esto es debido a que entre más grande sea el espacio de búsqueda, más sencillo será para el algoritmo asociar segmentos de código a un feature. El algoritmo tiene en cuenta las técnicas más usadas en la literatura, por lo que establece una línea base inicial basado en los conceptos teóricos utilizados.

Por otra parte, los tiempos de ejecución para la lectura de las clases Java es la tarea más demorada de la implementación (Ilustración 49), además se puede ver que el rendimiento de aplicar SFS + LSI es bastante rápido sin importar el número de variantes. Por lo que, la técnica por si misma, es escalable, ya que, al utilizarse en uno de los casos de estudio considerado de gran tamaño, obtuvo bastantes resultados. En otras palabras, el mayor reto en términos de optimización de tiempo está en encontrar la forma de representar el código.

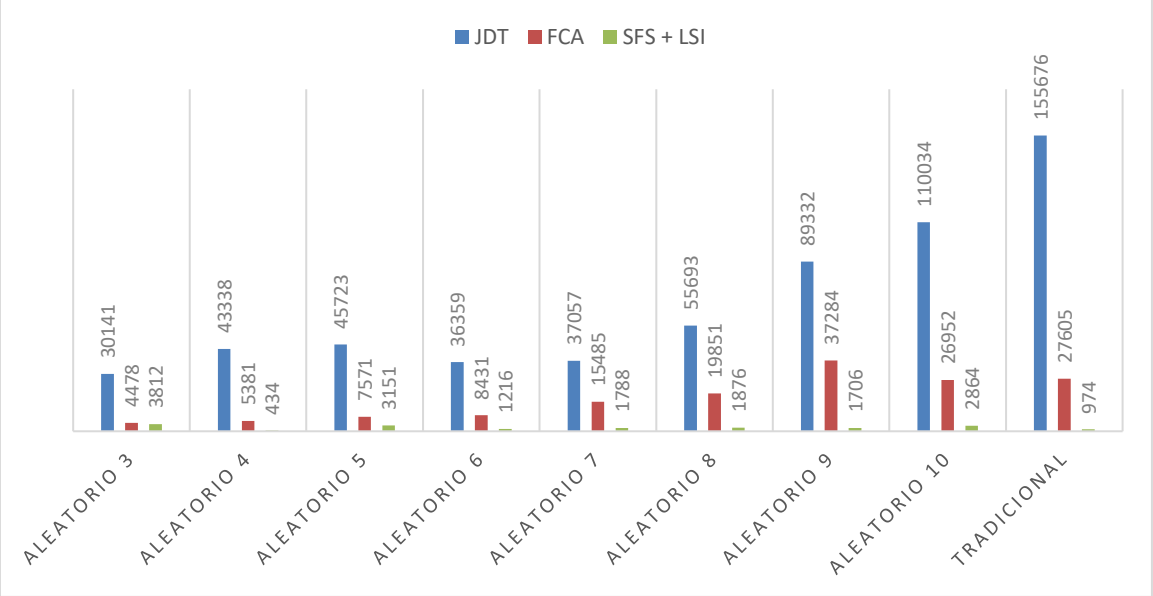

Ilustración 49 Comparación de tiempo de ejecución diferenciado por las etapas que intervienen en el algoritmo de línea base propuesto.

Finalmente, y como es de esperarse, aumentar el número de variantes que participan en un escenario facilita la tarea del algoritmo de encontrar trazas asociadas a features

(Ilustración 50). Esto se debe a que entre mayor sea el número de variantes, más fácil se podrá asociar una traza a un elemento de bloque diferente.

