Graduada en Ingeniería de la Salud

Biblioteca de transformaciones de modelos de características para gestionar la variabilidad de familias de productos software

Realizado por

Adriana Novoa Hurtado

Dirigido por

Lidia Fuentes Fernández y José Miguel Horcas Aguilera

Departamento

Lenguajes y Ciencias de la Computación

MÁLAGA, SEPTIEMBRE 2023

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADUADA EN INGENIERÍA DE LA SALUD

Biblioteca de transformaciones de modelos de características para gestionar la variabilidad de familias de productos software

Realizado por

Adriana Novoa Hurtado

Dirigido por

Lidia Fuentes Fernández y José Miguel Horcas Aguilera

Departamento

Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA

MÁLAGA, SEPTIEMBRE DE 2023

Abstract

Current systems exhibit high variability, necessitating the search for effective methods to manage the large number of configurations they present. Software Product Lines (SPL) represent one of the primary technologies enabling this capability. To model this variability effectively, it is crucial to find resources that can foster interoperability among existing tools that work with feature models. As this technology is still emerging, many variability constructs remain unsupported by most existing tools. Therefore, the main objective of this TFG (Final Degree Project) is to implement a set of transformations, known as edits, that enable interoperability across all existing tools, facilitating progress in this emerging area. The specific type of transformations to be employed is called refactorings, as they are the only ones that preserve the original feature model semantics among all possible transformations. This TFG proposes a reusable feature model refactoring library that allows transforming variability concepts into another supported by feature modeling tools.

Keywords: variability, configurations, SPL, interoperability, feature model, refactorings.

Resumen

La alta variabilidad de los sistemas actuales hacen necesario buscar maneras para gestionar la cantidad de configuraciones que presentan. Para ello se dispone de una de las principales tecnologías que lo hacen posible, que son las Líneas de Producto Software (SPL, del inglés *Software Product Lines*). Para modelar dicha variabilidad es necesario buscar algún recurso que permita generar interoperabilidad entre las herramientas que existen hoy día, que trabajan con modelos de características. Como esta tecnología conforma un campo emergente y novedoso, hay muchos constructores de variabilidad que no son soportados por la mayoría de herramientas existentes. Es por eso que el objetivo de este TFG es implementar una serie de transformaciones que brinden interoperabilidad entre todas las herramientas existentes para facilitar el avance de esta área en desarrollo. De todas las posibles transformaciones que se pueden hacer a un modelo, se ha decidido implementar una serie de *refactorings*, ya que este tipo de transformaciones no modifican la semántica del modelo original. El TFG propone una biblioteca reusable de refactorings para modelos de características que permite transformar los conceptos de modelo de la variabilidad usados por otros que soporten todas las herramientas existentes de modelos de características.

Palabras clave: variabilidad, configuraciones, SPL, interoperabilidad, modelo de características, transformaciones, *refactorings*.

Índice general

Índice de figuras

Índice de tablas

1
Introducción

1.1. Contexto

Los sistemas de hoy en día son altamente configurables y exponen mucha variabilidad. Una de las tecnologías principales para gestionar la variabilidad y las diferentes configuraciones de un sistema son las **Líneas de Producto Software** (SPL del inglés *Sofware Product Line*) [Pohl et al.(2005)]. Las SPLs son familias de productos software relacionados que comparten muchas características comunes (similitudes) pero que también tienen diferencias (características variables). En una SPL, los sistemas se descomponen en **características** (*features*) que representan funcionalidades o comportamientos del sistema [Apel et al.(2013)].

El principal artefacto para modelar la variabilidad en las SPLs son los **modelos de variabilidad** (*variability models*) y más concretamente los **modelos de características** (*feature models*) [Kang et al.(1990)], que son los modelos de variabilidad más usados en la práctica [Raatikainen et al.(2019), Horcas et al.(2023a)]. Un modelo de características es una herramienta para especificar las similitudes y diferencias (variabilidad) en los sistemas software descomponiendo las características del sistema en una estructura jerárquica en forma de árbol con relaciones padre-hijo. El objetivo del uso de este tipo de modelos es expresar de una manera clara y concisa el conjunto en sí de productos que comparten características comunes y que también tienen algunas diferencias. En el contexto de las SPLs, un producto se representa como un conjunto de características de un modelo de características [Apel et al.(2013)]. El hecho de expresar las SPLs con modelos de características permite ahorrar tiempo y esfuerzo, al tener codificados una gran cantidad de productos en un único modelo.

En la Figura 1.1 se puede observar un primer modelo simple de características que servirá para introducir algunos tipos de relaciones que se pueden dar entre las características de dicho modelo. En este modelo se han representado los diferentes productos que existen en un restaurante de comida rápida de pizzas. Un modelo de características contiene dos partes principales bien diferenciadas:

- **El árbol de características (*feature tree*)**: son las características que describen el modelo y las relaciones que existen entre ellas.
    - **Características (*features*)**: A nivel del modelo se diferencias dos tipos de características: las **características concretas** (*concrete features*) que son aquellas características imprescindibles que describen la funcionalidad o comportamiento del sistema, y las **características abstractas** (*abstract features*) que son sólo para or-

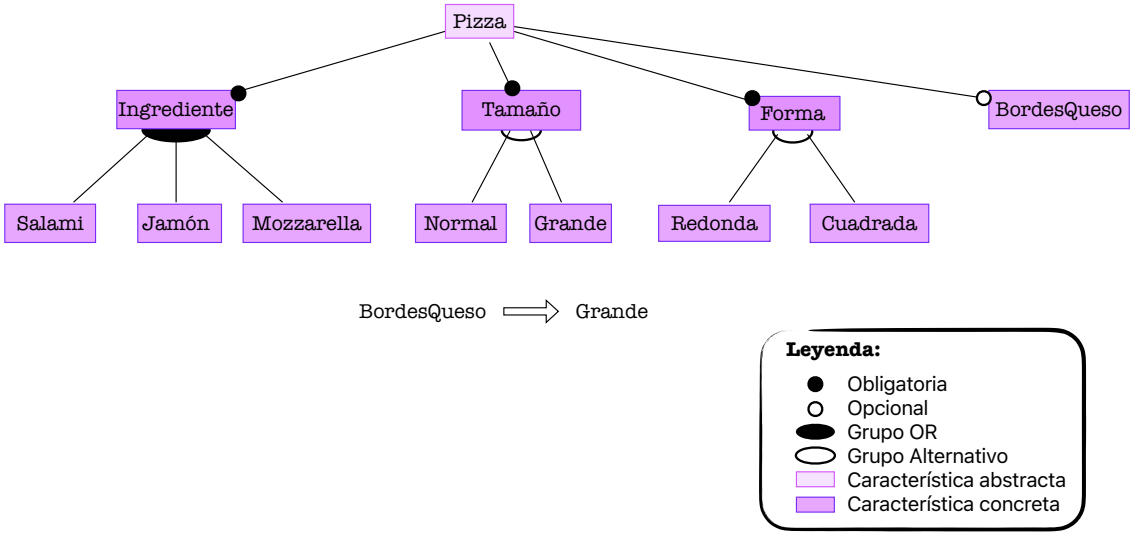

Figura 1.1: Modelo simple de características representando el menú de una pizzería.

ganizar el resto de características dentro del modelo, pero que no tienen ninguna asociación directa con una funcionalidad o comportamiento del sistema.

- **Relaciones (*relations*)**: Las características son descompuestas en el árbol usando diferentes tipos de relaciones padre-hijo. Pueden ser relaciones simples (con un solo hijo) o grupos (con varios hijos). Cada característica hija tiene siempre un solo padre. De esta forma algunas características son **obligatorias**, otras son **opcionales**, y también hay diferentes grupos asociados a una característica, como **grupos de selección ("or")** donde hay que seleccionar al menos una característica si la característica padre ha sido seleccionada, y **grupo alternativos ("xor")** donde sólo se puede escoger una de las características hija.

- **Restricciones textuales (*cross-tree constraints*)**: Un modelo de característica puede tener restricciones textuales para especificar relaciones más complejas entre las características que no pueden ser modeladas en el árbol. Las restricciones textuales suelen especificarse con fórmulas lógicas de diferente complejidad, pero los tipos de restricciones más simples incluyen restricciones del tipo *requires* (una característica requiere la presencia de otra) y del tipo *excludes* (una característica solo puede estar si no está la otra).

Siguiendo con el ejemplo de la Figura 1.1, la primera característica que aparece (`Pizza`) en la parte superior del modelo es la característica raíz (*root*), y no tiene ninguna característica

padre. En este caso la característica Pizza tiene cuatro características hijas. Como se puede observar en este modelo, tanto las características Ingrediente, Tamaño y Forma son obligatorias, lo que quiere decir que las tres irán siempre escogidas en todos los productos. La cuarta característica hija de Pizza es BordesQueso para representar la opción de una pizza con los bordes rellenos de queso. Esta característica es opcional porque irá seleccionada si el usuario la pide, pero no es necesaria para completar una pizza. A su vez, la característica Ingredientes es un grupo "or", que quiere decir que tiene cardinalidad $[1..n]$, es decir, se pueden escoger de 1 a 3 ingredientes de entre sus características hijas: Salami, Jamón y Mozzarella. Por otro lado, tanto en la característica Tamaño como Forma son grupos "xor" que tienen una cardinalidad $[1..1]$, lo que quiere decir que solo se puede escoger un único hijo. Así, solo podemos escoger uno de los tamaños: Normal o Grande; y una de las formas: Redonda o Cuadrada para las pizzas.

Los modelos de características [Benavides et al.(2010)] se pueden formalizar a lógica proposicional para obtener de forma automática el conjunto de configuraciones válidas que representa. Una **configuración válida** es un conjunto de características que cumple con todas las relaciones y restricciones del modelo. Por ejemplo, una de las configuraciones del modelo de características de la Figura 1.1 sería:

Listado 1.1: Una posible configuración para el modelo de Pizzas.

```
[Pizza, Ingrediente, Salami, Tamano, Grande, Forma, Cuadrada, BordesQueso]
```

La diferencia entre una configuración y un producto viene dada por el contraste entre las características abstractas y concretas. Un **producto** es una configuración válida del modelo donde las características abstractas han sido eliminadas. Así, el producto resultante para la configuración del ejemplo del Listado 1.1 sería:

Listado 1.2: Producto resultante de la configuración de ejemplo para el modelo de Pizzas.

```
[Ingrediente, Salami, Tamano, Grande, Forma, Cuadrada, BordesQueso]
```

En el Listado 1.2 se puede observar que no aparece la característica Pizza, ya que es abstracta y sólo sirve para organizar el modelo.

1.2. Motivación

Los modelos de características se pueden especificar en varios lenguajes de dominio específico o en base a meta-modelos [Horcas et al.(2023b)]. Uno de los lenguajes más usados recientemente en la comunidad de SPL para especificar los modelos de características es el lenguaje **UVL** (*Universal Variability Language*) [Sundermann et al.(2021)]. El propósito de UVL es ser lo más simple posible y ser extensible, cumpliendo las necesidades de las SPL. El problema es que actualmente UVL está en sus comienzos y soporta solamente conceptos de modelado de la variabilidad muy simples. En el código 1.3 se puede observar cómo se especifica el modelo de características de la Figura 1.1 en el lenguaje UVL.

Listado 1.3: Código del modelo de características simple para Pizza

```
namespace Pizza

features
  Pizza {abstract}
    mandatory
      Ingrediente
        or
          Salami
          Jamon
          Mozzarella
      Tamano
        alternative
          Normal
          Grande
      Forma
        alternative
          Redonda
          Cuadrada

    optional
      BordesQueso

constraints
  BordesQueso => Grande
```

Además de UVL, existen otros lenguajes y formatos para especificar los modelos de características así como algunas herramientas capaces de entender los modelos de características en estos lenguajes. Por ejemplo, los modelos de características se pueden definir en el lenguaje **Clafer** [Juodisius et al.(2019)] usado por la herramienta con el mismo nombre, en **XML** usado por la herramienta FeatureIDE [Thüm et al.(2014)], en **SXFM** usado por la herra-

mienta SPLOT [Mendonca et al.(2009)], o en **JSON** usado por la herramienta Glencoe, entre otros [Anna Schmitt(2018)].

Uno de los mayores problemas que se han encontrado a la hora de trabajar con modelos de características ha sido la falta de interoperabilidad entre los lenguajes y herramientas que se utilizan en las SPLs. Esto se debe a que para modelar la variabilidad de los sistemas actuales se necesitan conceptos de modelado más avanzados que aquellos que soportan la mayoría de herramientas existentes. Por ejemplo, una restricción textual del modelo de características puede ser una expresión booleana bastante compleja, sin embargo, la mayoría de las herramientas de modelado de la variabilidad solo soportan restricciones textuales del tipo "requires" y "excludes". Otro ejemplo es la posibilidad de especificar grupos de características con cardinalidad genérica $[a..b]$, pero solo unas pocas herramientas soportan esta característica. Las herramientas que se utilizan en este TFG son FeatureIDE, Flama, Glencoe, SPLOT y Clafer. Por ejemplo, Glencoe solo es capaz de leer y trabajar con modelos que estén definidos en JSON. Tanto SPLOT como Clafer soportan únicamente modelos definidos en SPLOT y Clafer respectivamente. FeatureIDE es capaz de leer los modelos de características definidos en XML, pero también soporta SXFM y UVL. Flama es capaz de leer modelos en UVL, XML, SXFM, y JSON.

Para lidiar con estos problemas, surgen las **transformaciones de modelo** aplicadas a los modelos de características (*edits*) [Thüm et al.(2009)]. Existen cuatro tipos de transformaciones de modelo: *refactorings*, *specializations*, *generalizations* y *arbitrary edits*. En este TFG nos centraremos en los *refactorings*, porque son las únicas transformaciones de modelo que mantienen la semántica de los modelos de características. Es decir, los productos representados por los modelos de características se mantienen [Horcas et al.(2023b)]. Hay muchos tipos de *refactorings*, desde aquellos que modifican la jerarquía de características del modelo hasta aquellos *refactorings* que modifican las restricciones textuales expresadas en lógica booleana, o aquellos que modifican ambos (el árbol de características y las restricciones). A lo largo del TFG se describirán cada uno de los *refactorings* que se han implementado y se estudiará de qué manera alteran los modelos de características para conseguir interoperabilidad entre las distintas herramientas que se usan para las SPL

El TFG surge de la necesidad de proporcionar interoperabilidad entre diferentes notaciones de modelos de características soportados por las herramientas existentes; de simplificar el análisis automático de los modelos de características; y del estudio de las propiedades de los

modelos de características, como su *naturalidad*, *correctitud* y *completitud*.

1.2.1. Objetivos

Se pretende identificar, formalizar e implementar un conjunto de *refactorings* que permitan un mejor análisis de sistemas y sus modelos de variabilidad, y permitan interoperabilidad entre diferentes tipos de herramientas, entre otras aplicaciones. Se van a desarrollar una serie de tests para probar el buen funcionamiento de cada uno de los *refactorings*. Una vez se compruebe el buen funcionamiento de cada *refactoring*, así como la estructura, naturalidad, y capacidad de análisis de los modelos resultantes.

Concretamente, se definen los siguientes objetivos:

- Desarrollar una biblioteca de *refactorings* (transformaciones de modelos de características sin modificaciones de productos) para los modelos de características con el propósito de proporcionar interoperabilidad entre las diferentes herramientas que existen.
- Identificar, formalizar, e implementar transformaciones de modelos basados en *refactorings* para los modelos de características dando como resultado una biblioteca de *refactorings*.
- Evaluar de forma cualitativa y cuantitativa los *refactorings* desarrollados, estudiando su naturalidad, correctitud y completitud.
- Integrar los *refactorings* en una herramienta web existente (Rhea) para la gestión de los modelos de características de forma que facilite la interoperabilidad de los *refactorings* con otras herramientas de SPL como FeatureIDE, Glencoe o SPLOT.

La herramienta principal que se ha usado en este TFG para el desarrollo de la biblioteca de transformaciones de modelo (*refactorings*) ha sido la librería de Flama [Galindo et al.(2023)], que está implementada en el lenguaje Python. Se ha escogido esta herramienta porque proporciona soporte para leer y serializar modelos de características en diferentes formatos, incluido UVL. Además, permite modelar conceptos avanzados de variabilidad para los cuales podremos definir *refactorings* y convertirlos en constructores de la variabilidad más simples.

1.3. Estructura de los contenidos

A continuación se describe brevemente el contenido de cada uno de los capítulos de este documento:

Capítulo 1: Introducción. Se presenta la motivación del proyecto y los objetivos del mismo, así como las aportaciones realizadas; y se realiza un breve repaso a la actualidad en el ámbito del uso de *refactorings*.

Capítulo 2: Estado del arte. Se hace una revisión del estado del arte de las líneas de productos software, de los modelos de variabilidad y modelos de características, así como de transformaciones de modelos y *refactorings*.

Capítulo 3: Metodología. Capítulo donde se estructura las fases del desarrollo, así como las pruebas y resultados.

Capítulo 4: Biblioteca de *refactorings*. Se describe de forma general la biblioteca de *refactorings* propuesta. Se especifican los requisitos generales de la biblioteca y se muestra su arquitectura.

Capítulo 5: *Refactorings* de modelos de características. Se detalla cada *refactoring* y se ilustra con un ejemplo en un *feature model* antes y después de realizar dicho *refactoring*.

Capítulo 6: Implementación y evaluación. Se muestra la implementación de la biblioteca de *refactorings*, así como las pruebas que se han llevado a cabo para comprobar la *correctitud* y *validez* de los mismos.

Capítulo 7: Conclusiones y Líneas Futuras. Se exponen las conclusiones derivadas de la ejecución del proyecto y se detallan las perspectivas futuras que se proponen en relación al mismo, incluyendo posibles ampliaciones y mejoras.

Apéndice A: Manual de usuario. El contenido de este apéndice es el manual de usuario de la biblioteca de *refactorings*; explica su instalación y su manejo básico.

2
Estado del arte

En este capítulo se van a introducir los conceptos más importantes relacionados con las líneas de productos software, los modelos de características, así como de las transformaciones de modelos y los *refactorings*. Es necesario tener en cuenta que es un campo emergente y aún hay muchas limitaciones. Actualmente se está trabajando para hacer frente a estas limitaciones, bien proponiendo la creación de un nuevo lenguaje de variabilidad (como puede ser UVL), o bien proporcionando interoperabilidad entre los lenguajes existentes como se propone en este TFG con ayuda de la biblioteca de *refactorings*.

2.1. Modelos de características y extensiones para modelar la variabilidad

Los modelos de características [Kang et al.(1990)] han sido ampliamente usados para modelar la variabilidad desde 1990 cuando fueron introducidos en FODA (Feature-Oriented Domain Analysis). Dado el éxito de los modelos de características, se han propuesto muchas extensiones y lenguajes para incrementar la expresividad de los modelos [Horcas et al.(2023b)].

2.1.1. Conceptos de modelado de la variabilidad básicos

Los conceptos o constructores básicos para modelar la variabilidad se han presentado en el modelo de la Figura 1.1. Esto incluye:

- Características concretas y abstractas.
- Características obligatorias y opcionales.
- Grupos de características, incluyendo:
  - Grupos or, con cardinalidad [1..n].
  - Grupos alternativos (o xor), con cardinalidad [1..1].
- Restricciones básicas, incluyendo:
  - Restricciones del tipo “requires”.
  - Restricciones del tipo “excludes”.

Ciertos autores [Sepúlveda et al.(2016)] indican que hay un bajo nivel de expresividad en los lenguajes existentes para poder modelar la variabilidad presente en los sistemas de hoy en día. Para solventar esto, diferentes autores han ido proponiendo extensiones de los modelos de características añadiendo nuevos constructores de lenguajes o conceptos avanzados para modelar la variabilidad. A continuación, se presentan algunas de estas extensiones, de las cuales se desarrollarán los *refactoring* para representarlos con constructores básicos.