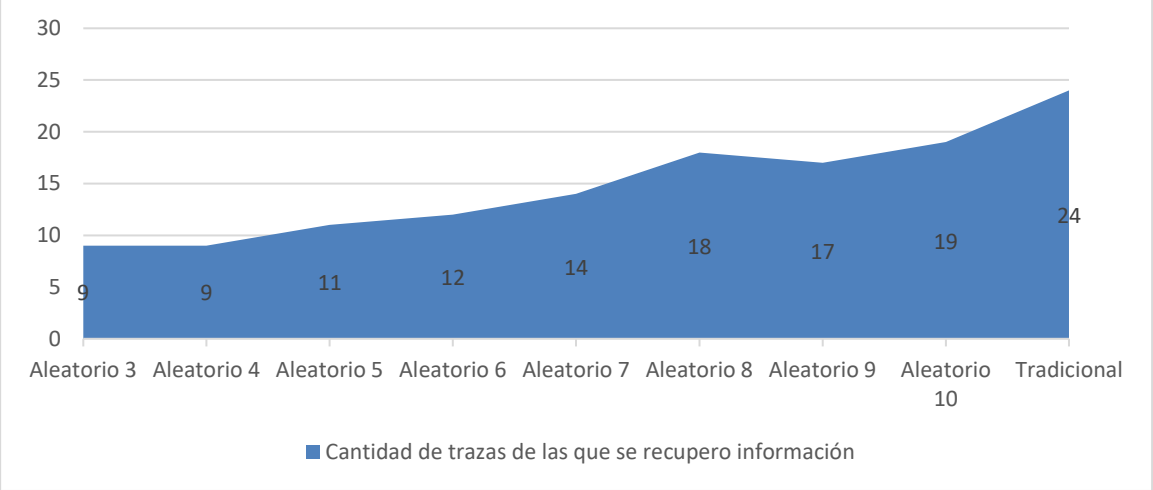

Ilustración 50 Cantidad de features con trazas asociadas por número de escenario.

5.6.3 Desempeño con respecto a otras técnicas

Con el fin de incentivar el uso de un escenario estandarizado, se realizó la comparación utilizando el escenario tradicional con otras técnicas ya implementadas en BUT4Reuse. Esto debido a que es el escenario más utilizado como se mencionó en la sección 2.4. 4..

NAME	PRECISION	RECALL	FSCORE
FCA + FEATURE SPECIFIC HEURISTIC	0,0087	0,1621	0,0160
FCA + SFS + TF	0,4376	0,3175	0,3566
FCA + SFS + TFIDF	0,4376	0,3175	0,3566
FCA + SFS + LSI (LÍNEA BASE)	0,7466	0,6943	0,6484

Tabla 13 Resultados de métricas utilizando diferentes algoritmos.

El resultado presentado por la técnica de línea base muestra un mejor desempeño a comparación de las otras (Tabla 13). Lo que permite afirmar que la técnica propuesta supera en rendimiento algunas otras técnicas de la literatura, esto se puede visualizar más fácil en la ilustración 49. Lo que supone pensar que esto incentivará a la invención de nuevos métodos que intenten superar los valores dados.