2.1.2. Conceptos de modelado de la variabilidad avanzados

En el modelo de la Figura 2.1 se pueden observar nuevos constructores avanzados para modelar la variabilidad que no estaban en el modelo básico de la Figura 1.1. El problema de estas extensiones es que no están soportadas por la mayoría de las herramientas que trabajan con modelos de características, de ahí la necesidad de una biblioteca como la propuesta en este TFG. Estos constructores avanzados son los siguientes:

- **Grupo XOR con una característica obligatoria *(XOR+Mandatory)***: en este caso la idea es muy similar a la del grupo "xor", pero en este caso una de las características hijas que pertenece al grupo es obligatoria. Al ser posible solamente elegir una característica perteneciente a un grupo alternativo, las restantes quedan "muertas", lo que quiere decir que no van a aparecer en ninguna configuración ni producto del modelo de características. En la Figura 2.2 se puede observar el grupo "xor+mandatory" aislado.

- **Grupo OR con una característica obligatoria *(OR+Mandatory)***: es la misma idea de grupo "or", pero en este caso una de las características hijas que pertenecen a ese grupo es obligatoria. Por tanto, si solo se escoge una característica, debe ser esa misma la elegida. En el caso de que se escojan más de una característica dentro de ese grupo, seguirá siendo obligatorio que una de las características elegidas sea la marcada como obligatoria. En la Figura 2.3 se puede observar el grupo "or+mandatory" aislado.

- **Grupos de descomposición múltiple *(Multiple Group Decomposition)***: son cualquier combinación de relaciones agrupadas bajo una misma característica que sería su padre. Este constructor de variabilidad está presente en el lenguaje usado en el artículo [Czarnecki and Wasowski(2007)] que permite tener grupos de descomposición múltiple (por ejemplo, un grupo "xor" y otro grupo "or" bajo la misma característica). Como se puede observar en la Figura 2.4, en este caso tenemos por un lado un grupo "xor", y después bajo la misma característica `Particularidades` hay otro grupo "or".

- **Grupo Mutex *(Mutex Group)***: Un grupo mutex especifica un grupo de características donde como mucho una puede ser seleccionada en una configuración. Esto sería equivalente a un grupo cuya cardinalidad sea $[0..1]$. En la Figura 2.5 se puede observar la característica `Extras` representando un grupo mutex con 4 sub-características: `Orégano`,

`AceitePicante`, `Tomillo`, y `Romero`. En una configuración de la pizza, podemos escoger entre 0 y 1 extras.

Los grupos mutex [Berger et al.(2013)] son un tipo de relación de descomposición que ocurre frecuentemente en el dominio de los sistemas operativos (e.g., en los sistemas KConfig y CDL).

- **Grupo de Cardinalidad *(Cardinality Group)***: consiste en un grupo similar al "or", pero en este caso la cardinalidad del grupo es aleatoria, es decir, $[a..b]$ (donde $a \geq 0$, $b \leq n$, y n es el número total de características dentro del grupo de cardinalidad). Por tanto, se pueden escoger entre un número cualquiera, a, y otro número cualquiera, b, de características dentro de ese grupo. En la Figura 2.6 se puede observar el grupo "cardinality" aislado.

- **Restricción textual pseudo-compleja *(Pseudo Complex constraint)***: es una manera de relacionar dos características que se encuentran en diferentes subárboles del modelo de características. No todas las restricciones textuales complejas son del mismo tipo. Algunas restricciones complejas pueden ser traducidas a un conjunto equivalente de restricciones simples. Por ejemplo, la restricción compleja del tipo $f_1 \vee f_2 \implies f_3$ es equivalente a la conjunción de restricciones de la forma $f_1 \implies f_3$ y $f_2 \implies f_3$. Las restricciones pseudo-complejas pueden ser convertidas a un conjunto de restricciones textuales simples, mientras que las restricciones textuales estrictamente complejas no. Dicho de una manera más formal, una restricción textual pseudo-compleja es una restricción textual compleja que tiene la forma $\Psi^{cnf} = \bigwedge c_i$ donde Ψ^{cnf} es el conjunto de restricciones textuales en forma normal conjuntiva, $c_i \equiv (\neg f_1 \vee f_2)$ o $c_i \equiv (\neg f_1 \vee \neg f_2)$ para características aleatorias f_1 y f_2. En otro caso se dice que la restricción textual es estrictamente compleja [Knüppel et al.(2017)]. En la Figura 2.7 se pueden observar las restricciones textuales pseudo-complejas que se presentan en el modelo de Pizza de la Figura 2.1.

- **Restricción textual estrictamente compleja *(Strict Complex Constraint)***: es una manera de relacionar dos características que se encuentran en diferentes subárboles del modelo de características. Las restricciones textuales complejas son fórmulas de lógica proposicional sobre las características del modelo de características. Las restricciones textuales estrictamente complejas no pueden separarse en varias restricciones simples

como ocurría con las pseudo-complejas. En la Figura 2.8 se puede observar la restricción textual estrictamente compleja que se presenta en el modelo de Pizza de la Figura 2.1.

- **Restricciones textuales simples:** Aunque las restricciones textuales simples son constructores de variabilidad simples, se considerarán como avanzados a la hora de aplicar sobre ellas un *refactoring*, ya que no son soportadas por algunas herramientas de variabilidad:

  - **Restricción textual simple del tipo** $A \implies B$ ***(Requires)***: es una manera de relacionar dos características que se encuentran en diferentes subárboles del modelo de características. En este caso, si se escoge la primera característica, entonces obligatoriamente hay que escoger también la segunda. No funciona de forma inversa, lo que quiere decir que la segunda característica debe aparecer en todos los productos que aparezca la primera, pero si en un producto aparece la segunda característica, la primera característica de la restricción textual no tiene por qué aparecer junto a ella en ese producto. En la Figura 2.9 se puede observar la restricción textual simple del tipo $A \implies B$ que se presenta en el modelo de Pizza de la Figura 2.1.
  - **Restricción textual simple del tipo** $A \implies \neg B$ ***(Excludes)***: es una manera de relacionar dos características que se encuentran en diferentes subárboles del modelo de características. En este caso, si se escoge la primera característica, entonces no se puede escoger la segunda. Funciona de forma inversa, lo que quiere decir que ninguna de las dos características implicadas en la restricción textual podrán aparecer en los mismos productos. En la Figura 2.10 se puede observar la restricción textual simple del tipo $A \implies \neg B$ que se presenta en el modelo de Pizza de la Figura 2.1.

Figura 2.1: Modelo extendido de características representando el menú de una pizzería.

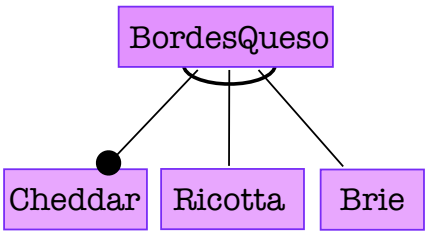

Figura 2.2: Grupo **XOR** con una característica obligatoria del modelo de Pizza de la Figura 2.1.

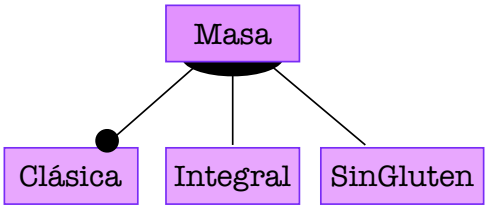

Figura 2.3: Grupo **OR** con una característica obligatoria del modelo de Pizza de la Figura 2.1.

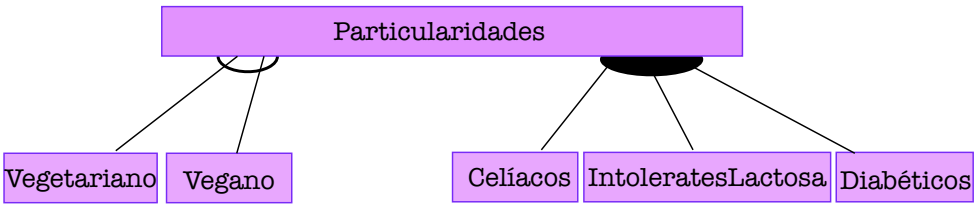

Figura 2.4: Descomposición de grupos múltiples del modelo de Pizza de la Figura 2.1.

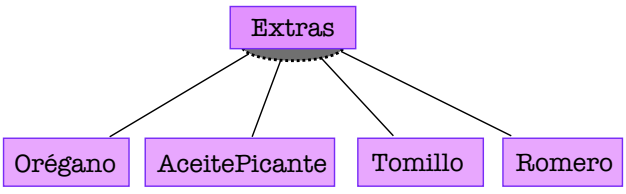

Figura 2.5: Grupo Mutex del modelo de Pizza de la Figura 2.1.

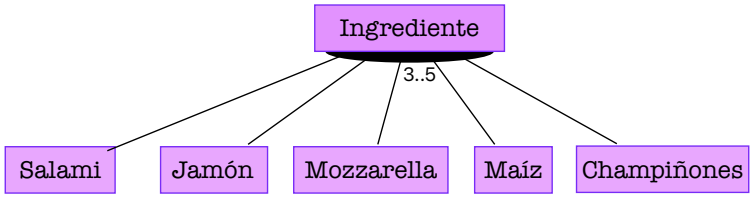

Figura 2.6: Grupo Cardinalidad del modelo de Pizza de la Figura 2.1.

$$\text{Vegano} \implies \neg\text{Salami} \wedge \neg\text{Jamón} \wedge \neg\text{Mozzarella}$$
$$\text{Vegetariano} \implies \neg\text{Salami} \wedge \neg\text{Jamón}$$

Figura 2.7: Restricciones textuales pseudo-complejas del modelo de Pizza de la Figura 2.1.

$$\text{BordesQueso} \wedge \text{Mozzarella} \implies \text{Grande}$$

Figura 2.8: Restricción textual estrictamente compleja del modelo de Pizza de la Figura 2.1.

$$\text{Celíacos} \implies \text{SinGluten}$$

Figura 2.9: Restricción textual del tipo $A \implies B$ *(Requires)* del modelo de Pizza de la Figura 2.1.

$$\text{IntolerantesLactosa} \implies \neg\text{Mozarella}$$

Figura 2.10: Restricción textual del tipo $A \implies \neg B$ *(Excludes)* del modelo de Pizza de la Figura 2.1.

2.2. Lenguajes y herramientas de soporte para modelos de características

Hay muchos lenguajes y herramientas para definir modelos de características. En esta sección se va a poner el foco en las herramientas seleccionadas en el estudio práctico publicado en [Horcas et al.(2023a)], que corresponde a los lenguajes de modelos de características conocidos y ampliamente usados en la comunidad SPL para modelos de características.

En la Sección 1.2 se habló de la interoperabilidad entre herramientas que sirven para trabajar con modelos de características. En la Tabla 2.1 se pueden observar los constructores de variabilidad que soporta cada herramienta.

Como se puede observar en la Tabla 2.1, los constructores básicos (características obligatorias, características opcionales, grupos "or", grupos "xor", y restricciones simples del tipo "requires" y "excludes") son soportados por todas la herramientas que se han usado para realizar este TFG. Las características abstractas solamente son soportadas por Flama, FeatureIDE y Clafer. De los constructores avanzados solamente Flama y Clafer son capaces de soportar la mayoría de ellos (grupos mutex, grupos de cardinalidad, restricciones textuales complejas, etc.). Los grupos de cardinalidad son también soportados por Glencoe. La única herramienta capaz de soportar los grupos de descomposición múltiple es Flama. Por otra parte, las restricciones textuales simples son soportadas por todas las herramientas, y las restricciones textuales complejas (de lógica proposicional) son soportadas completamente tanto por Flama, FeatureIDE, Glencoe y Clafer, siendo parcialmente soportadas por SPLOT.

Por lo tanto, la herramienta que más constructores soporta es Flama, seguida muy de cerca por Clafer. En este TFG se usará Flama, ya que además de dar mayor soporte al modelado y análisis de la variabilidad, se encuentra en continuo desarrollo.

Tabla 2.1: Constructores del lenguaje soportados por las diferentes herramientas de modelos de características [ter Beek et al.(2019), Horcas et al.(2023b), Horcas et al.(2023a)].

Herramientas	Característica abstracta	Característica obligatoria	Característica opcional	Grupo OR	Grupo XOR	Grupo mutex	Grupo cardinalidad	Grupo de descomposición múltiple	Grupo OR + obligatoria	Grupo XOR + obligatoria	Restricción textual simple	Restricción textual compleja	Características clonables	Características con atributos	Características numéricas
FeatureIDE	●	●	●	●	●	○	○	○	○	○	●	●	○	◐	○
FlaMa	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Glencoe	○	●	●	●	●	○	●	○	○	○	●	●	○	○	○
SPLOT	○	●	●	●	●	○	○	●	○	○	●	◐	○	○	○
Clafer	●	●	●	●	●	●	●	○	○	○	●	●	●	◐	●

●: soportado. ◐: parcialmente soportado. ○: no soportado.

2.3. Transformaciones de modelos

Las transformaciones de modelos son técnicas utilizadas en ingeniería de software para convertir o adaptar modelos de una representación a otra, lo que permite trabajar con diferentes perspectivas o enfoques en el desarrollo de software. Estas transformaciones son especialmente relevantes en el contexto de la ingeniería dirigida por modelos (MDA - Model-Driven Architecture) y otros enfoques de ingeniería de modelos. Básicamente, una transformación de modelo es un proceso de transformación (relaciones y mapeo) de elementos de un modelo a elementos correspondientes de otro modelo. Los modelos están definidos por sus meta-modelos, los cuales son los encargados de brindar la sintaxis para la construcción de los mismos.

Las principales transformaciones de modelos son las siguientes:

- **T2M (*Text-to-Model*)**: Esta transformación se refiere a la generación automática de modelos a partir de un texto o una descripción textual. Por ejemplo, se podría utilizar para transformar una especificación escrita en lenguaje natural a un modelo formal expresado en un lenguaje de modelado como UML (Unified Modeling Language) o BPMN (Business Process Model and Notation). En nuestro caso, usamos este tipo de transformación para leer o parsear los modelos de características descritos en ficheros (por ejemplo en UVL) al meta-modelo usado por la herramienta software que gestiona los modelos de características (por ejemplo Flama que ya dispone de varios parsers).

- **M2T (*Model-to-Text*)**: La transformación M2T implica la generación automática de texto o código a partir de un modelo. Es decir, toma como entrada un modelo y produce como resultado un documento de texto, código fuente u otro artefacto textual. Esta técnica es comúnmente utilizada en la generación automática de código a partir de modelos, lo que permite ahorrar tiempo y reducir errores en el desarrollo de software. En nuestro caso, usamos las transformaciones M2T para serializar los modelos de característica a fichero (por ejemplo en formato UVL).

- **M2M (*Model-to-Model*)**: La transformación M2M se refiere a la conversión o adaptación de un modelo a otro. Es decir, toma como entrada un modelo en un lenguaje o nivel de abstracción y produce como resultado un modelo equivalente o relacionado en otro lenguaje o nivel de abstracción. Las transformaciones M2M son útiles cuando se necesita trabajar con diferentes representaciones de un sistema o para mantener la consistencia

entre diferentes modelos utilizados en el proceso de desarrollo. Este el tipo de transformación principal en este TFG ya que todos los *refactorings* son transformaciones de este tipo.

En resumen, para poder trabajar con los modelos UVL, necesitamos unas operaciones de lectura y escritura de dichos modelos a los modelos de características en memoria. Las operaciones de lectura suelen verse como transformaciones T2M o "Readers/Parsers". Las operaciones de escritura o serialización suelen verse como transformaciones M2T o "Writters". Por último, las transformaciones entre modelos en memoria son transformaciones M2M. Flama ya dispone de los *readers* y *writters* apropiados para los diferentes lenguajes (UVL, XML,...).

En este TFG nos centramos en las transformaciones M2M para convertir un modelo de características en otro similar. Para el caso de los modelo de características, hay cuatro tipos de transformaciones M2M, que se denominan *edits* [Thüm et al.(2009)] (Figura 2.11):

- *Refactoring*. Un *refactoring* **no cambia** los productos del modelo original, por lo que después de la transformación el modelo mantiene su semántica.
- *Generalization*. Después de aplicar la transformación hay **más** productos en el modelo resultante de los que había en el modelo original.
- *Specialization*. Después de aplicar la transformación hay **menos** productos en el modelo resultante de los que había en el modelo original.
- *Arbitraty edit*. Después de aplicar la transformación hay productos diferentes de los que había en el modelo original, pero no tiene por qué haber ni más ni menos. Se añaden y eliminan productos, por lo que el resultado es arbitrario.

En la Figura 2.11 se muestra un esquema de los cuatro tipos de transformaciones existentes que se pueden realizar a los modelos de características. En este TFG se han desarrollado los *refactorings* porque son las únicas transformaciones en las que la semántica del modelo no varía porque se mantienen intactos sus productos.

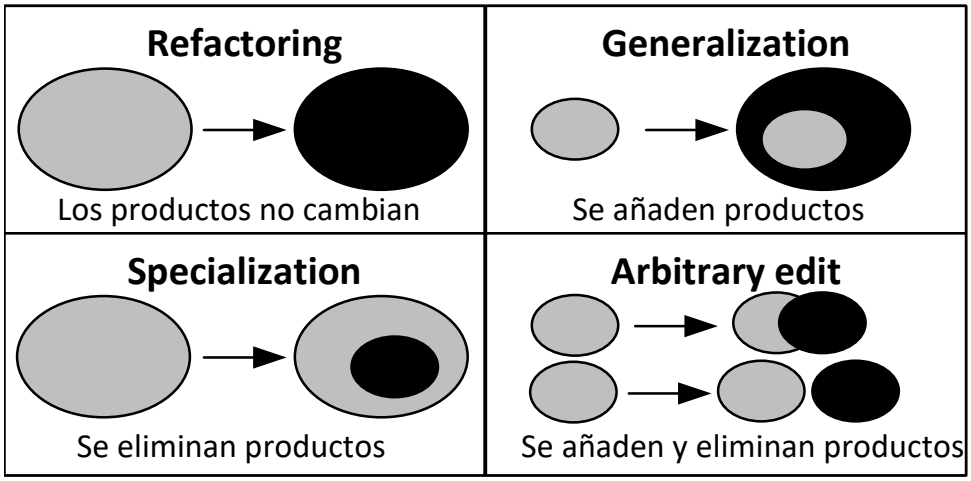

Figura 2.11: *Edits* o transformaciones M2M en los modelos de características.

3
Metodología

Este capítulo presenta el desarrollo y la metodología usada en el TFG para cumplir los objetivos del proyecto, así como las tecnologías que se han usado.

3.1. Desarrollo del trabajo fin de grado

Este trabajo fin de grado se enmarca como parte de un contrato de investigación de 7 meses (desde julio de 2022 a enero de 2023) en el que he pertenecido al equipo de trabajo del grupo CAOSD[1]. Durante el desarrollo del TFG, se ha colaborado con otros integrantes del grupo CAOSD participando en diferentes tareas de desarrollo relacionadas con las tecnologías de líneas de producto software y variabilidad.

[1] CAOSD: https://caosd.lcc.uma.es/

3.2. Metodología

Para el desarrollo de este proyecto se ha seguido una metodología incremental, que permite abordar el proyecto en etapas iterativas y manejables, facilitando la detección temprana de errores y adaptaciones. Además, también se ha hecho uso de la metodología basada en componentes, dada la definición de la interfaz de los *refactorings* que reduce la complejidad del desarrollo y permite su reutilización en distintas aplicaciones. Ambas metodologías combinadas han sido clave para alcanzar los objetivos propuestos en este TFG. Para la descripción de los requisitos se ha seguido el *desarrollo guiado por comportamiento* (BDD del inglés *Behaviour Driven Development*).

En la Figura 3.1 se muestra la metodología iterativa e incremental guiada por los *refactorings* que se han desarrollado para la biblioteca.