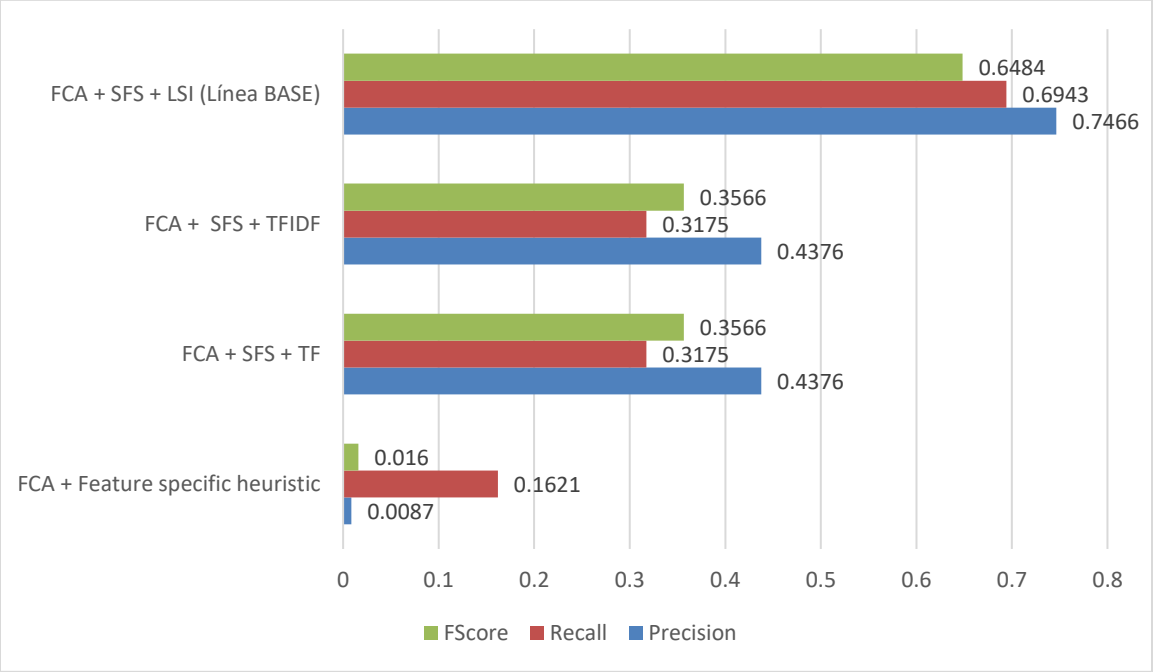

Ilustración 51 Comparación de técnicas aplicadas al escenario tradicional del benchmark.

5.7 **Resumen**

A lo largo del capítulo se mostró una explicación detallada de los algoritmos que se utilizan para la recuperación de características y de cómo se complementan entre sí para el desarrollo de la técnica de línea base planteada. También, se mostró su adaptación al GroundTruth establecido en al capítulo anterior para que se pudiera comparar con futuros estudios. Como valor agregado se realizó dicha adaptación para tres algoritmos adicionales y se compararon los resultados. Acorde al objetivo tres del trabajo de tesis, se implementó una técnica que mitigará las dificultades de comparación de algoritmos. Se encontró que el conjunto de algoritmos que se decidió establecer como línea base propone un gran reto para futuras investigaciones presentando resultados prometedores pero mejorables.

6. Conclusiones y recomendaciones

6.1 Conclusiones

El trabajo anterior constó de: Revisión del estado del arte del caso de estudio de SPL más utilizado, y revisión de algoritmos utilizados para la identificación automática de características; creación de una prueba capaz de unificar las distintas investigaciones, lo que permitiera comparar los resultados entre las mismas; y finalmente, el desarrollo de una técnica que sirva como línea base utilizando los algoritmos comúnmente utilizados en el campo de identificación de características.

Por lo anterior, el trabajo ha sido capaz de mostrar el estudio realizado en torno a un caso de estudio de interés para el campo de SPL, cumpliendo con el objetivo principal de la tesis. El análisis presentado es una recopilación de los algoritmos, escenarios y metodologías de evaluación que giran en torno a la identificación de features, y que proponen mejoras al momento de realizar futuras investigaciones.

Adicionalmente, se ha promovido el uso de él por medio de escenarios de pruebas establecidos que permitan comparar las diferentes técnicas de identificación de features, este último ha sido tan relevante que fue presentado en SPLC2018[12] y sigue presente para SPLC2019[13]. Ya que por medio del estudio sistemático realizado se puede evidenciar la dificultad que existe para comparar el rendimiento de distintas técnicas. Esto, puede explicar el motivo por el cual el desempeño no muestra resultados buenos (Ver sección 5.5.), ya que la falta de estándares de medición dificulta la tarea de diseñar diferentes enfoques que contrasten las fortalezas y debilidades de las técnicas.