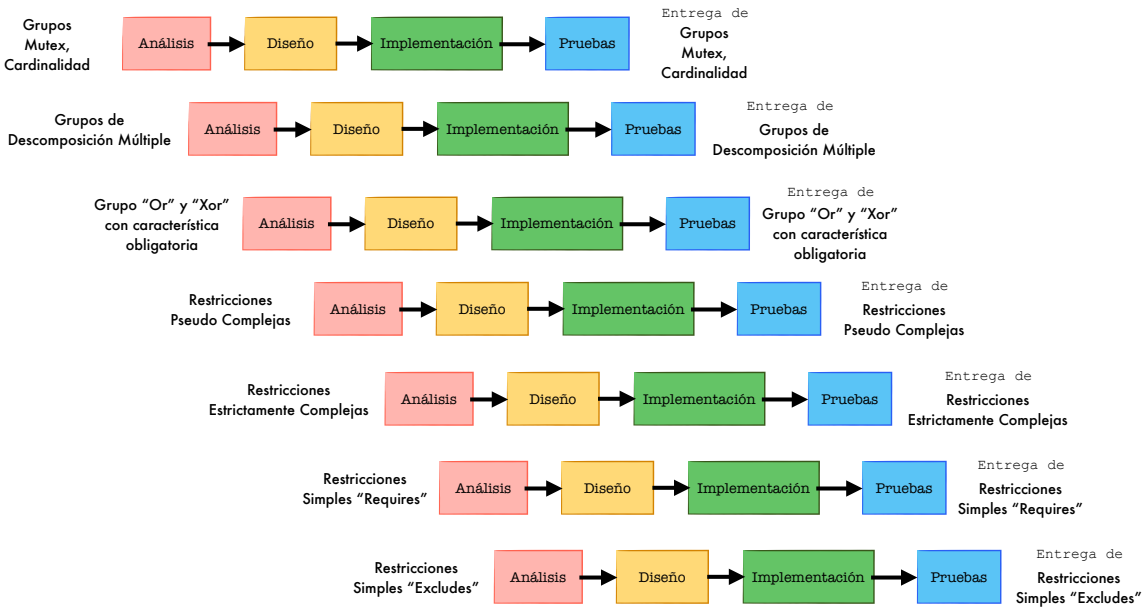

Figura 3.1: Modelo incremental e iterativo aplicado a la biblioteca de *refactorings*.

3.2.1. Desarrollo incremental e iterativo

El modelo de desarrollo incremental es un enfoque de desarrollo de software en el que se divide un proyecto en una serie de incremento, cada uno proporcionando una parte de la funcionalidad total requerida. Los incrementos en nuestro caso son un subconjunto de los *refactorings*. Los requisitos de la biblioteca y de los *refactorings* (descritos en el Capítulo 4 y 5

respectivamente) se priorizan y se entregan en orden de prioridad dentro de cada incremento. Además, en la planificación se definen el tamaño, el número y las etapas de construcción de los incrementos. Este modelo ofrece varias ventajas, tales como:

1. Los usuarios pueden aprovechar el sistema antes de que esté completamente terminado, ya que se obtienen funcionalidades progresivamente en cada incremento.
2. La tasa general de fallos del proyecto es reducida, aunque no se descarta la posibilidad de que puedan surgir problemas en algunos incrementos.
3. Los incrementos iniciales pueden ser utilizados como prototipos para obtener experiencia y retroalimentación relacionada con los requisitos de los incrementos posteriores del sistema.

3.2.2. Desarrollo basado en componentes

La metodología basada en componentes es un enfoque de desarrollo de software que se enfoca en la reutilización de componentes independientes y probados para construir sistemas más grandes. Los componentes son unidades de software que encapsulan funcionalidades específicas y se pueden combinar para formar aplicaciones completas. Esto ayuda a reducir costos, tiempo de desarrollo y mejora la eficiencia en la creación de software. El objetivo principal de esta metodología es mejorar la eficiencia y la calidad del desarrollo de software mediante la reutilización de componentes probados y bien diseñados. Al hacerlo, se reduce el tiempo de desarrollo, ya que los componentes ya han sido desarrollados y probados previamente, lo que disminuye la necesidad de construir cada parte del sistema desde cero.

En este TFG, además de la metodología basada en iteraciones, se ha seguido también la metodología basada en componentes, ya que cada *refactoring* tiene muy bien definida su interfaz, de forma que puede reutilizarse en diferentes contextos (aplicaciones, herramientas, webs,...).

3.2.3. Desarrollo guiado por comportamiento

El Desarrollo Guiado por Comportamiento (BDD, por sus siglas en inglés) es una práctica ágil que se enfoca en la colaboración entre desarrolladores, analistas de negocios y usuarios finales para definir los requisitos de un proyecto mediante historias de usuario y escenarios de validación. El objetivo principal es garantizar que el software se construya de manera centrada en el comportamiento esperado y en la creación de valor para el cliente.

En los Capítulos 4 y 5 el comportamiento de la biblioteca y los *refactorings* se describen mediante historias de usuario. Las historias de usuario son declaraciones simples y centradas en el usuario que describen la funcionalidad que debe tener el sistema desde el punto de vista del usuario final. Por lo general, se expresan en el formato ”Como [tipo de usuario], quiero [realizar una acción] para [lograr un objetivo]”. Estas historias proporcionan una comprensión clara de lo que se espera del software desde la perspectiva del usuario.

Para cada historia de usuario se definen varios escenarios de validación. Los escenarios de validación son ejemplos concretos que describen cómo debería comportarse el sistema en situaciones específicas. Estos escenarios se basan en las historias de usuario y se utilizan para validar que el software cumpla con los requisitos establecidos. Los escenarios de validación están escritos en un lenguaje claro y no técnico para que todas las partes interesadas puedan entenderlos.

Un aspecto importante del BDD es la automatización de pruebas basadas en los escenarios de validación. Estas pruebas automatizadas verifican que el software cumpla con los criterios definidos en los escenarios. Esto permite una validación continua del desarrollo y proporciona retroalimentación rápida en caso de que algo no funcione según lo esperado.

3.3. Tecnologías

Como se ha expuesto anteriormente, la biblioteca de *refactorings* está desarrollada 100 % en el lenguaje Python. Además, para trabajar con los modelos de características se ha usado la biblioteca Flama. Aparte, y por la naturaleza propia del proyecto, se han usado varias herramientas más de modelado de la variabilidad como FeatureIDE, SPLOT, y Glencoe. A continuación, se describen las herramientas más importantes y el uso que se le ha dado a cada una:

- **Flama** (https://flamapy.github.io/). Flama es una biblioteca para analizar y trabajar con modelos de variabilidad en diferentes formalizaciones (modelos de características, SAT, CNF, BDD o diagramas de decisión binarios,...). Se ha trabajado con los plugins de modelos de características para desarrollar los *refactorings* así como con los plugins de SAT y BDDs para el análisis de los modelos (por ejemplo para extraer los productos representados por el modelo).

- **FeatureIDE** (https://featureide.github.io/). FeatureIDE es un pluging de Eclipse que permite modelar y trabajar con modelos de característica de forma gráfica (mediante el diagrama de características) y de forma textual (con UVL). Se ha usado FeatureIDE para definir la mayoría de los modelos de características que se han usado en el TFG.

- **SPLOT** (http://www.splot-research.org/). SPLOT es un repositorio online de modelos de características que también permite definir nuevos modelos. Se ha usado SPLOT para definir algunos modelos de características muy básicos y comprobar la interoperabilidad con las otras herramientas.

- **Glencoe** (https://glencoe.hochschule-trier.de/). Glencoe es un editor online de diagramas de características. Se ha usado Glencoe para definir algunos modelos de características con constructores que no soportan otros modelos como los grupos de cardinalidad.

Aparte de estas herramientas de líneas de producto software y modelos de variabilidad propias de este TFG, también se han usado otras tecnologías para el desarrollo del TFG como *Git* (para la gestión del código fuente), *GitHub* (para el alojamiento del repositorio de código), y *Pytest* (una librería de test unitarios en Python para la implementación de las pruebas).

4

Biblioteca de refactorings para modelos de características

En este capítulo se describe la biblioteca de *refactorings* propuesta. Primero se especifican los requisitos generales de la biblioteca que deben cumplir todos los *refactorings*, y se muestran los casos de uso. A continuación, se describe la arquitectura software de la biblioteca.

4.1. Requisitos generales de la biblioteca de refactorings

La biblioteca de *refactorings* permite al usuario realizar tres acciones principales, que están descritas en el diagrama de casos de uso de la Figura 4.1:

1. La biblioteca permite aplicar un *refactoring* sobre una instancia específica (característica o restricción) de un modelo de características. En tal caso, la biblioteca recibe el modelo y la instancia que se quiera transformar, que puede ser una característica o una restricción textual. La biblioteca identifica automáticamente qué *refactoring* hay que aplicar en ese caso y lo aplica, devolviendo un modelo de características con la misma semántica y donde esa instancia ha sido transformada (refactorizada).

2. La biblioteca también permite aplicar un refactorizar de un constructor de variabilidad específico sobre todas las instancias de un modelo de características. En este caso la biblioteca recibe un modelo y el nombre de un *refactoring*, y se aplica dicho *refactoring* a todas las instancias del modelo a las que sea aplicable.

3. Por último, la biblioteca permite simplificar un modelo de características completamente, refactorizando los constructores de variabilidad complejos a otros más simples. En este caso, la biblioteca recibe únicamente un modelo y aplica todos los *refactorings* disponibles en la biblioteca que puedan aplicarse a dicho modelo en un orden específico. Los *refactorings* y el orden de aplicación se detallarán en el Capítulo 5.

Para describir los requisitos se va a hacer uso de las historias de usuario y criterios de aceptación para la biblioteca de *refactorings*.

En las Figuras 4.2, 4.3 y 4.4 se pueden observar tres plantillas generalizadas para historias de usuario que pueden ser aplicadas a todos los *refactorings* que se han desarrollado en este TFG, con sus correspondientes criterios de aceptación. En las tres, la nota superior corresponde con la historia de usuario, y en las dos tarjetas inferiores se muestran sus correspondientes criterios de aceptación o escenarios. El ejemplo de la Figura 4.2 es para aplicar un *refactoring* que modifique un modelo de características que contiene un constructor de variabilidad que no es soportado por la mayoría de herramientas de modelos de características, sustituyéndolo por constructores simples y manteniendo la semántica del modelo, es decir, sin alterar sus productos. Por otra parte, en la Figura 4.3 el ejemplo muestra la necesidad de poder aplicar varios *refactorings* de manera consecutiva a un modelo de características, y también se mantiene

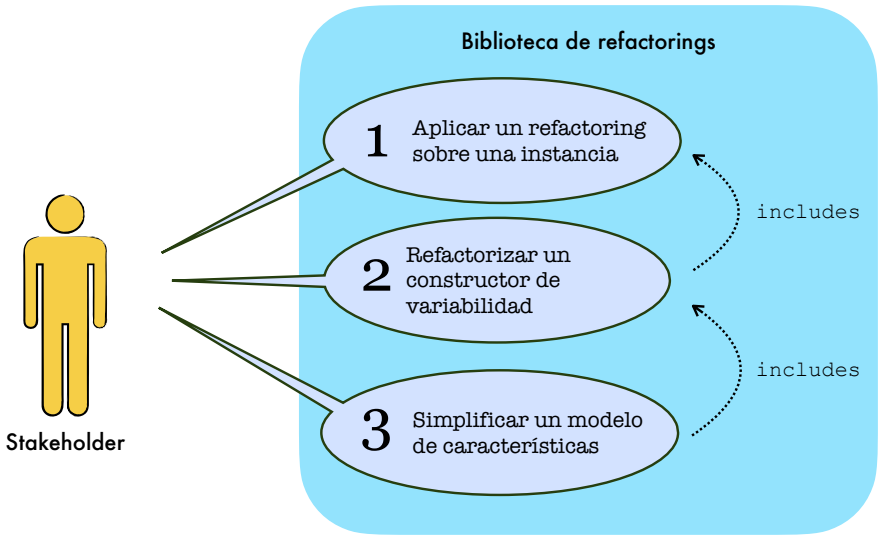

Figura 4.1: Diagrama de casos de uso de la biblioteca de *refactorings*.

la semántica del modelo en este caso. Finalmente en la Figura 4.4 se puede observar la historia de usuario y criterios de aceptación en caso de que querer simplificar el modelo completo aplicando todos los *refactorings* existentes y que el modelo no cambie su semántica.

Las tres figuras representan tanto historias de usuario como criterios de aceptación generales, lo que quiere decir que se pueden aplicar para cada uno de los *refactorings* que se van a describir en el Capítulo 5. Además, en cada uno de ellos se añaden nuevas historias de usuario con sus correspondientes criterios de aceptación, que serán propios de cada *refactoring* específico.

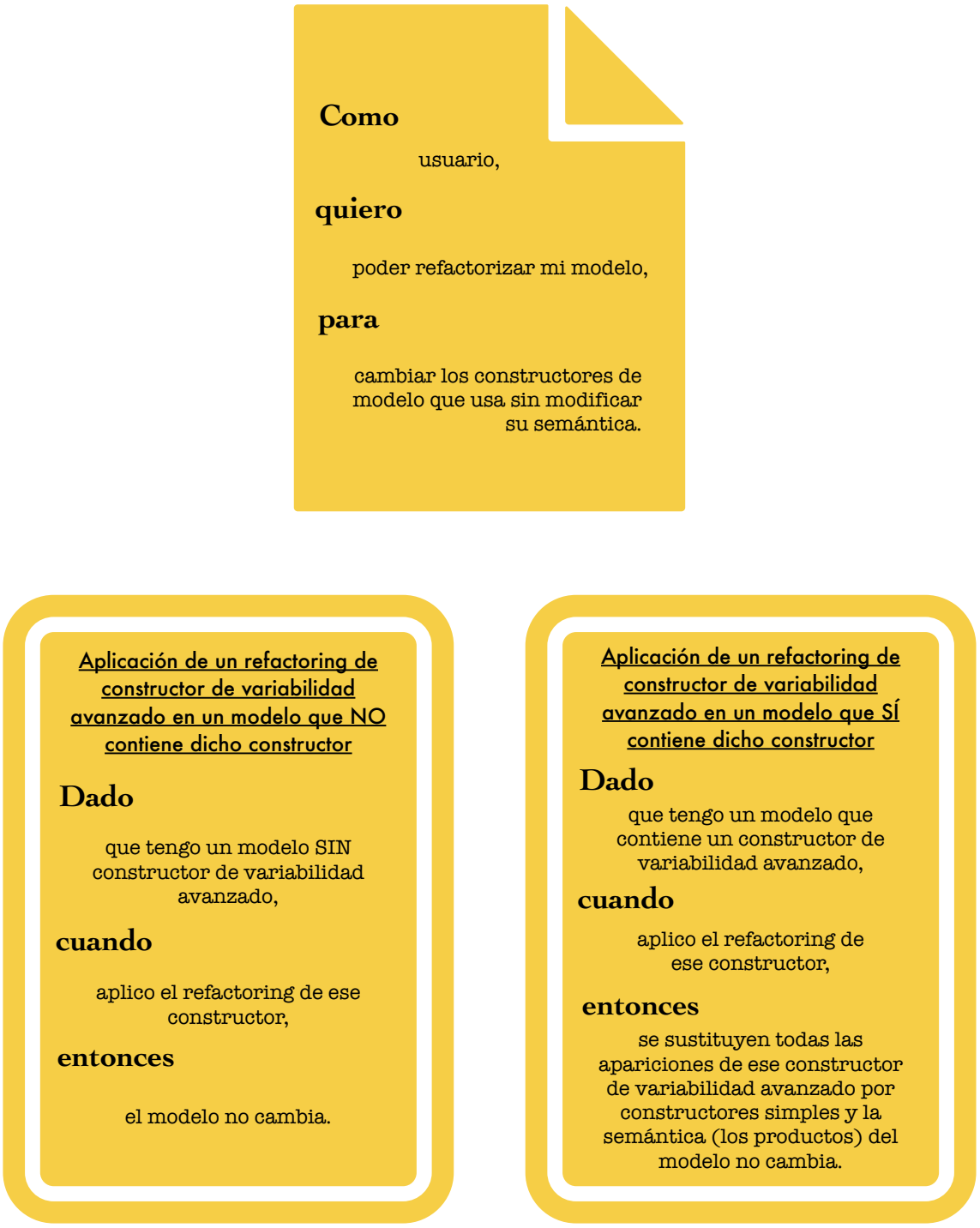

Figura 4.2: Ejemplo de historia de usuario y sus correspondientes criterios de aceptación en el caso de aplicación de un *refactoring* a un modelo de características.

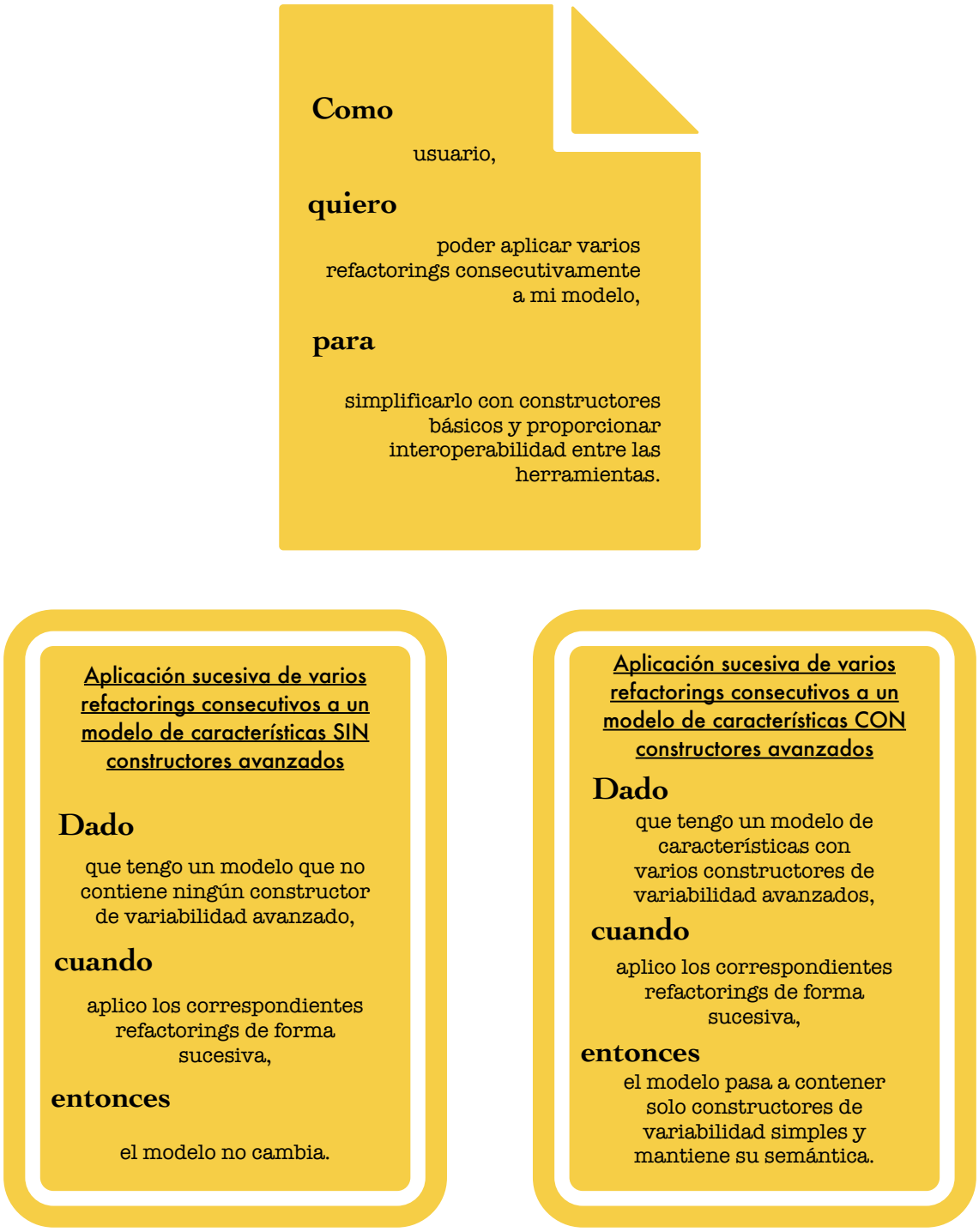

Figura 4.3: Ejemplo de historia de usuario y sus correspondientes criterios de aceptación en el caso de una aplicación sucesiva de *refactorings* a un modelo de características.

Figura 4.4: Ejemplo de historia de usuario y sus correspondientes criterios de aceptación en el caso de una simplificación completa de un modelo de características.

4.2. Arquitectura software de la biblioteca de refactorings

En la Figura 4.5 se puede observar la arquitectura que se ha construido para la biblioteca de *refactorings*. En ella, vemos que la biblioteca hace uso de la biblioteca Flama para gestionar y analizar los modelos de características. La estructura de la biblioteca refleja la misma estructura que Flama. Esto permitirá en un futuro integrar la propia biblioteca dentro de Flama como un plugin adicional. Esta estructura se clasifica en "Modelos", "Operaciones" y "Transformaciones". La interfaz `FMRefactoring` de "Modelos" contiene las definiciones que deben seguir todos los *refactorings*. "Operaciones" consta de un `RefactoringEngine` que es donde se llevan a cabo todas las operaciones necesarias para aplicar los *refactorings*. Por último, en "Transformaciones" están incluidos todos los *refactorings* que se han implementado en este TFG. Los *refactorings* hacen uso de un módulo `Utils` que se ha definido para completar la arquitectura, y de una serie de `Tests` que sirven para comprobar la *correctitud* o *validez* de los *refactorings* que se han implementado.

Cuando se pasa un modelo de características (**input**) al que es necesario aplicar un *refactoring*, este se pasa a través de **RefactoringEngine**, que aplica los *refactorings* necesarios o indicados según el caso de uso solicitado, siguiendo la interfaz `FMRefactoring`. Cuando se ha especificado o identificado la operación a realizar, se llama a la correspondiente transformación, que son los *refactorings* propiamente dichos. Finalmente, como se puede observar en la Figura 4.5, se devuelve (**output**) un modelo modificado, que estará adaptado a las necesidades del usuario.

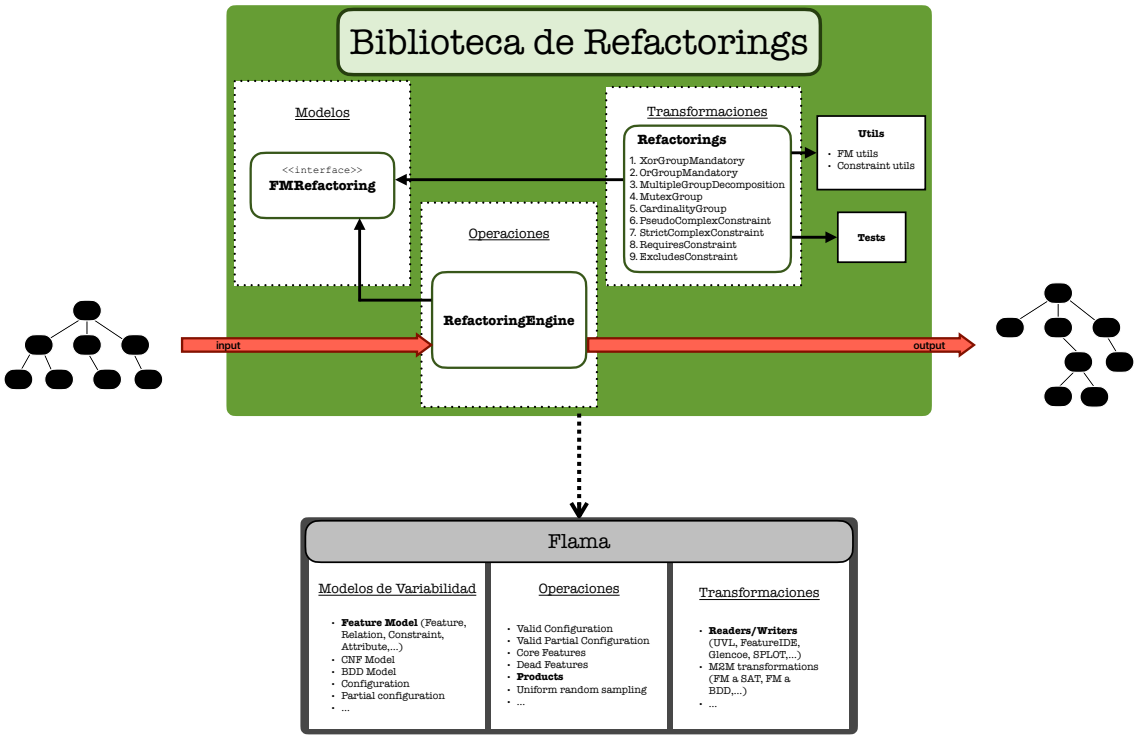

Figura 4.5: Arquitectura de la biblioteca de *refactorings*.

5

Refactorings de modelos de características

En este capítulo se describen los requisitos, pruebas y diseño para cada uno de los *refactorings* desarrollados en la biblioteca. Cada subsección describe un *refactoring*, incluyendo (1) su descripción (qué hace el *refactoring*); (2) sus requisitos en forma de historia de usuario; junto con sus criterios de aceptación, en forma de escenarios asociados a la historia de usuario, que sirven de pruebas automáticas para la corrección y completitud del *refactoring*; (3) su diseño en forma de esquema genérico de la transformación de modelo correspondiente; y (4) un ejemplo aplicado sobre el modelo de características de nuestro de caso de estudio presentado en el Capítulo 2.

Los *refactorings* necesarios para cumplir con las especificaciones del usuario son los siguientes:

1. Grupo “xor” con una característica obligatoria
2. Grupo “or” con una característica obligatoria
3. Grupos de descomposición múltiple
4. Grupos mutex
5. Grupos de cardinalidad

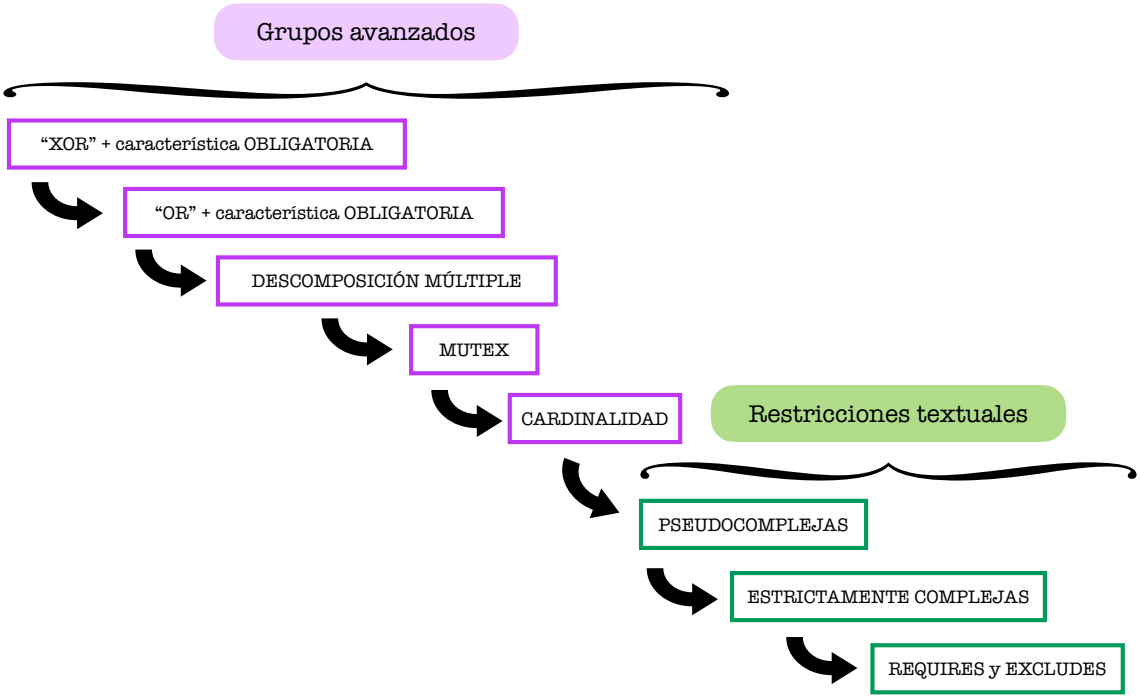

Figura 5.1: Orden de prioridad a la hora de aplicar varios *refactorings* de manera sucesiva a un modelo de características.

6. Restricciones textuales pseudo-complejas
7. Restricciones textuales estrictamente complejas
8. Restricciones textuales simples “requires”
9. Restricciones textuales simples “excludes”

Al ser posible aplicar varios *refactorings* de manera sucesiva a un modelo de características, es necesario establecer un orden de prioridades para asegurar que se mantenga la semántica del modelo. El orden de prioridades se establece en la Figura 5.1. Este orden de prioridades hay que tenerlo en cuenta porque existen ciertos *refactorings* que dan lugar a un nuevo grupo avanzado de variabilidad, o bien se pueden añadir restricciones textuales complejas o simples que anteriormente no existían en el modelo, y que por tanto hacen que sea necesario volver a aplicar el correspondiente *refactoring* para seguir simplificando el modelo.

5.1. Grupo “xor” con una característica obligatoria

Descripción. Para realizar el *refactoring* de un grupo “xor” con una característica obligatoria dentro hay que sustituir el grupo “xor” por un grupo de descomposición múltiple, en el que la característica que era obligatoria en el grupo original se mantiene obligatoria, y el resto de características que pertenecían al grupo “xor” pasan a ser parte de un grupo de cardinalidad $[0..0]$. Las características que no eran la obligatoria ya estaban muertas y se mantienen muertas.

Dicho de otra manera consiste en separar el grupo “xor” que contiene una característica obligatoria en un grupo de descomposición múltiple, que a su vez tiene como hijas la característica obligatoria por una parte, y por otra parte, el resto de las características dentro a un nuevo grupo, que ahora es un grupo con cardinalidad $[0..0]$.

Es necesario tener en cuenta que el resultado de este *refactoring* es un grupo de descomposición múltiple, por lo que después de aplicarlo habría que realizar el correspondiente *refactoring* de grupos de descomposición múltiple que se explicará en la Subsección 5.3.

Requisitos.

- Historia de usuario: “**Como** usuario, **quiero** aplicar el *refactoring* de grupos “xor” con una característica obligatoria a mi modelo, **para** transformarlos en grupos de descomposición múltiple con la misma característica obligatoria y el resto en un grupo de cardinalidad $[0..0]$, manteniendo la semántica del modelo.”

- Criterios de aceptación:

  1. ***Transformación del grupo “xor” con una característica obligatoria***

     - **DADO** un modelo de características,

       Y una característica f de dicho modelo representando un grupo “xor” con subcaracterísticas $f_1, f_2, ...f_n$,

       Y siendo f_m ($f_m \in [f_1, f_2, ..., f_n]$) una característica obligatoria,

       CUANDO aplico el *refactoring* de grupo “xor“ con característica obligatoria sobre la característica f,

       ENTONCES el modelo contiene una instancia menos de grupos “xor”,

       Y el modelo contiene una instancia más de grupo de descomposición múltiple,

       Y las características hijas de f pertenecen todas a un grupo de cardinalidad

$[0..0]$ excepto f_m,

Y las características hijas de f que no son f_m están muertas,

Y la característica f_m es obligatoria.

Diseño. La Figura 5.2 muestra el esquema de la transformación de modelo correspondiente al *refactoring* del grupo "xor" con una característica obligatoria que sigue la especificación anterior.

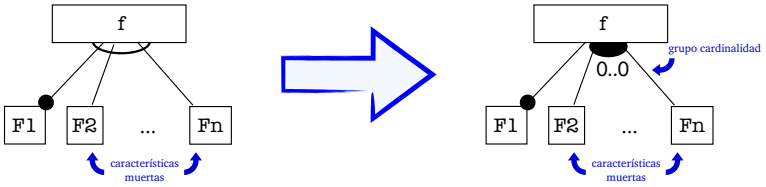

Figura 5.2: *Refactoring* del grupo "xor" con una característica obligatoria general.

Ejemplo. La Figura 5.3 muestra el *refactoring* de grupos "xor" con una característica obligatoria aplicado al caso de estudio de la SPL de pizzas.

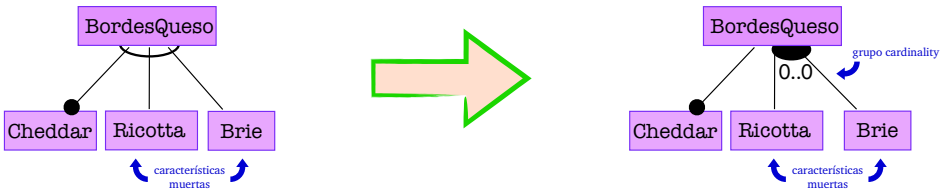

Figura 5.3: *Refactoring* del grupo "xor" con una característica obligatoria del ejemplo en la Figura 2.3.

Como se puede observar en la Figura 5.3, la característica `BordesQueso`, que era el grupo "xor" y tenía dentro de ese grupo una característica obligatoria, pasa a ser una característica con grupo de descomposición múltiple, en el que hay una característica obligatoria por una parte y un grupo de cardinalidad $[0..0]$ por otra parte. Las hijas de `BordesQueso` son `Cheddar`, `Ricotta` y `Brie`. De las tres hijas de `BordesQueso`, la característica `Cheddar` se mantiene obligatoria como ocurría con el grupo original, y las otras dos características, `Ricotta` y `Brie`, pasan a formar parte de un grupo de cardinalidad $[0..0]$ y se mantienen muertas como lo estaban en el grupo original.

5.2. Grupo “or” con una característica obligatoria

Descripción. Para realizar el *refactoring* de un grupo “or” con una característica obligatoria dentro hay que sustituir el grupo “or” por un grupo “and”, en el que la característica que era obligatoria en el grupo original se mantiene obligatoria, y el resto de características que pertenecían al grupo “or” pasan a ser características simples opcionales.

Requisitos.

- Historia de usuario: “**Como** usuario, **quiero** aplicar el *refactoring* de grupos “or” con una característica obligatoria a mi modelo, **para** transformarlos en grupos “and” con la misma característica obligatoria y el resto opcionales, manteniendo la semántica del modelo.”
- Criterios de aceptación:

  1. ***Transformación del grupo “or” con una característica obligatoria***

     - **DADO** un modelo de características,
       Y una característica f de dicho modelo representando un grupo “or” con subcaracterísticas $f_1, f_2, ...f_n$,
       Y siendo f_m ($f_m \in [f_1, f_2, ..., f_n]$) una característica obligatoria,
       CUANDO aplico el *refactoring* de grupo “or“ con característica obligatoria sobre la característica f,
       ENTONCES el modelo contiene una instancia menos de grupos “or”,
       Y el modelo contiene una instancia más de grupo “and”,
       Y las características hijas de f son todas opcionales excepto f_m,
       Y la característica f_m es obligatoria.

Diseño. La Figura 5.4 muestra el esquema de la transformación de modelo correspondiente al *refactoring* del grupo “or” con una característica obligatoria que sigue la especificación anterior.

Ejemplo. La Figura 5.5 muestra el *refactoring* de grupos “or” con una característica obligatoria aplicado al caso de estudio de la SPL de pizzas.

Como se puede observar en la Figura 5.5, la característica `Masa` que era el grupo “or” y tenía dentro de ese grupo una característica obligatoria pasa a ser una característica con grupo

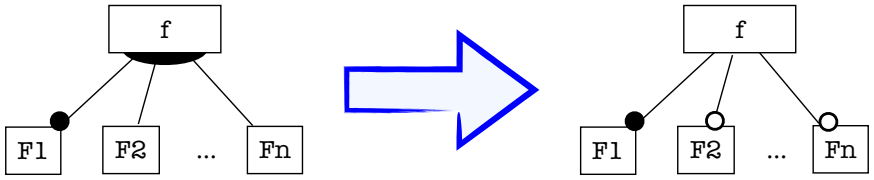

Figura 5.4: *Refactoring* del grupo "or" con una característica obligatoria general.

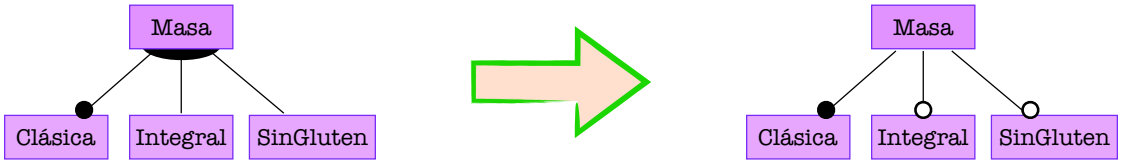

Figura 5.5: *Refactoring* del grupo "or" con una característica obligatoria del ejemplo en la Figura 2.3.

"and". Las características hijas de `Masa` pertenecían al grupo y de ellas, la característica `Clásica` era obligatoria y las otras dos hijas, `Integral` y `SinGluten` pertenecían al grupo sin más. Después de hacer el *refactoring* la característica `Masa` tiene un grupo "and", y de sus tres hijas del grupo original, `Clásica` se mantiene obligatoria, mientras que `Integral` y `SinGluten` pasan a ser características opcionales.

5.3. Multiple Group Decomposition

Descripción. El *refactoring* para grupos de descomposición múltiple también ha sido formalizado en Knüppel et al. [Knüppel et al.(2017)] y se describe así: Para eliminar los grupos múltiples $g_1, g_2, .., g_n$ bajo una característica f, se colocan las características del grupo de descomposición en un grupo "and" (siendo la cardinalidad del grupo and $[n..n]$, y siendo n el número de grupos que había en el grupo de descomposición múltiple original), y se sustituye cada grupo g_i por una característica f_p_i auxiliar, obligatoria y abstracta, tal que f sea padre de f_p_i, y f_p_i sea padre de su correspondiente grupo en el grupo de descomposición múltiple original, es decir, g_i para $i = 1, ..., n$. Las características obligatorias y opcionales que perteneciesen a f se mantienen intactas.

Dicho de otra manera consiste en descomponer en grupos independientes cada subgrupo que antes pertenecía a una misma característica padre. Para ello, se añade una característica obligatoria abstracta para cada subgrupo.

Requisitos.

- Historia de usuario: "**Como** usuario, **quiero** aplicar el *refactoring* de grupos de descomposición múltiple a mi modelo, **para** separar cada grupo de manera independiente, manteniendo la semántica del modelo."

- Criterios de aceptación:

  1. ***Transformación del grupo descomposición múltiple***

     - **DADO** un modelo de características,
       Y una característica f de dicho modelo representando un grupo de descomposición múltiple con n sub-grupos $g_1, g_2, ..., g_n$,
       CUANDO aplico el *refactoring* de grupo descomposición múltiple sobre la característica f,
       ENTONCES el modelo contiene una instancia menos de grupos de descomposición múltiple,
       Y el modelo contiene n características nuevas f_p_i (siendo $i = 1, 2, ..., n$),
       Y todas las características f_p_i son abstractas,
       Y todas las características f_p_i son obligatorias,

Y cada característica f_p_i tiene un grupo correspondiente con el que tenía el grupo de descomposición múltiple original,

Diseño. La Figura 5.6 muestra el esquema de la transformación de modelo correspondiente al *refactoring* del grupo de descomposición múltiple que sigue la especificación anterior.