Finalmente, el desarrollo de una técnica de línea base construido en base a los algoritmos más utilizados en la literatura, establecen un punto de partida en el cual se propone un objetivo al cual superar. Lo anterior favorece al desarrollo de investigaciones que giran en

[12] Challenge case. Software Product Line Conference 2018. http://kishi-lab.sakura.ne.jp/splc2018/call-for-papers/call-for-challenge-solutions/challenge-case-2-feature-location-benchmark-with-argouml-spl/
[13] Call for solutions. Software Product Line Conference 2019. https://splc2019.net/call-for-papers/call-for-challenge-solutions/

torno a la identificación de features para que se llegue al estado deseado de recuperar un alto número de features basado en un bajo número de variantes como entrada. Lo que permitirá que se pueda establecer en algún momento, basado en unos valores de métricas muy altos, la implementación en un ambiente industrial.

6.2 Recomendaciones

Al realizar el presente trabajo se mostró lo importante que es considerar la utilización de escenarios generales que permitan comparar resultados para cualquier caso de estudio que se utilice como caso de prueba de un enfoque investigativo.

6.3 Trabajo futuro

Existen muchos factores a investigar aún, siguiendo en línea con el actual enfoque:

- Desarrollo de nuevos escenarios de prueba en casos de estudio utilizados como MobileMedia.
- Construcción de un repositorio de benchmark que organice los diferentes tipos de prueba por categorías.
- Desarrollo de nuevas metodologías que promuevan el aumento de rendimiento de las técnicas.
- Evaluación del estudio de herramientas en la nube y utilización de GPUs para el desarrollo de nuevas metodologías que permitan procesar más información.

SPL es un campo que aún tiene muchas opciones de investigación tanto en el desarrollo de identificación de features como en otras áreas. Según los resultados presentados en la sección 5.5.3., se puede evidenciar la necesidad de utilizar otras técnicas de inteligencia artificial que puedan ayudar al desempeño. Más aún, con respecto a los rendimientos en términos de tiempo, se debe plantear una forma óptima de representar un proyecto de software que tenga en cuenta un alto nivel de granularidad, lo que ayude a que el proceso completo posea un mayor grado de escalabilidad.

A. Anexo: Comparación de artículos que utilizan ArgoUML-SPL

B. Anexo: Análisis de los elementos de software reflejados en formato del benchmark propuesto para el caso de estudio: Casa domótica.

7. Bibliografía

1. Pohl, K., Buckle, G., & Van Der Linden, F. (2005). Software product line engineering: Foundations, principles, and techniques. Software Product Line Engineering: Foundations, Principles, and Techniques.
2. Lenz, G., & Weinands, C. (2006). Practical Software Factories in .net. Source.
3. Krueger, C. Easing the Transition to Software Mass Customization. Proceedings of the 4th International Workshop on Product Family Engineering. October 2001. Bilbao, Spain. Springer-Verlag, New York, NY.
4. Charles W. Krueger. Software Reuse. ACM Computing Surveys, 24(2):131–183, 1992.
5. Software Engineering Institute | Carnegie Mellon University. (2014). A Framework for Software Product Line Practice, Version 5.0 - Organizational Risk Management, 1–2.
6. Kang, K. C., Cohen, S. G., Hess, J. a, Novak, W. E., & Peterson, a S. (1990). Feature-Oriented Domain Analysis (FODA) Feasibility Study. Distribution, 17(November), 161.
7. M. D. McIlroy (1968). Mass produced software components, NATO-sponsored meeting on Software Engineering, Garmisch, Germany.
8. Edsger W. Dijkstra (1969), Notes in Structure Programming.

9. David Parnas (1976). On the design and development of program families. IEEE transactions on software engineering. Vol. SE 2, N° 1.
10. Martinez, J. (2016). Mining Software Artefact Variants for Product Line Migration and Analysis. PHD Dissertation.
11. Gomaa, H. (2004). Designing Software Product Lines with UML: From Use Cases to Pattern---Based Software Architectures. Addison---Wesley Professional.
12. Metzger, A., & Pohl, K. (2014). Software Product Line Engineering and Variability Management: Achievements and Challenges. Future of Software Engineering (FOSE).
13. Heradio, R., Perez-Morago, H., Fernandez-Amoros, D., Javier Cabrerizo, F., & Herrera-Viedma, E. (2016). A bibliometric analysis of 20 years of research on software product lines. Information and Software Technology, 72.
14. Wolfram Fenske and Sandro Schulze (2015). Code Smells Revisited: A Variability Perspective. In International Workshop on Variability Modelling of Software-Intensive Systems, pages 3–10. ACM.
15. Assunciao, W. K. G., Lopez-Herrejon, R. E., Linsbauer, L., Vergilio, S. R., Egyed, A., Assunção, W. K. G., Egyed, A. (2017). Reengineering legacy applications into software product lines: a systematic mapping. Empirical Software Engineering, 22(6), 1–45.
16. Couto, M. V., Valente, M. T., & Figueiredo, E. (2011). Extracting Software Product Lines: A Case Study Using Conditional Compilation. 2011 15th European Conference on Software Maintenance and Reengineering.
17. S. Apel and C. Kastner, (2009) “Virtual separation of concerns ¨ - a second chance for preprocessors,” Journal of Object Technology, vol. 8, no. 6, pp. 59–78.
18. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, (1997) “Aspect-oriented programming,” in 11th European

Conference on ObjectOriented Programming (ECOOP), ser. LNCS, vol. 1241. Springer Verlag, pp. 220–242.

19. Liu, J., Batory, D., & Lengauer, C. (2006). Feature oriented refactoring of legacy applications. Proceeding of the 28th International Conference on Software Engineering - ICSE ’06.
20. Eisenbarth, T., Koschke, R., & Simon, D. (2003). Locating features in source code. IEEE Transactions on Software Engineering, 29(3), 210–224.
21. Christopher D. Manning Prabhakar Raghavan Hinrich Schütze,(2009) Introduction to Information Retrieval. Cambridge University Press Cambridge, England.
22. Eyal-Salman, H., Seriai, A. D., & Dony, C. (2013). Feature-to-code traceability in a collection of software variants: Combining formal concept analysis and information retrieval. Proceedings of the 2013 IEEE 14th International Conference on Information Reuse and Integration, IEEE IRI 2013, 209–216.
23. G. Salton, A. Wong, C.S. Yang, A vector space model for automatic indexing, Communications of the ACM 18 (11) (1976) 613–620.
24. Martinez et al. (2017) ESPLA: A Catalog of Extractive SPL Adoption Case Studies. SPLC 2017.
25. Couto, M. V, Valente, M. T., & Figueiredo, E. (2011). Extracting software product lines: A case study using conditional compilation.
26. Martinez, J., Ordoñez, N., Tërnava, X., Ziadi, T., Aponte, J., Figueiredo, E., (2018). Feature Location Benchmark with ArgoUML SPL. SPLC 2018.
27. Wesley K. G. Assunção, Roberto E. Lopez-Herrejon, Lukas Linsbauer, Silvia R.Vergilio, and Alexander Egyed. 2017. Reengineering legacy applications into software product lines: a systematic mapping. Empirical Software Engineering 22, 6 (2017), 2972–3016
28. Sven Apel, Sergiy S. Kolesnikov, Norbert Siegmund, Christian Kästner, and Brady Garvin. 2013. Exploring feature interactions in the wild: the new feature- interaction challenge. In 5th International Workshop on Feature-

Oriented Software Development, FOSD ’13, Indianapolis, IN, USA, October 26, 2013, Andreas Classen and Norbert Siegmund (Eds.). ACM, 1–8.

29. Assunção, W. K. G., & Vergilio, S. R. (2014). Feature location for software product line migration. Proceedings of the 18th International Software Product Line Conference on Companion Volume for Workshops, Demonstrations and Tools - SPLC ’14.
30. Rubin, J., & Chechik, M. (2013). A survey of feature location techniques. Domain Engineering: Product Lines, Languages, and Conceptual Models, 29–58.