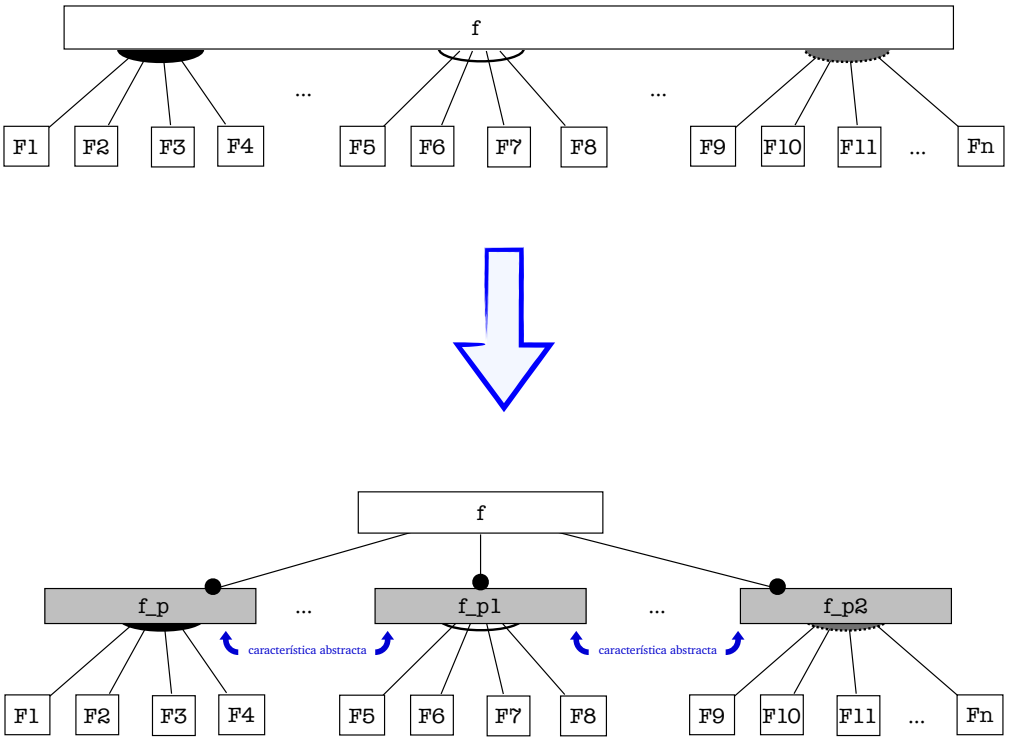

Figura 5.6: *Refactoring* general de descomposición de grupos múltiples.

Ejemplo. La Figura 5.7 muestra el *refactoring* de grupos de descomposición múltiple aplicado al caso de estudio de la SPL de pizzas.

Como se puede observar en la Figura 5.7, la característica `Particularidades`, que era el grupo descomposición múltiple, pasa a ser una característica normal. Se sustituye el grupo descomposición múltiple que anteriormente tenía dos grupos diferentes, que eran un grupo "xor" y un grupo "or" por dos características abstractas y obligatorias, `Particularidades_p` y `Particularidades_p1`, que son hijas de `Particularidades`.Cada característica auxiliar nueva es un grupo correspondiente con los respectivos grupos que había en el grupo de descomposición múltiple original. Ahora `Particularidades_p` es una característica obligatoria abstracta y tiene

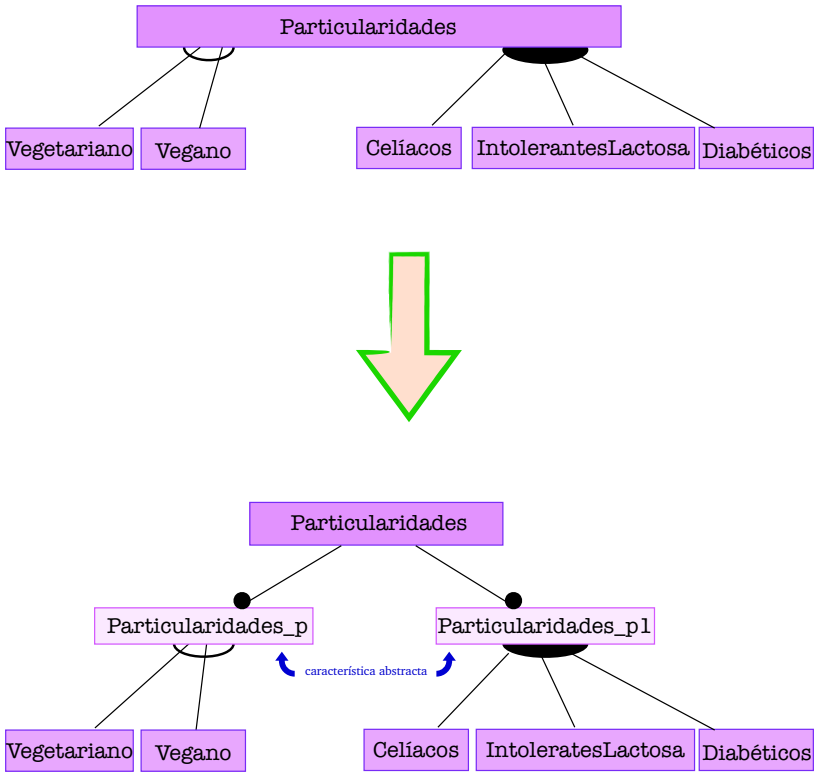

Figura 5.7: *Refactoring* de descomposición de grupos múltiples del ejemplo en la Figura 2.4.

un grupo "xor", cuyas hijas son `Vegetariano` y `Vegano`; y la característica `Particularidades_p1` es también una característica obligatoria y abstracta que tiene un grupo "or" cuyas hijas son `Celíacos`, `IntolerantesLactosa` y `Diabéticos`.

5.4. Grupos mutex

Descripción. El *refactoring* para los grupos mutex ha sido ya formalizado por Knüppel et al. [Knüppel et al.(2017)] de la siguiente forma: Dada una característica f que representa un grupo mutex cuyas sub-características son $f_1, f_2, ..., f_n$, transformamos el tipo de descomposición de f a una característica simple con una nueva sub-característica opcional abstracta f'. La característica f' es un grupo alternativo (xor) con las sub-características $f_1, f_2, ..., f_n$. Dicho de otra forma, consiste en convertir el grupo mutex en un grupo alternativo (con cardinalidad $[1..1]$) cuyo padre sea una nueva característica opcional abstracta, que a su vez tiene como padre el que antes tenía el grupo mutex. De este modo, se puede seleccionar o no esa característica opcional abstracta, lo que determinará si se escoge o no el grupo en su totalidad.

Requisitos.

- Historia de usuario: "**Como** usuario, **quiero** aplicar el *refactoring* de grupos mutex a mi modelo, **para** transformarlos en grupos "xor" opcionales manteniendo la semántica del modelo."

- Criterios de aceptación:

  1. ***Transformación del grupo mutex***

     - **DADO** un modelo de características,
       Y una característica f de dicho modelo representando un grupo mutex con sub-características $f_1, f_2, ...f_n$,
       CUANDO aplico el *refactoring* de grupo mutex sobre la característica f,
       ENTONCES el modelo contiene una instancia menos de grupos mutex,
       Y el modelo contiene una instancia más de grupos "xor",
       Y el modelo contiene una característica nueva f',
       Y la característica f' es un grupo "xor",
       Y la característica f' es abstracta,
       Y la característica f' es opcional,
       Y la característica f contiene como único hijo a f',
       Y la característica f' contiene como hijos a $f_1, f_2, ..., f_n$.

Diseño. La Figura 5.8 muestra el esquema de la transformación de modelo correspondiente al *refactoring* del grupo mutex que sigue la especificación anterior.

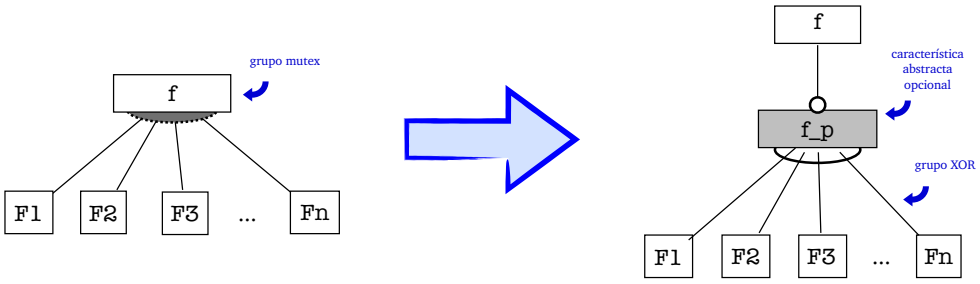

Figura 5.8: *Refactoring* del grupo mutex general.

Ejemplo. La Figura 5.9 muestra el *refactoring* de grupos mutex aplicado al caso de estudio de la SPL de pizzas.

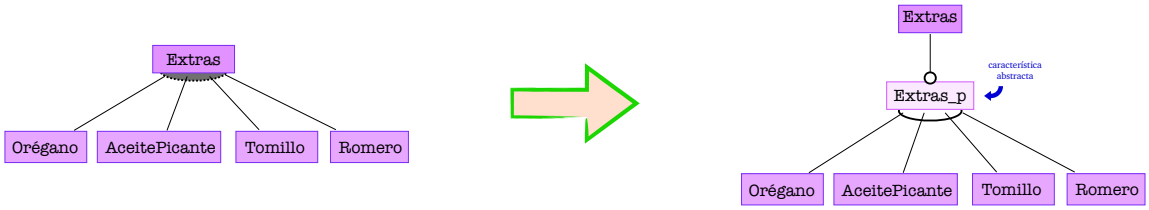

Figura 5.9: *Refactoring* del grupo mutex del ejemplo en la Figura 2.5.

Como se puede observar en la Figura 5.9, la característica `Extras` que era el grupo mutex pasa a ser una característica que no es padre de un grupo. Se crea una nueva característica opcional y abstracta `Extras_p` como hija única de `Extras`. `Extras_p` es un grupo "xor" (cardinalidad $[1..1]$) con las hijas originales de `Extras`: `Orégano`, `AceitePicante`, `Tomillo` y `Romero`.

5.5. Grupos de cardinalidad

Descripción. El *refactoring* para grupos de cardinalidad también ha sido formalizado en Knüppel et al. [Knüppel et al.(2017)] de la siguiente forma: Si una característica f es un grupo de cardinalidad con multiplicidad $[a..b]$, se cambia el tipo de descomposición de f a una característica con una relación simple y n características hijas opcionales, siendo n el número total de características hijas que había anteriormente en el grupo de cardinalidad, y se añade la siguiente restricción textual de lógica proposicional:

$$f \Rightarrow \bigvee_{M \in P_{a,b}} \left(\bigwedge_{f_i \in M} f_i \wedge \bigwedge_{f_j \in \{f' | f' \text{ is child of } f\} \setminus M} \neg f_j \right), \tag{5.1}$$

con $P_{a,b} = \{A \in 2^{\{f' | f' \text{ is child of } f\}} | a \leq |A| \leq b\}$, siendo el conjunto de todas las combinaciones de características hijas de f donde cada combinación tiene al menos a y como mucho b elementos (ver caso general en la Figura 5.10).

Dicho de otra manera consiste en cambiar los grupos de cardinalidad $[a..b]$ (siendo a el número de características mínimas, y b el número de características máximo a escoger, $a < b$), por un grupo "and" cuyas características hijas son todas opcionales, y añadir una restricción que haga que sea posible escoger las características con las mismas posibilidades que antes de aplicar el *refactoring*.

Es necesario tener en cuenta que el resultado de este *refactoring* añade una restricción compleja al modelo, por lo que después de aplicarlo habría que realizar el correspondiente *refactoring* de eliminación de restricciones textuales complejas que se explicará en las Subsecciones 5.6 y 5.7.

Requisitos.

- Historia de usuario: "**Como** usuario, **quiero** aplicar el *refactoring* de grupos de cardinalidad a mi modelo, **para** transformarlos en grupos "and" opcionales, añadiendo una restricción textual, manteniendo la semántica del modelo."

- Criterios de aceptación:

  1. ***Transformación del grupo cardinalidad***

     - **DADO** un modelo de características,

       Y una característica f de dicho modelo representando un grupo de cardinali-

dad con sub-características $f1, f2, ..., f_n$,

CUANDO aplico el *refactoring* de grupo de cardinalidad sobre la característica f,

ENTONCES el modelo contiene una instancia menos de grupos cardinalidad,

Y el modelo contiene n relaciones nuevas representando a las características hijas,

Y todas las características hijas que pertenecían al grupo ahora son opcionales,

Y se añade una restricción textual nueva,

Y la restricción textual se construye mediante la ecuación 5.1.

Diseño. La Figura 5.10 muestra el esquema de la transformación de modelo correspondiente al *refactoring* del grupo cardinalidad que sigue la especificación anterior.

Ejemplo. La Figura 5.11 muestra el *refactoring* de grupos cardinalidad aplicado al caso de estudio de la SPL de pizzas.

Como se puede observar en la Figura 5.11, la característica `Ingredientes`, que era el grupo cardinalidad, pasa a ser una característica normal. Se sustituye el grupo cardinalidad que anteriormente tenía cinco características por cinco relaciones simples opcionales que asocian la característica `Ingredientes` padre con sus cinco hijas originales: `Salami`, `Jamón`, `Mozzarella`, `Maíz` y `Champiñones`. Como anteriormente el grupo original tenía cardinalidad $[2..5]$, ahora se añade una restricción textual de lógica proposicional que nuevamente permite que solamente se escojan características de esa misma manera, restringiendo la libertad de escoger todas las características libremente que en un principio ofrecen las características opcionales.

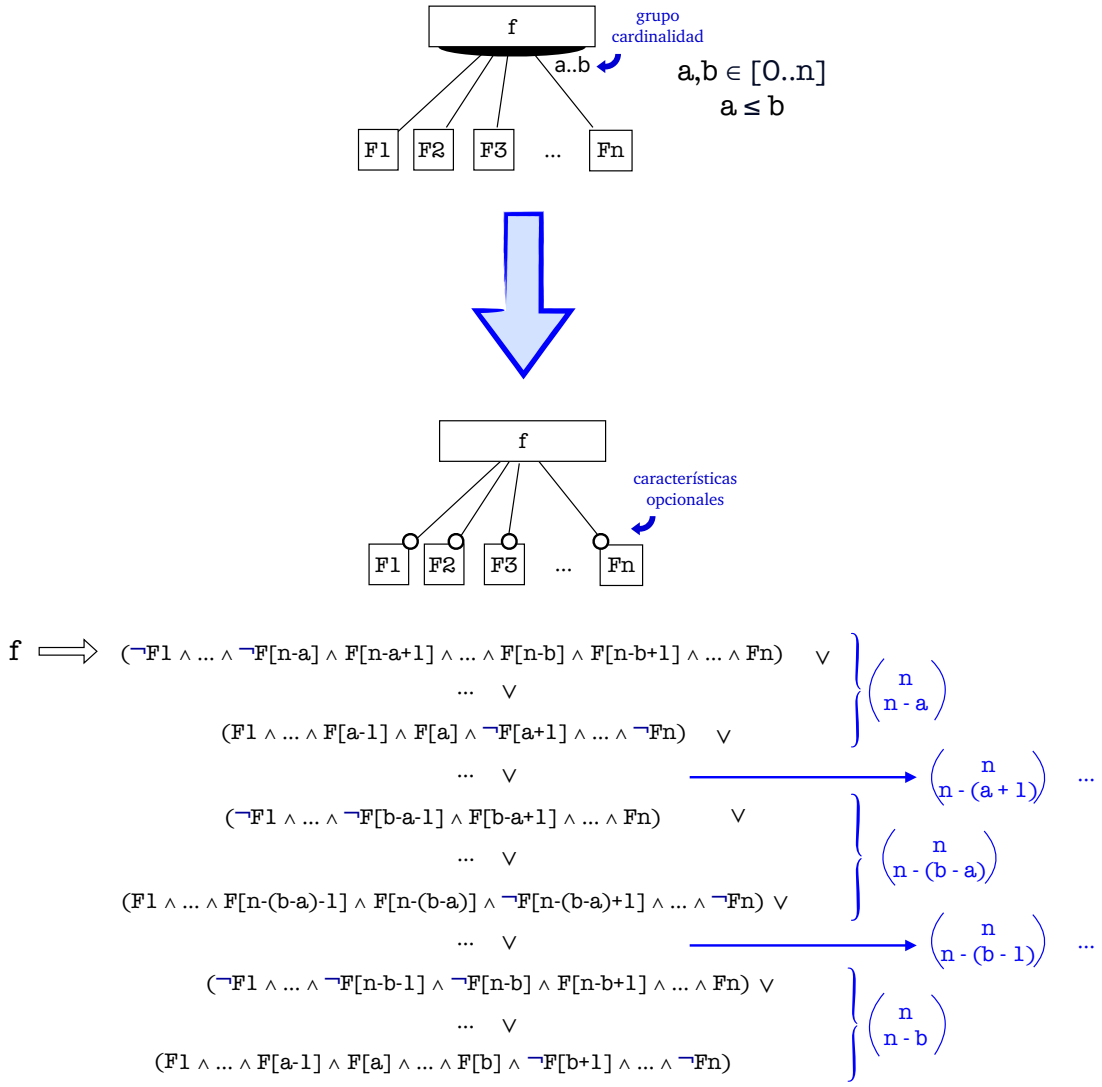

Figura 5.10: *Refactoring* del grupo cardinalidad general.

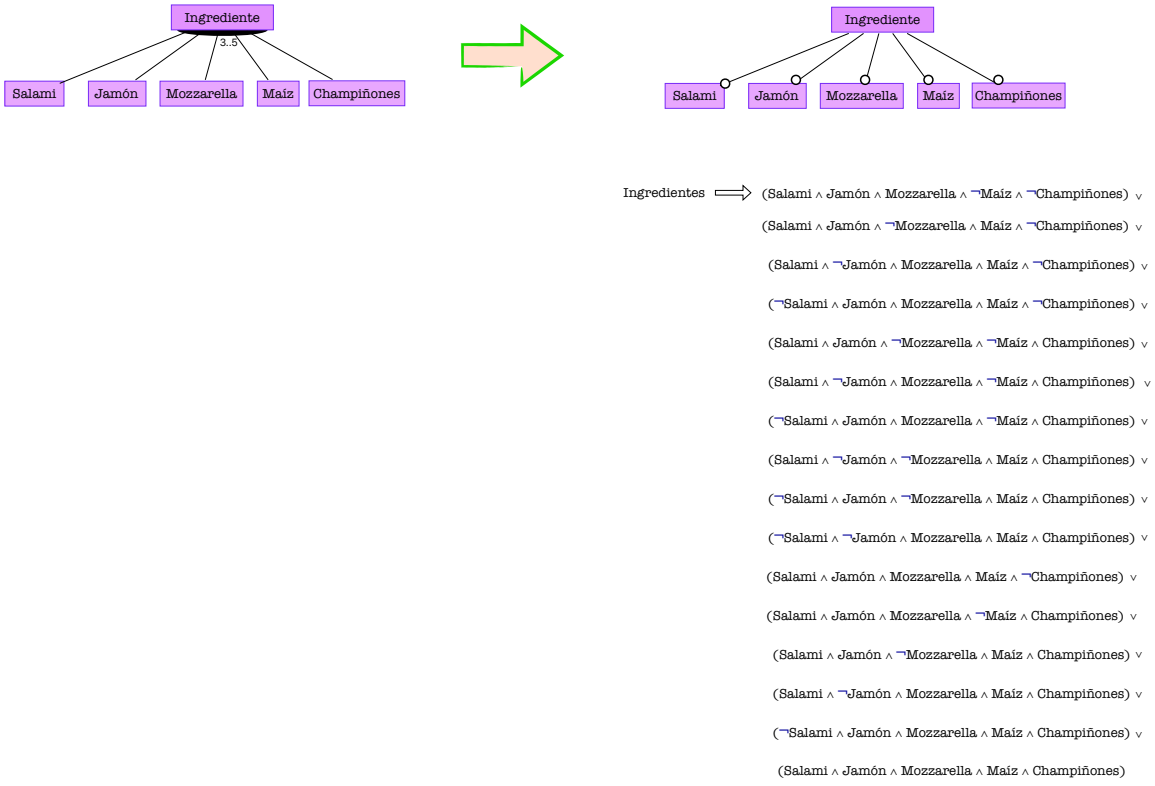

Figura 5.11: *Refactoring* del grupo cardinalidad del ejemplo en la Figura 2.6.

5.6. Pseudo Complex Constraint

Descripción. El *refactoring* para la eliminación de restricciones textuales pseudo-complejas ha sido ya formalizado por Knüppel et al. [Knüppel et al.(2017)] de la siguiente forma: Las restricciones complejas pueden ser convertidas a un conjunto de restricciones textuales simples.

Es necesario tener en cuenta que el resultado de este *refactoring* añade una serie de restricciones simples al modelo, por lo que después de aplicarlo habría que realizar el correspondiente *refactoring* de eliminación de restricciones textuales simples que se explicará en las Subsecciones 5.8 y 5.9.

Requisitos.

- Historia de usuario: "**Como** usuario, **quiero** aplicar el *refactoring* de eliminación de restricciones textuales pseudo-complejas a mi modelo, **para** transformarlos en grupos "xor" opcionales manteniendo la semántica del modelo."
- Criterios de aceptación:

  1. ***Eliminación de restricciones textuales pseudo-complejas***

     - **DADO** un modelo de características,
       Y una restricción textual pseudo-compleja c,
       CUANDO aplico el *refactoring* de eliminación de restricciones textuales pseudo-complejas,
       ENTONCES el modelo contiene una instancia menos de restricción textual pseudo-compleja,
       Y el modelo contiene un número de instancias más de restricciones textuales complejas equivalente al número de características implicadas en la restricción textual pseudo-compleja.

Diseño. La Figura 5.12 muestra el esquema de la transformación de modelo correspondiente al *refactoring* de eliminación de restricciones textuales pseudo-complejas que sigue la especificación anterior.

Ejemplo. La Figura 5.13 muestra el *refactoring* de eliminación de restricciones textuales pseudo-complejas aplicado al caso de estudio de la SPL de pizzas.

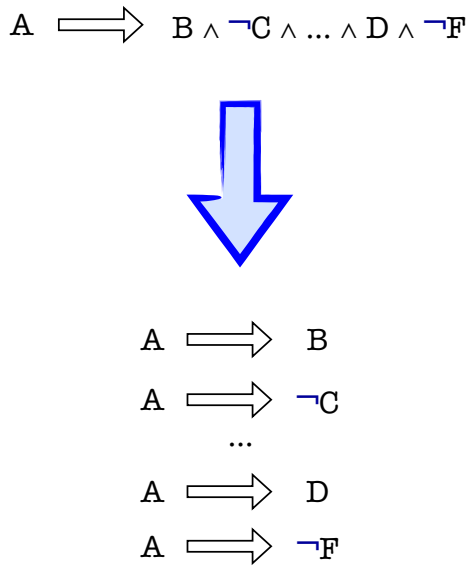

Figura 5.12: *Refactoring* de eliminación de restricciones textuales pseudo-complejas general.

Como se puede observar en la Figura 5.13, al aplicar el *refactoring* de eliminación de restricciones textuales pseudo-complejas, se sustituye la restricción textual pseudo-compleja inicial y por cada característica que estaba en la parte derecha de cada restricción hay una nueva restricción textual simple. Ahora la característica `Vegano` implica que no se puede escoger ni `Salami`, ni `Jamón` ni `Mozzarella`, y si se escoge `Vegetariano` entonces no se puede escoger ni `Salami` ni `Jamón`. En ambos casos no cambia la semántica del modelo porque ambas expresiones significan lo mismo, pero en el segundo caso no hay ninguna restricción textual pseudo-compleja.

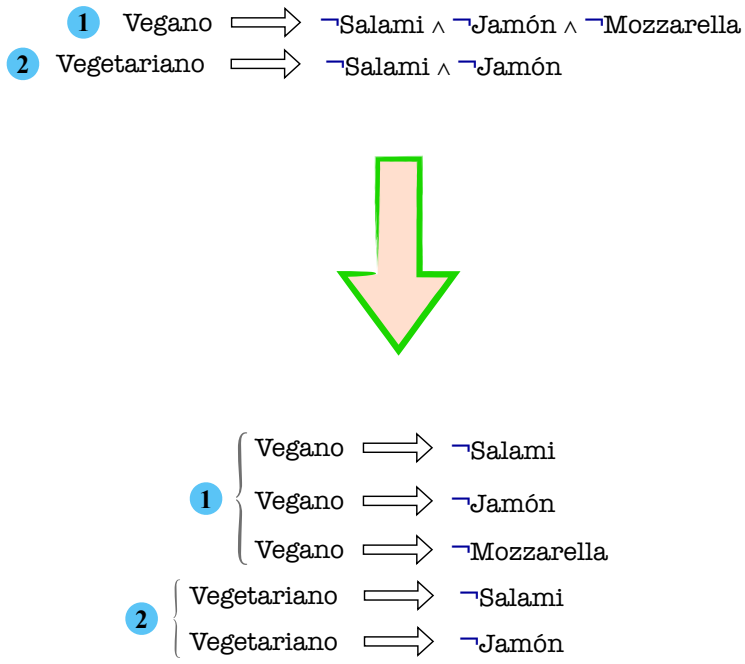

Figura 5.13: *Refactoring* de eliminación de restricciones textuales pseudo-complejas del ejemplo en la Figura 2.7.

5.7. Strict Complex Constraints

Descripción. El *refactoring* para la eliminación de restricciones textuales estrictamente complejas ha sido ya formalizado por Knüppel et al. [Knüppel et al.(2017)] de la siguiente forma: La idea de transformar modelos de características con restricciones estrictamente complejas a modelos sin ellas es traducir las restricciones textuales estrictamente complejas a características abstractas adicionales y restricciones textuales simples sin modificar la semántica del modelo de características. Para eliminar las restricciones textuales complejas de los modelos de características se usan árboles abstractos. La suposición inicial es que se puede transformar cualquier restricción textual a un árbol abstracto ya que la restricción compleja es semánticamente equivalente al árbol abstracto en un modelo de características dado (restringen las mismas combinaciones de características que no pueden ser escogidas simultáneamente). Las nuevas características abstractas se unen en un nuevo grupo "or" que pertenece a una nueva característica abstracta opcional, que será hija de la raíz del modelo de características.

Es necesario convertir antes a **Forma Normal Conjuntiva** (**CNF**, ***Conjunctive Normal Form***) la restricción textual para poder generar el nuevo árbol abstracto.

Hay que tener en cuenta que el resultado de este *refactoring* añade una serie de restricciones simples al modelo, por lo que después de aplicarlo habría que realizar el correspondiente *refactoring* de eliminación de restricciones textuales simples que se explicará en las Subsecciones 5.8 y 5.9.

Requisitos.

- Historia de usuario: "**Como** usuario, **quiero** aplicar el *refactoring* de eliminación de restricciones textuales estrictamente complejas a mi modelo, **para** que desaparezcan manteniendo la semántica del modelo."

- Criterios de aceptación:

  1. ***Eliminación de restricciones textuales estrictamente complejas***

     - **DADO** un modelo de características,
       Y una restricción textual estrictamente compleja c,
       CUANDO aplico el *refactoring* de eliminación de restricciones textuales estrictamente complejas sobre la restricción c,

ENTONCES el modelo contiene una instancia menos de restricción textual estrictamente compleja,
Y el modelo contiene una instancia más de árbol abstracto,
Y el modelo contiene una instancia más de grupos "or",
Y el modelo contiene tantas características abstractas nuevas como característcas estuvieran implicadas en la restricción textual estrictamente compleja,
Y el modelo contiene una nueva característica *or*,
Y la característica *or* es abstracta,
Y la característica *or* es obligatoria,
Y la característica *or* contiene como hijas a todas las características que estuvieran implicadas en la restricción textual estrictamente compleja,
Y el modelo contiene tantas restricciones textuales simples nuevas como características estuvieran implicadas en la restricción textual estrictamente compleja.

Diseño. La Figura 5.14 muestra el esquema de la transformación de modelo correspondiente al *refactoring* de eliminación de restricciones textuales estrictamente compleja que sigue la especificación anterior.

Ejemplo. La Figura 5.15 muestra el *refactoring* de eliminación de restricciones textuales estrictamente complejas aplicado al caso de estudio de la SPL de pizzas.

Como se puede observar en la Figura 5.15, para eliminar la restricción textual estrictamente compleja `BordesQueso` $\wedge$ `Mozzarella` $\implies$ `Grande`, es necesario generar una nueva característica `OR` que es abstracta y obligatoria. La característica `OR` es un grupo "or" que contiene las tres características implicadas en la restricción textual del modelo original. Esas tres características nuevas, `BordesQueso_st`, `Mozzarella_st` y `Grande_st` son características abstractas hijas de `OR` que pertenecen al grupo "or". Además, hay tres restricciones simples nuevas, que impiden escoger `BordesQueso` o `Mozzarella` si se escogen `BordesQueso_st` o `Mozzarella_st`, y que hacen que sea obligatorio escoger `Grande_st` en caso de que se escoja `Grande`.

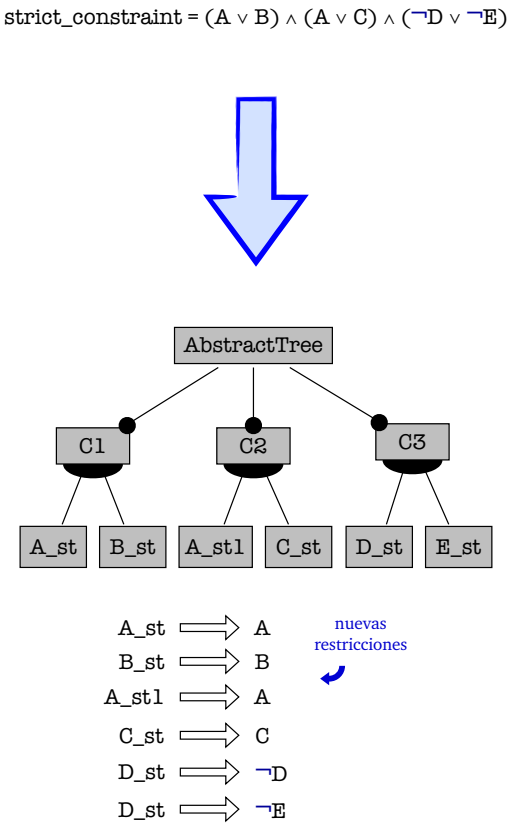

Figura 5.14: *Refactoring* de eliminación de restricciones textuales estrictamente complejas general.

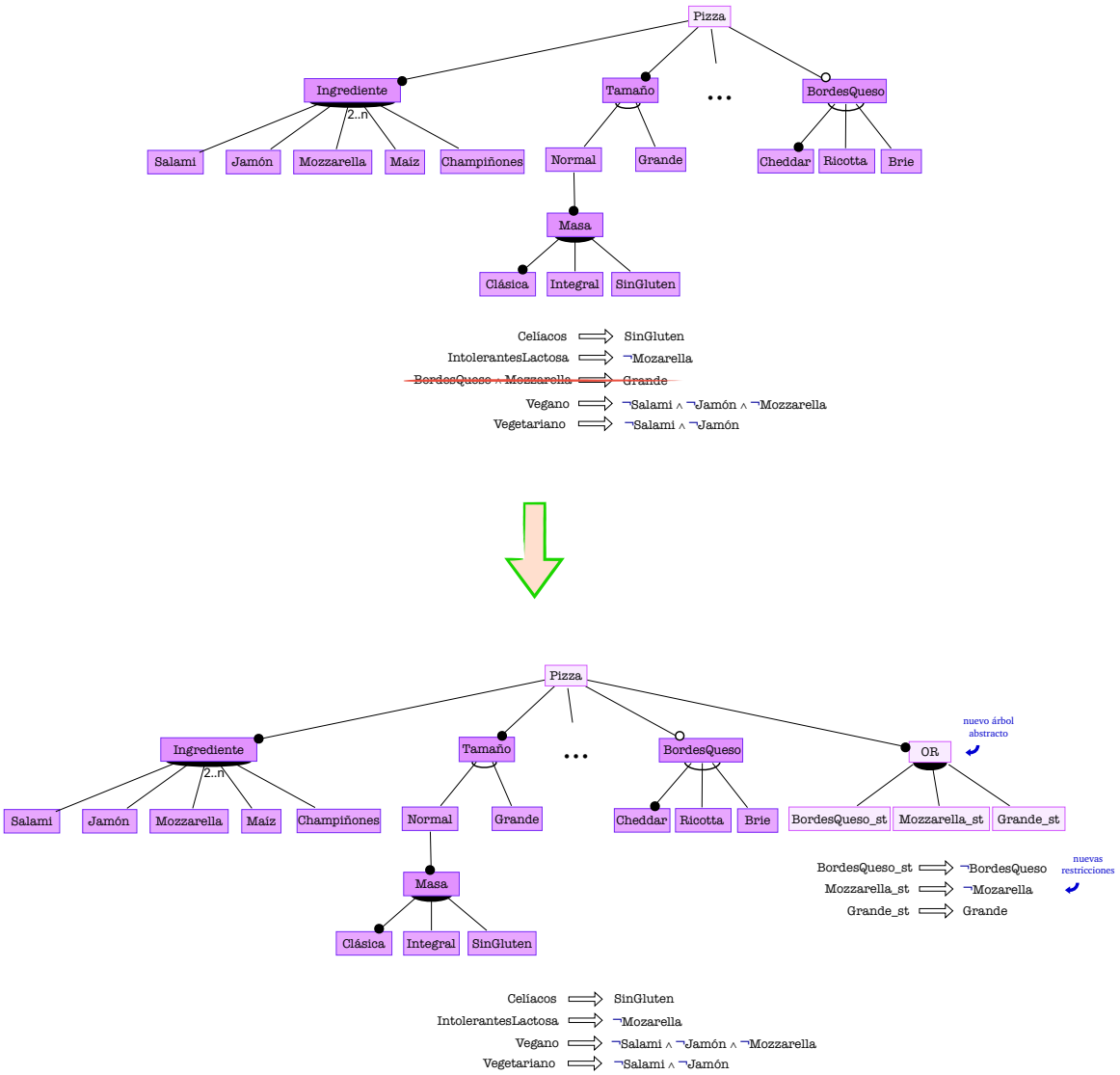

Figura 5.15: *Refactoring* de eliminación de restricciones textuales estrictamente complejas del ejemplo en la Figura 2.8.

5.8. Requires

Descripción. El *refactoring* de eliminación de restricciones simples del tipo $A \implies B$ ha sido descrito en [van den Broek et al.(2008)] de la siguiente manera: sea un modelo de características dado por un árbol T y una restricción textual del tipo $A \implies B$. Se desea construir un modelo cuyos productos sean aquellos productos de T que contienen B cuando se escoge A. Este conjunto de productos es la unión de los conjuntos de productos de $T(+B)$ y $T(-A-B)$. En este caso $T(-A-B)$ es una manera más corta de expresar $(T(-A))(-B)$. El conjunto de productos de $T(+B)$ y $T(-A-B)$ forman una disyunción. Por lo tanto, el modelo de características requerido puede obtenerse creando una nueva característica como raíz que sea un grupo "xor" y que tenga como hijos a $T(+B)$ y $T(-A-B)$. El algoritmo para eliminar las restricciones textuales de este tipo es el que se muestra en el Listado 5.1.

Listado 5.1: Algoritmo para eliminar las restricciones textuales del tipo $A \implies B$ de T.

```
Construye el árbol T(+B) y T(-A-B).
Si ambos árboles no son iguales a NULL, entonces el resultado consiste en una nueva raíz,
que es un grupo ``xor'' con subárboles T(+B) y T(-A-B).
Si T(-A-B) es igual a NULL, entonces el resultado es T(+B),
Si T(+B) es igual a NULL, entonces el resultado es T(-A-B).
```

Dicho de otra forma el *refactoring* para eliminar las restricciones textuales de la forma $A \implies B$ consiste en duplicar el subárbol original donde aparecen las características implicadas en la restricción, realizando ciertas transformaciones que obligan a que si hay posibilidad de escoger la característica A, se tenga que escoger obligatoriamente también la característica B.

Requisitos.

- Historia de usuario: "**Como** usuario, **quiero** aplicar el *refactoring* de eliminación de restricción textual simple del tipo "requires" a mi modelo, **para** que desaparezca la restricción textual del tipo $A \implies B$ manteniendo la semántica del modelo."
- Criterios de aceptación:

  1. ***Eliminación de restricción textual de la forma*** $A \implies B$

     - **DADO** un modelo de características,

       Y una restricción textual del tipo $A \implies B$,

CUANDO aplico el *refactoring* de eliminación de restricciones textuales del tipo "requires" sobre el modelo,

ENTONCES el modelo contiene una instancia menos de restricción textual del tipo $A \implies B$,

Y el modelo contiene dos subárboles que representan las mismas características,

Y el modelo contiene una característica nueva $requires$,

Y la característica $requires$ es un grupo "xor",

Y la característica $requires$ es abstracta,

Y la característica $requires$ es la nueva raíz del modelo,

Y la característica $requires$ contiene como hijos a la raíz del árbol original de manera duplicada.

Diseño. La Figura 5.16 muestra el esquema de la transformación de modelo correspondiente al *refactoring* de eliminación de restricciones textuales de la forma $A \implies B$ que sigue la especificación anterior.

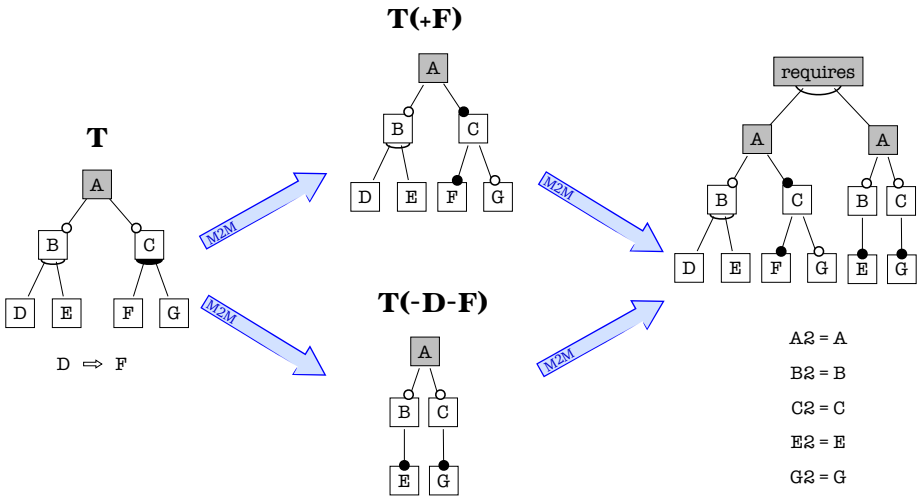

Figura 5.16: [van den Broek et al.(2008)] *Refactoring* de eliminación de restricción textual simple del tipo $A \implies B$ general.

Ejemplo. La Figura 5.17 muestra el *refactoring* para eliminar restricciones textuales del tipo $A \implies B$ aplicado al caso de estudio de la SPL de pizzas.

Como se puede observar en la Figura 5.17, se ha eliminado la restricción textual simple de la forma $A \implies B$, que en este caso es `Celíacos` $\implies$ `SinGluten`. Para poder eliminarla y que no cambie la semántica del modelo, se han creado dos árboles a partir del modelo de características principal. La modificación que se ha llevado a cabo en el primero de ellos es que se ha convertido a obligatoria la característica `Celíacos` ($T(+$ `Celíacos`$)$), mientras que en el segundo se ha eliminado esa misma característica `Celíacos` y también se ha eliminado `SinGluten` ($T(-$ `Celíacos` $-$ `SinGluten`$)$). Además, ahora ambos árboles que antes tenían como raíz la característica `Pizza` son hijos de una nueva característica abstracta llamada `requires` que es un grupo "xor".

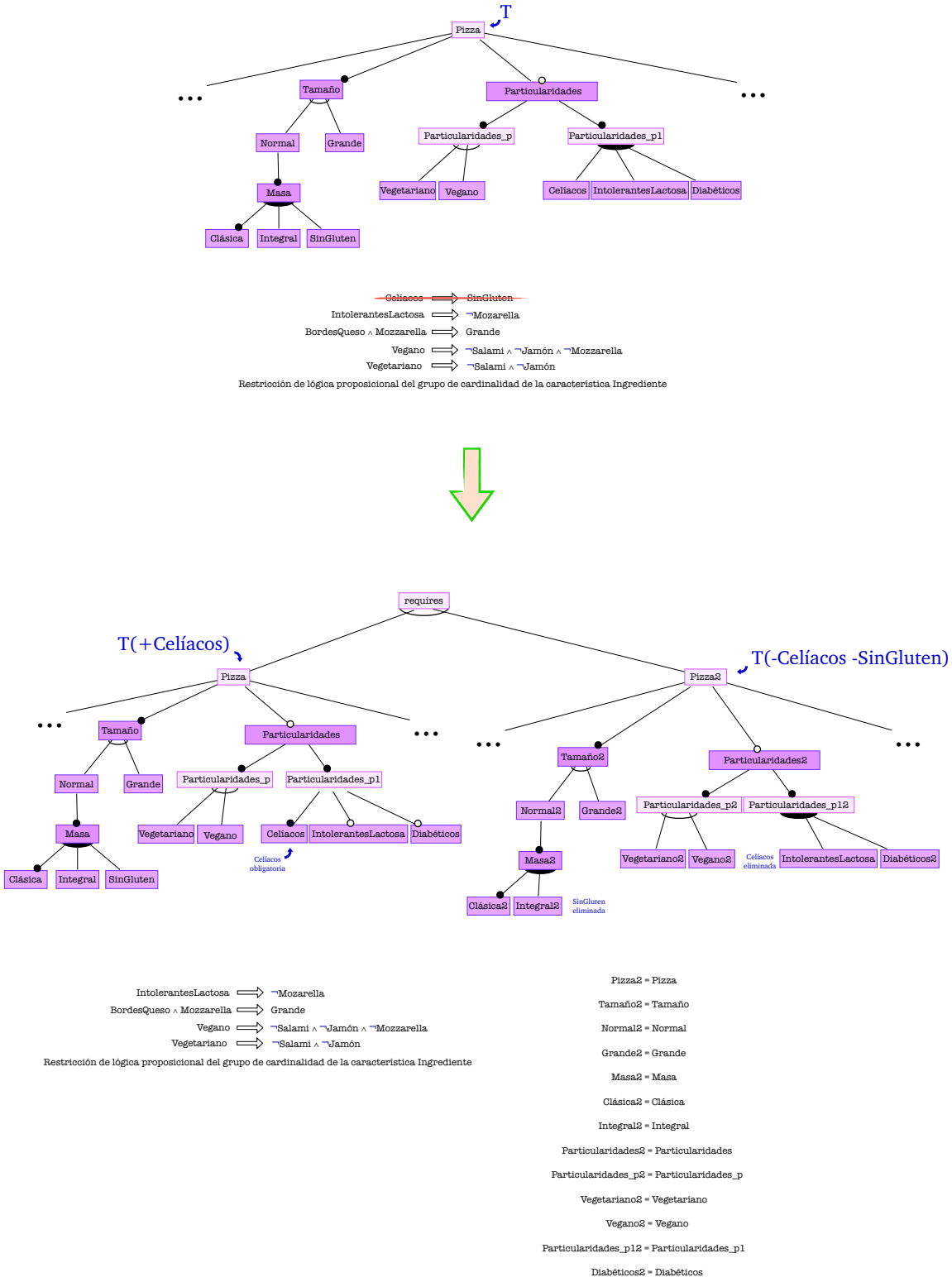

Figura 5.17: *Refactoring* de eliminación de restricción textual simple del tipo $A \implies B$ del ejemplo en la Figura 2.9.

5.9. Excludes

Descripción. El *refactoring* de eliminación de restricciones simples del tipo $A \implies B$ ha sido descrito en [van den Broek et al.(2008)] de la siguiente manera: sea un modelo de características dado por un árbol T y una restricción textual del tipo $A \implies \neg B$. Se quiere construir un modelo de características cuyos productos sean aquellos productos de T que no contienen a la vez las características A y B. Este conjunto de productos es la unión de los conjuntos del árbol $T(-B)$ y $T(-A+B)$. Además, los conjuntos de productos de $T(B)$ y $T(-A+B)$ están en una disyunción. Por lo tanto, el modelo de características requerido se puede obtener tomando una nueva característica "xor" como la raíz y que tiene los árboles $T(B)$ y $T(-A+B)$ como hijos de ella. El algoritmo para eliminar las restricciones textuales del tipo $A \implies \neg B$ de T es el que se indica en el Listado 5.2.

Listado 5.2: Algoritmo para eliminar las restricciones textuales del tipo $A \implies \neg B$ de T.

```
Construye el árbol T(-B) y T(-A+B).
Si ambos árboles no son iguales a NULL, entonces el resultado consiste en una nueva raíz,
que es un grupo ``xor'' con subárboles T(-B) y T(-A+B).
Si T(-B) es igual a NULL, entonces el resultado es T(-A+B),
Si T(-A+B) es igual a NULL, entonces el resultado es T(-B).
```

Dicho de otra manera, el *refactoring* para eliminar las restricciones textuales de la forma $A \implies \neg B$ consiste en duplicar el subárbol original donde aparecen las características implicadas en la restricción, realizando ciertas transformaciones que impiden que si hay posibilidad de escoger la característica A, se pueda escoger también la característica B.

Requisitos.

- Historia de usuario: "**Como** usuario, **quiero** aplicar el *refactoring* de eliminación de restricción textual simple del tipo "excludes" a mi modelo, **para** que desaparezca la restricción textual del tipo $A \implies \neg B$ manteniendo la semántica del modelo."
- Criterios de aceptación:

  1. ***Eliminación de restricción textual de la forma*** $A \implies \neg B$

     - **DADO** un modelo de características,
       Y una restricción textual del tipo $A \implies \neg B$,
       CUANDO aplico el *refactoring* de eliminación de restricciones textuales del

tipo “excludes” sobre el modelo,

ENTONCES el modelo contiene una instancia menos de restricción textual del tipo $A \implies \neg B$,

Y el modelo contiene dos subárboles que representan las mismas características,

Y el modelo contiene una característica nueva $excludes$,

Y la característica $excludes$ es un grupo “xor”,

Y la característica $excludes$ es abstracta,

Y la característica $excludes$ es la nueva raíz del modelo,

Y la característica $excludes$ contiene como hijos a la raíz del árbol original de manera duplicada.

Diseño. La Figura 5.18 muestra el esquema de la transformación de modelo correspondiente al *refactoring* de eliminación de restricciones textuales de la forma $A \implies \neg B$ que sigue la especificación anterior.

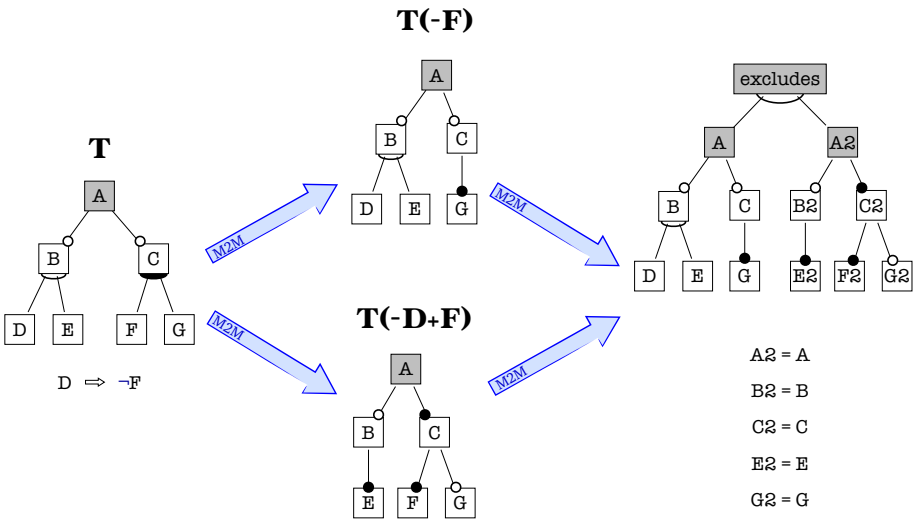

Figura 5.18: [van den Broek et al.(2008)] *Refactoring* de eliminación de restricción textual simple del tipo $A \implies \neg B$ general.

Ejemplo. La Figura 5.19 muestra el *refactoring* para eliminar restricciones textuales del tipo $A \implies \neg B$ aplicado al caso de estudio de la SPL de pizzas.

Como se puede observar en la Figura 5.19, se ha eliminado la restricción textual simple de la

forma $A \implies \neg B$, que en este caso es `IntolerantesLactosa` $\implies \neg$ `Mozzarella`. Para poder eliminarla y que no cambie la semántica del modelo, se han creado dos árboles a partir del modelo de características principal. La modificación que se ha llevado a cabo en el primero de ellos es que se ha eliminado la característica `Mozzarella` ($T(-$ `Mozzarella`)), mientras que en el segundo se ha eliminado la característica `IntolerantesLactosa` y se ha convertido `Mozzarella` a una característica obligatoria ($T(-$ `IntolerantesLactosa` $+$ `Mozzarella`)). Además, ahora ambos árboles que antes tenían como raíz la característica `Pizza` ahora son hijos de una nueva característica abstracta llamada `excludes` que es un grupo "xor".

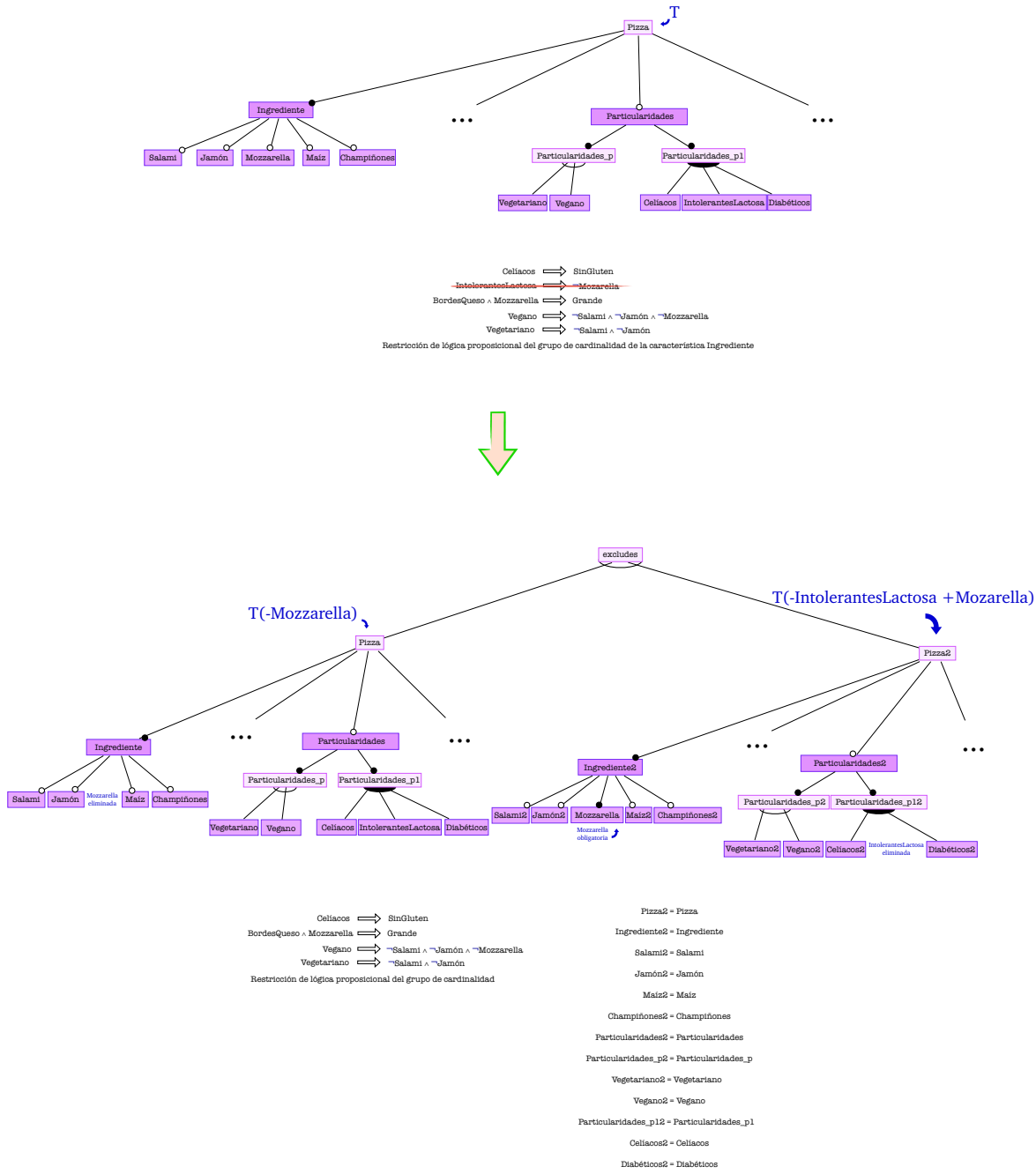

Figura 5.19: *Refactoring* de eliminación de restricción textual simple del tipo $A \implies \neg B$ del ejemplo en la Figura 2.10.

6

Implementación y evaluación

En este capítulo se muestra la implementación de la biblioteca de *refactorings*, así como las pruebas que se han llevado a cabo para comprobar la *correctitud* y *validez* de los mismos. Tras implementar los *refactorings* se han construido una serie de modelos de prueba para compararlos y comprobar que se obtenía el resultado esperado.

6.1. Implementación

En la Figura 6.1 se muestra el diagrama de clases de los *refactorings* que se han implementado en este TFG. En él se puede observar la interfaz **FMRefactoring**. En esta interfaz se definen los métodos que deben implementar todos los *refactorings*. El "motor" que hace que funcionen los *refactorings* es **RefactoringEngine**. En él se encuentran las tres operaciones principales que se definieron en los casos de uso en la Figura 4.1. En la clase **FMUtils** se encuentran las operaciones de ayuda necesarias para los modelos de características, mientras que la clase **ConstraintsUtils** contiene las operaciones de ayuda para las restricciones textuales, por ejemplo para poder diferenciar entre restricciones textuales simples y complejas.

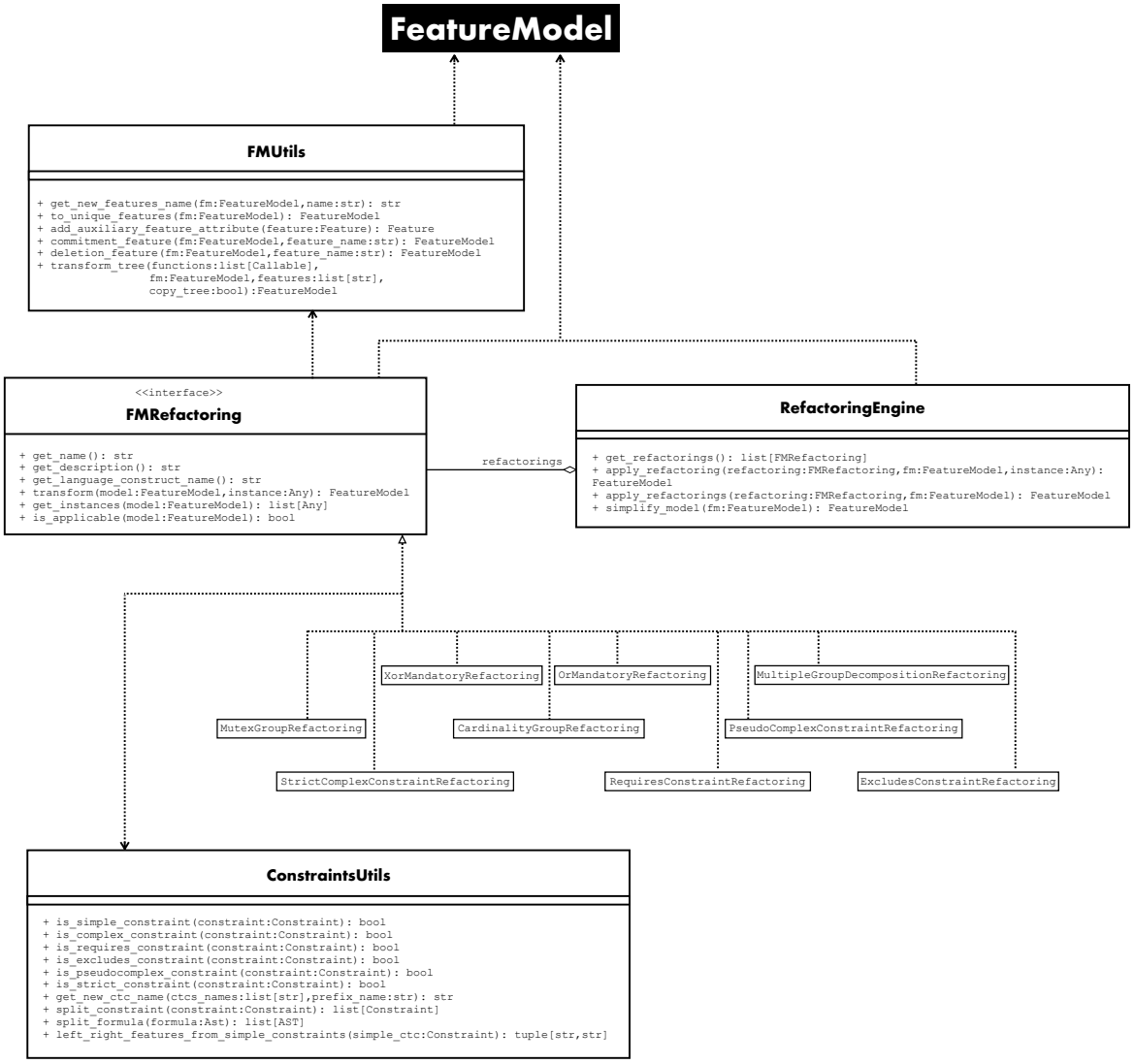

Figura 6.1: Diagrama de clases de los *refactorings*.

La implementación en código Python de la biblioteca de *refactorings* implementada para este TFG está disponible en la biblioteca:

- Repositorio GitHub de la **Biblioteca de *refatorings***:
  https://github.com/anhe2p9/biblioteca_refactorings.git

6.2. Testing

Para desarrollar los test se han construido una serie de modelos de prueba que contienen o bien los grupos avanzados o bien las restricciones textuales pertinentes. Se han creado también los modelos que tengan la misma semántica pero que contengan solo constructores de variabilidad simples. El objetivo era ir pasando todos los modelos con constructores avanzados de variabilidad por **RefactoringEngine** y comparar el resultado con el modelo que se había creado inicialmente con constructores simples. Se considera que un *refactoring* verifica su *correctitud* o *validez* si al comparar los productos de ambos modelos el resultado es el mismo.

Para hacer dicha comprobación se usa la operación de **products** disponible en Flama mediante el plugin de SAT que transforma el modelo de característica a lógica proposicional para razonar sobre el modelo. Se han desarrollado tests unitarios siguiendo la especificación de los criterios de aceptación del Capítulo 5. Todos los modelos se han creado a mano para comprobar los *refactorings*.

En la Figura 6.2 se muestran los casos base que se han montado para asegurar la *correctitud* o *validez* de los *refactorings*. Como se puede observar, para cada *refactoring* hay varios casos base que se comprueban, como por ejemplo que el constructor avanzado de variabilidad esté en diferentes lugares del modelo de características, que se relacione de diferentes formas con el resto de características, o que se incluyan varias instancias de ese constructor en diferentes partes del modelo (incluido uno dentro de otro).

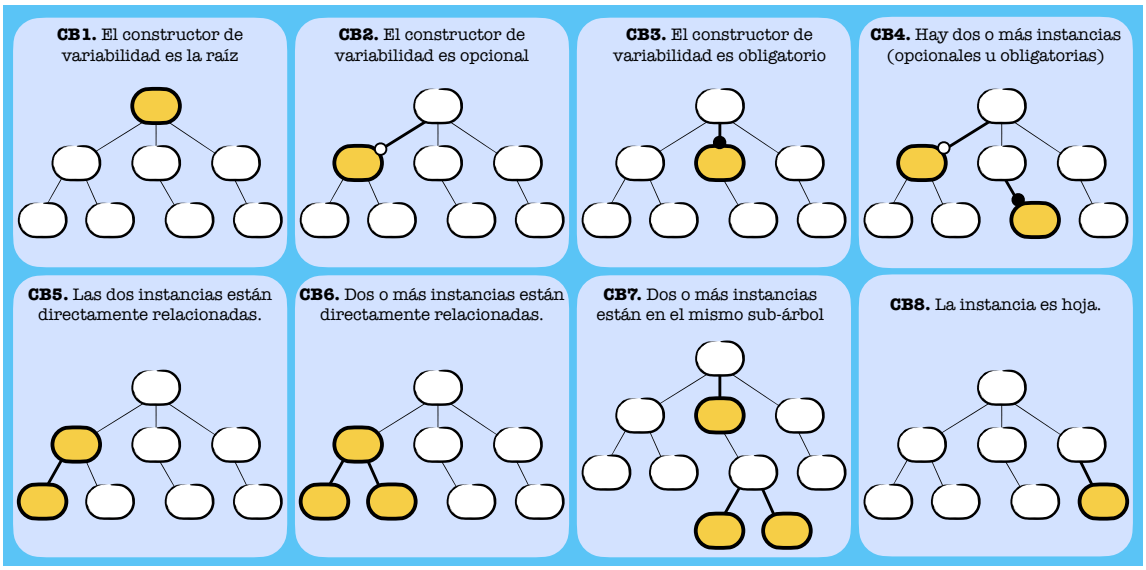

Figura 6.2: Casos base para comprobar la *correctitud* de los *refactorings* [Horcas et al.(2023b)].

6.3. Extensión de la librería para nuevos refactorings

En caso de querer implementar un nuevo *refactoring* para modelos de características sería necesario seguir el Diagrama de clases de la Figura 6.1. Como se puede observar en la clase **FMRefactoring**, el nuevo *refactoring* debe tener los siguientes métodos:

- `get_name()`: devuelve un *string* con el nombre del nuevo *refactoring*.
- `get_description()`: devuelve un *string* con la descripción de lo que hace el *refactoring*.
- `get_language_construct_name()`: devuelve un *string* con el nombre del constructor de variabilidad al que afecta el *refactoring*.
- `transform(model:FeatureModel,instance:Any)`: devuelve el modelo de características con las modificaciones pertinentes que se realiza para quitar la instancia necesaria en cada caso. Aquí es donde se debe implementar el código del *refactoring*.
- `get_instances(model:FeatureModel)`: devuelve una lista con todas las instancias que contiene el modelo de características que se ha pasado a las que se les pueda aplicar el *refactoring*.
- `is_applicable(model:FeatureModel)`: devuelve un *booleano* que indica si es posible aplicar el *refactoring* o no (porque esté la instancia que lo hace necesario o no).

6.4. Resultados

Los resultados de este TFG contribuyen a las tareas del proyecto de investigación *Rhea: Lenguaje y ecosistema para el análisis, derivación, resolución y materialización de la variabilidad centrado en la arquitectura y en los atributos de calidad* (Ref. *P18-FR-1081*) financiado por la Junta de Andalucía y cuya IP principal es Lidia Fuentes. La biblioteca se ha integrado en la herramienta *Rhea*[1]. Rhea (actualmente en desarrollo) es una herramienta online para la edición y refactorización de modelos de características con el fin de proporcionar interoperabilidad entre las herramientas de modelado y análisis de la variabilidad.

Merece también la pena señalar que parte de los *refactorings* de la biblioteca propuesta en este TFG han servido para la publicación de varios artículos científicos. Concretamente en una publicación de revista indexada en el JCR como Q2 y en un artículo de congreso indexado en el SCIE como Clase 2 (A-):

- José Miguel Horcas, Mónica Pinto, Lidia Fuentes. *A modular metamodel and refactoring rules to achieve software product line interoperability*. Journal of Systems and Software (JSS). Vol: 197. Año: 2023. DOI: https://doi.org/10.1016/j.jss.2022.111579

  En ese artículo se presentan dos *refactorings* (MutexGroup y CardinalityGroup) desarrollados en Java y en Henshin (un lenguaje de transformación de modelos gráfico). La complejidad en el desarrollo de ambos *refactorings* usando esos lenguajes propició que el desarrollo de este TFG se llevara a cabo usando Python, reimplementando ambos *refactorings*.

- José Miguel Horcas, Joaquín Ballesteros, Mónica Pinto, Lidia Fuentes. *Elimination of constraints for parallel analysis of feature models*. 27th ACM International Systems and Software Product Line Conference (SPLC 2023).

  En ese artículo se usan los *refactorings* de la biblioteca que afectan a las restricciones textuales. Los *refactorings* fueron adaptados a las necesidades del proyecto para integrarlos en la herramienta propuesta en el artículo.

[1] Rhea: https://rhea.caosd.lcc.uma.es/

7

Conclusiones y líneas futuras

En este capítulo se muestran las conclusiones que se han obtenido durante el desarrollo del TFG y las líneas futuras de investigación que partirán de la base del contenido de este trabajo.

7.1. Conclusiones

Durante el desarrollo de este TFG se ha podido comprobar que la interoperabilidad entre las herramientas de modelos de características es un campo en continua expansión y que está en sus primeras fases de desarrollo. Los *refactorings* son necesarios y han podido proporcionar la interoperabilidad que se necesitaba entre herramientas que trabajan con modelos de características, pues los constructores avanzados de variabilidad proporcionan una mayor expresividad en comparación con los simples y rompen con las limitaciones que dichos constructores simples de variabilidad presentaban anteriormente.

Los objetivos planteados se han cumplido:

- Se ha desarrollado una biblioteca de *refactorings* (transformaciones de modelos de características sin modificaciones de productos) para los modelos de características que proporcionan interoperabilidad entre las diferentes herramientas que existen.
- Se han identificado, formalizado e implementado transformaciones de modelos basados en *refactorings* para los modelos de características dando como resultado una biblioteca de los siguientes *refactorings*:
    1. Grupo "xor" con una característica obligatoria.
    2. Grupo "or" con una característica obligatoria.
    3. Grupos de descomposición múltiple.
    4. Grupos mutex.
    5. Grupos de cardinalidad.
    6. Restricciones textuales pseudo-complejas.
    7. Restricciones textuales estrictamente complejas.
    8. Restricción textual simple "requires".
    9. Restricción textual simple "excludes".
- Se han evaluado los *refactorings* desarrollados de forma cualitativa, estudiando su, correctitud y completitud. El estudio de la escalabilidad para modelos de características muy grandes se desarrollará como línea futura de investigación.

- Se han integrado los *refactorings* en una herramienta web existente (Rhea) para la gestión de los modelos de características de forma que facilita la interoperabilidad de los *refactorings* con otras herramientas de SPL como FeatureIDE, Glencoe o SPLOT. Rhea es una herramienta externa a este TFG que se encuentra actualmente en desarrollo, pero en cuyo backend se han incorporado todos los refactorings de la biblioteca propuesta en este TFG.

Es una ventaja que los modelos no cambien su semántica al ser sometidos a una transformación sin cambios de productos. El hecho de mantener los productos intactos en un modelo de características al operar sobre él permite ahorrar tiempo y esfuerzo en comprobar la relevancia que tendrían los cambios de semántica si se produjesen. Además, se han implementado unos tests unitarios que comprueban la correctitud de los *refactorings* y su validez.

Este TFG también ha aportado el conocimiento y aprendizaje de un área nueva de ingeniería del software como son las líneas de producto software y la variabilidad de los sistemas. Aunque las líneas de producto son un concepto bastante antiguo introducido en la industria del automóvil por Ford, en ingeniería del software aún es una área que necesita mayor investigación. En ese sentido, este TFG también aporta su granito de arena para dar avanzar un paso más el estado del arte de las transformaciones de modelo aplicadas a modelos de características.

7.2. Líneas futuras de investigación

Este trabajo sirve como base para seguir expandiendo las fronteras de la tecnología SPL. A partir del contenido del mismo, la biblioteca propuesta se puede ampliar para desarrollar nuevos refactorings para extensiones más avanzadas de los modelos de características.

7.2.1. Nuevos refactorings para extensiones más avanzadas

Se pretende ampliar la biblioteca de *refactorings* para que también se puedan aplicar soluciones de interoperabilidad en características con diferentes propiedades más avanzadas, como que tengan **atributos**, que sean características **numéricas** y también características **clonables** [Horcas et al.(2023a)].

- Las características con atributos son aquellas que permiten escoger entre varias propiedades de la característica que los contiene. Al escoger una característica, es posible definir además una propiedad intrínseca de dicha característica (por ejemplo el coste, rendimiento, eficiencia energética,...).

- Las características numéricas son aquellas que contienen un cuantificador intrínseco en ellas (por ejemplo un valor entero). Cada característica numérica puede añadir un número arbitrario de productos, pues es una nueva manera de definir las características. Hasta ahora solo se ha trabajado con características **booleanas**, que se podían o bien escoger o bien no escogerlas. A diferencia de las características booleanas, las numéricas se definen con un cuantificador que genera una nueva configuración y por tanto un nuevo producto en cada cantidad que se define al resolver el modelo de características.

- Las características clonables son aquellas que encabezan un subárbol que puede darse de manera repetida en un sistema. El hecho de que existan este tipo de características facilita la configuración del sistema porque evita repeticiones innecesarias al poder repetir la configuración de un conjunto de características haciendo pequeñas variaciones. Las características clonables introducen también restricciones más complejas como los cuantificadores universales y existenciales.

7.2.2. Estudio de la escalabilidad de los refactorings

Por otro lado, la biblioteca de refactorings puede usarse para estudiar la escalabilidad de los refactorings en modelos de variabilidad colosales. Sin embargo, los modelos de características que existen de gran tamaño ya se encuentran simplificados (con constructores de variabilidad simples) ya que estos modelos se han diseñado con las herramientas de modelado más básicas [Horcas et al.(2023a)]. Para poder realizar un estudio de la escalabilidad de los refactorings, primero se deben generar modelos de características sintéticos de diferente tamaño y que contengan los diferentes constructores de variabilidad necesarios para poder aplicar los refactorings.

Bibliografía

[Anna Schmitt(2018)] Georg Rock Anna Schmitt, Christian Bettinger. 2018. Glencoe – A Tool for Specification, Visualization and Formal Analysis of Product Lines. In *Transdisciplinary Engineering Methods for Social Innovation of Industry 4.0 (Advances in Transdisciplinary Engineering, Vol. 7)*. 665–673. https://doi.org/10.3233/978-1-61499-898-3-665

[Apel et al.(2013)] Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines - Concepts and Implementation.* Springer. https://doi.org/10.1007/978-3-642-37521-7

[Benavides et al.(2010)] David Benavides, Sergio Segura, and Antonio Ruiz Cortés. 2010. Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.* 35, 6 (2010), 615–636. https://doi.org/10.1016/j.is.2010.01.001

[Berger et al.(2013)] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki. 2013. A Study of Variability Models and Languages in the Systems Software Domain. *IEEE Trans. Software Eng.* 39, 12 (2013), 1611–1640. https://doi.org/10.1109/TSE.2013.34

[Czarnecki and Wasowski(2007)] Krzysztof Czarnecki and Andrzej Wasowski. 2007. Feature Diagrams and Logics: There and Back Again. *11th software product line conference*, 23–34. https://doi.org/10.1109/SPLINE.2007.24

[Galindo et al.(2023)] José A. Galindo, Jose-Miguel Horcas, Alexander Felferning, David Fernandez-Amoros, and David Benavides. 2023. FLAMA. A collaborative effort to build a new framework for the automated analysis of feature models. In *26th ACM International Systems and Software Product Lines Conference (SPLC)*, Vol. B. ACM.

[Horcas et al.(2023a)] José Miguel Horcas, Mónica Pinto, and Lidia Fuentes. 2023a. Empirical analysis of the tool support for software product lines. *Softw. Syst. Model.* 22, 1 (2023), 377–414. https://doi.org/10.1007/s10270-022-01011-2

[Horcas et al.(2023b)] José Miguel Horcas, Mónica Pinto, and Lidia Fuentes. 2023b. A modular metamodel and refactoring rules to achieve software product line interoperability. *J. Syst. Softw.* 197 (2023), 111579. https://doi.org/10.1016/j.jss.2022.111579

[Juodisius et al.(2019)] Paulius Juodisius, Atrisha Sarkar, Raghava Rao Mukkamala, Michal Antkiewicz, Krzysztof Czarnecki, and Andrzej Wasowski. 2019. Clafer: Lightweight Modeling of Structure, Behaviour, and Variability. *Programming Journal* 3, 1 (2019), 2. https://doi.org/10.22152/programming-journal.org/2019/3/2

[Kang et al.(1990)] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. 1990. *Feature-oriented domain analysis (FODA) feasibility study.* Technical Report. Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst.

[Knüppel et al.(2017)] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. 2017. Is There a Mismatch between Real-World Feature Models and Product-Line Research?. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 291–302. https://doi.org/10.1145/3106237.3106252

[Mendonca et al.(2009)] Marcilio Mendonca, Moises Branco, and Donald Cowan. 2009. S.P.L.O.T.: Software Product Lines Online Tools. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications* (Orlando, Florida, USA) *(OOPSLA '09)*. ACM, New York, NY, USA, 761–762. https://doi.org/10.1145/1639950.1640002

[Pohl et al.(2005)] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer-Verlag, Berlin, Heidelberg.

[Raatikainen et al.(2019)] Mikko Raatikainen, Juha Tiihonen, and Tomi Männistö. 2019. Software product lines and variability modeling: A tertiary study. *Journal of Systems and Software* 149 (2019), 485 – 510. https://doi.org/10.1016/j.jss.2018.12.027

[Sepúlveda et al.(2016)] Samuel Sepúlveda, Ania Cravero, and Cristina Cachero. 2016. Requirements modeling languages for software product lines: A systematic literature review. *Information and Software Technology* 69 (2016), 16–36.

[Sundermann et al.(2021)] Chico Sundermann, Kevin Feichtinger, Dominik Engelhardt, Rick Rabiser, and Thomas Thüm. 2021. Yet another textual variability language?: a community effort towards a unified language. In *25th ACM International Systems and Software Product Line Conference (SPLC)*, Vol. A. 136–147. https://doi.org/10.1145/3461001.3471145

[ter Beek et al.(2019)] Maurice H. ter Beek, Klaus Schmid, and Holger Eichelberger. 2019. Textual variability modeling languages: an overview and considerations. In *Proceedings of the 23rd International Systems and Software Product Line Conference, SPLC 2019, Volume B, Paris, France, September 9-13, 2019*, Carlos Cetina, Oscar Díaz, Laurence Duchien, Marianne Huchard, Rick Rabiser, Camille Salinesi, Christoph Seidl, Xhevahire Tërnava, Leopoldo Teixeira, Thomas Thüm, and Tewfik Ziadi (Eds.). ACM, 82:1–82:7. https://doi.org/10.1145/3307630.3342398

[Thüm et al.(2009)] Thomas Thüm, Don S. Batory, and Christian Kästner. 2009. Reasoning about edits to feature models. In *International Conference on Software Engineering (ICSE)*. 254–264. https://doi.org/10.1109/ICSE.2009.5070526

[Thüm et al.(2014)] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming* 79 (2014), 70 – 85. https://doi.org/10.1016/j.scico.2012.06.002 Experimental Software and Toolkits (EST 4): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT-3 2010).

[van den Broek et al.(2008)] Pim van den Broek, Ismênia Galvão, and Joost Noppen. 2008. Elimination of Constraints from Feature Trees. In *Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings. Second Volume (Workshops)*, Steffen Thiel and Klaus Pohl (Eds.). Lero Int. Science Centre, University of Limerick, Ireland, 227–232.

Apéndice A.0
Manual de usuario

A.1. Manual de usuario

En este apéndice se detalla el procedimiento para instalar y usar la biblioteca de *refactorings* para modelos de características.

A.1.1. Requerimientos técnicos

- Python 3.9+
- Flama

La biblioteca ha sido probada en Linux (Mint y Ubuntu) y Windows 11.

A.1.2. Descarga e instalación

1. Instalar Python 3.9+
2. Descargar/Clonar el repositorio GitHub y entrar al directorio principal:
   `git clone https://github.com/anhe2p9/biblioteca_refactorings.git`
3. Crear un entorno virtual: `python -m venv env`
4. Activar el entorno virtual:
   - En Linux: `source env/bin/activate`
   - En Windows: `.\env\Scripts\Activate`

   ** En caso de que esté usando Ubuntu, instale el paquete `python3-dev` con el comando `sudo apt update && sudo apt install python3-dev` y actualice wheel y setuptools con el comando `pip install --upgrade pip wheel setuptools` justo después del paso 4.
5. Instale las dependencias: `pip install -r requirements.txt`

A.1.3. Ejecución de la biblioteca de refactorings (RefactoringEngine)

Se puede usar cualquier modelo de la carpeta de modelos o en la carpeta de tests para ejecutar y probar la biblioteca.

- **Ayuda:** proporciona ayuda para ejecutar **RefactoringEgine**.
  `python main.py -h`

- **Aplicar un *refactoring* sobre una instancia:** Aplica un *refactoring* a una instancia determinada (característica o restricción textual) del modelo de características proporcionado por el usuario.

  - Ejecución: `python main.py -fm FEATURE_MODEL -i INSTANCE`
  - Inputs:
    - El parámetro `FEATURE_MODEL` especifica el camino del archivo del modelo de características en formato UVL.
    - El parámetro `INSTANCE` especifica el nombre de la característica o el número de restricciones (comenzando en 0) sobre la que el correspondiente *refactoring* será aplicado.
  - Outputs:
    - Un fichero con un modelo de características en formato UVL con la instancia dada refactorizada.
  - Ejemplo: `python main.py -fm models/Pizzas_complex.uvl -i Specials`

- **Refactorizar un constructor de variabilidad:** Aplica un *refactoring* dado a todas las instancias en el modelo de características proporcionado.

  - Ejecución: `python main.py -fm FEATURE_MODEL -r REFACTORING`
  - Inputs:
    - El parámetro `FEATURE_MODEL` especifica el camino del archivo del modelo de características en formato UVL.
    - El parámetro `REFACTORING` especifica el nombre del *refactoring*. Este puede ser uno de los *refactorings* disponibles en la biblioteca: ['MutexGroupRefactoring', 'CardinalityGroupRefactoring', 'MultipleGroupDecompositionRefactoring', 'OrMandatoryRefactoring', 'XorMandatoryRefactoring', 'PseudoComplexConstraintRefactoring', 'StrictComplexConstraintRefactoring', 'RequiresConstraintRefactoring', 'ExcludesConstraintRefactoring'].
  - Outputs:

    - Un modelo de características en formato UVL con todas las instancias del *refactoring* especificado transformadas.
  - Ejemplo: `python main.py -fm models/Pizzas_complex.uvl -r MutexGroupRefactoring`

- **Simplificar un modelo de características:** aplicar todos los posibles *refactorings* a todas las instancias dadas en el modelo de características.
  - Ejecución: `python main.py -fm FEATURE_MODEL`
  - Inputs:
    - El parámetro `FEATURE_MODEL` especifica el camino del archivo del modelo de características en formato UVL.
  - Outputs:
    - Un modelo de características en formato UVL con todas las instancias de todos los *refactorings* posibles transformadas.
  - Ejemplo: `python main.py -fm models/Pizzas_simple.uvl`

A.1.4. Ejecución de los tests

```
pytest -v tests/Test_refactorings.py
```

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga

E.T.S. DE INGENIERÍA INFORMÁTICA