    Fat, R., Deen, A. A., Seriai, A., Huchard, M., Fat, R., Deen, A. A., … Huchard, M. (2014). Reengineering Software Product Variants into Software Product Line : REVPLINE Approach To cite this version : HAL Id : lirmm-00981473.
31. Dubinsky Y, Rubin J, Berger T, Duszynski S, Becker M, Czarnecki K (2013) An exploratory study of cloning in industrial software product lines 17th European Conference on Software Maintenance and Reengineering (CSMR), pp 25–34.
32. Alves V, Schwanninger C, Barbosa L, Rashid A, Sawyer P, Rayson P, Pohl C, Rummler A (2008) An exploratory study of information retrieval techniques in domain analysis. In: 12th international software product line conference, SPLC, pp 67–76.
33. Petersen, K., Vakkalanka, S., & Kuzniarz, L. (2015). Guidelines for conducting systematic mapping studies in software engineering: An update. In Information and Software Technology (Vol. 64).
34. Young, T. J. (2005). Using AspectJ to Build a Software Product Line for Mobile Devices. University of British Columbia, Department of Computer Science, (August), 1–67.
35. Figueiredo, E., Cacho, N., Sant’Anna, C., Monteiro, M., Kulesza, U., Garcia, A., … Dantas, F. (2008). Evolving Software Product Lines with Aspects: An

Empirical Study on Design Stability. Software Engineering, 2008. ICSE ’08. ACM/IEEE 30th International Conference On, 261–270.

36. Powers, David M W (2011). Evaluation: From Precision, Recall and F-Measure to ROC, Informedness, Markedness & Correlation. Journal of Machine Learning Technologies. 2 (1): 37–63.
37. Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2014. Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants. In ICSME. IEEE Computer Society, 391–400.
38. Eyal-Salman, H., Seriai, A., & Dony, C. (2014). Feature-Level Change Impact Analysis Using Formal Concept Analysis. *Proceedings of the 26th International Conference on Software Engineering and Knowledge Engineering (SEKE 2014)*, 447–452.
39. Martinez, J., & Traon, Y. Le. (n.d.). Name Suggestions during Feature Identification : The VariClouds Approach, 119–123.
40. B. Ganter, R. Wille, Formal Concept Analysis: Mathematical Foundations, 1st ed., Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997
41. Linsbauer, L., Lopez-Herrejon, E. R., & Egyed, A. (2013). Recovering traceability between features and code in product variants. Proceedings of the 17th International Software Product Line Conference on - SPLC ’13, 131.
42. Al-msie ’deen, R., Seriai, A.-D., Huchard, M., Urtado, C., & Vauttier, S. (2013). Mining features from the object-oriented source code of software variants by combining lexical and structural similarity. 2013 IEEE 14th International Conference on Information Reuse & Integration (IRI).
43. Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., & Harshman, R. (1990). Indexing by latent semantic analysis. Journal of the American Society for Information Science, 41(6), 391–407.
44. Grossman, Frieder’s (2004). Information Retrieval, Algorithms and Heuristics. Springer Netherlands

45. Landauer, T. K., Folt, P. W., & Laham, D. (1998). An introduction to latent semantic analysis. Discourse Processes, 25(2), 259–284.
46. Xing, E. (n.d.). LECTURE : LATENT SEMANTIC INDEXING (LSI).
47. Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., & Harshman, R. (1990). Indexing by latent semantic analysis. Journal of the American Society for Information Science, 41(6), 391–407.
48. Travassos, G. H., Santos, P. S. M. dos, Mian, P. G., Neto, A. C. D., & Biolchini, J. (2008). An Environment to Support Large Scale Experimentation in Software Engineering. 13th IEEE International Conference on Engineering of Complex Computer Systems (iceccs 2008). doi:10.1109/iceccs.2008.30
49. Wohlin, C. (2014). Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering.