Instituto Tecnológico y de Estudios Superiores de Monterrey

Campus Monterrey

División de Mecatrónica y Tecnologías de Información

Programa de Graduados

Maestría en Ciencias en Tecnología Informática

Tesis

Pynion: Prototipo de Herramienta para Mecanismos de Variación para Líneas de Productos para Ambientes Heterogéneos y Legados

por

Alejandro Pérez Muñoz
604718

Monterrey, N.L., Diciembre de 2008

Instituto Tecnológico y de Estudios Superiores de Monterrey

Campus Monterrey

División de Mecatrónica y Tecnologías de Información

Programa de Graduados

Los miembros del comité de tesis recomendamos que la presente tesis de Alejandro Pérez Muñoz sea aceptada para desarrollar el proyecto de tesis como requisito parcial para obtener el grado académico de **Maestro en Ciencias,** especialidad en:

Tecnología Informática

Comité de Tesis:

_____

Dr. Guillermo Jímenez Pérez

Asesor Principal

_____ _____

Dr. Juan Carlos Lavariega Jarquin

Sinodal

M.E. Mario Manuel Martinez Garza

Sinodal

_____

Dr. Joaquin Acevedo Mascarúa

Director del Programa de Graduados

Diciembre de 2008

Índice

Índice de figuras

Índice de cuadros

Resumen

La gran mayoría de los sistemas de software son sistemas complejos que operan en ambientes heterogéneos. En una misma aplicación puede utilizarse diversos lenguajes de programación y formatos de almacenamiento de datos para adaptarse a las necesidades y diferentes vistas de un mismo proyecto. En esta tesis desarrollaremos Pynion un prototipo de herramienta para mecanismos de variación diseñado para facilitar el desarrollo de líneas de productos de software en ambientes heterogéneos. También implementaremos una línea de productos utilizando esta herramienta para mecanismos de variación para poder verificar su factibilidad como herramienta de desarrollo de software.

1. Introducción

Las líneas de productos de software son una gran herramienta para la reutilización de recursos de software. Las líneas de productos de software permiten reutilizar no sólo secciones particulares de código sino la arquitectura entera del programa e incluso requerimientos completos [1]. Cuando un grupo de aplicaciones de software comparte más funcionalidad de la que difieren estas son idóneas para ser transformadas en una linea de productos de software. Estas aplicaciones que contienen funcionalidad similar son miembros de un mismo dominio. A partir del análisis de este dominio se puede empezar a diseñar una línea de productos de software. Las líneas de productos de software, gracias a su reuso, nos permiten desarrollar software más rápido y con mucha mayor calidad. Uno de los factores que perjudica y limita la adopción de la ingeniería de las lineas de productos de software es la dificultad de utilizar esta cuando se desea trabajar con ambientes heterogéneos. Esta no es una limitación pequeña pues un gran número de aplicaciones y sistemas se desarrollan en ambientes heterogéneos. En esta tesis analizaremos, diseñaremos y evaluaremos un prototipo de una herramienta para mecanismos de variación para líneas de productos de software en ambientes heterogéneos llamada *Pynion*. *Pynion* busca aminorar los efectos negativos de la heterogeneidad sobre el desarrollo de las líneas de productos de software.

1.1. Ambientes Heterogéneos

"Las diferencias honestas son a menudo una señal saludable de progreso"

Mahatma Ghandi

Los sistemas de software modernos son heterogéneos; regularmente estos están conformados por subsistemas construidos sobre muchas plataformas diferentes. La heterogeneidad, es causada por muchas y diversas razones: restricciones físicas (tamaño, costo, duración de batería, capacidad), condiciones del mercado (ciertos componentes solo están disponibles para ciertas plataformas), habilidades de los desarrolladores disponibles (conocimientos en Java, Perl, C, etc.) o históricas (sistemas legados)[2]. Usualmente, la heterogeneidad se ve como una molestia que debe ser corregida. Si bien es cierto que esto es verdad en algunos casos, la heterogeneidad puede ser vista como una ventaja y no una molestia[3]. La heterogeneidad se debe, entre otras razones, a que diferentes herramientas pueden funcionar mejor para diferentes problemas y situaciones. Un ambiente heterogéneo, como vemos, puede ser causado por el hecho de haberse escogido las herramientas adecuadas para realizar un trabajo más rápido, de modo más eficiente, más seguro o una combinación de estas virtudes. Se debe reconocer entonces que si bien los efectos secundarios de la heterogeneidad (como pueden ser más mantenimiento y mayor necesidad de entrenamiento) no son deseables es importante notar que muchas veces la heterogeneidad si es deseable en un sistema de software.

Antes y por sobre todo, los ambientes de producción son heterogóeneos[4]. En los ambientes distribuidos este heterogeneidad se puede observar en 3 aspectos: hardware, red y software. La *heterogeneidad en hardware* se refiere a diferentes equipos de computo como son los servidores, las computadoras de escritorio y los dispositivos móviles entre otros. Esta heterogeinidad es común pues de la mayoría de las aplicaciones distribuidas se espera que funcionen bien en diferentes configuraciones de hardware disponible. La *heterogeneidad en red* se refiere a que las interconexiones entre sistemas no se conforman a una sola arquitectura (Ethernet, red inalámbrica), protocolo (Http, Ftp y otros) o tecnología (equipos de distin-

tas marcas). La *heterogeneidad en software* se refiere a los diferentes sistemas operativos y aplicaciones que se ejecutan sobre estos[5]. La heterogeneidad de software es el tipo de hete¬ rogeneidad que se mantuvo como principal objetivo para *Pynion* en su búsqueda de facilitar el desarrollo de líneas de productos de software.

La *heterogeneidad del software* se trata de aquella que esta presente en las aplicaciones en toda la pila de software incluyendo el sistema de operativo, el middleware(como servicios web y otros) y las diversas aplicaciones finales del usuario. Esta heterogeneidad es más im¬ portante cuando se considera que cada uno de estos componentes probablemente provenga de proovedores diferentes[4]. Existen diversas razones por las cuales una aplicación termine siendo heterogénea. Para algunas aplicaciones es requerimiento que trabajen de manera tras¬ parente sobre distintos sistemas operativos. Además de esto, una aplicación puede llegar a ser desarrollada en varios lenguajes de programación por diversas razones; una de ellas puede ser cuando una aplicación tiene que trabajar a bajo nivel o, por ejemplo, la aplicación esta divida en cliente y servidor y cada uno esta desarrollado en su propio lenguaje. Incluso las aplicaciones más sencillas en muchas ocasiones deben De almacenar datos; esto puede involu¬ crar herramientas como bases de datos o formatos de archivos como CSV, XML y otros. Otra razón es que muchas aplicaciones también persisten sus configuraciones en archivos en dife¬ rentes formatos que pueden ser desde XML, ini hasta incluso un formato propietario. Algunas aplicaciones se tienen que comunicar con otros servidores por medio de servicios web (SOAP o REST ambos creados sobre XML) o formatos propietarios y tal vez requiera de transfor¬ maciones de XSLT para lograr comunicarse con servidores que operan con otros esquemas. Las aplicaciones web usualmente contienen archivos con código en HTML, CSS, Javascript diseñados para trabajar del lado del cliente *(navegador web*). También pueden incluir conteni¬ do embebido en Flash (construido por medio de ActionScript o Flex) o Silverlight (escrito en el lenguaje Javascript o Python). Unido a todo esto existen diversas tecnologí**a**s del lado del servidor de diferentes provedores. Todo esto hace que la heterogeneidad de las aplicaciones aumente considerablemente.

Las proyectos de software modernos también suelen tener diferentes vistas o representa¬ ciones. Batory y otros en [6] mencionan que de la mayoría de las aplicaciones tienen varias representaciones diferentes y no sólo el código fuente las representa. Una aplicación puede tener otras vistas como pueden ser la configuración, el diseño (UML por ejemplo), la documentación y el proceso de compilación. Además los diferentes módulos de los que se componen una aplicación de- be tomar en cuenta todas las diversas representaciones que existe para una misma aplicación. Por esta razón es necesario que cualquier mecanismo de variación pueda ser utilizado de modo similar en las diferentes representaciones que existan de una línea de productos de sof- tware. Esta *heterogeneidad de software* debido a diferentes representaciones se debe tomar en cuenta a la hora de diseñar una línea de productos de software. Se puede ver que la *heterogeneidad del software,* para beneficio o detrimento de los desarrolladores de software, se ha vuelto una parte inevitable para la mayoría de las aplicaciones, sistemas y líneas de productos de software modernos.

1.2. Motivación

"El precio de la libertad es la eterna vigilancia"

Thomas Jefferson

Para esta tesis se ha identificado una dominio donde se pueda probar el correcto funcio¬namiento y factibilidad de *Pynion:* la vigilancia remota. El primer encuentro con aplicaciones en este dominio fue a petición de un pequeño proyecto para implementar un prototipo de una aplicación para el monitoreo de un autobús escolar. Se tomó el control del proyecto cuando el diseño original que el cliente deseaba no fue posible debido a que se había tomado como base una aplicación para vigilancia en industrias; en donde las cámaras se conectaban directamente a la red local y hacían la función de servidores web. Estos cámaras/servidores eran después accedidas por personal de seguridad de manera directa desde sus estaciones de trabajo. Este diseño de arquitectura no fue el adecuado para este prototipo debido a diversos factores que a continuación enumeramos. Primero, la conexión del autobús era intermitente asó que no se podía garantizar la disponibilidad permanente. Segundo, el ancho de banda era muy limitado y no permitía que múltiples usuarios pudiese revisar las mismas cámaras al mismo tiempo; lo cual era un requerimiento explícito del cliente. Otro problema fue el tipo de conexión, al ser inalámbrica, no permitía que fuera posible tener una dirección IP fija en los autobuses, requerimiento necesario para el esquema seleccionado. Finalmente el *firewall* (cortafuego en español) del ISP (Proveedor de servicios de internet, por sus siglas en ingles) hacia imposible que el equipo de computo montado en el autobús pudiera aceptar llamadas externas desde *internet.* Todos estos problemas fue lo que llevo a que el equipo buscara replantear la dirección del proyecto y además buscar una arquitectura que se adaptará mejor al contexto del ambiente de trabajo.

El desarrollo inicial de este proyecto y las modificaciones subsecuentes pedidas por el cliente fueron lo que hizo evidente que se trataba de un dominio de aplicaciones. El diseño de la aplicación fue cambiado para adaptarse mejor a las limitantes propias de trabajar en unidades móviles a través de una conexión inalámbrica. Primero, se replanteo la arquitectura de la aplicación para que las computadoras en los autobuses funcionaran como clientes de un servidor central que funciona como intermediario a diferencia del modelo original en donde las cámaras eran los servidores directos. Las cámaras después envían periódicamente las imágenes tomadas de la *webCam* conectadas a su puerto USB al servidor central. Este servidor después se encargaba de darle acceso a los usuarios a las imágenes de cada cámara. Sin embargo, una vez realizado el prototipo de la aplicación se decidió que la calidad de imagen provista por las cámaras del tipo *webCam* disponibles no era lo suficientemente buena para las necesidades del proyecto. Tomado esto en cuenta, se reestructuro el diseño y la implementación para utilizar *cámaras IP* en lugar de las *webCam* que se habían utilizado en el diseño inicial originalmente. Al hacer los cambios requeridos fue evidente que esta transformación del diseño de la aplicación implicaba en realidad crear otro miembro de la misma línea de productos de software. Incluso al ser sólo un prototipo su implementación contaba ya con código en varios lenguajes de programación: Python y archivos de configuración .ini en la computadora conectada a las cámaras y C#, Html y Javascript en la aplicación web en el servidor; esto crea un ambiente de programación heterogéneo y no propicio a las herramientas para los *mecanismos de variación* tradicionales que se utilizan en las líneas de productos de software que están ligados a ambientes homogéneos (más información sobre los diferentes mecanismos de variación en la subsección 2.2).

La aplicación original y sus subsecuentes modificaciones forman parte de un solo dominio de aplicaciones: la vigilancia remota por vídeo. Todas las aplicaciones comparten el mismo tema central, la trasmisión de imágenes unidireccional de un sitio remoto hacia uno o varios clientes ubicados en redes diferentes que la fuente de las imágenes. El uso principal actual para la vigilancia remota es la seguridad en donde su uso ya es extendido en bancos, tien-

das y estacionamientos de oficinas y tiendas departamentales[7]. Otro uso importante de las cámaras, y que en efecto es el uso que se le dio para el proyecto ya descrito, es el monitoreo remoto y vigilancia de personas como puede ser en escuelas, hospitales y asilos de ancianos. Además de estos usos, existen otros menores pero igualmente importantes como pueden ser monitoreo de trafico y monitoreo para encontrar estadísticas de flujo de personas en tiendas y parques de diversiones[8]. También existen usos científicos para la vigilancia remota como puede ser el monitoreo de volcanes activos o para monitorear, vigilar y calcular estadísticas acerca de especies de animales en vías de extinción. Como vemos existen muchas aplicaciones de software que son usadas para la vigilancia remota; pero al analizarlas con detenimiento podemos notar que nos estamos refiriendo a aplicaciones que son miembros de una misma línea de productos de software.

1.3. Solución Propuesta

El propósito de esta tesis es crear un prototipo de herramienta para mecanismos de variación para facilitar el desarrollo de líneas de productos en ambientes de software heterogéneos.
Para probar que esta herramienta es factible y beneficiosa, se utilizará un prototipo de esta para crear una línea de productos de software en el dominio de la vigilancia remota. Para lograr esto se identificara la funcionalidad común de la línea de productos de software para la vigilancia remota y la funcionalidad variable de esta. Una vez creada la línea de productos de software, se analizaran el proceso de desarrollo y los resultados para después poder evaluar las cualidades y los defectos que tienen esta herramienta.

Los mecanismos de variación utilizados para desarrollar líneas de productos de software cuentan con 3 componentes importantes: la selección de los rasgos *(features* en inglés), la validación de la selección y la composición de los rasgos. El esfuerzo principal de *Pynion* se centra en la tercera etapa: la composición de los rasgos. En este etapa se adoptará una estrategia doble para permitir el uso de diferentes lenguajes de programación o de almacena¬ miento de datos en la misma línea de productos de software. La propuesta es permitir dos niveles diferentes de modificaciones al código fuente y los demás archivos que componen una aplicación. El primer nivel de modificación es permitir los cambios a nivel archivos, esto se refiere a poder agregar, eliminar o sustituir por completo archivos en la aplicación de salida. Esto permite que los archivos fuente puedan ser construidos sin necesidad de modificación o adaptación alguna para la línea de productos de software permitiendo así el uso natural de las herramientas de desarrollo ya disponibles para el desarrollador de la línea de productos de software. La segunda es la identificación y marcación de puntos de variación dentro de un archivo. La marcación de los archivos se hará por medio del soporte para comentarios de cada lenguaje, permitiendo así que los marcadores de sintaxis de los diferentes editores de texto e IDEs sigan operando sin problemas en el archivo fuente. En estos puntos de variación se podrán realizar las variaciones de dos modos: ejecutando funciones de Python y sustituyendo su resultado en el punto de variación o utilizando *plantillas* para realizar la substitución de código en el punto de variación. El propósito de *Pynion* es permitir el uso de diversos lengua¬ jes de programación dentro de una misma línea de productos software de la manera menos invasiva sobre el código fuente original, sin perder la flexibilidad necesaria para el desarrollo de las líneas de productos de software.

Para evaluar la solución se buscará una serie de características deseables. Estas características deben hacer que el desarrollo de líneas de productos de software en ambientes heterogéneos en primera instancia sea factible, pero ademas es necesario que este sea relati-

vamente sencillo. Esta facilidad de desarrollo es necesario si se quiere que haya una mayor adopción de las estrategia de las líneas de productos de software para proyectos en ambien¬ tes heterogéneos. Si estas características estan presentes, el prototipo sera considerado; si no estan presentes el prototipo no habra cumplido con su objetivo.

Las características esenciales en el prototipo son las siguientes:

1. *Soporte para ambientes heterogéneos.* El soporte para ambientes heterogéneos es la base para *Pynion. Pynion* debe soportar el desarrollo en varios lenguajes de programación y en varios formatos de almacenamiento o transferencia de datos.

2. *Poca invasividad.* La poca invasividad permite reutilizar los conocimientos ya adquiridos en la tecnologías pues se exige pocas modificaciones sobre los recursos ya disponibles. Esta propiedad es una propiedad subjetiva; sin embargo, la facilidad de desarrollo puede ser una evidencia indirecta de la poca invasividad de la herramienta.

3. *Soporte para modificaciones de diferentes granularidades.* El soporte para modificaciones en diversas granularidades se referiere a la capacidad de la herramienta para hacer agregar, quitar o modificar desde componentes completos hasta una sóla linea de código. El permitir diferentes granularidades nos permite lograr un mayor reuso de los recursos disponibles.

4. *Soporte para diferentes estrategias de desarrollo de software.* El soporte de las diferentes estrategías de desarrollo de líneas de productos de software permite que *Pynion* se pueda utilizar en diversos proyectos. Dado que *Pynion* sólo será probado en un sólo proyecto (el dominio de la vigilancia remota) buscaremos en este proyecto pistas que nos indique el potencial de esta herramienta.

5. *Soporte para diferentes tipos de variabilidad.* El soporte de la variabilidad es importante para implementar las diferentes estrategias de desarrollo de software. Además el sopor¬ te para los diferentes tipos de variabilidad es un buen indicio de la flexibilidad de la herramienta para mecanismos de variación.

Ademas de estas características existen otras características deseables en el prototipo como son que el código generado sea fácil de leer y sea rastreable. Estas propiedades son importantes para permitir el desarrollo de las líneas de productos en ambientes colaborativos y empresariales. El conjunto de estas características nos permitirá evaluar los resultados de *Pynion.*

1.4. Estructura de la Tesis

En el capítulo 2 de esta tesis documenta el marco teórico que fundamenta su desarrollo. En ese capítulo, primero se discutirán los términos y conceptos claves que se utilizan en el desarrollo de las líneas de productos de software. En esta sección nos concentraremos en los términos utilizados para describir una línea de productos de software y los conceptos que describen sus diferentes fases de desarrollo. Después de presentarse estos conceptos, se analizarán las herramientas actuales que pueden funcionar como mecanismos de variación en las líneas de productos, discutiendo las virtudes y defectos que tienen estas.

En el capítulo 3 se discutirá a fondo el diseño y la implementación de *Pynion,* comen¬ zando por el diseño general. Después se revisará las decisiones tomadas durante el diseño y

se analizará este diseño más a fondo. Tercero, se describirán las partes más distintivas hasta describir detalles de como fue implementado. También se identificaran y describirán las herramientas e ideas y los autores de estas que fueron tomadas para el desarrollo de *Pynion*. Finalmente, se incluirá un breve manual de usuario describiendo la funcionalidad completa de *Pynion*. Este manual incluirá breves ejemplos de su uso de la herramienta explicando cuando debe ser usada cada estrategia.

En el capítulo 4 se utilizara *Pynion* para implementar una línea de productos de software en el dominio de la vigilancia remota. Primero se definirá el dominio y se realizara el análisis y modelado de rasgos para la línea de productos de software. Después se implementaran el ambiente de desarrollo creando los recursos base de la ingeniería de dominio para la ingeniería de aplicación. Finalmente, se instanciaran varios miembros de la familia de productos utilizando los recursos creados anteriormente. Todo este proceso se hará bien documentado mostrando, especialmente, las decisiones importantes tomadas y el código fuente que valga la pena analizar a fondo para probar las hipotesis de esta tesis. Además esta sección identifica el tipo de variabilidad encontrada en la línea de productos y como ayudo *Pynion* a representarla. Finalmente, esta sección incluye los patrones de diseño encontrados durante el desarrollo y las mejores practicas de usar *Pynion*.

En el capítulo 5 se presentaran los resultados de la línea de productos de software para evaluar el desempeño de *Pynion* como herramienta para mecanismos de variación para ambientes heterogéneos. Finalmente en el capítulo 6 se analizará la tesis completa. En esta sección se revisará los resultados de *Pynion* como herramienta para mecanismos de variación y se evaluará sus resultados. En este capítulo se hará especial énfasis en como *Pynion* facilite o dificulte el desarrollo de lineas de productos en ambientes heterogéneos de software. También en esta sección se propondrán futuros desarrollos que podrán hacer de *Pynion* una mejor herramienta para el desarrollo de líneas de productos de software para ambientes heterogéneos.

2. Marco Teórico

Existen varios temas que se deben conocer para poder entender por completo el desarrollo de esta tesis. En esta sección se explicaran estos conceptos a fondo sentando las bases de la tesis. Ademas se justificara, en la medida en que lo permita los limites de espacio, su definición y se dará a conocer el origen de estos conceptos. El objetivo de este capítulo es que incluso las personas no familiarizadas con el desarrollo de líneas de productos de software puedan comprender el desarrollo de este documento en toda su extensión.

En la subsección 2.1 daremos una referencia mínima de los conceptos claves de las líneas de productos de software. En la subsección 2.2 veremos los mecanismos de variación que actualmente se utilizan para desarrollar las líneas de productos de software. En resumen esta sección nos servirá para establecer los conocimientos necesarios para poder entender a fondo esta tesis y las decisiones tomadas durante el desarrollo de este proyecto.

2.1. Conceptos Clave

Antes de poder entender a fondo las líneas de productos de software y el contenido de esta tesis es necesario entender los conceptos claves que rodean el desarrollo de estas técnicas. El objetivo central de las líneas de productos de software es el manejo de las similitudes y la variabilidad entre diferentes programas y sistemas de software similares. En las próximas subsecciones se analizan y describirán a detalle los diferentes conceptos que son parte del dominio de las líneas de productos de software.

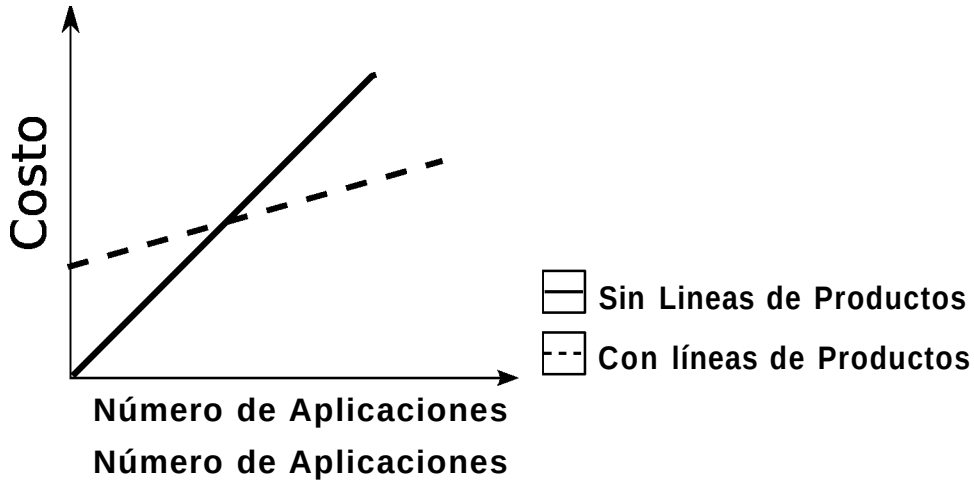
2.1.1. Linea de Productos de Software

"Nada es particularmente difícil si lo divides en pequeñas tareas"

Henry Ford

Las líneas de productos de software es una nueva rama de la ingeniería de software que enfatiza el reuso. Las líneas de productos de software también son conocidas por otros nombres como *familias de productos, familia de sistemas, aplicaciones de dominio* y fábricas de software. Las líneas de productos de software son una familia de sistemas de software que tiene una funcionalidad común y otra funcionalidad variable. En una línea de productos de software la funcionalidad común debe ser significativamente mayor que una funcionalidad variable. Para tomar ventaja de la funcionalidad en común (como pudieran ser los requerimientos, los diseños, los componentes, la documentación y otros), los recursos comunes son desarrollados para luego ser utilizados en varios miembros de la línea de productos de software [1]. Una de las definiciones más conocidas para la líneas de producto de software es la de Clements y Northrop en [9] que define a una línea de productos de software como: "Un juego de sistemas intensivos de software que comparten un juego de rasgos manejados en común que satisfacen las necesidades específicas de un mercado particular o misión y son desarrollados a partir de un juego de común de activos base de modo sistemático". El proposito esencial es aumentar el reuso de recursos que siempre se busca en el desarrollo de software para aumentar la reutilización hasta llegar a ser a nivel arquitectura.

Tradicionalmente las aplicaciones de software se han desarrollado como sistemas indivi¬ duales, esto es cada aplicación se ha desarrollado como si fuera única [1]. Asó los ingenieros de software se encuentran constantemente ante un dilema. Por un lado se pide que se cree



Número de Aplicaciones

Figura 1: Diagrama de recuperación de inversión

un software que atraiga a los clientes con su funcionalidad, facilidad de uso y confiabilidad, que a la vez sea fácil de mejorar y evolucionar conforme las necesidades del cliente y los cambios tecnológicos. Cubrir estos requerimientos requiere de una ingeniería cuidadosa. Por el otro lado los ingenieros de software se encuentran presionados para producir software que pueda ser lanzado al mercado antes que la competencia. Este requerimiento requiere rápido desarrollo del sistema. La solución para este dilema son las líneas de productos de software. La reutilización permite utilizar código ya probado mejorando la calidad. Además la reutilización significa que menos código se tiene que producir por cada aplicación lo cual hace que el desarrollo sea considerablemente más rápido[10].

En la industria del software las líneas de productos son un concepto nuevo, sin embargo en otras industrias este es un concepto viejo y probado. Para Gomaa en [1] el concepto es tan antiguo que incluso las pirámides egipcias se pueden llegar a considerar como un primer ejemplo de las líneas de productos. Un ejemplo más actual proviene de las industria aeronáutica con los aviones A-318, A-319, A-320 y A-321 de European Airbus que comparten rasgos comunes incluyendo motores de jet, equipo de navegación y equipo de comunicación [9]. Este proceso de reutilizar la misma línea de ensamblaje para construir diferentes productos se le llama *personalización en masa.* La *personalización en masa* es la producción a gran escala de productos hechos a la medida [11].

La creación de las líneas de productos tiene un costo de inversión inicial fuerte. Este costo es debido a la necesidad de crear los recursos bases que se van a reutilizar dentro de la línea de productos de software. Sin embargo a medida que se reutilizan los recursos y se va aplicando la experiencia de haber creado otros miembros del mismo dominio de aplicaciones los ahorros de utilizar la estrategia de línea de productos de software se van incrementando y eventualmente son mayores que los gastos debidos por la inversión inicial. Esto es regularmente representado por la figura 1 que esta basada en una figura similar de Weiss en [10]. Weiss establece en [10] que este punto de recuperación de inversión se encuentra ligeramente por arriba de la creación de la tercera aplicación cuando se compara la creación de aplicaciones usando líneas de productos de software con crear las aplicaciones de manera tradicional. Asó vemos que minimizar el costo por producto, al maximizar el reuso de recursos es la clave de las ventajas del desarrollo de aplicaciones y sistemas por medio de líneas de producto de software.

Tradicionalmente, las estrategias clasicas de las líneas de productos han sido con el *costo al inicio* esto quiere decir se invierte en la creación inicial de los recursos compartidos. Sin embargo, esta no es la única estrategia que se puede seguir. Otra visión es considerar la línea de productos de software como una inversión acumulativa en que se van tomando los recursos reutilizables a medida que se van creando las diversas aplicaciones. Estas diferentes visiones producen diferentes estrategias de líneas de productos.

2.1.2. Clasificación de las estrategias de líneas de productos de software

> "Siempre dicen que el tiempo cambia las cosas pero la verdad es que las tienes que cambiar tu mismo"
>
> Andy Warhol

Al igual que el la ingeniería de software en donde se han creado estrategias dinámicas como las técnicas á*giles de desarrollo de software.* En la ingeniería de líneas de productos de software también se han creado nuevas técnicas mas ágiles para el desarrollo. La intención de estas técnicas no es sustituir las estrategias tradicionales sino dar caminos nuevos para la adopción de las líneas de productos de software a empresas y equipos que no lo han realizado. Las estrategias de líneas de productos de software pueden ser clasificadas en dos categorías generales: las *heavyweight* (o pesadas) y las *lightweight* (o ligeras)[12]. Las estrategias *heavyweight* son las tradicionales con una inversión inicial fuerte mientras que las herramientas ágiles corresponden a las *lightweight.* En cuanto al beneficio económico McGregor et al. en [12] nos muestra una diferencia clara. Las estrategias *heavyweight* tardan más en llegar al punto de retorno de inversión sin embargo, a medida que crece el número de aplicaciones estas resultan mas productivas que las estrategias *lightweight.* Ademas de las diferencias económicas cada una de las estrategias ofrecen diferentes beneficios y son mejores para diferentes tipos de proyectos.

Las estrategias de *adopción* de líneas de productos de software también se dividen en tres categorías: *proactivas, reactivas* y *extractivas* [13]. Las proactivas se refieren a la estrategias en que todos los costos se pagan al inicio, la planeación, análisis se realizan antes de que se comience a codificar para el proyecto. Las reactivas son similares a la programación extrema *(eXtreme Programming,* en inglés) y se trata de armar una espiral de desarrollo que se repite constantemente. Finalmente, las extractivas se refieren a extraer los recursos para la línea de productos de software a partir de uno o varios productos ya existentes. El tipo de adopción recomendada dependen de las experiencia de los desarrolladores en el dominio y de las aplicaciones en el dominio ya existente.

Para Paul Clements en [14], las virtudes de trabajar de manera proactiva beneficia eventualmente a las empresas pues les permite identificar de mejor manera que aplicaciones pertenecen a una línea de productos de software y cuales no; Este hecho a la larga trae consigo una ventaja competitiva para la empresa que las utiliza. En cambio, Krueger piensa que no se debe descartar las opciones reactivas y extractivas para las empresas para las cuales el costo de la inversión inicial del método proactivo sea excesivo y demasiado riesgoso[13].

2.1.3. El proceso de la ingeniería de líneas de productos de software

> "El primer problema del ingeniero en cualquier situación de diseño es descubrir cual es el verdadero problema"

El proceso de la ingeniería de líneas de productos esta dividido en dos procesos: la *ingeniería del dominio* y la *ingeniería de la aplicación* [10]. Estos procesos son iterativos y se deben pulir constantemente. Como explica Weiss en [10] estos procesos no son enteramente independientes el uno y del otro y requieren de la retroalimentación constante entre ellos. Los ingenieros de aplicación necesitan conocer a fondo la plataforma creada por los ingenieros del dominio para poder crear las diferentes aplicaciones de la línea de productos de software de manera independiente. Mientras tanto los ingenieros de dominio requieren la retroalimentación de los ingenieros de aplicación para mejorar la plataforma de tal modo que se pueda sacar mayor provecho de estas con el menor costo económico y en horas hombre. Tanto la *ingeniería de dominio* como la *ingeniería de aplicación* son partes esenciales de cualquier proyecto de líneas de productos de software.

Pohl et al.[11] definen la *ingeniería de dominio* como el proceso de crear la plataforma reutilizable y al hacerlo se define las partes en común y las variables de la línea de productos de software. El primer paso de la ingeniería de dominio es el análisis de las partes en común. Esta análisis es vital pues en el se define si el proyecto de la línea de productos es viable [10]. El análisis debe permitir saber si la línea sera aceptada en el mercado y busca a la vez sacar el mayor provecho de las partes en común y la variabilidad de la línea de productos de software. Ademas de esto, la línea de productos debe establecer la plataforma de trabajo para la ingeniería de aplicación [11]. Es vital que esta plataforma evolucione iterativamente al mejorar el conocimiento del dominio y cuando sea necesario debe poder ampliarlo.

Los pasos esenciales para crear los recursos base o dominio son:

1. Determinar cuales recursos base son comunes para todas las aplicaciones y cuales son los variables.

2. Escoger un mecanismo de variación que permita aplicar las variaciones sin modificar la comunalidad de la línea de productos

3. Proveer las instrucciones para crear instancias de aplicaciones basada en los recursos base [15]

El desarrollo en la ingeniería de dominio debe ser meticuloso ya que en la correcta selección de las partes en común, las partes variables y el mecanismo de variación que se va a utilizar es donde se obtienen los mayores beneficios económicos.

La *ingeniería de aplicación* se define, de acuerdo a Pohl et al.[11], como el proceso de derivar aplicaciones específicas de la plataforma establecida en la línea de productos. La *ingeniería de aplicación* trabaja de manera más cercana con el cliente. Su propósito es indagar en los requerimientos de cliente para desarrollar una aplicación para él. Se espera que el ingeniero de aplicaciones utilice la plataforma de dominio de modo de poder pasar de los requerimientos del cliente a la implementación de manera más sencilla. Un ingeniero de aplicaciones debe de lidiar primordialmente con el análisis y la especificación de la aplicación, pues la plataforma se debe hacer cargo de la mayor parte del desarrollo y la implementación de la misma. El proceso de ingeniería de aplicación también se debe hacer de manera iterativa para lograr refinar la aplicación que se entrega al cliente [10]. El proceso de la *ingeniería de aplicación* trabaja con los requerimientos de cada cliente individualmente.

La introducción de las líneas de productos de software ha traído grandes beneficios a las empresas que las implementan, sin embargo el método tradicional aquí descrito se ha

enfrentado con algunas barreras de adopción. Recientemente nuevos enfoques han surgido para remontar estas barreras aunque la idea básica detrás de las líneas de producto de software sigue siendo las mismas. Esta nueva generación de metodologías, modelos y herramientas han empezado a sustituir las herramientas que se diseñaron por primera vez hace más de 15 años. Estas nuevas herramientas han tratado de suprimir el dominio de aplicación (el proceso de ensamblar una aplicación debe ser mínimo y de preferencia automatizado), ser mínimamente invasivas (uno de los temas de esta tesis) y sus combinaciones validas deben ser limitadas (las composiciones de configuración crecen exponencialmente y nunca se requieren tantos miembros de la línea de productos de software)[16]. Con lo anterior se busca que la transición a las metodologías de líneas de productos de software sean más sencillas y más similares al desarrollo de los productos tradicionales a las que están acostumbrados los desarrolladores de software.

2.1.4. Variabilidad

> "La turbulencia es una fuerza de vida. Es una oportunidad. Amemos la turbulencia y usemos para el cambio"
>
> Ramsay Clark

La variabilidad es parte esencial en el desarrollo de las líneas de productos de software. La variabilidad en un sistema se presenta a nivel de requerimientos, arquitectura, pruebas, documentación y componentes[9]. La variabilidad es la habilidad de un sistema, recurso o ambiente de desarrollo para soportar el desarrollo de aplicaciones que difieren de una a otra de un modo planeado. Esta variabilidad es la que permite adaptar el producto a las necesidades del cliente[15]. La variabilidad debe ser identificada en el proceso de la ingeniera de dominio y debe ser explotada en el proceso de aplicación [9]. El objetivo de la variación en una línea de productos de software es maximizar el retorno de la inversión *(ROI* por sus siglas en inglés) de construir o mantener productos por un periodo de tiempo o un número finito de productos[15]. Esto quiere decir que una variación sólo debe ser agregada a la línea de productos cuando esta tenga un sentido económico. La variabilidad en la línea de productos de software puede ser de dos tipos: *variabilidad del tiempo* que se refiere a la evolución normal con el tiempo de cualquier proyecto de software y la *variabilidad de espacio* que es el énfasis principal de las líneas de productos de software donde existen varias opciones para ejecutar la misma función en lugares predeterminados [14]. La variabilidad también puede ser clasificada como esencial o local. La variabilidad esencial se refiere a la que es debida a las decisiones basadas en los requerimientos del cliente. Mientras que la variabilidad local se refiere a cualquier otro tipo de variabilidad diferente a la esencial[15].

Existen varios términos que encapsulan los diferentes conceptos que describen de variación en una línea de productos de software. A continuación definiremos cada uno de estos conceptos:

- *Comunalidad.* La comunalidad o parte en común se refiere a la parte de las aplicaciones que esta presente de todas la aplicaciones de la línea de productos de software y que siempre se aplica de la misma manera[9].
- *Variabilidad.* La variabilidad o partes variables se refiere a todas los funciones o recursos que no pertenecen a todas las aplicaciones de la línea de productos de software [9].

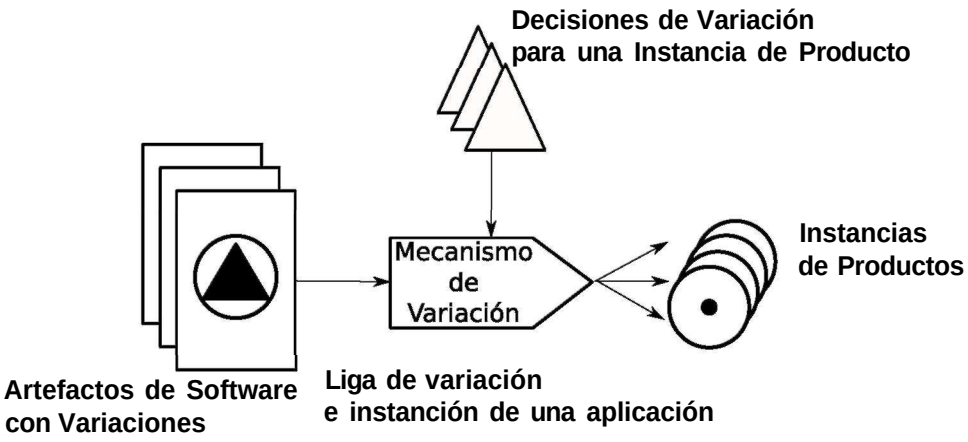


Figura 2: Diagrama de ligado de variación

- *Punto de Variación.* Un punto de variación una representación de la variación real en la aplicación. Representa un sólo punto en donde se puede realizar un cambio[9]. Los puntos de variación se refieren exclusivamente a cambios visibles de funcionalidad en el sistema de manera externa y no a cambios internos en las aplicaciones [15].

- *Mecanismo de Variación.* Los mecanismos de variación son mecanismos utilizados para producir variantes de un recurso base de manera controlada [15]. La figura 2 nos muestra un diagrama de un mecanismo de variación que liga las diferentes variantes a un punto de variación para crear diferentes aplicaciones en una línea de productos de software. Esta figura esta basada en una similar de Krueger que aparece en [17]. Analizaremos más a fondo este concepto y las diferentes opciones existentes en la subsección 2.2.

- *Variante.* Una variante es la aplicación de una opción específica en un punto de variación por medio de un mecanismo de variacióon[9].

2.1.5. Análisis de Rasgos

> "Ninguna mujer puede ser hermosa solo por la fuerza de sus rasgos, ni lista solo con la ayuda del habla"
>
> Langston Hughes

Como fue mencionado en las secciones anteriores, el análisis de la variabilidad del dominio es parte esencial del desarrollo del diseño de una línea de productos de software. Principalmente se trata de identificar los diferentes rasgos que describen el dominio. En las líneas de productos de software, un *rasgo* es un requerimiento o característica que es provisto por uno o más miembros de la línea de productos de software[1]. Los *rasgos* deben ser funcionalmente diferenciables para los usuarios finales [18]. Estos rasgos son los que identifican a la línea de productos de software y también son los que diferencian a los diferentes miembros de la línea de productos de software. Los *rasgos* que son comunes a todos los miembros identifican su comunalidad, mientras que los *rasgos* que difieren entre cada uno de ellos identifican su variabilidad. El hecho de que los rasgos describen tanto la variabilidad como la comunidad

de una línea de productos de software es la razón por la cuál es un concepto importante en el desarrollo de estas.

El análisis de requerimientos en las líneas de productos de software es mías complicado que el análisis para un sistema tradicional. En una aplicación individual, un sistema no esta completo hasta que se implementan todos los requerimientos; esto es, el sistema provee de todos los características definidas en el análisis. En un sistema tradicional estos rasgos pueden ser prioretizados para poder hacer las entregas de la funcionalidad en varias etapas. Para las líneas de productos, se debe ir más altó e identificar las características de variabilidad de los rasgos[1]. El análisis de requerimiento de líneas de productos requiere, en realidad, considerar varias aplicaciones del mismo dominio a la vez, haciéndolo más difícil que el análisis tradicional que sólo analiza una aplicación a la vez.

En el proceso del análisis de rasgos debemos de identificar la categoría en la comunalidad o variabilidad a la que pertenece cada rasgos. Gomaa en [1] clasifica los rasgos en los siguientes tipos:

- *Rasgos Comunes.* Los rasgos en común son aquellos que son contenidos en todos los miembros de la línea de productos de software. La funcionalidad en común puede ser encapsulada en un solo rasgo o puede ser distribuido entre varios rasgos. Regularmente cuando existen varios rasgos en común es por que no se conoce el dominio lo bastante a fondo para garantizar que ninguno de ellos se vuelva opcional. En la literatura, los rasgos comunes pueden ser conocidos por otros nombres como *forzosos(mandatory* en ingles), necesarios(necessary en ingles) o núcleo *(kernel o core* en ingles).

- *Rasgos Opcionales.* Los rasgos opcionales son aquellos que solo aparecen en algunos miembros de la líneas de productos. Un rasgo opcional tradicionalmente cuenta con que los rasgos comunes se encuentren disponibles.

- *Rasgos Alternativos.* Los rasgos alternativos se refieren a una serie de rasgos en que dos o mías pueden ofrecer la misma funcionalidad. Estos rasgos son mutuamente exclusivos entre si. Los rasgos alternativos siempre aparecen agrupados con otros que ofrecen la misma funcionalidad. En estos grupos puede aparecer un rasgo por omisión (o por defecto) que este seleccionado si ningún rasgo esta seleccionado por defecto.

- *Rasgos Parametrizados.* Un rasgo parametrizado es aquel que puede ser modificado al tiempo de configuración con algún valor. Un rasgo parametrizado debe definir un tipo de valor, un rango de valores validos y opcionalmente un valor por omisión.

  Rasgos Prerrequisitos. Cuando una funcionalidad depende de otra funcionalidad, la fun¬ cionalidad de la que depende es una rasgo prerrequisito.

- *Rasgos de Inclusión Mutua.* Los rasgos de inclusión mutua se refieren cuando los rasgos deben aparecer unidos en un miembro de la línea de productos de software.

Gacek y Michalis Anastasopoulos en [19] incluyen inclusión mutua o exclusión mutua como dos tipos más de rasgos. Inclusión mutua se refieren a aquellos rasgos en que solo se puede incluir un rasgo en caso de que otro ya haya sido agregado. La exclusión mutua es similar a la alternativa pero en este caso los rasgos no proveen funcionalidad equivalente. Sin embargo en otra literatura este tipo de rasgos es considerado por dependencias, condiciones y restricciones y no como una nueva clase completa [20] [18].

La introducción de rasgos en un código introduce consigo cierta variabilidad dentro del código. Esta variabilidad ha sido identificada y clasificada por Sharp en [21] en las siguientes categorías:

- *Positiva.* En la variabilidad positiva se agrega nueva funcionalidad al sistema.
- *Negativa.* En la variabilidad negativa se remueve funcionalidad del sistema.
- *Opcional.* En la variabilidad opcional se agrega código del sistema.

  Alternativa. En la variabilidad opcional se sustituye código del sistema.
- *Funcional.* En la variabilidad funcional se cambia la funcionalidad del sistema.
- *Plataforma o Ambiente.* En la variabilidad de plataforma o ambiente se adapta a una nueva plataforma o ambiente.

2.1.6. Modelado de Rasgos

"Los modelos son para ser usados, no creídos"

H Theil

Una vez que se ha identificado y clasificado la variabilidad es importante documentar lo aprendido en este proceso. Al igual que los demás requerimientos los rasgos se pueden docu¬ mentar de dos maneras: *textual* y *gráfica.* La *documentación textual* ofrece una expresividad ilimitada a costa de generar una mayor ambigüedad en la definición de los requerimientos. Los modelos o *documentación gráfica* tienen un juego más limitado de posibilidades pero definen de manera no ambigua los requerimientos[11]. La documentación explicita de los rasgos es de vital importancia en el desarrollo de una línea de productos de software pues permite el desarrollo ordenado de los componentes de la línea.

Pohl et al. en [11] notan cuales son las tres áreas donde realizar la documentación explicita trae los beneficios más importantes: la toma de decisiones, la comunicación y el rastreo. La toma de decisiones mejora pues uno documenta que decisiones y por que se han tomado; asó otros pueden tomar en cuenta estas decisiones para tomar las suyas propias. También mejora la comunicación acerca de la variabilidad en el proyecto al dar un punto de vista con una alta abstracción que todos pueden utilizar. Además ayuda al rastreo al permitir ligar los diferentes artefactos con los requerimientos que los provocaron. Esta es la razón por la cuál una buena estrategia de modelado de la variabilidad es vital para que un proyecto de línea de productos de software llegue a lograrse a tiempo, en presupuesto y de acuerdo con los requerimientos. El modelado de la variabilidad para líneas de productos de software a seguido tres vertientes: Diagramas de Rasgos (FD por sus siglas en inglés *[Feature Diagram*]), Modelo de Variación Ortogonal (OVM por sus siglas en inglés) y adaptaciones de UML *(Unified Modeling Language*).

Los Diagramas de Rasgos (FD) son una familia popular de lenguajes de modelado usa¬ dos para la ingeniería de requerimiento en líneas de productos de software [22]. La primera propuesta para este tipo de modelado fue la de Kang para FODA (*Feature Oriented Domain Analysis* o Análisis de Dominio Orientado a Rasgos) [18] utilizando la descomposición de la aplicación en sus rasgos funcionales. FODA se basa en la reutilización de abstracciones ba¬ sados en los conceptos de agregar/descomponer, generalizar/especializar y parametrizar. El

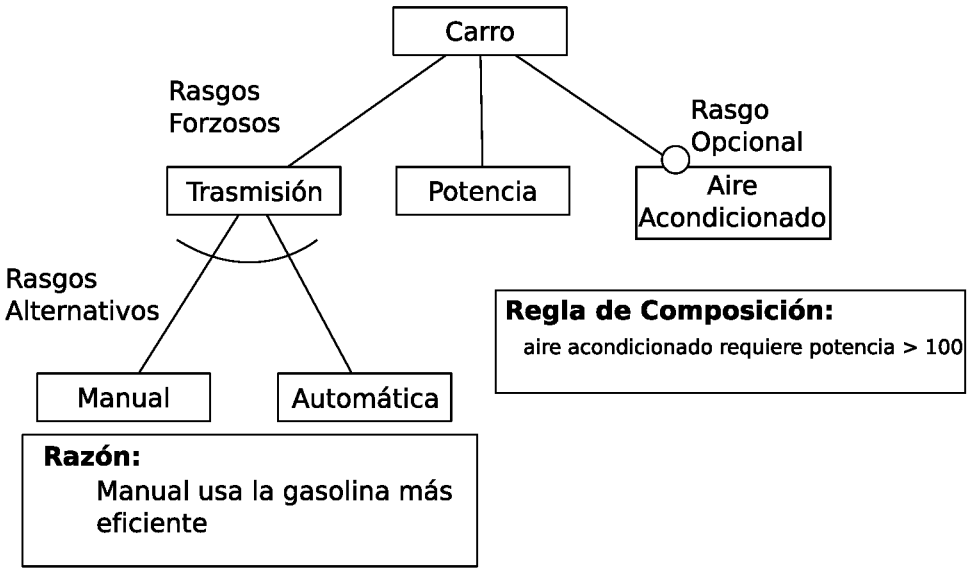


Figura 3: Diagrama de Rasgos de FODA

gran legado de FODA es la introducción del modelo de rasgos para diagramar una línea de productos de software. Kang en [18] identifica tres tipos de componentes: esenciales, alterna¬ tivos y opcionales. Los componentes esenciales son aquellos base a todas las aplicaciones del dominio. Ademas de sus componentes y jerarquías, Kang define en el diagrama más restric- ciones que debe cumplirse para que una selección sea valida. En la figura 3 podemos ver un diagrama similar al propuesto por FODA para una línea de productos de carros. Czarnecki en [20] para Programación Generativa amplia un poco el concepto permitiendo la selección múltiple cuando existen componentes alternativos. Los diagramas de rasgos son esenciales, no sólo por que permiten visualizar la estructura y jerarquía de los componentes si no que también nos permiten ver las restricciones que los rigen. En la figura 4 vemos un ejemplo del diagrama de rasgos de Czarnecki diseñado para una línea de productos de carros. Desde entonces ha habido diversas revisiones por diferentes autores para mejorar la expresividad y aminorar la ambigüedad de los diagramas de rasgos [22]. Los diagramas de rasgos han sido tan aceptados en la industria de la líneas de productos de software a tal grado que incluso existe un *plugin* para soportarlos para el ambiente de desarrollo integrado Eclipse con soporte para diferentes cardinalidades y parametrizaciones [23].

Los Modelos de variación ortogonal (OVM por sus siglas en inglés) son presentados por Pohl et al. en [11] como una manera de modelar la variabilidad en una línea de productos de software. Pohl et al. [11] definen los OVM como: "un modelo que define la variabilidad de una línea de productos de software. Relaciona la variabilidad con otros modelos de desarrollo como modelos de rasgos, modelos de caso de uso, modelo de diseño, modelos de componentes y modelos de pruebas." Los OVM buscan suplir algunas de las carencias que presenta los diagramas de rasgos: la ambigüedad de su definición, su falta de herramientas para agrupa- miento y falta de claridad. Para lograr estos objetivos los OVM tiene varias características particulares. Primero, los OVM distinguen entre una puntos de variación y variantes. Segun¬ do, los OVM tiene una notación más extensa que permite hacer una descripción más puntual de la variabilidad de la línea de productos de software. Finalmente, los OVM contiene un me-

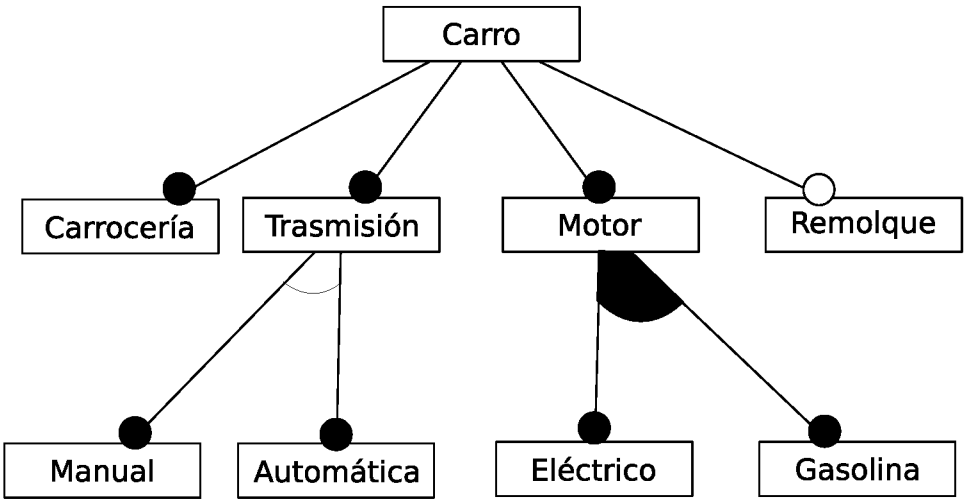


Figura 4: Diagrama de Rasgos de Programación Generativa

canismo para agrupar varias variantes y puntos de variación como si fueran uno sólo. Además de todo esto los OVM están diseñados para ligar esta variabilidad con otros modelos, en particular con los diagramas de rasgos, UML(Diagramas de Caso de Uso, de Secuencia, de Clases, de Flujo de Datos), maquina de estados y otros artefactos relacionados con el análisis y diseño de software [11]. Los OVM están intimamente relacionados con los diagramas de rasgos pero son más precisos en su capacidad de definición así ellos pueden ser usados para quitar ambigüedades de los diagramas de rasgos[24].

Gomaa ofrece a la Ingeniería de Software basada en UML para Líneas de Productos o PLUS, por sus siglas en inglés(Product Line UML-Based Software Engineering) en [1]. UML encapsula una diversidad de puntos de vista para diferentes fases del diseño de una aplicación. Gomaa propone aprovechar esta diversidad de diagramas y la capacidad de UML para ser extendidos para usar sus diagramas en el modelado de las líneas de productos. De manera general PLUS permite una modelación similar que cuando se diseñan aplicaciones individuales, pero extiende UML para modelar la comunalidad y variabilidad de las líneas de productos de software de manera explícita. El reuso de los conocimientos ya adquiridos por la mayoría de los diseñadores de software, además del reuso de herramientas ya existentes es la gran ventaja que ofrece la propuesta de Gomaa sobre otras propuestas.

El modelado de los rasgos es una parte importante del análisis y diseño de una línea de productos de software. Los diferentes modelos nos permiten visualizar y definir con claridad de la comunalidad y variabilidad de un dominio. Estos modelos necesitan ser bajados al mundo real para crear las herramientas para implementar esta variabilidad. Las herramientas que nos ayudan comúnmente para este proceso son los mecanismos de variación. Los mecanismos de variación nos permiten unir diferentes rasgos en una aplicación final que podamos entregar al cliente.

2.2. Mecanismos de Variación

> "El hecho es que la civilización requiere de esclavos. Los griegos estaban en lo cierto. A menos que haya esclavos para hacer los trabajos feos, horribles y poco

interesantes, la cultura y la contemplación se vuelve casi imposible. La esclavitud humana es mala, insegura y desmoralizante. Pero de la esclavitud mecánica, de la esclavitud de las máquinas, depende el futuro del mundo"

Oscar Wilde

Los mecanismos de variación son la manifestación de la variación a nivel de código. Los mecanismos de variación están ligados con la *realización del dominio* y la *realización de la aplicación.* El proceso de la *realización de dominio* se refiere al proceso de crear los recursos reutilizables de una línea de productos de software[9]. Antes de crear estos recursos es necesario escoger el mecanismo de variación, pues el mecanismo de variación define como se deben crear los recursos. El proceso de la *realización de la aplicación* es el proceso de escoger las variantes para crear una aplicación [9]. Así el proceso de la *realización de la aplicación* es, simplemente, el utilizar él o los mecanismos de variación para crear una miembro de la línea de productos de software. Los mecanismos de variación son independientes de los modelos de la variación, entonces un mismo modelo puede ser implementado a través de diferentes mecanismos.

Existen diversos mecanismos de variación que se pueden aplicar a las líneas de productos de software. La selección de uno o varios mecanismos de variación para una línea de produc¬ tos de software impacta de manera directa en la evolución y desarrollo del dominio y de las aplicaciones individuales. Bachmann et al en [15] establecen que el objetivo de la variabilidad de software es maximizar el retorno de la inversión (ROI) para construir y mantener pro¬ ductos por un periodo de tiempo y/o número de productos. Al analizar esto, se puede ver que difícilmente podrá existir un mecanismo único de variación que sea superior a todos los demás en todas las condiciones. Existen mecanismos de productos que tienen un alto costo en la ingeniería del dominio, pero hacen la generación de aplicaciones virtualmente gratuitas. Este tipo de mecanismos son idóneos cuando se quieren crear un gran número de aplicaciones y se pueden esperar un tiempo largo en crear las primeras aplicaciones. También hay meca¬ nismos cuya ingeniería de dominio es mínimo pero el tiempo de desarrollo de cada aplicacióon individual es alto. Este tipo de mecanismos de variación es idóneo cuando se quieren sólo unas cuantas aplicaciones y se necesita la primera urgentemente. La selección de él o de los mecanismos de variación dentro de una línea de productos de software es de suma impor¬ tancia y es por eso esencial conocer y considerar las distintas características de los diversos mecanismos de variación para poder hacer la selección correcta.

En los siguientes párrafos describiremos algunos de los mecanismos de variación que se han explorado en la literatura de las líneas de productos de software. Para poder comprender mejor los distintos mecanismos de variación enumeraremos las distintas características que deseamos analizar:

1. *Tiempo de Ligado.* El momento en que se deciden y se aplican las variantes en un punto de variación cambia de mecanismo a mecanismo. Krueger en [17] enlista los diferentes tiempos de ligado: tiempo de reuso de código (frameworks, especializaciones, etc), tiempo de desarrollo (copiar y mantener, etc), instanciación de código estático (copiar directorios, etc), tiempo de compilación (preprocesadores, generadores, aspectos, etc), tiempo de empaquetado (recursos, etc), modificaciones del consumidor, tiempo de instalación *(scripts* de instalación), tiempo de arranque (configuración del programa) o tiempo de ejecución (preferencia de usuarios).

2. *Tecnologías y habilidades necesarias.* Un mecanismo de variación puede aprovechar los conocimientos que ya se tienen en el lenguaje de desarrollo de la aplicación (herencia, *templates* y otros) o puede requerir nuevos conocimientos. Además que tan buen soporte hay para el mecanismo con las herramientas ya existentes o que nuevas herramientas son necesarias para su desarrollo. Al hablar de las herramientas y habilidades nos referimos a las utilizadas tanto en la *ingeniería de dominio* como en la *ingeniería de aplicación* pues unas y otras pueden ser similares o completamente diferentes.

3. *Validación y limitación de las combinaciones.* La validación y limitación de las com¬ binaciones es muy importante para los mecanismos de variación pues son las que li¬ mitan que productos se pueden crear con la línea de productos de software. Esto es especialmente importante si recordamos que las líneas de productos crecen de manera exponencial dependiendo del número de puntos de variación y variantes que contengan. En los mecanismos de variación su validación puede ser externa o interna: en la externa el desarrollador debe especificar las restricciones, en las internas el mecanismo de la variación las descubre automáticamente (compilación, etc). Las internas son las más efectivas para utilizar; sin embargo, siempre habrá necesidad de validaciones externas para adaptarse a las reglas del dominio que no pueden ser representadas por el código del sistema.

4. *Soporte para ambientes heterogéneos.* Como discutimos en la sección 1.1 los ambientes heterogéneos son inevitables en los proyectos de software. Es por esto que creemos que es necesario para cualquier línea de productos tener la habilidad de trabajar en ambientes heterogéneos y poder incluir puntos de variación en cualquier documento del sistema, sin importar el formato en que fue creado éste.

5. *Estrategia de la línea de productos de software.* Como mencionado antes existen tres tipos de estrategias de líneas de productos: proactivas, reactivas y extractivas [13]. Cada tipo de mecanismo funciona mejor para algunos tipos de estrategias. En particular se valoraran mejor los mecanismos que permitan trabajar con la mayoría o todas las diferentes estrategias.

6. *Invasividad.* Las líneas de productos deben de tratar de ser mínimamente invasivas en su transición de desarrollo centrado en aplicaciones a desarrollo centrados en línea de productos de software[16]. Esto es importante pues que permite utilizar los conocimien¬ tos y recursos ya creados con anterioridad en otros proyectos en una línea de productos de software. Asó para ser poco invasiva, debe de tratar de conservar la mayor parte de la estructura, jerarquía, formatos y sintaxis de los archivos ya existentes.

7. *Variabilidad soportada.* En [19] Gacek y Anastasopoulos definen varios tipos de varia¬ bilidad que pueden soportar los mecanismos de variación, estos son: positiva, negativa, opcional, alternativo, funcional y de plataforma o ambiente. Ademas de estos tipos de variación podemos identificar también la variación *derivativa* [25] en donde la varia- ción producida depende de las variantes seleccionadas para la aplicación. Idealmente un mecanismo de variación debe poder soportar todo tipo de variaciones, sin embargo regularmente los mecanismos de variación solamente soporta algunas cuantas.

2.2.1. Lenguajes de Un Dominio Específico

Los *Lenguajes de un Dominio Específicos* o *DSL* (por sus siglas en inglés) han sido una de las herramientas más recurridas para el desarrollo de líneas de productos de software. La diferencia entre un *DSL* y un lenguaje de propósito general es que los *DSLs* suelen tener un propósito limitado y una sintaxis más sencilla. Es en esta sintaxis más sencilla que se encuentra la razón por la cual son buscados como mecanismos de variación para las líneas de productos. Estas sintaxis sencillas permiten el desarrollo de aplicaciones de manera mas rápida siendo un pegamento entre componentes. Existen varias maneras de crear *DSLs*[26]: Como "piggybacking" (ejemplo Lex y Yacc), *pipeline* (se procesa en varios pasos), *procesador léxico* (sustituciones simples), extensión del lenguaje (se agrega funcionalidad a un lenguaje ya existente) y especialización del lenguaje(quitar funcionalidad de un lenguaje existente).Los *DSLs* pueden ser clasificados como internos y externos dependiendo de si utilizan un lenguaje ya existente o no como su base de su desarrollo[27]. Los beneficios de los lenguajes internos son su facilidad de desarrollo explotando el potencial de metaprogramación de los lenguajes (Ruby y Lisp son dos ejemplos muy usados) a costa de una validación, seguridad y flexibilidad mas limitadas. La literatura esta llena de ejemplos de *DSLs* utilizados en líneas de productos de software. Weiss en [10] nos muestra un ejemplo de un *DSL* para una linea de productos para estaciones del clima flotantes. Batory en [28] mezcla los generadores que veremos en la sección 2.2.3 con un *DSLs* para máquinas de estados para generar una línea de productos de software para soporte de fuego. Podemos notar que los *DSLs* se han usado en diversas ocasiones para crear líneas de productos de software, he incluso el resto de los mecanismos de variación puede ser considerado como casos especiales de *DSLs*.

Los *DSLs* regularmente trabajan antes o durante la compilación, pero no es difícil con¬cebir que estos trabajen en cualquier otro tiempo, incluyendo pero no limitado a el tiempo de ejecución si se diseñan con ese proposito. Los *DSLs* son uno de los mecanismos de variación mas versátiles y flexibles que existen para las línea de productos de software. Esto hace posible que los *DSL* puedan incluir validaciones y verificaciones desde su diseño. Además un mismo archivo en *DSL* puede generar diferentes salidas que indica un buen soporte para ambientes heterogéneos[10]. Dado la versatilidad de las soluciones *DSLs* también es posible que éstas implementen todos los tipos de variabilidad. Sin embargo esta versatilidad tiene un costo, el soporte de las herramientas para los *DSLs* es mínimo y en general requiere herramientas hechas a la medida. Ademas los *DSLs* son muy invasivos al crear un lenguaje enteramente nuevo, lo que impide el fácil reuso de recursos ya existentes incluso cuando se trata de *DSLs* internos (Aunque es en menor medida). Además los *DSLs* requieren de mucho entrenamiento para los desarrolladores antes de que estos sean efectivos. Al ser muy invasivos es la razón por la que los *DSLs* son un buen mecanismo de variación si se desea utilizar una estrategia del tipo proactiva o reactiva pero no tan buena para una estrategia extractiva.

2.2.2. Plantillas C++

Las plantillas de C++ son un caso particular de plantillas, pues estas están integra-das directamente al lenguaje. Las plantillas de C++ son un sublenguaje Touring-Completo embebido en C++[20]. Esto las hace una herramienta poderosa de *metaprogramación*. Por todo esto, se ha explorado extensivamente el uso de las plantillas C++ para el desarrollo de líneas de productos de software. Czarnecki en [20] explorar el utilizar las plantillas de C++ y explotar su herencia para crear una línea de productos de software. En el trabajo de VanHilst

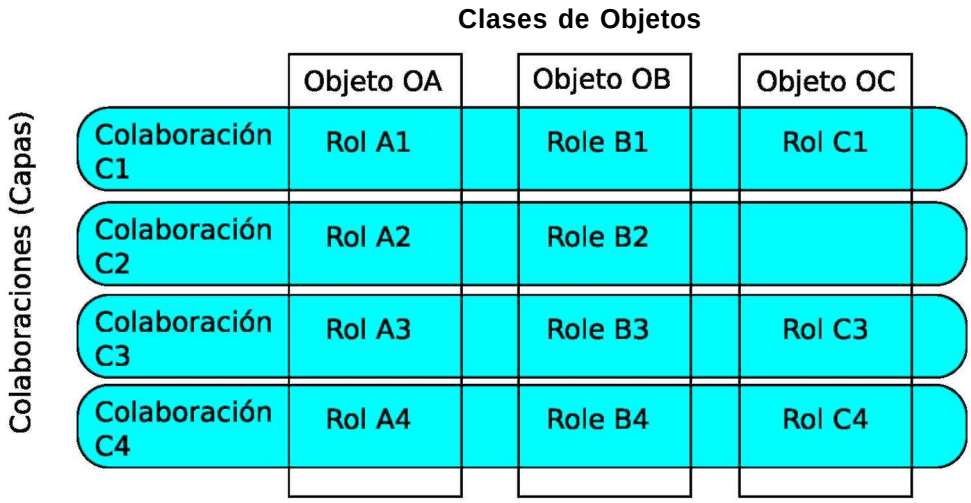


Figura 5: Ejemplo de descomposición en colaboraciones para Mixin Layers

y Notkin en [29] se establece que se puede hacer diseños de colaboración y roles con el uso de las plantillas de C++. Esta investigación sentó las bases para el trabajo de Smaragdakis y Batory de *Mixin Layers* [30] como mecanismo de variación para líneas de productos de software. Los *mixins* se refieren a un concepto de programación en donde un fragmento de una clase es implementado de manera externa a la clase principal. Los *Mixin Layers* extienden el concepto al crear grupos o capas de *mixins* que se agregan a diversas clases a la vez. Los *Mixin Layers* tienen la capacidad de crear colaboraciones de las plantillas de C++ para crear refinamientos de varias clases a la vez a través organizadas en capas. Estos refinamientos pueden agregar nueva funcionalidad a varias clases en un solo momento dentro de la aplicación, efectivamente haciendo modificaciones del tipo *"cross-cut concern"* sobre las clases. En la figura 5 basada en una figura similar en [30] podemos ver la interrelación entre las capas y las clases. Dado lo anterior, podemos ver que las plantillas C++ son una herramienta poderosa integrada al lenguaje que nos puede ayudar en la creación de línea de productos de software.

Los *mixin layers* son un mecanismo de variación de precompilación que hace uso de la funcionalidad ya existentes en el lenguaje C++. Al estar ligado al lenguaje, los Mixin Layers, son totalmente soportados por las herramientas y conocimientos disponibles para los desarrolladores. Esta integración también hace que se pueda reutilizar una gran cantidad de los recursos ya existentes pues es un mecanismo poco invasivo. Sin embargo, los Mixin Layers no se adaptan bien a ambientes heterogéneos puesto que el formato factible de los archivos que participen en la línea de productos de software esta recluido únicamente a C++. El soporte de la variabilidad no es tan extenso y se limita principalmente a la de adición de funcionalidad y no soporta, por ejemplo la variabilidad derivativa. La verificación es bastante buena ya que el compilador puede verificar que el código generado sea válido, sin embargo el soporte para la validación y limitación de las combinaciones es mínimo. Dadas sus características las plantillas de C++ pueden ser utilizado en cualquier tipo de estrategia de desarrollo: ya sea proactiva, reactiva o extractiva; sin embargo, en el caso de las extractivas sólo se pueden reutilizar recursos creados originalmente en C++.

2.2.3. Generadores

Al igual que los mixin los generadores GenVoca utilizan el desarrollo por composiciones en donde la funcionalidad se va agregando por capas. Las similitudes con los Mixin Layers son impresionantes y con justificada razón: Don Batory participó en la creación de ambos. Sin embargo, en lugar de explotar la funcionalidad en los lenguajes ya existentes como las plantillas de C++; GenVoca utiliza extensiones sobre los lenguajes en que se quiere desarrollar. El objetivo de estas extensiones es permitir el desarrollo composicional de manera mas completa para permitir verificaciones y validaciones. Los conceptos básicos de los generadores fueron establecidos por Batory y O'Malley en [31]. Los principales extensiones se refieren al concepto de componentes *(components* en ingles) y reinos *(realm* en ingles). Cada componente puede agregar una funcionalidad al sistema y estos componentes pueden ser clasificados en diversos reinos o categorías. Con esta herramienta podemos definir un modelo que represente que combinaciones jerárquicas son validas. Este modelo, que es muy similar a una definición de una gramática, se le llama Modelo GenVoca. A continuación podemos ver un ejemplo de una gramática para una generador de base de datos similar a la encontrada en [32].

```
1 S = {a , b, c}
2 T = {d[S] , e[S], f[S]}
3 W = {n[W] , m[W] , p, q[T,S]}
```

Estas gramáticas pueden representar información muy similar a la información a los dia¬ gramas de rasgos vistos en 2.1.6. Ademas los generadores incluyen reglas de diseño *(Design rules* en ingles) que permiten agregar restricciones extras de las creadas por la gramática [32]. Estas reglas de diseño son de 4 tipos: postcondiciones, precondiciones, postrestricciones y prerrestricciones. El proceso de verificar estas reglas se le conoce como *DRC* (Design Rule Checking). Finalmente al lenguaje se le agrega funcionalidad para soportar el desarrollo me¬ diante componentes jerárquicos, en el caso de C++ se creo el lenguaje P++[33]. El desarrollo de los generadores a continuado y se desarrolló el *AHEAD Tool Suite*[6] que busca soportar las jerarquías en múltiples lenguajes y tipos de artefactos (diferentes al código fuente). También se ha buscado que la validación de las composiciones sea mas automáticas en [34]. Asó los generadores han ido aumentando su funcionalidad buscando ser un mecanismo de variación formal y ampliamente funcional para el desarrollo de líneas de productos de software.

Los generadores son uno de los mecanismo de variación mas intensamente desarrollados para su uso en línea de productos de software. Este mecanismo de variación con tiempo de ligado de compilación puede limitar y validar las composiciones de manera estricta y soporta muchos de los diferentes tipos de variabilidad. La última versión, el AHEAD Tool Suite, busca mejorar la capacidad de los generadores para trabajar en ambientes heterogéneos, sin embargo el soporte para cada lenguaje o tipo de artefacto se tiene que incluir o crear por separado. Incluso con el AHEAD Tool Suite el sistema es todavía invasivo pues se extienden todos los lenguajes en que se quiere desarrollar la línea de productos de software. Aún asó dado que las extensiones son relativamente pequeñas es factible que se puedan reutilizar alguna de las herramientas ya existentes y que los generadores se puedan utilizar para estrategias proactivas, reactivas y extractivas. Esta es la razón por la cuales consideramos los generadores uno de los mecanismo de variación mas completos disponibles para los desarrolladores de líneas de productos de software.

2.2.4. Aspectos

La programación orientada a aspectos o AOP (por las siglas en ingles de *Aspect Oriented Programming)* es un desarrollo reciente de la ingeniería de software [35]. En la programación orientada a aspectos se busca separar las diferentes preocupaciones de un sistema *(concerns* en la literatura) para después entretejerlas *(weaving* en la literatura) en una sola aplicación. Si consideramos que en lugar de utilizar preocupaciones secundarias utilizamos rasgos po¬ demos utilizar aspectos para crear una línea de productos de software. El uso de Aspectos para crear líneas de productos ya ha sido explorado [36] [37] particularmente en AspectJ con buenos resultados. AspectJ es la implementación más popular de la programación orientada a aspectos y es una extensión del lenguaje Java. En los Aspectos se puede agregar nuevo fun¬ cionalidad o código (consejos o *advices)* en puntos seleccionados del código (puntos de corte o *pointcuts).* El utilizar consejos como rasgos en lugar de preocupaciones es la manera en que puede desarrollarse una línea de productos de software a través de aspectos. Los aspectos, a diferencia de otros mecanismos de variación, pueden hacer las mismas modificaciones en varios puntos de corte al mismo tiempo. Esto lo diferencia fuertemente de otros mecanismos de variación y lo hace idoneo para ciertos tipos de rasgos (seguridad, rastreo y otros) que se deben aplicar de manera dispersa en un sistema.

Para el caso de AspectJ el tiempo de ligado del mecanismo de variación es la compila- ción. Para utilizar este mecanismo se tiene que aprender los conceptos de la programación orientada a aspectos, pero afortunadamente muchos de los conocimientos y herramientas de los lenguajes orientados a objetos se aplican igual que siempre. También es cierto que actual¬ mente existen nuevas herramientas para soportar el desarrollo basado en aspectos disponibles en el mercado. Sin embargo, la validación y la limitación de las combinaciones no es posible con este enfoque, siendo este hecho una de sus mayores debilidades de esta herramienta como mecanismo de variación para líneas de productos de software. Estas estrategias tienen un soporte muy limitado para la heterogeneidad, al menos hasta que los aspectos sean mejor soportados por diversos lenguajes. Al poder agregar los aspectos de manera completamente externa la invasividad es nula siendo una de los mejores mecanismos en este punto. Sin em¬ bargo la variabilidad *negativa* no es fácil de implementar con estas herramientas[19]. Dado que es poco invasivo, el desarrollo basado en aspectos puede funcionar en cualquiera de las estrategias (proactiva, reactiva y extractiva) haciéndola una herramienta bastante versátil para el desarrollo de línea de productos de software.

2.2.5. Frames

La tecnología de Frames es otro de los mecanismos de variación que se han investigado como factible para ser utilizado dentro de las líneas de productos de software. XVCL [38] es la mas nueva implementación de frames y esta basado en la tecnología de Frames creada por Paul Basset y documentada en [39]. Los Frames son una tecnología para crear variantes jerárquicas de código y otros documentos. El sistema de Frames es, básicamente, una serie de plantillas que son mezcladas para crear los documentos finales. En el caso de XVCL estas plantillas están encapsuladas en el formato XML. Dado el poder que se tiene para escoger configuraciones los Frames son una excelente opción para crear líneas de productos de software.

El tiempo de ligado de este mecanismo es de nuevo es la compilación. Sin embargo, no existe buenas herramientas para dar soporte a XVCL, incluso no se puede hacer uso las herramientas del lenguaje de los documentos originales pues los Frames son altamente

invasivos para los documentos al cambiar enteramente su formato por XML y alterar su estructura para crear nuevas jerarquías. Esta invasividad es la razón por la cual la estrategia mas recomendada para este mecanismo de variación es la proactiva. Sin embargo, cualquier tipo de documento de texto puede ser generado mediante el sistema asó que el soporte para la heterogeneidad es bastante bueno, ademas de que la estructura puede soportar todos tipo de variabilidad enumerados con anterioridad. Uno de los limites del uso de frames como mecanismo de variación para líneas de productos de software es que los Frames no incluyen ningún método con el cual se puede restringir las composiciones válidas. En conclusión, los Frames son un muy buen mecanismo de variación si se desea soportar la heterogeneidad del sistema, pero a un alto costo, al ser altamente invasivo de la estructura de los documentos.

2.2.6. Resumen de los mecanismos de variación

Actualmente existen muchas opciones como mecanismo de variación en líneas de pro¬ductos de software. Tanta diversidad de opciones existe pues los diferentes proyectos tienen distintas necesidades. Sin embargo, actualmente se ha comenzado a buscar en las estrategias que los mecanismos de variación sean menos invasivas. El que un mecanismo de variación sea menos invasivo permite que se pueda utilizar otras estrategias de líneas de productos de software ademas de la proactiva. Mas importante, para el tema de esta tesis, el crear una herramienta menos invasiva nos permite poder aplicar una línea de productos de software para ambientes heterogéneos y reutilizar recursos ya existente. Estas mejoras han ocurrido con el tiempo, pero a la vez buscando no descuidar la funcionalidad esencial necesaria para crear una línea de productos de software.

Al principio de esta sección se describió que es un mecanismo de variación y como se relaciona con las líneas de productos de software. Luego definimos un conjunto de características deseables en un mecanismo de variación para una línea de productos de software. Finalmente, en las subsecciones anteriores hemos dado un rápido vistazo a algunas de las diferentes categorías de mecanismos de variación y los hemos contrastado contra las características deseables. Tradicionalmente, las herramientas para los mecanismos de variación se enfocan en un sólo mecanismo de variación, mientras tanto *Pynion* trata de funcionar para varios mecanismos de variación. Esto nos da una buena idea de marco teórico actual en donde se va desarrollar *Pynion.*

2.3. Conclusión

En esta sección se presentó una breve introducción a los conceptos relacionados con las líneas de productos y los conceptos claves que hay que entender para su desarrollo. Ademas dimos un breve repasó en otras técnicas que han sido utilizadas en el dominio de la línea de productos de software como mecanismo de variación. En esta sección se presentaron los fundamentos esenciales para entender el desarrollo de esta tesis: conocimiento del dominio (de línea de productos de software) y conocimiento de herramientas similares (revisión de otros mecanismos de variación).

En la siguiente sección presentaremos de manera detallada a *Pynion.* Comenzaremos con el diseño completo de *Pynion* y una vista a detalle de los componentes que lo confor¬man. Después presentaremos una guía para el usuario de esta herramienta para mecanismos de variación con sencillos ejemplos para entender su funcionamiento completo. Finalmente, revisaremos a fondo secciones importantes del código para entender a mayor profundidad el

funcionamiento interno de la herramienta.

3. Pynion: prototipo de herramienta para mecanismos de variación para líneas de productos en ambientes heterogéneos

No siempre las líneas de productos son concebidas como tales. Muchas veces estas aparecen una vez que se han creado varios productos con funcionalidad similares. Esto hace que sea común que se quiera desarrollar una línea de productos de software reutilizando los recursos ya creados con anterioridad. Las aplicaciones también suelen contener diversos lenguajes de programación y configuración para ocuparse de distintas partes del sistema: vistas, configuración, control, modelo, etc. Esto hace que sea común encontrarse con que se desea crear una linea de productos de software utilizando recursos ya existentes y/o en ambientes heterogéneos. La propuesta de esta tesis es crear un prototipo de una herramienta para un mecanismo de variación que minimice el impacto en los recursos ya existentes causados por la adaptación a una estrategia de línea de productos de software y que ésta pueda ser usados en ambientes heterogéneos. Esta herramienta, llamada *Pynion,* busca ayudar a resolver los retos que los ambientes heterogéneos presentan a líneas de productos de software.

El nombre de *Pynion* para esta herramienta surge de unir dos palabras: Python y piñon *(pinion* en ingles). El prefijo *Py* es común entre las aplicaciones desarrolladas en el lenguaje de programación Python y Pynion no es una excepción. Un piñon es un tipo de engrane pequeño que puede ser usado en varios mecanismos, esta es la razón por la cuál el nombre fue adoptado. Al ser un engrane es fácil identificarlo con las líneas de producción del mundo real. Al ser pequeño (y poco visible) toma otra de las características importantes de la herramienta: la no invasividad. Asó el nombre *Pynion* es una referencia tanto al nombre del lenguaje de programación en que la herramienta se creo como a la funcionalidad diseñada para este.

Este capítulo presentará una descripción completa y a fondo de de *Pynion* y los principios que le dieron sustento. En las primera sección describiremos de manera genérica las características de la propuesta. En la segunda subsección entraremos más a detalle en la descripción de Pynion y lo analizaremos pieza por pieza de manera individual. La tercera subsección consiste en un manual completo de usuario de este prototipo de herramienta para mecanismos de variación. En la cuarta subsección se incluirá secciones pequeñas del código notable que sean interesantes para realizar un estudio mas a detalle. Finalmente, en la quinta sección analizaremos los componentes e ideas externas utilizadas por *Pynion* y explicaremos por que fueron elegidas y quienes las crearon o las idearon. Asó, en esta sección se busca dar un entendimiento profundo, claro y completo de *Pynion* que permita entender como esta herramienta para mecanismos de variación puede facilitar el desarrollo de líneas de productos para ambientes heterogéneos.

3.1. Vista General

Pynion es un prototipo de una herramienta para mecanismo de variación para ambientes heterogéneos y productos legados. Para lograr esto *Pynion* busca llegar a varios objetivos. El primer objetivo es ser poco invasivo, de modo que se pueda utilizar en varios lenguajes de programación y formatos de texto. El segundo objetivo es permitir la flexibilidad en las estrategias de creación de líneas de productos. El tercer y último objetivo es permitir la mayor cantidad de tipos de variabilidad posibles. Una de las características de *Pynion* es que permite la variación a varios niveles de alcance permitiendo mayor flexibilidad en el estilo de desarrollo y estrategias para el desarrollador de la línea de productos de software. Como todos los mecanismos de variación, Pynion funciona mezclando los rasgos que se han seleccionado

de la línea de productos de software. A la combinación de rasgos que forman un miembro de la línea de productos de software se le conoce como composición. La representación de cada rasgo en *Pynion* es un juego de archivos dentro de una carpeta cuyo nombre es el mismo nombre del rasgo. La combinación y orden en que se seleccionan los rasgo de una composición, esto es, las instrucciones y contenidos de las carpetas, es lo que permite crear los diversos miembros de la línea de productos de software.

Cada carpeta contiene todas las modificaciones necesarias para unir un rasgo a un miembro cualquiera de la línea de productos de software. Estas modificaciones pueden tener varios alcances. Pynion permite modificaciones de tres alcances: archivo, plantilla y función. A nivel archivo permite agregar, eliminar o sustituir archivos completos independientemente si son textuales o binarios. A nivel plantilla se puede sustituir fragmentos de un archivo de texto por otro contenido a partir de plantillas. También existe una sustitución similar a plantilla, pero en lugar de sustituir un fragmento de archivo por otro, se sustituye un fragmento de un archivo por el valor de retorno de una función de *Python.* Estos diversos alcances permite ajustar la granularidad con la que se desea trabajar las modificaciones permitiendo conservar mas código intacto o con modificaciones mínimas.

Otra de las características importantes de *Pynion* es el restricción de composiciones. En *Pynion* cada rasgo puede verificar si la composición del miembro de la línea de productos de software completo es válida. Para realizar las validaciones cada rasgo se tiene que hacer cargo de dos funciones: publicar sus *valores* y verificar su propia validez en el producto de la línea. Cada capa puede publicar sus *valores*; los *valores* son palabras que describen el rasgo. Tradicionalmente, los valores suelen ser el nombre del rasgo y su categoría. *Pynion* recolec¬ ta estas propiedades que despues serán mandadas de regreso a cada rasgo individualmente como prevalores y postvalores. A partir de ahí, cada rasgo será encargado de verificar que la composición a la cual pertenece sea valida. El rasgo tendrá a su disposición los *valores* expuestas por los demás rasgos y su posición dentro de la composición para evaluar la validez del producto. Esta herramienta permite limitar el número de combinaciones válidas para cada línea de productos de software, hacinándolas mas manejables y garantizando su validez.

Pynion es una herramienta para mecanismos de variación para línea de productos de software que funciona a través de sustitución de archivos, plantillas o funciones. Esto le permite ser una buena herramienta para traba jar en ambientes heterogéneos o legados, que regularmente no son fáciles de adaptar a una línea de productos de software. El poder trabajar en estos tipos de ambientes es necesario para una mejor aceptación de las líneas de productos de software como una herramienta útil en la ingeniería de software. Pynion trata de ser una herramienta para mecanismos de variación abierto y flexible que busca poder ser adaptado a las diferentes estrategias de la líneas de productos. Es por esto que *Pynion,* en la medida de lo posible, busca no limitarse a ninguna metodología o modo de diseñar la línea de productos de software y facilitar su uso en cualquiera de ellas. *Pynion* busca ser asó una herramienta versátil para ayudar en el desarrollo de líneas de productos en ambientes heterogéneos y/o legados.

3.2. Descripción Detallada

Pynion permite realizar las variaciones en varios alcances. Los alcances mas poderosos y versátiles de *Pynion* son el alcance a nivel plantilla y función. Estos alcances se lleva por medio del preprocesamiento de archivos de textos llevada a cabo por cada uno de los rasgos. Este tipo de preprocesamiento se puede hacer en todos los formatos de archivos de texto

que comprendan una aplicación mientras cumplan un par de condiciones: los espacios en blanco no son significativos en la sintaxis y existe un modo de embeber comentarios dentro del documento. La razón de estas limitaciones es que se puede aprovechar mayor provecho de *Pynion* cuando se utiliza el alcance a nivel plantillas. *Pynion* hace un muy buen esfuerzo en lidiar con la indentación dentro de formatos de texto donde los espacios en blanco son significativos(como el lenguaje Python), sin embargo al trabajar con plantillas es fácil no respetar correctamente los espacios en blanco. Esto hace que lenguajes o formatos donde los espacios en blanco sean significativos sean más difícil de trabajar con esta herramienta para mecanismos de variación. La razón de la exigencia de que el lenguaje soporte comentarios es que *Pynion* utiliza la funcionalidad de agregar comentarios en cada lenguaje para marcar los puntos de variación en donde se harán las sustituciones de las plantillas y funciones. Dado esta restricción *Pynion* no podría trabajar en lenguajes que no soporten comentarios. Afortunadamente, la gran mayoría de los lenguajes y formatos de texto, no tienen problemas con estas restricciones.

Pynion esta desarrollado en Python que un lenguaje dinámico. El uso de Python permite modificar la línea de productos de software sin necesidad de recompilar el código fuente. Además, dado que la mayor parte del esfuerzo de desarrollo se debe enfocar en la creación de recursos y no en en la definición de como estos recursos son mezclados, el desarrollo del código fuente para unir los rasgos debe ser lo mas sencillo posible. El uso de un lenguaje para *scripting* nos permite un desarrollo mas veloz de la definición del proceso de mezclado de los rasgos en la línea de productos de software. Finalmente el uso de Python permite el uso de las platillas de Mako [40] como base para el preprocesado de archivos por medio de plantillas. Python provee a *Pynion* con la flexibilidad necesaria para un rápido de desarrollo de los rasgos de una línea de productos de software.

El proceso mediante el cual se realiza la mezcla de los diferentes rasgos debe ser sencillo, previsible y determinístico. En la figura 6 vemos el proceso en que se mezclan los diferentes rasgos para formar una aplicación miembro de línea de productos de software. Para realizar la mezcla de rasgos se ejecuta *Pynion* de la línea de comando. Los argumentos de la línea de comandos son los nombres y el orden en que se mezclaran los rasgos para formar la aplicación final. Estos nombres deben corresponder a los nombres de las carpetas en la carpeta de la línea de productos de software. El controlador *root* de cada rasgo se encarga de publicar los valores del rasgo y validar el rasgo y su posición. Una vez que cada rasgo ha sido encontrado válido se procede a la mezcla de los rasgos. La información en el archivo *feature.ini* representa la configuración de cada rasgo. Este archivo contiene la información de como el rasgo debe ser mezclado. El archivo *feature.ini* incluye tres tipos de instrucciones de mezclado: *agregar*, *eliminar* y *actualizar*. La instrucción de actualizar se realiza por medio de *plantillas* o *funcio¬ nes*. Si no existe ningún error, la aplicación preprocesada se deberá encontrar en la carpeta de salida. Este proceso se puede variar seleccionando otros rasgos o cambiandolos de orden, o incluso cambiando los archivos de configuración.

La validación de la aplicación miembro de la línea de productos de software se realiza a nivel rasgo. Cada rasgo se encarga de validarse a si mismo; una vez que todos los rasgos se logran validar sin error la aplicación se considera válida y se procede al proceso de mezclado. Cada carpeta de rasgo puede contener un archivo llamado *root.py*. El archivo *root.py* es un archivo de *Python* que contiene una clase que contiene dos funciones: *publish* y *validate*. *Publish* se encarga de publicar los valores (palabras o *strings)* asociadas con este rasgo. Por omisión si no existe el archivo *root.py* publish regresa el nombre del rasgo como su valor. La función de *validate* verifica que el rasgo y su posición sean validos. La función *validate* recibe

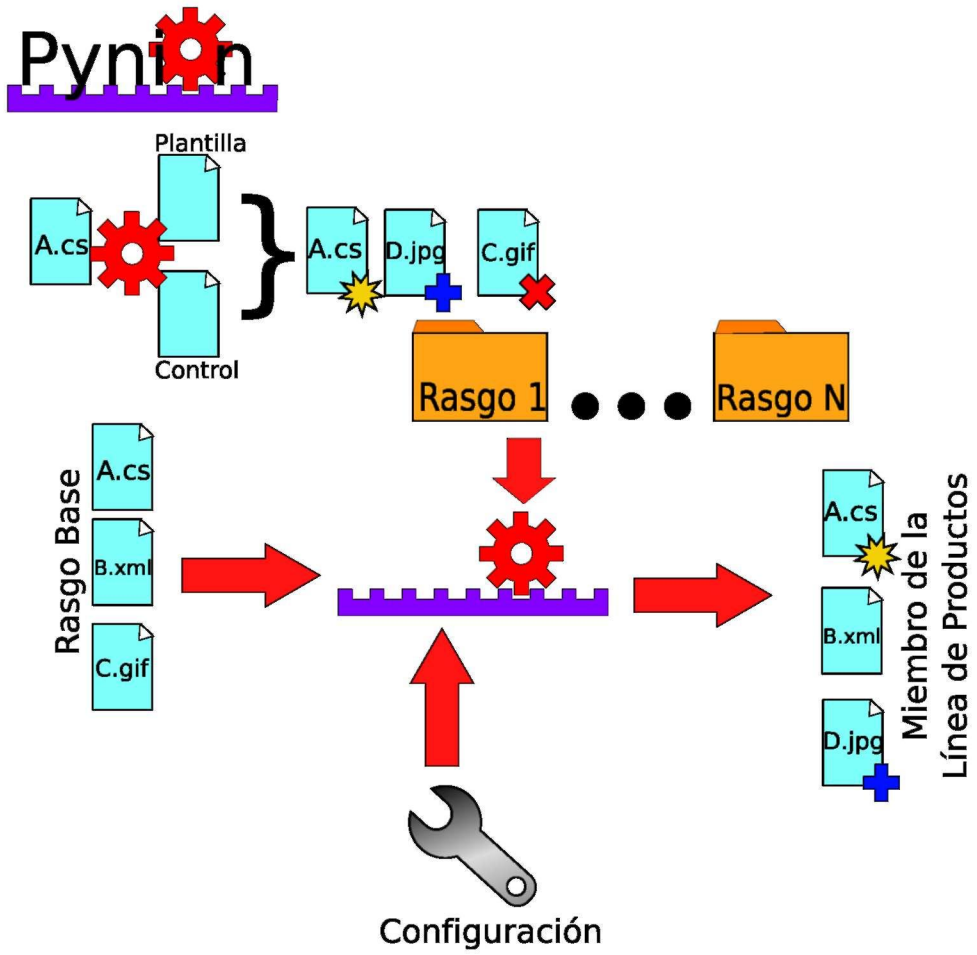


Figura 6: Diagrama del funcionamiento de Pynion

como parámetros todos los valores publicados por los rasgos anteriores como *prevalores* y los valores publicados por los rasgos posteriores como *postvalores.* Esta estrategia es similar a la estrategia usada en [32] por Batory e igual de poderosa. El valor de retorno de la función *validate* es la lista de errores encontrados o *nulo* o un arreglo vacío en caso de que sea válido. El proceso de validación es esencial para garantizar que las aplicaciones miembro de la línea de productos de software funcionen como se espera.

Cada rasgo contiene la información de como este podría ser integrado a la aplicación miembro de la línea de productos de software. El desarrollador tiene completo control sobre la integración del código; los archivos y plantillas tienen que ser integrados explícitamente por el desarrollador de la línea de productos de software. El archivo *feature.ini* es el archivo que contiene las instrucciones generales para mezclar el rasgo en la aplicación. El formato de este archivo es el formato tópico de los archivos *.ini.* El archivo contiene 3 secciones: add, *remove* y *update* que se encargan de la funcionalidad de agregar, remover y modificar archivos respectivamente. En la sección de *add* se dan de alta las trayectorias a los archivos que se desean agregar(o sustituir integramente) de la aplicación para este rasgo. La siguiente sección, *delete*, contiene las trayectorias de los archivos que se eliminaran durante el proceso de mezclado. Finalmente, la sección de *update* contiene la información para realizar los dos tipos de modificaciones existentes sobre un archivos de texto: por plantilla o por función. Las modificaciones por plantillas o función son las variaciones mas poderosas y flexibles permitidas al desarrollador por medio de *Pynion.*

La funcionalidad de modificación por medio de plantillas o funciones se encuentra en el corazón de *Pynion.* Esto permite hacer cambios mas localizados sobre un sólo archivo de texto. Para que un archivo pueda ser transformado este debe de contener puntos de variación. Los puntos de variación indican cuales son los lugares donde sera modificado el contenido del archivo. Las marcas de los puntos de variación pueden ser ocultados detrás de comentarios (independientemente de como los comentarios se agreguen en el formato del archivo) de modo que el marcaje no afecte con la compilación, ni la ejecución de los archivos a modificar. El primer modo de modificar un archivo de texto es mediante plantillas, en este caso se crea un archivo con una plantilla para cada punto de variación que se desea modificar del documento original. El soporte para las plantillas en *Pynion* es provisto por Mako Templates[40], un sistema de plantillas para Python. En el caso de las funciones se debe crear una clase y esta clase debe de contener una función para cada punto de variación. El valor de retorno de estas funciones sustituirán el código original dentro del punto de variación. Esta gran gama de opciones permite que el desarrollador pueda reutilizar todo el código que necesita a un menor costo de desarrollo y con toda la flexibilidad para implementar diversas estrategias.

El verdadero poder del uso de plantillas y funciones en *Pynion* se basa en el *contexto* disponible para ellas. El contexto se refiere a la información que esta disponible para modificar y tomar decisiones acerca de la salida de las plantillas y funciones. En el caso de las plantillas los patrones tienen un objeto llamado *c* que contienen el contexto. En el caso de las funciones el contexto es pasado como un parametro llamado *context.* El *contexto* contiene la siguiente información: el texto original en los puntos de variación, los rasgos anteriores y posteriores, el nombre y posición del rasgo, los prevalores y postvalores y la configuración tanto de la familia completa de productos como del rasgo. Toda esta información permite gran flexibilidad en como se puede integrar el rasgo al resto del producto permitiendo, entre otras cosas, realizar variaciones *derivativas* que dependan de rasgos previos o posteriores (como visto en [25]). *Pynion* esta diseñado para dar soportar el mayor número de tipos de variaciones diferentes posibles.

El orden en el cual se ejecutan las instrucciones de variación también pueden afectar el producto final. Las instrucciones de agregar se realizan primero, eliminar se realizan segundo y actualizar se realizan al final. Sin embargo, no es recomendable modificar por varios métodos el mismo archivo. El no modificar varias veces el mismo archivo en el mismo rasgo le dará mas limpieza al código y al rasgo, ademas de una mayor certeza de que el resultado es el esperado por el desarrollador.

Esta sección cubrió las características y forma de funcionamiento de *Pynion.* Se describió a fondo como se puede usar a *Pynion* como una herramienta para mecanismo de variación y cuales son los beneficios de usarlo. La siguiente sección es un manual de usuario, donde se demostrará como se utiliza y se desarrolla las líneas de productos de software para sacar provecho a toda la funcionalidad de *Pynion.*

3.3. Manual de Usuario

> "Siempre hay un momento en que la personas se dan por vencidas y hacen lo impensable: leen el manual"
>
> Anónimo

Pynion es una herramienta mecanismos de variación para línea de productos de software. El objetivo primordial de *Pynion* es mejorar el soporte para líneas de productos legados cuando se tiene ambientes heterogéneos y/o productos legados. En esta sección veremos paso a paso como utilizar *Pynion* para la creación de una línea de productos de software. El objetivo es presentar una visión detallada de la funcionalidad completa de *Pynion* y como aprovecharla. Esta succión permitirá al lector entender como instalar *Pynion,* crear una línea de productos de software, administrar sus rasgos y crear miembros de esta línea de productos de software.

3.3.1. Instalación

Pynion fue escrito en un lenguaje de *scripting,* asó que para permitir su ejecución sólo se necesita copiarlos a la carpeta correcta o agregar su ubicación a la variable de ambiente *path.* Sin embargo para que Pynion funcione hay ciertos prerrequisitos que tienen que ser instalados en el equipo de computo. Estos incluyen el interprete del lenguaje Python y varias bibliotecas de las que hace uso *Pynion.*

- *Python.* Python es el lenguaje y plataforma donde Pynion esta construido. La versión actual liberada al mercado de Python es la versión 2.6. Python puede ser instalado en la gran mayoría de los sistemas operativos modernos. En el caso de Mac OSX y los sistemas operativos variantes y derivados de UNIX, Python se encuentra precargado en el sistema operativo desde su instalación. Para el caso de los sistemas operativos de Microsoft (Windows 2003 Server, Windows XP, Vista, etc.) se puede descargar el programa instalador desde la página *http://python.org.* Es importante agregar la carpeta de instalación de Python a la variable de ambiente *path* [41] para que Python pueda ser ejecutado desde cualquier carpeta sin problemas.

- *ConfigObj.* ConfigObj es un modulo para Python para la lectura y escritura de archivos de configuración en el formato *.ini.* La gran virtud de ConfigObj es que su interfase de uso es sencilla e intuitiva [42]. ConfigObj provee dos funciones diferentes para

Pynion. La primera es la configuración que controla las diferentes variaciones para cada rasgo en la línea de productos de software. La segunda es permitir configuraciones adicionales personalizadas para los desarrolladores (mas información en la siguientes secciones). ConfigObj puede ser descargado de *h*ttp://www.voidspace.org.uk/cgi-*bin/voidspace/downman.py?file=configobj.py* sólo hay que agregarlo a los módulos de Python o a la carpeta de instalación de la línea de productos de software.

- *Mako.* Mako es una librería eficiente para plantillas en Python [40]. A diferencia de otras plantillas Mako no esta basado en XML. En Pynion, es Mako quien provee los servicios de plantillas para la variación. Mako puede ser descargado desde *http://www.makotemplates.org/downloads/Mako-0.2.2.tar.gz.*

Las librerías de Pynion también pueden ser instaladas mediante *EasyInstall*[43]. *EasyInstall* es un modulo de Python para la administración de la descarga, instalación y compilación de paquetes de Python. Para instalar *EasyInstall* sólo se tiene que descargar el archivo *ez_setup.py* desde *http://peak.telecommunity.com/dist/ez-setup.py* y ejecutar desde la línea de comandos:

```
python ez_setup.py
```

Easy_Install buscar por omisión los paquetes en el *Python Package Index* o *PYPI[44].* Este repositorio esta formado por paquetes que son distribuidos por medio de archivos con extensión . *egg* [43] .Para instalar las librerías es sólo necesario ejecutar el comando de *easy_install* y el nombre del paquete a instalar. Para instalar las librerías que son requeridas por *Pynion* sería necesario ejecutar las siguientes instrucciones desde la línea de comandos:

```
easy_install ConfigObj
easy_install Mako
```

Easy_Install ofrece muchas y diversas opciones para administrar los paquetes de Python, sin embargo estas opciones quedan fuera del alcance de esta tesis.

3.3.2. Ejecución y Disposición de Archivos

Pynion esta formado por varios archivos con extensión *.py.* Estos archivos son módulos de Python. Estos archivos forman los diferentes componentes de *Pynion.* Los módulos que forman *Pynion* y su función es la siguiente:

Pynion.py. Es el programa principal de Pynion y el punto de inicio.

Classloader.py. Se encarga de cargar a memoria las clases encargadas de variar los archivos por medio de funciones.

PynionUtils.py. Utilería de funciones para el manejo de bloques de texto (para sustituirlos por otros bloques) e indentación.

ProductTrace.py. Clases encargada de mantener un registro las acciones ejecutadas para crear una aplicación miembro de la línea de productos de software.

Controller.py. Controller.py contiene la clase base para todas las validaciones en Pynion.

```
1  SubBase/Pynion.py
2            UPDATE Feature
3               ADD Base
4  readme.txt
5               ADD Base
6  ProductTrace.py
7            REMOVE Feature
8               ADD Base
```

Figura 7: Ejemplo del contenido de un archivo "output.txt"

Crear un miembro de la línea de productos de software en *Pynion* es sencillo. Lo primero es abrir una terminal o línea de comando en la carpeta donde se encuentra la línea de productos de software. Una vez hecho esto se debe ejecutar la próxima instrucción:

```
python pynion feature1 feature2 .. featureN
```

Donde *feature1, feature2, featureN* son los nombres de los rasgos (que corresponden a los nombres de las carpetas) que se desean agregar a la aplicación miembro de la línea de productos de software en el orden que se desean mezclar. Los archivos de salida pueden ser encontrados en la carpeta *output* y el registro de acciones ejecutadas se pueden encontrar en el archivo *output.txt.* El archivo *output.txt* enlista todos los archivos que se generaron en el proceso de generar un miembro de la línea de productos de software. Debajo de cada uno de los nombres se enlistan que acciones se ejecutaron *(ADD,REMOVE o UPDATE* esto es agregar, remover o actualizar respectivamente) y que rasgo fue el encargado de realizar ese cambio para generar el archivo. El archivo *output.txt* sirve entonces para poder buscar errores en el proceso de crear una línea de productos de software.

En la figura 7 vemos un ejemplo de un archivo *output.txt.* En este caso esta aplicación cuenta con tres archivos: *Pynion.py* en la subcarpeta *subBase* (en la línea 1), *readme.txt* (en la línea 4) y *ProductTrace.py*(en la linea 6). El archivo *Pynion* es agregado por el rasgo *base* (en la línea 3) y actualizado por el rasgo *Feature* (en la línea 2). El archivo *readme.txt* es agregado por el rasgo *Base* (en la línea 5) y jamás es modificado. Finalmente, el archivo *ProductTrace.py* no aparece en la aplicación final pues fue eliminado por el rasgo *Feature* (en la línea 7). Esta figura nos muestra que podemos aprender bastante del registro de variaciones hechas para crear una aplicación miembro de una linea de productos de software.

El concepto central de *Pynion* son los rasgos y como estos son incorporados a un miembro de la línea de productos de software. Los rasgos en *Pynion* están representados por archivos dentro de una carpeta. La manera en que estos archivos están organizados es básico para el funcionamiento de los mecanismos de variación. En la figura 8 podemos ver la jerarquía de un rasgo. La carpeta de cada rasgo contiene los siguientes archivos:

- *Carpeta "Add".* A partir de esta carpeta es de donde se empiezan a buscar los archivos para realizar la acción de agregar. La carpeta puede contener archivos (como *archivoAgregado1.ejemplo* y *archivoAgregado2.ejemplo*) y subcarpetas que pueden contener mas archivos formando su propia jerarquía. Todos estos archivos pueden o no ser agregados o sustituir archivos en la aplicación final, dependiendo de la configuración del rasgo y si de otro rasgo agrego el archivo con anterioridad.

- *Carpeta "Update".* La carpeta *Update* contiene los archivos para actualización median¬ te plantillas (como *plantillaActualizacion1.mako* y *plantillaActualizacion2.mako).* Estas plantillas están escritas en el lenguaje para plantillas de Mako [40] y terminan con la extensión *.mako.* Más información sobre el manejo de plantillas se puede encontrar en la sección 3.3.6.

- *archivo _ _init_ _py".* Los archivos *.py* en un rasgo forman parte de un paquete de Python[41]. Un paquete en Python es simplemente un conjunto de módulos relacionados. Todos los paquetes en Python deben de contener un archivo *_ _init_ _.py* para que carpetas con nombres comunes no interfieran con las clases estándares de Python [45]. Para mayor información acerca de la funcionalidad de los archivos *_ _init_ _.py* puedes visitar el tutorial de Python en *http://docs.python.org/tut/.*

- *Feature.ini.* El archivo *Feature.ini* contiene la configuración del rasgo (el proceso de mezclado) y las configuración extras que desee el desarrollador para el rasgo. El formato del archivo es el formato tópico de los archivos *.ini.* Más información sobre este archivo de configuración en la sección 3.3.4.

- *Root.py.* El archivo *Root.py* debe contener un clase llamada *Root.* Esta clase debe de heredarse de la clase *RootController.* Esta clase se usa en la validación de la línea de productos de software. Mas información se pueden encontrar en la sección 3.3.3.

- *Clases para Actualización.* El resto de los archivos con extensión *.py* son clases escritas en Python para manejar la actualización de archivos de texto por medio de funciones. Mas información se puede encontrar en la sección 3.3.7.

Esta estructura de archivos debe de ser repetidos para cada uno de los rasgos que se deseen agregar a una línea de productos de software. La única excepción regla es el rasgo llamado *base.* Este rasgo forzosamente tiene que existir, ya que se agrega como primer rasgos a todas las aplicaciones. Ademas este rasgo no debe de contener la carpeta *update* pues no existen archivos a modificar. Esta jerarquía a pesar de lucir compleja, permite una gran flexi¬ bilidad en el modo de variar cada rasgo de la línea de productos de software y la granularidad con la que se desea hacer estas variaciones.

3.3.3. Validación

La validación de una composición de rasgos, esto es un miembro probable de la línea de productos de software, permite restringir cuales son los productos válidos en la línea de productos de software. Esta restricción trae consigo dos beneficios: garantizar el correcto funcionamiento de todos los miembros de la línea de productos de software y limitar el número de combinaciones válidas para hacerlas mas manejables. La validación de un miembro de una línea de productos de software se realiza antes de que se ejecute el proceso de mezclado los rasgos. El proceso de validación se realiza en cada rasgo agregado a la familia de productos individualmente. Si cada uno de los rasgos del miembro de la línea de productos de software pasa la validación entonces este miembro de la línea de productos de software es valido. Ya que *Pynion* es un prototipo de una herramienta para mecanismos de variación independiente del lenguaje, este no puede asumir nada acerca de la sintaxis del documento de texto. Debido a esto es que la validación de las composiciones es completamente externa y debe correr a cargo del desarrollador de la línea de productos de software.

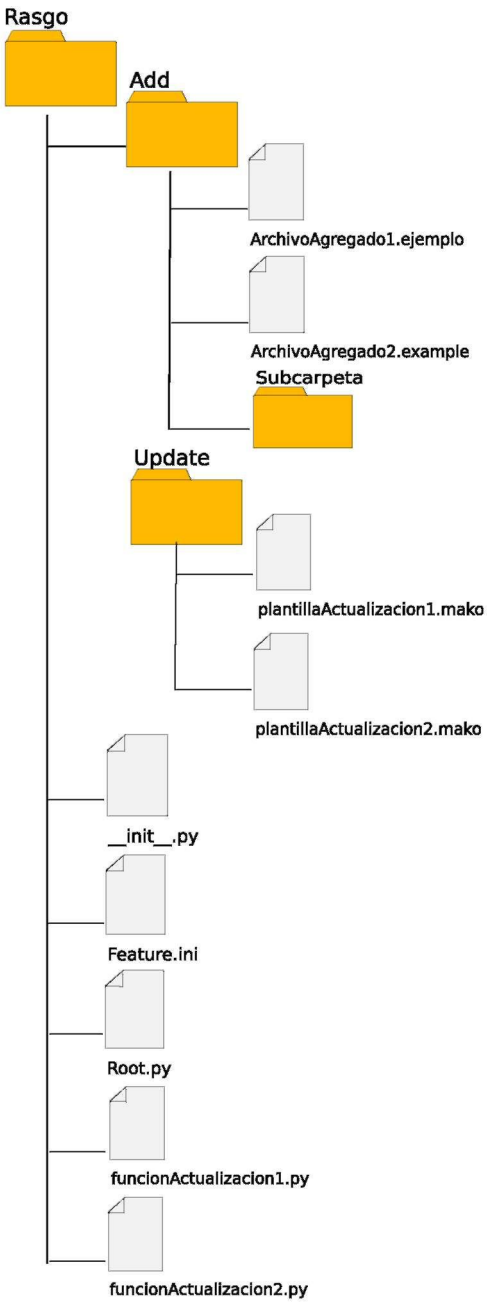


Figura 8: Jerarquía de Archivos de Pynion

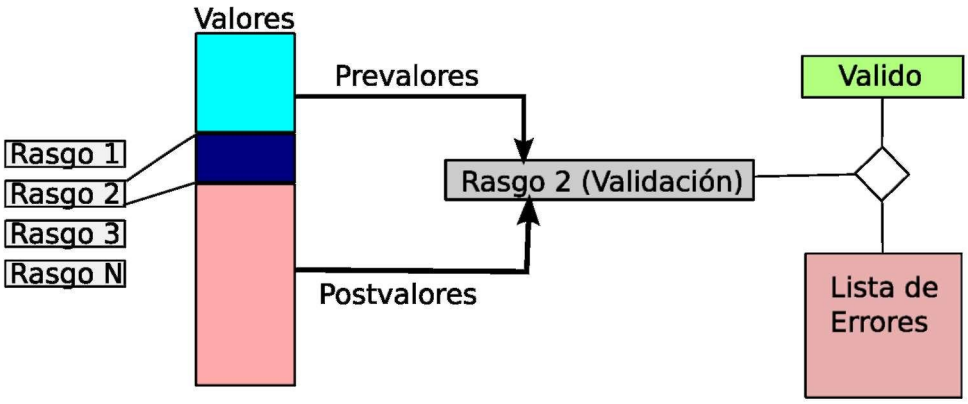


Figura 9: Proceso de Validación de Pynion

La validación en *Pynion* se realiza en dos fases: exponer valores y verificar la validez. Durante la exposición de valores cada capa puede exponer uno o varios valores que lo describen y que luego serón consumidos por las demás capas. La segunda etapa consiste en verificar la validez de cada rasgo. Para verificarlo cada rasgo recibe los valores expuestos por las capas anteriores, llamados prevalores, y los valores expuestos por las capas posteriores, llamados postvalores. Esta habilidad de poder verificar contra los valores anteriores y posteriores hace que el sistema de validación de *Pynion* sea equivalente a la técnica de validación de Batory en [32]. *Pynion* realiza estas verificaciones a través de clases de Python.

En la figura 9 vemos el proceso de validación de una composición de *Pynion.* En la figura se presenta el caso de la validación del rasgo 2. Primero vemos como el rasgo 2, al igual que el resto de los rasgos exponen sus valores. A partir de ahí vemos que los valores antes de los valores del rasgo 2 son renombrados como prevalores y los valores después de los valores del rasgo 2 son renombrados como postvalores. Los prevalores y postvalores son pasados a la validación del rasgo 2 donde el rasgo 2 decide si la composición es válida. Si la composición no fuera valida el rasgo reporta los errores que se encontraron.

Toda la validación de *Pynion* recae sobre el archivo *root.py* dentro de cada una de las carpetas de rasgo. El archivo *root.py* debe contener una clase *root* que extienda la clase *RootController* que se encuentra en el módulo *root.* Esta clase debe de contener dos métodos: *publish* y *validate.* El método *publish* deberá regresar un tupla con los valores (palabras clave) para identificar este rasgo. El método *validate* debe de regresar un arreglo con los mensajes de error (en *cadena de caracteres*) que se encontraron en este rasgo, si no se encontrase ningún error debe de regresar nulo o un arreglo vacío. La función *validate* recibe dos parámetros: *prevalues* y *postvalues. Prevalues* es una tupla que contiene los valores expuestos por los rasgos agregados al miembro de línea de productos de software antes del rasgo actual y *postvalues* es un arreglo que contiene los valores públicados por los rasgos posteriores. Si no llegase a existir este archivo o esta clase por omisión se publica como valor el nombre del rasgo y la validación regresa un arreglo vacío.

En la figura 10 podemos ver como sería un archivo *root.py* típico. Como todos los archivos *root.py,* esta contiene una clase llamada *root* que desciende de la clase base *RootController* del módulo *Root.* En este caso, esta clase publica un sólo valor: *feature* que es el nombre que identifica el rasgo (en las líneas 11 y 12). La función de *validate* hace uso del soporte

```
1  import sys
2
3 #agrega la carpeta padre donde se encuentra el archivo controlador
4  sys.path.append("..")
5
6 #importa la clase RootController del módulo controller
7 from Controller import RootController
8
9  class Root(RootController):
10     #publica el valor de "feature"
11     def publish(self):
12         return ("Feature",)
13
14     #valida que no este varias veces rasgo en la lista de "rasgos"
15     def validate(self, prevalues, postvalues):
16         errors = []
17         if "Feature" in prevalues:
18             errors.append("Feature cannot be added twice")
19         if "Feature" in postvalues:
20             errors.append("Feature cannot be added twice")
21         return errors
```

Figura 10: Ejemplo de un archivo feature/root.py

para conjuntos de Python[46](más información sombre el soporte de conjuntos de Python *en:http://docs.python.org/lib/lib.html)* para impedir que este rasgo sea utilizado más de una vez en el mismo producto miembro de la línea de productos de software (en la función *validate* de las líneas 15 a 21).

3.3.4. Configuración

Los archivos de configuración son una parte vital en el uso de *Pynion* como una herramienta para mecanismo de variación. Los archivos de configuración se encargan primordialmente de indicar como se deberá unir un rasgo con el resto de la aplicación. En Pynion los archivos de configuración deben ser escritos en el formato *.ini.* Existen dos tipos de archivos de configuración: *Pynion.ini* y *feature.ini.* El archivo *Pynion.ini* contiene la configuración a nivel línea de productos de software completa. El contenido de este archivo es libre siempre y cuando sea un archivo en formato *.ini* válido. La razón por la cual es libre es por que la configuración almacenada en este archivo es directa y únicamente pasada al código del desarrollador de la línea de productos de software y no es utilizado por *Pynion* en su proceso de mezclado de rasgos. En el archivo *Pynion.ini* se podrían almacenar, por ejemplo, la configuración de las conexiones a base de datos, ubicación de archivos o servidores que se deseen utilizar en el proceso de mezclado, etc. El archivo *feature.ini* contiene información del proce¬ so de mezclado ademas de cualquier configuración que desee agregar el desarrollador para su propio uso. Mas información sobre como los desarrolladores debe acceder a esta información

```
1 add = *.py, *.cpp
2 remove = ProductTrace.py
3 [update]
4 Readme.Readme = readme.txt
5 makoTest = makoTest.py
```

Figura 11: Ejemplo de un archivo feature.ini

de configuración se puede encontrar en la subsecciones 3.3.6 y 3.3.7.

En la figura 11 vemos un ejemplo típico de un archivo *feature.ini.* El archivo *feature.ini* contiene que acciones se van a realizar en el proceso de mezclado para incluir un rasgo en particular. Existen tres tipos de acciones: agregar, remover y actualizar. El valor de *add* contiene una lista de las trayectorias de los archivos partiendo desde la carpeta *Add* dentro de la carpeta del rasgo separados por coma. El valor de *remove* contiene la lista de las trayectorias de los archivos a eliminar de la aplicación final separadas por coma. Las trayectorias a los archivos pueden contener comodines como lo son "*" y "?" lo que lo hace equivalente a como funcionan en el sistema operativo UNIX. La tercera parte es una sección llamada *update* que permite agregar todas las modificaciones que se harán en archivos marcados con puntos de variación (mas información sobre el marcaje de puntos de variación en la sección 3.3.5). En esta sección se encuentran una serie de pares separados por un signo de igual. El primer miembro del par se refiere a la *plantilla* o *clase* que se desea utilizar para realizar las sustituciones que se harán en el archivo marcado para modificación, que es el segundo miembro del par. Pynion realiza primero una búsqueda entre las plantillas *.mako* dentro de la carpeta *update* antes de buscar una clase en el rasgo para hacer las substituciones. Por ejemplo, *Readme.Readme* se puede referir a las plantillas en el archivo en la trayectoria *update/Readme/Readme.mako* o a la clase *Readme* en el módulo *Readme.* La substitución por plantilla siempre lleva prioridad sobre la sustitución por funciones en caso de que existiesen tanto la plantilla como la clase. A partir de aquí el contenido de archivo *feature.ini* es de libre uso para el desarrollador de la línea de productos de software para incluir sus propis configuraciones pertinentes al rasgo. En las siguientes secciones veremos mas información sobre el porceso de marcaje de archivos (3.3.5) y la sustitución por plantillas (3.3.6) y funciones (3.3.7).

3.3.5. Marcaje de Archivos con Puntos de Variación

La manera ms versátil y flexible de alterar el producto miembro de una línea de productos de software a trav´és de un rasgo es utilizando la acción de actualización (esto es *update*). El uso de la acción de actualización sólo esta permitido para archivos que están marcados apropiadamente con uno o varios puntos de variación. El marcaje de un punto de variación se hace encapsulado el código que se desea modificar entre dos renglones marcados que definen el inicio y el final del punto de variación. El código o texto que se encuentra entre las líneas de marcaje es sustituido por el código provisto por el rasgo. Las marcas utilizadas sólo detectan que la línea contenga el patrón adecuado y no que la línea completa cumpla con el patrón completo. La razón por la cual la línea sólo debe contener la marca es por que esto permite que la marca este escondida como comentarios en el archivo, dejándolo el archivo de texto semánticamente intacto. La línea de marcaje de inicio debe contener la siguiente cadena de caracteres *[[[Py:Nombre]]]* donde *Nombre* es un identificador del punto de variación que debe

Python

```
1 #[[[Py:HelloWorld ]]]
2 print "hello world"
3 #[[[End]]]
```

C

```
1 /* [[[Py:HelloWorld ]]] */
2 printf( "hello world");
3 /*[[[End]]]*/
```

C++

```
1 //[[[Py:HelloWorld ]]]
2 cout<<"hello world";
3 //[[[End]]]
```

SQL

```
1 --[[[Py:HelloWorld ]]]
2 Select HELLO FROM WORLD
3 --[[[End ] ] ]
```

XHTML

```
1 <!--[[[Py: HelloWorld]]]-->
2 <div>Hello World</div>
3 <!--[[[End]]]-->
```

LateX

```
1 %[[[Py: HelloWorld ]]]
2 \emph{Hello World}
3 %[[[End]]]
```

Figura 12: Ejemplos de marcaje en diversos lenguajes

ser único en el archivo de texto. El nombre del punto de variación también indica de que plantilla o función se extraerá el código para su sustitución. La marca de final de un punto de variación debe contener la cadena de caracteres *[[[End]]]*. Estos puntos de variaciones despues son utilizados por las plantillas (3.3.6) y funciones de la clase (3.3.7). En la figura 12 podemos ver ejemplos de como funcionaria el marcaje en varios lenguajes de desarrollo comunes.

En las siguientes secciones veremos como se puede utilizar este marcaje para modificar el código de los archivos. Estas modificaciones puede ser a partir de plantillas o por funciones. En la figura 13 vemos como se altera un archivo en los puntos de variación. Existen tres operaciones diferentes que se pueden realizar: *positiva* (agregar código al punto de variación), *negativa* (remover el código del punto de variación) y *alternativa* (cambia el código del punto de variación por código completamente nuevo). En la figura podemos apreciar como afectan este tipo de modificaciones al archivo final.

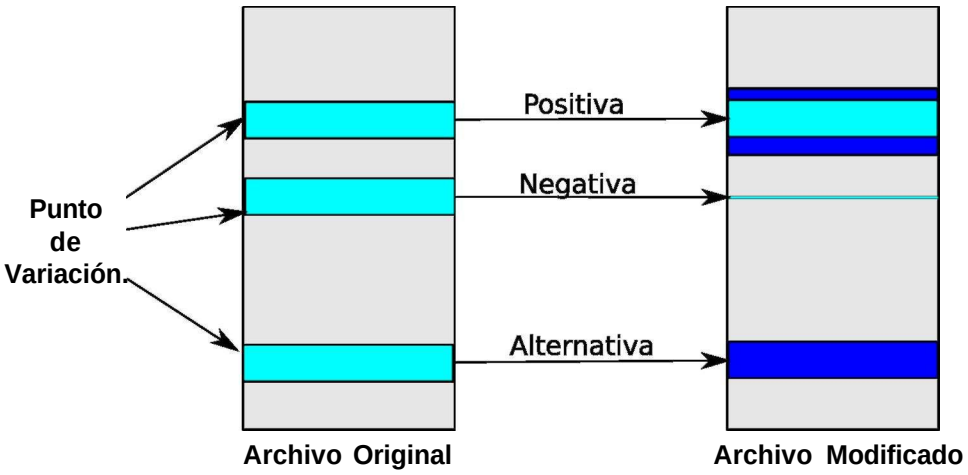


Figura 13: Modificación de archivos en puntos de variación

3.3.6. Variación por Plantillas

Al buscar un método de actualización Pynion primero busca en la carpeta *update* un archivo con extensión *.mako* que corresponda con el configurado en el archivo *feature.ini.* Si logra encontrar el archivo con extensión *.mako* este sera utilizado para la actualización. Las plantillas de actualización deben ser escritas en el lenguaje de programación Mako [40]. Mako fue desarrollado a su vez en Python y esa es la razón por la cual su sintaxis es similar a la de este. Una de las virtudes de Mako es su poder de definir varias plantillas dentro de un mismo archivo. Estas plantillas se identifican por un nombre, este nombre es el mismo con el que identificamos los puntos de variación en los documentos de texto. El resultado de la evaluación de las plantillas es sustituido en los puntos de variación originales.

La salida de las plantillas en muchas ocasiones necesita poderse adaptar dependiendo de los rasgos anteriores, posteriores y en particular del contenido acumulado actual del documen¬ to. Para lograr esto, las plantillas reciben como parámetros los bloques de textos extraidos en los puntos de variación. Los nombres de los parametros corresponden directamente a los que se les dio en el punto de variación pero incluyendo el prefijo 6_.Además de esto, las plantillas define un objeto de la clase *FileContext* llamado *c* que le da a la plantilla información sobre el contexto en que se esta ejecutando. Los bloques originales y el contexto de ejecución ayudan al desarrollador a ajustar la modificaciones hechas por el rasgo a la composición de la línea de productos de software.

El parámetro *c* contiene los siguientes atributos:

- *name.* Es el nombre de la plantilla y punto de variación que se esta ejecutando.
- *featureName.* Es el nombre del rasgo que se esta ejecutando.
- *config.* Es un objeto del tipo ConfigObj [42] que representa la configuración encontrada en el archivo *Pynion.ini* que es compartido con toda la línea de productos de software.
- *featureConfig.* Es un objeto del tipo ConfigObj [42] que representa la configuración encontrada en el archivo *feature.ini* de este rasgo.

- *blocks.* Es un diccionario con los bloques que se extrajeron de los puntos de variación del archivo original (estos también pueden ser accedidos directamente como parámetros).
- *toplayers.* Son los rasgos declarados a ejecutarse despues del rasgo actual.
- *bottomlayers.* Son los rasgos que se han ejecutado antes del rasgo actual.
- *prevalues.* Son los valores publicados por los rasgos anteriores al actual (ver subsección 3.3.3 para más información).
- *postvalues.* Son los valores publicados por los rasgos posteriores al actual (ver subsección 3.3.3 para más información).

Ademas de esos atributos el parámetro *c* del tipo *FileContext* también contiene las siguientes funciones:

- *named (nombre).* Transforma el parámetro *nombre* al formato *nombre_featureName#* donde *featureName* es el nombre de rasgo actual y *#* es la posición del rasgo en la combinación del producto actual. Este cambio nos permite obtener nombres únicos que pueden ser reutilizado en la plantilla sin posibilidades de que colisionen con nombres dentro de otros rasgos.
- *snamed(nombre).* Transforma el parámetro *nombre* al formato *nombre-#* donde # es la posición del rasgo en la combinación del producto actual. La función *snamed* puede ser utilizada cuando se quiere ahorrar el máximo de caracteres en la aplicación final.

Finalmente existen varias funciones que puede ser utilizados como filtros en Mako. Los filtros en Mako son funciones de Python que reciben un *string* como parámetro y regresa otro *string* como parámetro. Los filtros son utilizados con el operador "—" [47] (Mas información sobre los filtros de Mako se puede encontrar en *http://www.makotemplates.org/docs/*). En *Pynion* existen los filtros disponibles en las plantillas de *Pynion* son:

- nm. *nm* es una manera abreviada de usar *c.named* para crear nombres únicos asociados al rasgo.
- sn. *sn* es una manera abreviada de usar *c.snamed* para crear nombres únicos cortos asociados al rasgo.
- i. *i* representa la palabra *indentación* y se utiliza para corregir la indentación de los bloques de texto extraídos del original para que ajusten con la indentación de la plantilla. Esto puede ser útil por dos razones: lenguajes como Python donde los espacios en blanco son significativos y para darle una mejor vista al código final para facilitar su lectura y depuración.

En la figura 14 vemos como se puede marcar un archivo para ser modificado. En la figura 15 vemos una plantilla diseñada para el documento anterior. En esta figura vemos que se puede utilizar la función *named* de *FileContext* en varías formas. **`${c.name('main')>*`** y **`${'main' |nm}*`** son equivalentes ya que la segunda útiliza la función *name* con la sintaxis de filtro de Mako. Por último en la figura 16 vemos el resultado de procesar el archivo original con la plantilla. Aquí podemos ver que *Pynion* conserva la correcta indentación del documento.

En esta subsección se ha demostrado como *Pynion* puede trabajar con Mako para ofrecer variaciones por medio de plantillas. Además vimos que *Pynion,* a través de plantillas, ofrece

Documento Original

```
1  public string HelloWorld()
2  {
3      //[[[Py: HelloWorld ]]]
4      string s = "Hello World";
5      //[[[End]]]
6      return s;
7  }
8
9  public void main()
10 {
11     //[[[Py:Main]]]
12     print(HelloWorld());
13     //[[[End]]]
14 }
15 //[[[Py:Repos]]]
16 //[[[End]]]
```

Figura 14: Ejemplo de un archivo marcado con puntos de variación

Plantilla Mako

```
1  <%def name="HelloWorld()">\
2  ${b_HelloWorld | i}\
3  s += "Alex";
4
5  </%def>
6  <%def name="Main()">\
7  ${'main'|c.named}();
8  ${c.named('main')}();
9  </%def>
10 <%def name="Repos()">\
11 public void ${'main'|nm}()
12 {
13     ${b_Main | i}\
14 }
15 </%def>
```

Figura 15: Ejemplo de un archivo de extensióon *.mako*

Documento Resultado

```
1   public string HelloWorld()
2 {
3       //[[[Py: HelloWorld ]]]
4       string s = "Hello World";
5       s += "Alex";
6       //[[[End]]]
7       return s;
8 }
9
10  public void main()
11  {
12      //[[[Py:Main]]]
13    main_Feature0   ();
14    main_Feature0   ();
15      //[[[End]]]
16  }
17  //[[[Py:Repos]]]
18  public void main_Feature0 ()
19  {
20      print(HelloWorld());
21  }
22  //[[[End]]]
```

Figura 16: Resultado de procesar un archivo con una plantilla

funcionalidad para mantener la correcta indentación del documento, esto es siempre útil, pero es esencial en lenguajes como *Python* donde la indentación es significativa. Mako es un lenguaje que encapsula todo el poder de Python, lo cuál da gran flexibilidad al desarrollador. A pesar de que *Pynion* depende de Mako para las plantillas la descripción completa de la funcionalidad de Mako esta fuera del alcance de este documento.

3.3.7. Variación por Funciones

La manera más flexible en que *Pynion* permite hacer modificaciones es a través de *funciones.* Cada archivo en Pynion puede ser asignado a una clase dentro del rasgo. Cada bloque del rasgo original es procesado por una función y remplazado por su resultado. El desarrollo de las variaciones por medio de *funciones* es más lento que el desarrollo por medio plantillas, sin embargo también permite al desarrollador aún mas control sobre el resultado final. La ma¬ nera en que funciona la variación por funciones es la siguiente: Antes de llamar a las funciones se crea un instancia de la clase que contiene las funciones. Después, sobre esta instancia, se llaman las funciones para cada bloque (las funciones y los bloques correspondientes deben de compartir el nombre) pasando un parámetro llamado *context.* El objeto *context* es un objeto del tipo *FileContext* que se discutió a profundidad en la subsección 3.3.6. A diferencia del caso de variación por medio de plantillas, no se tiene acceso a la función de *indentación,* ni los atajos para acceder a las funciones *named* y *snamed,* ni a los bloques de texto renombrados con el sufijo 6_. El *string* que regresa la función es es texto que será sustituido en el archivo original en lugar del bloque marcado. En la figura 17 vemos un ejemplo de una clase que se utiliza para realizar las variaciones en *Pynion,* esta es relativamente equivalente a la plantilla mostrada en 15 (exceptuando la indentación). En la figura 17 podemos ver que las funciones no son tan fácilmente legibles como la plantilla de la figura 15, pero tiene acceso a todo el poder del lenguaje *Python* y sus librerías. Así, la variación por *funciones* sólo se recomien¬ da cuando se necesita la mayor flexibilidad posible y acceso a la funcionalidad avanzada de Python.

3.4. Detalle del Código

"El diablo esta en los detalles"

Dicho americano

En esta sección se revisará con mayor detenimiento ciertas secciones y fragmentos de las clases y el código fuente que forman parte de *Pynion.* Esto nos ayudará a visualizar como funciona la aplicación a profundidad. Los segmentos de código escogidos para esta sección fueron seleccionados por ser los que su conocimiento aporta mayores beneficios al desarrollador de líneas de productos de software utilizando *Pynion.* Para entender el código a mostrar en esta sección no es necesario estar familiarizado con el lenguaje de programación *Python* o sus librerías, sin embargo este conocimiento recomendable.

A continuación vemos el contenido del archivo *RootController.py*:

Archivo Controller.py (Clase RootController)

```
1 class RootController:
2     """
3     Clase que es un controlador Root que controla la validación
4     """
```

Variation.py

```
1  class Variation
2          def HelloWorld(self, context):
3                  return context.block['HelloWorld'] += "s+='Alex;'\
                       n"
4
5          def Main(self, context):
6                  mainCall = context.named('main') + '();\n'
7                  return mainCall + mainCall
8
9          def Repos(self, context):
10                 retval = 'public void ' + context.named('main') +
                       "()\n{"
11                 retval += '    ' + context.blocks['main'] + '\n'
12                 retval += '}'
13                 return retval
```

Figura 17: Ejemplo de una funcion para sustituciún en Pynion

```
5      def publish(self):
6          return (self.__name__,)
7
8      def validate(self,prevalues, postvalues):
9          return []
```

El archivo *Controller.py* contiene la clase *RootController*. *RootController* es la clase base de donde descienden todas las clases que realizan todas las validaciones de rasgos en *Pynion*. Como vemos, la implementación por omisión de la función *publish* es publicar el nombre de la clase donde se encuentra. Esta es la función que se utiliza si se crea una clase descendiente de *RootController* pero no se implementa el método *publish*. Como visto antes, si no se crea esta clase, el valor por omisión es el nombre del rasgo. En el caso de la función *validate* el valor de retorno es un arreglo vacío, esto es, no hay errores por omisión en la función.

A continuación veremos la función *execute* que se encuentra en la clase *FeatureEngine* en el archivo *Pynion.py*:

FeatureEngine.execute en Pynion.py

```
1        def execute(self):
2            """
3            Ejecuta todos los cambios en la aplicación para este rasgo
4             """
5            self.__addFiles()
6            self.__removeFiles()
7            self.__updateFiles()
```

Este segmento de código pertenece a la clase *FeatureEngine*. La clase *FeatureEngine* es una clase interna a *Pynion* que representa los cambios que va a realizar un rasgo. En el método *execute* vemos en que orden se aplican los cambios. El orden es: primero se agregan archivos (línea 5), segundo se eliminan archivos (línea 6) y tercero se actualizan los archivos (línea 7).

No es recomendable que los cambios de un rasgo dependan del orden en que se aplican los diferentes tipos de variaciones; pero se puede realizar, sin embargo, si así se desea.

El siguiente segmento de código es la función *identBlock* que puede ser accedida por medio del filtro *i* desde las plantillas *Mako*:

función identBlock en Pynion.py

```
1       def indentBlock(self, text):
2           """Indenta un bloque para sustituirlo en una plantilla """
3           lines = text.splitlines(True)
4           indentation = self.getIndent()
5           return indentation.join(lines)
```

La función *identBlock* pertenece a la clase *PynionBuffer*. Las instancias de *PynionBuffer* capturan la salida de cada plantilla. La función *indentBlock* prepara el bloque de texto a indentar, para que conserve la indentación de la plantilla.

En esta secciones hemos revisado algunos pequeños fragmentos del código de *Pynion*, para entender mejor su funcionamiento. Con esta información tenemos mayor capacidad de entender y prever el comportamiento de *Pynion* en más situaciones.

3.5. Bases de Pynion

"Si logre ver más lejos, es por que estoy parado en los hombros de gigantes"

Isaac Newton

En esta sección revisaremos los cimientos sobre los cuales esta construido *Pynion*. Además de los conceptos revisados en esta sección y en la anterior, *Pynion* toma concep¬tos e ideas de otros proyectos y herramientas. *Pynion* también utiliza directamente ciertos lenguajes y herramientas ya existentes actualmente en parte de su desarrollo. En esta sección justificaremos la selección de estas herramientas e ideas que se han escogido para el desarrollo de esta tesis. Para cada una mencionaremos cuales son sus virtudes de estas, mencionaremos cuales son esenciales para la herramienta y finalmente como estas encajan entre los demás componentes de *Pynion*.

3.5.1. Python

Python es un lenguaje dinámico, orientado a objetos que puede ser utilizado para muchos tipos de desarrollo de software [41]. Python es conocido como un lenguaje de *scripting* por su facilidad de uso y su habilidad de trabajar como "pegamento" para componentes [48]. Python es el lenguaje sobre el cuál se desarrollará *Pynion*. Dado las expectativas que se tienen del funcionamiento de *Pynion* hay tres cualidades de las cuales se puede sacar provecho de Python: "interpretado", integración de componentes, portabilidad. Python no es en realidad interpretado, es compilado, sin embargo al ser compilado automáticamente cada vez que se realiza un cambio en el contenido de un módulo, hace que parezca comportarse como si fuera un lenguaje interpretado. *Pynion* utiliza este poder de interpretación para ejecutar tanto plantillas (que veremos a más profundida en la subsección 3.5.2) como funciones que generen código, que pueden ser constantemente modificadas sin requerir recompilación de la aplicación. Tradicionalmente, el código interpretado es más lento que el compilado, y esto también es cierto para Python, sin embargo, hay que recordar que las aplicaciones generados por *Pynion*

no sufrirán de problemas de velocidad pues estas deberán ser compiladas aparte. La segunda razón para escoger Python sobre otros lenguajes es la facilidad de integración de componentes. El propósito de *Pynion* es ser una herramienta para mecanismos de variación muy abierto para soportar todos los tipos de variaciones posibles y todas las estrategias de líneas de productos existentes. Con este objetivo en mente, la capacidad de Python de interactuar con muchos lenguajes y de tener una librería estándar muy completa, permitirá a las líneas de productos de software creadas en *Pynion* tener acceso a muchísima funcionalidad para generar el código dependiente de una gran cantidad variables(como pueden ser esquema de base de datos, configuraciones de red, etc). La tercera razón es la portabilidad, Python es capaz de correr en una gran cantidad de sistemas operativos diferentes, manteniendo en en todos la gran mayoría de la funcionalidad intacta. Esta portabilidad es bien recibida por *Pynion* ya que le permite trabajar en un ambiente heterogéneo, esto es, en ambientes desarrollados en varios o para varios sistemas operativos a la vez. Estas virtudes descritas de Python son la razón por la cual fue escogido como el lenguaje de programación para el desarrollo de *Pynion* como una herramienta para mecanismo de variación.

3.5.2. Plantillas Mako

En *Pynion* se usan las plantillas para poder modificar los documentos de manera sencilla. La parte mas importante para juzgar el motor de plantillas a utilizar es que la sintaxis de la herramienta de plantillas sea lo menos invasiva posible. Existen varias librerías para el manejo de plantillas en Python entre ellas se encuentran: Mako[40], Cheetah[49] y Genshi[50]. El más antiguo de los tres es Cheetah y su edad se empieza a notar en su diseño ya que tiene una sintaxis relativamente invasiva. Genshi por otro lado esta diseñado exclusivamente para XML y HTML, por lo cuál no podría funcionar dentro ambientes verdaderamente heterogéneos sin problemas. En cambio Mako tiene una sintaxis poco invasiva y soporte para varias plantillas en un solo archivo de texto, lo que lo hace ideal para las exigencias existentes de *Pynion.* Por esto, Mako resultó ser el motor de plantillas que mejor se acomoda al diseño de *Pynion.* La documentación completa del uso de Mako se puede encontrar en *http://www.makotemplates.org/docs/* [47].

3.5.3. Cog

Cog[51] es una herramienta para generación de código desarrollada en Python. Cog no sera utilizado directamente en *Pynion* pero varias de las ideas de este si serán implementadas directamente. Una de ellas es el usar marcadores de linea completa para marcar el código fuente que se quiere sustituir. Al usar la línea completa como identificador versus solo un fragmento de ella, en Cog, es posible ocultar los marcadores entre los comentarios y mantener el código de Python embebido detrás de ellos. Este proceso sera introducido a *Pynion* para poder introducir los puntos de variación en un documento ya existente de la manera menos invasiva posible y a la vez permitiendo que todas las herramientas de desarrollo sigan funcionando sin contratiempos. Otra virtud de Cog, que es virtualmente exclusiva para su uso con Python, es que mantiene un sistema de indentación y dedentación controlada. Este control de la indentación permite generar código para documentos o lenguajes de programación donde los espacios en blanco pueden ser significativos. También *Pynion* tratara de mantener la indentación y dedentación de las plantillas a través de una funcionalidad similar a la de Cog. Así vemos que el principal aporte de Cog a *Pynion* es permitir que los puntos de variación

en un documento sean introducidos de manera poco invasiva y de manera genérica de modo que pueda funcionar en cualquier lenguaje que contenga soporte para comentarios.

3.5.4. Reglas de Diseño

Pynion busca también permitir una validación y limitación de las composiciones de las aplicaciones de la línea de productos de software. Con este objetivo, *Pynion* toma ideas de los generadores GenVoca, las modifica y los expande. En los generadores existen cuatro conceptos importantes de validación: Postcondiciones y precondiciones, postrestricciones y prerrestricciones [32]. Las postcondiciones se refieren a los valores que expones a las capas inferiores. Las precondiciones son las condiciones que debe cumplir los valores de las capas superiores. Las postrestricciones son valores que se exportan a las capas superiores. Y las prerrestricciones son valores que se debe de cumplir de las capas inferiores. El sistema que esta integrado a *Pynion* funciona permitiendo que cada rasgo pueda exportar cuantos valores desee tanto para los rasgos anteriores como los rasgos posteriores. Ademas cada rasgo recibirá los valores publicados por las demás y la posición de la capa en que los valores fueron publicados. Esta información servirá para que cada rasgo verifique la validez de la composición y reporte cualquier error que haya ocurrido en la selección, limitando así, efectivamente, el número de composiciones factibles dentro de la línea de productos de software. Es por esto que, gracias al soporte para conjuntos *(sets* en inglés) de Python, la complejidad de las ecuaciones no aumenta demasiado en comparación de los generadores, pero si hace más profunda el conjunto de validaciones que se puede lograr hacer en *Pynion.*

3.5.5. Juegos de Archivos y Juegos de Cambios

Pynion busca como parte esencial el lograr la mayor reutilización posible del código fuente ya existente. Esto permitirá reutilizar recursos legados ya creados con anterioridad en otros proyectos de software. Con este objetivo es que *Pynion* permitirá agregar, eliminar y sustituir archivos completos desde cualquier rasgo. Esta estrategia es similar a la de AHEAD Tool Suite que permite agregar cualquier tipo de archivo a un componente[6] pero va mas allá al permitir sustituciones y remociones de archivos enteros. Esto permite el poder agregar y eliminar aplicaciones completas de un sistema, en caso de ser necesario por la selección de un rasgo, a un costo bajo ya que no se necesita modificar los archivos originales. Ademas de esto, en *Pynion,* se puede agregar clases auxiliares a librerías ya existentes que sean especificas de un sólo rasgo. Finalmente, aceptar archivos completos permite que se agreguen archivos binarios a la línea de productos de software, que de otro modo no sería factible incluir. Esto permite a *Pynion* a soportar parte de la heterogeneidad normal de un ambiente de desarrollo.

Otro concepto importante para *Pynion* es la automatización del proceso de creado de los diferentes rasgos. El propósito de esto, es que los programadores puedan cambiar sustitu¬ciones de archivos completos por sustituciones en sólo ciertos puntos de variación de manera automática. Henrickson y Van der Hoek en [52] analizaron ya el uso de juego de cambios (*Change Sets* en ingles) para la creación de lineas de productos a partir de recursos ya exis¬tentes. Además del concepto de juegos de cambios, Henrickson y Van der Hoek, también incluyen el concepto de relaciones que permite agrupar este grupo de cambios para limitar las composiciones factibles. A pesar de que *Pynion* no incluye estos conceptos como parte directa de la estrategia de creación de líneas de productos de software, si los tomará en cuenta como una manera de simplificar la transición del trabajo de desarrollo centrado en aplicaciones

hacia estrategias centradas en líneas de productos de software.

3.6. Resumen

Este capítulo nos da una profunda descripción de las características de Pynion. Este análisis incluyo una revisión minuciosa de como utilizar la funcionalidad completa de *Pynion.* Además en esta sección hace referencia a las demás personas, proyectos e ideas que se tomaron para la construcción de *Pynion.* Sin embargo, es importante probar que la herramienta para mecanismos de variación pueda funcionar con una herramienta útil en el desarrollo de líneas de producto de software reales. En el siguiente capítulo utilizaremos *Pynion* en el proceso de desarrollar una línea de productos de software en el campo de la vigilancia remota.

4. Línea de Productos de Software:Vigilancia Remota

En esta sección analizaremos el desarrollo de una línea de productos de software utilizando *Pynion* como una herramienta para mecanismos de variación. El objetivo de este capítulo es evaluar a *Pynion* como una herramienta que facilite el desarrollo de líneas de productos de software en ambientes heterogéneos. Para lograr esto desarrollaremos una línea de productos de software y la implementaremos utilizando *Pynion.* Una vez hecho esto analizaremos y evaluaremos tanto el resultado, como el proceso de desarrollo y así ver si *Pynion* es una herramienta para mecanismos de variación viable. Con estos objetivos desarrollaremos una línea de productos de software en el dominio de la *vigilancia remota.*

El dominio de la vigilancia remota y su línea de productos de software correspondientes sera analizada a profundidad en este capitulo. En la primera sección se justificara la selección de estos para esta tesis. En la segunda sección describirá el proceso de desarrollo que se siguió para esta línea de productos de software utilizando *Pynion.* Después veremos cuales son los tipos de variación encontrados en las aplicaciones miembros de la línea de productos de software y ejemplificaremos como se codificaron estas en *Pynion.* Finalmente, identificaremos los patrones de diseño que aparecieron durante el desarrollo de la línea de productos de software, los generalizaremos y describiremos a gran detalle. Al final de este ejercicio debe ser factible evaluar las virtudes y los límites de *Pynion* como una herramienta para mecanismos de variación, incluso al ser tan sólo un prototipo.

4.1. Justificación

"El gran hermano te esta observando"

1984, George Orwell

En la sección 1.2 vimos como las diferentes aplicaciones en el dominio de la vigilancia remota fueron la motivación para la creación de *Pynion.* Además de las razones ya vista en la sección *MOTIVACIÓN* hay buenas razones para desarrollar nuestra primera línea de productos de software con *Pynion* en este dominio. Una de las razones principales es que ya hemos diseñado varias aplicaciones de la línea de productos de software. Esto nos da varios beneficios: la certeza que existe una línea de productos latente en este dominio y nos da recursos que podemos reutilizar para crear la línea de productos de software. Esta son las razones por las cuales el dominio de la vigilancia remota es un buen lugar para comenzar una línea de productos de software.

A medida que analizamos las aplicaciones que pertenecen a este dominio nos damos cuenta de algo importante en sus características: el ambiente de vigilancia remoto es un ambiente heterogéneo. Una de las razones para esta heterogeneidad es el hardware; en particular cuando nos referimos a las cámaras. Las diferentes necesidades que tienen las aplicaciones de vigilancia remota pueden utilizar cámaras con características muy diferentes. Las cámaras pueden tener diferentes cualidades, por ejemplo existen cámaras digitales fotográficas, cámaras de vídeo y *webCams*; unas cámaras tienen auto-enfoque y alta resolución mientras otras están carentes de ambas. Además las diferentes cámaras pueden tener diferente objetivo por ejemplo puede haber cámaras de visión nocturna (por ejemplo para captar imágenes de animales nocturnos), cámaras con capacidad de movimiento (para vigilancia de seguridad de bancos, tiendas departamentales y demás áreas abiertas), cámaras de detección de movimiento (para vigilancia de seguridad no supervisada) y otros. Las cámaras también pueden variar

en como capturan y se puede extraer las imágenes y vídeos de ellas. Por ejemplo, existen cámaras que se conectan al puerto USB como son las *webcams* o existen otras que se conectan a la red *Ethernet.* En estas cámaras incluso puede variar como son almacenadas y trasmitidas las imágenes que puede ser en formato *RAW* o *JPEG* u otros. El manejo de cada una de las variaciones en los tipos de cámaras provoca variaciones significativos en el código fuente y la configuración de las aplicaciones diseñadas para trabajar en el dominio de la vigilancia remota.

Otro punto de variación importante dentro del dominio es que se desea hacer y que se puede hacer con las imágenes ya capturadas. Por ejemplo, se puede desear únicamente almacenar las imágenes localmente, se puede querer almacenar remotamente, se puede desear la compartición de imágenes como un servicio (a través de un servidor web o un servicio web) o incluso una combinación de estas opciones. Además la disposición de los servidores o la presencia de *firewalls* puede restringir que protocolos y puertos de comunicación se pueden utilizar por la aplicación; estos pueden ser, por ejemplo, *http, ftp, smtp* o *smb* (el protocolo de compartición de archivos de Windows). Ademas el equipo que desee recibir o trasmitir las imágenes pueden tener diferentes tipos de sistemas operativos como pueden ser Windows, Unix y sus variantes, Linux, MacOX u otros sistemas operativos portátiles. Toda esta variación también impacta fuertemente en el código fuente de las aplicaciones del dominio de la vigilancia remota.

Una de las virtudes de la línea de productos de software para la vigilancia remota es que es un dominio fácil de identificar. La vigilancia remota es algo común y sus premisas básicas son conocidas, permitiendo que entender secciones selectas del código sean más fáciles de entender. Una segunda virtud, es que las aplicaciones base de donde se tomaron se utilizan diversos lenguajes de programación en el código fuente y formatos para almacenar configuraciones. Entre los lenguajes de desarrollo de las aplicaciones legado incluyen: *Python, C#, .ini, xml, html, css, javascript* y *aspx(aspx* es una combinación de *html* y plantillas). Un lenguaje que merece una mención particular es *Python.* El lenguaje *Python* es uno de los pocos lenguajes de programación en que los espacios en blanco son significativos. Tradicionalmente, generar código fuente para este tipo de lenguajes de programación a través de plantillas no es sencillo pues el uso plantillas suelen introducir o remover espacios en blanco en un código fuente. Esto hace que, si se puede usar *Pynion* para generar archivos de *Python,* se demuestra la versatilidad de esta herramienta para mecanismo de variación.

Las características de heterogeneidad de hardware, software y sistema operativo son la principal razón por el cual se seleccionó el dominio de la vigilancia remoto para este estudio. Además las aplicaciones del dominio de vigilancia remota son bastante conocidas. Esta fácil identificación es lo que lo hace ideal para presentar en esta tesis. Finalmente, el uso de diferentes lenguajes de desarrollo de las aplicaciones originales (de donde tomaremos recursos para la línea de productos de software) hace que se pruebe verídicamente que *Pynion* puede ser una herramienta para mecanismos de variación para líneas de producto de software en ambientes heterogéneos.

4.2. Desarrollo

Para el desarrollo de una línea de productos de software es necesario un buen conoci¬ miento del dominio de las aplicaciones. El máximo provecho de las estrategias de líneas de productos de software sólo se obtiene cuando uno esta extremadamente familiarizado con las aplicaciones del dominio que se quieren diseñar. El conocimiento del dominio permite saber

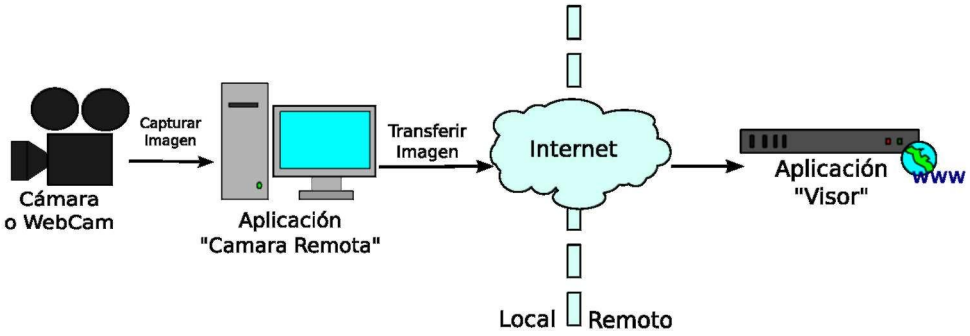


Figura 18: Arquitectura de la línea de productos de software para la vigilancia remota

que partes de una línea de productos de software son iguales para todos los productos y cuales puedan cambiar. Con el fin de desarrollar una línea de productos de software de calidad se escogió una con la cual se tenia una familiaridad previa: la vigilancia remota. Un par de aplicaciones de vigilancia remota sentaran las bases para esta línea de productos de software. Las aplicaciones fueron diseñadas para trasmitir imágenes capturadas de una de seguridad de un autobus a una aplicación web. Existieron dos versiones de esta aplicación, una trabajaba con una *webCam* y otra trabajaba con una *cámara ip*. El hecho de la existencia de dos aplicaciones de vigilancia remota tan similares descubre la existencia de una línea de productos de software subyacente.

El primer paso, y probablemente el más importante de todos, en la creación de una línea de productos de software es el análisis del dominio donde se crearan los productos. En la figura 18 vemos el funcionamiento esperado de los miembros de la línea de productos de software basados en las aplicaciones ya existentes. En esta figura podemos identificar tres componentes: *cámara*, la aplicación de la *cámara remota* y la aplicación *Visor*. La *cámara* se refiere al hardware que se utilizara como fuente de vídeo y desde donde se realizara la captura de imágenes. La aplicación de la *cámara remota* se encarga de obtener una serie de imágenes y transferirlas a la aplicación *Visor*. Finalmente la aplicación *Visor* se hace cargo de almacenar y desplegar estas imágenes para el usuario. Estos son los componentes esenciales de la línea de productos de software para la vigilancia remota.

En la figura 18 también podemos vislumbrar algunos lugares donde la variabilidad se puede presentar en la línea de productos de software. Como visto a partir de las aplicaciones base de vigilancia remota, la variabilidad se puede presentar en el momento de la captura de imágenes. Las aplicaciones que se habían desarrollado con anterioridad capturaban sus imágenes de dos diferentes tipos de cámaras. Una trabajaba con *webCams* y otras con *cámaras IP*. Otro lugar donde se puede presentar la variación es durante la trasmisión de estas imágenes pues existen diversos formas de transferir imágenes por medio de una red y varios de estos métodos podrían pertenecer a la línea de productos de software para la vigilancia remota.

Una vez que se conoce un dominio es momento de pasar al diseño de la línea de productos de software. El primer paso es identificar los recursos ya existentes y factorizarlos en los rasgos de la aplicación. En este caso, sabemos que existen dos aplicaciones dentro de la línea de productos de software: la aplicación de la *cámara remota* y la aplicación *Visor*. Una vez esto hecho esto podemos empezar a descubrir ciertos rasgos que puede contener la aplicación. Uno de los rasgos, que son mutuamente exclusivos, es el tipo de cámara a utilizar: *USB* (o

webCam) o *cámaras ip*. Una vez hecho esto es necesario complementar los rasgos actuales de las aplicaciones conocidas del dominio con los rasgos deseables de la línea de productos de software.

Una vez que se identificaron los rasgos conocidos de la línea de productos de software es necesario agregar todos los rasgos que deseamos que contenga la línea de productos de software. En la figura 19 vemos el diagrama de rasgos de la línea de productos de software para la vigilancia remota. Este diagrama de rasgos esta basado en la figura 4 de Czarnecki [20]. En este diagrama los círculos blancos representan rasgos opcionales, los círculos negros representan los rasgos obligatorios. En los diagramas propuestos por Czarnecki existen dos tipos de grupos de rasgos que son marcados por un arco. Si el arco no es relleno, entonces este grupo es un grupo alternativo, esto es, se debe seleccionar uno y a lo más uno de los rasgos del grupo (Como en el caso de los nodos descendientes del rasgo *captura de imagen*). Si el nodo es relleno, entonces se puede seleccionar uno o varios de los rasgos en la composición a la vez (como los descendientes del rasgo de *transferencia de imágenes*). De este modo este diagrama representa en su totalidad a la línea de productos de software para la vigilancia remota.

En la figura 19 empezamos por analizar los rasgos directamente dependientes del nodo padre (el nodo de vigilancia remota) del diagrama de rasgos. Los rasgos *cámara remota* y *Visor* se refieren a las dos aplicaciones que forman la línea de productos de software. En el diagrama podemos ver que la aplicación de la *cámara remota* es obligatoria mientras que la aplicación *Visor* es opcional. Esto se debe a que la aplicación tiene que capturar las imágenes de la cámara necesariamente, pero el destino de las imágenes no es forzosamente la aplicación *Visor*. La aplicación de la *cámara remota* puede ser modificado por tres rasgos: *timer, captura* y *transferencia. Timer* se refiere a si se harán pausas entre cada captura de imagen. El rasgo *captura* se refiere al tipo de hardware de la cámara que se quiere utilizar. Finalmente el rasgo de *transferencia* se refiere al protocolo que se utilizara para transferir las imágenes obtenida de la cámara. En la aplicación *Visor* existen dos rasgos, uno se refiere al almacenamiento (como se van a grabar las imágenes en el servidor) y otro rasgo de seguridad para agregar autorización y autentificaron de usuarios. Además de estos rasgos también existe el rasgo de *Move* que permite mover la dirección a la que apunta la cámara. En el diagrama tan bien se incluyen ciertas reglas para crear las composiciones válidas para esta línea de productos de software. Es importante notar que los rasgos descritos en este diagrama son rasgos lógicos y los rasgos finales que se van a agregar a la línea de productos de software pueden ser levemente diferentes. En la siguiente secciones veremos los rasgos reales que se generaron a partir de este diagrama y una descripción detallada de estos.

Una vez realizado el diagrama de rasgo es momento de empezar a trabajar sobre el código de la línea de productos de software. Para usar *Pynion* se debe de comenzar la línea de productos de software con una refactorización de las aplicaciones originales con dos objetivos. El primero es reestructurar la aplicación y marcar los diferentes puntos de variación (más información en la sección 4.4.3) donde se desea adaptar la línea de productos de software para los diversos rasgos. Al final de cumplir con este primer objetivo de la refactorización tendremos el rasgo *base* listo (El rasgo *base* siempre se agrega a la composición y contiene la estructura básica de cualquier miembro de la línea de productos de software). El segundo objetivo de la refactorización es desensamblar las aplicaciones originales en sus diferentes rasgos. Una vez cumplido este objetivo se debe contar con los rasgos suficientes para formar las composiciones necesarias para formar aplicaciones miembros de la línea de productos de software equivalentes en funcionalidad a las aplicaciones originales del dominio conocido.

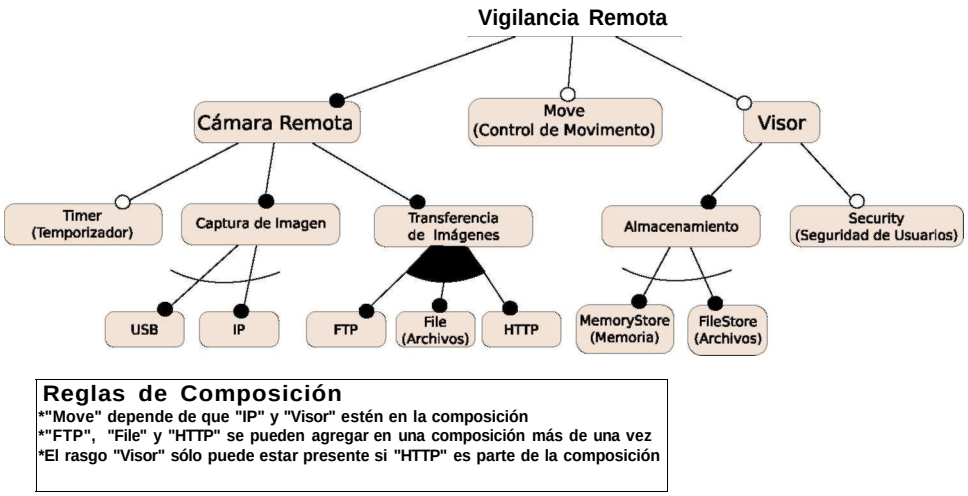


Figura 19: Diagrama de rasgos de la línea de productos de software para la vigilancia remota

Una vez que se tiene el diagrama de rasgos y se han implementado los rasgos extraídos a partir de las aplicaciones originales, es momento de empezar a trabajar sobre el resto de los rasgos identificados en el diagrama de rasgos. El diagrama de rasgos normalmente debe incluir diversos rasgos no existentes en las aplicaciones originales. Estos rasgos se deben de desarrollar uno a uno y verificar que sean compatibles con el resto de los rasgos de la línea de productos de software para la vigilancia remota. Es importante notar que se debe de crear los archivos *root.py* para todos estos rasgos de modo que se hagan cargo de la validación de las composiciones garantizado que todas las aplicaciones miembro generadas sean válidas. Una vez realizado esto, se pueden crear nuevas aplicaciones tan sólo seleccionando distintas composiciones de rasgos a través de la linea de comandos en *Pynion.* En total se desarrollaron 12 rasgos para esta línea de productos de software; en la siguiente secciones describiremos todos estos rasgos a gran detalle.

4.2.1. Lista de Rasgos

Los rasgos identificados en la sección anterior fueron reorganizados en las diferentes car¬petas de rasgos para ser usados en *Pynion* para poder crear las aplicaciones miembros de la línea de productos de software para la vigilancia remota. Cada uno de los rasgos en esta sección, si es incluido en la composición, agrega nueva funcionalidad a la aplicación miembro. Para cada uno de los rasgos describiremos su funcionalidad, sus beneficios y los componentes que modifican. Finalmente, para todos las rasgos enumeraremos su dependencias y la configuración que este acepta este rasgo para modificar el resultado de la linea de productos de software.

A continuación se enumeran los rasgos identificados en el orden cronológico en que fueron desarrollados:

- *Base.* El rasgo *base* es el rasgo básico exigido cualquier línea de productos de software creada en *Pynion.* Este rasgo es incluido por omisión en todas las composición que se deseen generar. En el caso de la línea de productos de software para la vigilancia

remota, el rasgo *base* se encarga de sentar los cimientos para la aplicación de la cámara remota únicamente. La version de la aplicacion de la *camara remota* contenida en el rasgo solamente simula una captura de una imagen (en realidad un valor nulo) y la imagen vacía resultante no se almacena en ningún lado. El rasgo *Base* no tiene ninguna funcionalidad pero si presenta los puntos de variación donde el resto de los rasgos podrán modificar la aplicación.

Todos los rasgos dependen tácitamente del rasgo *base* pues este forma parte forzosa de cada una de las composiciones. El rasgo *base* además esta encargado de garantizar que la composición contenga un juego de rasgos mínimos. En el caso de la línea de productos de software para vigilancia remota estos rasgos mínimos son al menos un rasgo que permita la captura de imágenes (como son *USB* e *IP*) y al menos un rasgo que transfiera la imágenes a una ubicación remota o local *(File, Ftp* y *Http).* El rasgo base no es configurable, pues este no contiene funcionalidad alguna.

- USB. El rasgo *USB* se refiere al componente que permite la obtención las imágenes a través de de un dispositivo *USB.* Así el rasgo *USB* es un rasgo de captura de imágenes desde una cámara. Generalmente, este dispositivo sera una *WebCam,* pero también pueden ser cámaras digitales y tarjetas de captura de vídeo siempre y cuando se comu¬ niquen con el equipo de computo a través del puerto *USB.* El rasgo *USB* sólo realiza alteraciones en la aplicación de cámara remota.

  Para que puede ser utilizado el rasgo *USB* debe de estar instalado el módulo *VideoCapture* [53] en el equipo de computo que puede ser descargado en *http://videocapture.sourceforge.net/.* Ademas de esto, si se desea utilizar el rasgo *USB* en una composición es necesario que no exista ningún otro método de captura de imágenes (en este caso, una opción sería el rasgo *IP)* pues cada cliente de cámara remota sólo puede manejar una sola cámara a la vez. La única configuración que contiene el objeto USB es la posición del dispositivo (el valor *Device* en la sección *USB* de la configuración) entre los dispositivos USB conectado al equipo de computo. El resto de las configuraciones típicas de *WebCams* como la calidad y la resolución se debe configurar con el software de la aplicación manejadora de la cámara directamente.

  File. El rasgo *File* es un rasgo de transferencia de imágenes. El rasgo *File* se encarga de almacenar las imágenes en el sistema de archivos en un sistemas remotos utilizando el sistema de compartición de archivos de Windows. Las imágenes se almacenan con el formato *JPEG.* El rasgo *file* es el único rasgo que permite, de manera natural, almacenar las imágenes localmente en el equipo de computo que esta conectado a la cámara. Este rasgo únicamente altera a la aplicación de la *cámara remota* de las aplicaciones que pueden resultar de la línea de productos de software.

  El rasgo *File* no depende de ninguno otro rasgo y se puede utilizar en múltiples ocasiones dentro de una misma composición. El rasgo *File* contiene dos valores de configuración: el nombre de archivo (llamado *filename*) y el número de imágenes que se desean almacenar(llamado *limit).* El valor *limit* es el número de imágenes máximo que se desean grabar. Una vez cumplido este límite se remplaza la imagen más antigua por la imagen recien tomada. Por esta razón es que *limit* tiene un valor mínimo de 1. El valor de *filename* es la trayectoria donde se desea almacenar las imágenes. La trayectoria en el valor de *filename* puede ser local o remota. Si se desea almacenar más de una imagen (esto es, *limit ¿1*) es importante que el nombre del archivo incluya la cadena de caracteres *%i*

que será sustituido por un número de imagen actual que va desde 1 al valor de *limit* y despúes de llegar a *limit* vuelve a comenzar.

- *IP.* El rasgo *IP* es otro rasgo de captura de imágenes. El rasgo *IP* puede obtener las imágenes desde cualquiere servidor web, pero en particular lo hace de cámaras que tienen una interfase de acceso por medio de *Http,* comúnmente conocidas como cámaras IP. Este rasgo es similar al rasgo *USB* pero cambia la fuente de donde se obtiene las imágenes. En este caso al igual que el rasgo *USB* este es un rasgo de captura así que sólo modifica la aplicación de la *cámara remota.*

  El rasgo *IP* es un rasgo de captura y no puede ser usado al mismo tiempo que otros rasgos de captura (esto es *USB* o el mismo). La configuración del rasgo *IP* se hace a través de la sección de configuración del mismo nombre. Existen cuatro valores de configuración para el rasgo *IP*: *server, address, username* y *password.* La dirección web de donde se descargan las imágenes es la concatenación de los valores *server* (que es el nombre del servidor) y *address* (que es la trayectoria en el servidor). Además de esto se puede configurar un nombre de usuario (el valor *username*) y contraseña (el valor *password*) para poder autenticarse con el esquema *BASIC* a la cámara IP.

- *Timer.* El timer es el único rasgo de la línea de productos de software para la vigilancia remota que pertenece de la clase *delay.* Los rasgos de la clase *delay* se encargan de controlar el tiempo que se espera antes de capturar la siguiente imagen. Si no se incluye el rasgo *Timer* en la composición entonces las imágenes se capturan tan rápido como sea posible. El rasgo *Timer* es otro rasgo más que solo modifica la aplicación de la *cámara remota.*

  El rasgo *Timer* es el rasgo más sencillo dentro de la línea de productos de software para la vigilancia remota. Este rasgo no depende de ningún otro, ni necesita de ninguna librería externa para funcionar. La configuración de *Timer* es mínima y se hace en la sección de configuración del mismo nombre. La sección *Timer* incluye un sólo valor: *duration.* El valor *duration* almacena el número de segundos que se debe esperar la aplicación de la *cámara remota* entre capturas de imágenes. El valor de *duration* debe ser cero o positivo, además de que puede ser un valor de punto flotante para mayor control de la duración.

- Ftp. El rasgo *Ftp* es un rasgo de transferencia de imágenes. El rasgo *Ftp* permite trans¬ ferir imágenes a un servidor por medio del protocolo *FTP (File Tranfer Protocol* o protocolo de transferencia de archivos en español). Las imágenes almacenadas en el ser¬ vidor *ftp* son almacenadas con el formato *JPEG.* El servidor abre la conexión al servidor al iniciar la aplicación de la *cámara remota* y la mantiene abierta durante todo el trans- curso de la ejecución de la aplicación. El rasgo *Ftp* hace modificaciones únicamente en la cámara remota.

  El rasgo *Ftp* es un rasgo de transferencia. Al igual que los otros rasgos de transferencia, se pueden usar varios de estos en la misma composición. El rasgo *Ftp* es un rasgo de repetible sin dependencias de otros rasgos o librerías. La configuración del rasgo *Ftp* se hace en la sección de configuración del mismo nombre. La sección *Ftp* de la configuración consta con los siguientes valores: *server, username, password, name* y *limit.* El valor *server* almacena el nombre o dirección donde se encuentra ubicado el servidor *ftp.* El valor de *username* y *password* almacenan las credenciales de autenticación que usará la

cámara remota para comunicarse con el servidor. El valor de *limit* y *name* funcionan de manera muy similar de como funcionan en el rasgo *File*. El valor de *name* es el nombre con el que se guardaran las imágenes en el servidor. El valor de *limit* es el número máximo de imágenes que se desea guardar, una vez alcanzado este límite se remplaza la imagen más antigua guardada. El valor de limit por lo mismo debe ser un valor entero mayor o igual a uno. Si el valor de *limit* es mayor que uno entonces el valor de *nombre* debe incluir la secuencia de caracteres "%i". La secuencia "%i" sera sustituida por los números de 1 al valor de *limit* cíclicamente.

- *Http*. El rasgo *Http* es un rasgo de transferencia similar a *File* y Ftp. El rasgo *Http* sube las imágenes a un servidor *Web* por medio del protocolo *Http* (Protocolo de transferencia de hipertexto por sus siglas en ingles). Al usar este rasgo las imágenes son transferidas en el formato *JPEG*. El rasgo sólo modifica la aplicación de la *cámara remota*.

  El rasgo *Http* al ser un rasgo de transferencia se puede usar con otros de la misma categoría (los rasgosFtp y *File*). Este rasgo puede ser utilizado en múltiples ocasio¬ nes en la misma composición. El rasgo *Http* no depende de ningún otro, pero si se desea utilizar la aplicación visor se debe agregar el rasgo *Http* a la composición. El rasgo *Http* utiliza el modulo *MultipartPostHandler.py* [54] descargable de la dirección: *http://odin.himinbi.org/MultipartPostHandler.py*. El rasgo *Http* se configura en la sec- ción de configuración con el mismo nombre. La sección *Http* contiene tres valores de configuración: *server, path, name*. El valor de *server* (nombre o dirección ip del servidor web) y el valor de *path* (la trayectoria del servidor web donde se subirán las imágenes) se concatenan y es la dirección donde se subirán las imágenes a través del método *POST*. El valor de *name* es el nombre con el cual se subirá la imagen al servidor web.

- *Visor*. El rasgo *Visor* es un rasgo básico de la línea de productos de software para la vigilancia remota. El rasgo *Visor* agrega la aplicación *Visor* al sistema miembro de la línea de productos de software. La aplicación *Visor* es una aplicación web donde se pueden desplegar las imágenes capturadas y transferidas desde varias aplicaciones de *cámara remotas*. El rasgo *Visor* no contiene la funcionalidad completa para recibir y desplegar las imágenes y otros rasgos deben ser agregados antes de que la aplicación sea completamente funcional. El rasgo *Visor* no altera la aplicación *cámara remota* y en cambio sienta las bases para la aplicación *Visor*.

  La presencia del rasgo *Visor* en una composición depende de varios otros rasgos para ser valido. El primero es el rasgo *Http* ya que la aplicación de la *cámara remota* y el *Visor* se comunican a través del protocolo *Http*. El segundo es un rasgo de la categoría de almacenamiento (los rasgos *FileStore* y *MemoryStore* son los de esta categoría) pues de otro modo las imágenes que se suben al servidor no serían almacenadas en ningún lugar. El rasgo *Visor* no es un rasgo configurable, ya que no contiene ninguna funcionalidad alguna si no que sienta las bases para que otros rasgos implementen su funcionalidad.

  • *MemoryStore*. El rasgo *MemoryStore* es uno de los dos rasgos de almacenamiento en *Visor*. El rasgo *MemoryStore* almacena la imagen actual de cada cámara remota. Las imágenes se almacenan en memoría utilizando los servicios de *Caching* de *Asp.net*. El rasgo *MemoryStore* almacena las imágenes como objetos de clase *Image*. Este rasgo únicamente agrega funcionalidad a la clase *Visor*.

  La clase *MemoryStore* tiene varias dependencias de otros rasgos. El primero de ellos

es el rasgo *Visor* pues como el rasgo *MemoryStore* lo modifica, se necesita *Visor* este antes presente en la composición. Este rasgo también depende de que no exista otro rasgo de almacenamiento de imágenes en la composición (en este caso el rasgo *FileStore* o el mismo). *MemoryStore* no es un rasgo configurable.

- *Move.* El rasgo *Move* permite controlar remotamente el movimiento de una cámara. Para utilizar este rasgo es necesario que se tenga el *hardware* adecuado que permita modificar la dirección a la que apunta el lente de la camara. En este caso, la cámara debe ser del tipo *IP* y debe poder ser manejada por medio del protocolo *http.* El rasgo *Move* permite cinco operaciones: arriba, abajo, izquierda, derecha y centrar. El rasgo *Move* es el único rasgo que modifica tanto la aplicación de la *cámara remota* como la aplicación *Visor.*

  El rasgo *Move* depende principalmente de la existencia del *hardware* necesario que permita el mover la dirección del lente de una cámara. El rasgo *Move* depende también de la presencia del rasgo *Visor* antes en la composición del producto miembro de la línea de productos de software de vigilancia remota, pues este agrega la aplicación *Visor* a la salida que requiere ser modificada. Ademas, el rasgo *Move* depende directamente de la presencia del rasgo *IP* antes en la composición, pues *Move* sólo funciona con *cámaras ip.* El rasgo *Move* permite configurar los comandos para mover la cámara en la sección *move* de la configuración de la aplicación de la *cámara remota.* La sección *move* contiene 6 valores: *path, left, right, up, down* y *center.* El valor *path* es la trayectoria en el servidor web de la *cámara ip* donde se enviaran los comandos de movimiento. El valor *path* deberá incluir el código *%s* donde se insertaran los comandos de movimientos respectivamente.

- *FileStore.* El rasgo *FileStore* permite almacenar imágenes en la aplicación *Visor.* A diferencia del rasgo *MemoryStore,* las imágenes se almacenan directamente en el disco duro. Las imágenes almacenadas por este rasgo son grabadas en el formato *JPEG.* El rasgo *FileStore* solo modifica la aplicación *Visor.*

  El rasgo *FileStore* al ser un rasgo de almacenamiento de imágenes, tiene varias de¬ pendencias. El rasgo *FileStore* sólo puede ser agregado a una composición si este es agregado despues del rasgo *Visor.* El rasgo *Visor* agrega la aplicación del mismo nom¬ bre que *FileStore* modifica. Ademas en una composición sólo puede haber un rasgo de almacenamiento de imágenes (esto es, no puede aparecer *MemoryStore* o el mismísimo *FileStore* en la misma composición). El rasgo *FileStore* puede ser configurado con 2 valores: *filename* y *count.* El rasgo *filename* es el nombre con el que serán almacenadas las imágenes, mientras que el valor de *count* almacena cuantas imágenes podrán ser al¬ macenadas por cada cámara en el equipo donde se ejecuta la aplicación *Visor.* El valor de *filename* debe incluir la cadena de caracteres "0" donde se insertaran los números de 0 a *count* cíclicamente, eliminando de manera secuencial las imágenes más viejas.

- *Security.* El rasgo *security,* como su nombre lo indica, se encarga de la seguridad. La funcionalidad esencial del rasgo es la autentificación y autorización de usuarios en la aplicación *Visor.* La autentificación se realiza mediante formas de html, mientras que la autorización se hace mediante trayectorias en el servidor. El rasgo de *security* se encarga de crear dos roles de usuario: el usuario ordinario y el usuario administrador. El acceso del usuario ordinario esta restringido a poder ver las imágenes capturadas

```
1 add = Visor/*.*, Visor/css/*.css, Visor/images/*.*
```

Figura 20: Contenido del archivo "feature.ini" dentro del rasgo "Visor"

de las diversas cámaras remotas. El usuario administrador tiene el mismo acceso que el usuario ordinario, pero también puede ver, dar de alta, modificar y eliminar otros usuarios, ya sean ordinarios o administradores. El rasgo *security* exclusivamente afecta a la aplicación *Visor*.

Al modificar la aplicación *Visor* es necesario que el rasgo del mismo nombre se encuentre antes que *security* en la composición. El rasgo *security* no es modificable, sin embargo los archivos de configuración que genera si lo son. Si se desea tener más control de como se aplica la seguridad en los miembros de la línea de productos de software es mejor modificar los archivos finales directamente.

4.3. Variabilidad utilizada

Uno de los propósitos principales de *Pynion* es soportar los diferentes tipos de variación existentes en las líneas de productos de software. En la sección 2.2 revisamos los diferentes tipos de variaciones: positiva (agregar funcionalidad), negativa (eliminar funcionalidad), opcional (agregar código), alternativa (cambiar código), plataforma o ambiente (contexto de la aplicación), funcional (cambio de la funcionalidad) y derivativa (modificado por la composición de la aplicación). Durante el desarrollo de la línea de productos de software para la vigilancia remota se utilizaron diversos tipos de variación. No todas las variaciones fueron utilizadas en esta línea de productos de software, por ejemplo, la variación negativa sólo se usa en líneas de productos desarrolladas con una estrategia extractiva (a partir de productos ya existente) y no tiene mucho sentido utilizarla en líneas de productos de software como la de vigilancia remota que es proactiva. En las siguientes subsecciones analizaremos los diferentes tipos de variación y como fueron aplicadas en la línea de productos de software.

4.3.1. Variación positiva

La variación positiva es la mías básica de todos los tipos de variación. Esta variación se encarga de agregar funcionalidad al miembro de la línea de productos de software de esta composición. En *Pynion* este tipo de variación corre a cargo de los rasgos. La manera más sencilla de agregar funcionalidad es agregando archivos a la salida de la línea de productos de software. Esto se puede hacer mediante el archivo *feature.ini*. En el archivo *feature.ini* se puede agregar trayectorias de archivos al valor *add* y estos serán agregados a la salida cuando el rasgo se agregue a la composición. Un ejemplo utilizado en la línea de productos de software para la vigilancia remota es el rasgo *Visor*. El rasgo *Visor* agrega la aplicación del mismo nombre a la composición. La aplicación *Visor* agrega nueva funcionalidad (la capacidad de ver imágenes vía una aplicación web) al resultado de la línea de productos de software. En la figura 20 vemos que el rasgo *Visor* agrega los archivos dentro de la carpeta *Visor* y de sus subcarpetas *css* y *images* a la composición final. El conjunto de estos archivos forman la nueva funcionalidad y es por eso un tipo de variación positiva.

```
1 <%def name="sections()">\
2  ${b_sections|i}
3  [timer]
4  duration=${c.config['timer']['duration']}
5  </%def>
```

Figura 21: Contenido del archivo "config.mako" del rasgo "timer"

```
1 <%def name="pause()">\
2  return self.${"pause"|nm}
3  </%def>
```

Figura 22: Extracto del archivo "client.mako" del rasgo "timer"

4.3.2. Variación opcional

La variación opcional es aquella que permite agregar código a un archivo existente. Este tipo de variación es similar a la positiva pero la variación se realiza a una mucho menor escala y con un alcance mucho más específico. En la línea de productos de software para vigilancia remota se utilizó en diversas ocasiones. Una de las razones mías comunes era agregar una nueva sección a los archivos de configuración *cliente.ini*. En la figura 21 vemos el contenido del archivo *config.mako* del rasgo *timer*. En la línea 1, vemos que le punto de variación se llama *sections*. En la línea 2 vemos que lo primero que se hace en la plantilla es agregar el código original que había en el punto de variación. El resultado final de la plantilla es que se agreguen las líneas 3 y 4 al código del archivo *cliente.ini*.

4.3.3. Variación alternativa

La variación alternativa se refiere a la variación en la cual se sustituye el código original enteramente por nuevo código. En este tipo de variación es específica en el nivel de los puntos de variación. La diferencia entre la variación opcional y la variación alternativa, es que la variación alternativa no referencia el contenido original del punto de variación. Al no agregar el contenido original el nuevo contenido del punto de variación es provisto por el rasgo. Este tipo de variación se presenta en dos tipos de rasgos principalmente: los rasgos opcionales y alternativos. Los rasgos opcionales son aquellos que se pueden o no aparecer en una combinación; los rasgos alternativos son aquellos que de un grupo sólo se puede seleccionar uno. En la figura 22 vemos un extracto del archivo *client.mako* del rasgo *timer*. En la línea 1 a la 3 vemos que no se referencia el contenido anterior del punto de variación *pause*. El nuevo contenido del punto de variación *pause* es el resultado de evaluar la línea 2 exclusivamente.

4.3.4. Variación de plataforma o ambiente

La variación de plataforma o ambiente es la variación que se lleva a cabo para ampliar la línea de productos de software a una nueva plataforma o ambiente. En el caso de la línea de productos de software para la vigilancia remota esta variación proviene directamente del diferente *hardware* de las diferentes cámaras. Existen tres rasgos que se refieren directamente al tipo de cámara que se utilizan: *USB*, *IP* y *Move*. El rasgo *USB* se utiliza exclusivamente

```
1 %if 'Move' in c.topLayers:
2 data = self.${"opener"|nm}.open("http://%s %s" %(self.${"server"|nm
    },self.${"path"|nm}), params)
3 headers = data.info()
4 self.action = headers['' x— action '']
5 %else:
6 self.${"opener"|nm}.open("http://%s%s" %(self.${"server"|nm},self.$
    {"path"|nm}), params)
7 %endif
```

Figura 23: Extracto del archivo *Client.mako* del rasgo *Http*

con cámaras que se conectan por medio del puerto *USB*, tradicionalmente estas son cámaras sencillas y económicas del tipo *webCam*. El rasgo *IP* se utiliza cuando uno se desea comu¬nicar con la cáamara por medio del protocolo *http* comúnmente conocidas como cámaras *ip*. Finalmente, el rasgo *Move* se puede utilizar exclusivamente con cámaras *IP* con control de la dirección a donde apunta el lente de la cámara remoto. La variación del tipo plataforma o ambiente es una variación que es útil cuando tenemos que manejar diferente *hardware* como base.

4.3.5. Variación derivativa

La variación derivativa se refiere a la variación que depende de la presencia de otros rasgos dentro la composición. La variación derivativa generalmente no es deseada pues esta rompe con la encapsulación que es deseable en los rasgos. Sin embargo, existen ocasiones en que este tipo de variación es útil (para aprovechar recursos ya obtenidos) o es necesaria debido dependencias fuera del control del desarrollador (por ejemplo causada por el hardware o la plataforma de desarrollo). En la línea de productos para software para la vigilancia remota, se utilizó la variación derivativa para reutilizar las peticiones a la cámara *ip* para enviar comandos de movimiento a la cámara. En la figura 23 vemos un ejemplo de una variación derivativa. En este caso, si entre los rasgos superiores se encuentra el rasgo *Move* (en la línea 1) el resultado incluye las líneas 2 a la 4, en caso contrario el resultado incluye la línea 6. La variación derivativa permite gran flexibilidad en una línea de productos de software.

4.4. Patrones de Diseño para Pynion

"El éxito es seguir el patrón de la vida que uno mas disfruta"

Al Capp

Los sistemas exhiben estructuras idiomáticas y recurrentes para resolver problemas y hacer los sistemas más flexibles, elegantes y finalmente reutilizable. Los patrones de diseño han sido propuestos como un método de representar, registrar y reutilizar estas estructuras de datos recurrentes y asociadas a la experiencia de diseño [55]. Durante el desarrollo de la línea de productos de software para vigilancia remota fue evidente la aparición de patrones que se repetían constantemente. La identificación de estos patrones permite sentar las bases para hacer recomendaciones que se apliquen de manera general al desarrollo de línea de

productos de software utilizando *Pynion* como herramienta para mecanismos de variación. Algunos patrones corresponden al desarrollo de líneas de productos de software independiente del mecanismo de variación que se desee utilizar en su desarrollo y otros son específicos de *Pynion* como herramienta de desarrollo.

Esta sección presentara los diferentes patrones de diseño que han sido descubiertos trabajando en la línea de productos de software para la vigilancia remota en *Pynion.* Cada uno de estos patrones sera descrito a gran detalle para poder ser identificado e utilizado con mayor facilidad. También para cada uno de ellos se enlistaran los beneficios que se pretenden obtener con el uso de patrón de diseño. En cada subsección se revisaría cuales son los usos probables que se le puede dar a este patrón dentro de una línea de productos de software. Finalmente, cada patrón de diseño incluirá uno o varios ejemplos del uso del patrón directamente extraído de la línea de productos de software para vigilancia remota con el código fuente que muestra el patrón.

4.4.1. Alcance de nombres: Rasgo

En un sistema creado en cualquier lenguaje de programación existen variables cuyo alcance esta restringido para evitar colisión entre nombres o accesos indebidos por alguna sección de código. Tradicionalmente el alcance de un nombre puede ser global, local, "namespace" o espacio de nombre, clase y otros. Por las características de las líneas de producto es necesario crear un nuevo alcance en los nombres de variables y evitar así colisiones indeseables entre los nombres. Este nuevo alcance es el alcance a nivel rasgo. Un nombre declarado a nivel rasgo debe ser sólo accesible por funciones y métodos declarados en el mismo rasgo.

En *Pynion* la función *named* de la clase *FileContext* y el filtro de *Mako* llamada nm cumplen la función de definir este alcance. Ademas de esto, existen la función *snamed* y el filtro *sn* que ofrecen el mismo servicio pero con una notación más corta (La sección 3.3.6 contiene más información acerca del uso de estas funciones). Estas funciones modifican un nombre haciéndolo único para el rasgo y sólo accesible desde este. La manera en que esto función es anexando al nombre el nombre del rasgo y la posición en que se encuentra este rasgo dentro de la composición del producto. Este tipo de alcance interactúa con los alcances ya definidos por un programa y los restringe más de modo que el nombre solo pueda pertenecer a un sólo rasgo dentro de la composición.

El uso del alcance a nivel rasgo ofrece varios beneficios para el desarrollo de líneas de productos de software. El primero es evitar la colisión de nombres entre rasgos. Esto es de gran importancia por que los rasgos deben de ser independientes unos de otros y así se puede evitar que el mismo nombre se útilice en los mismos rasgos o se generen efectos secundarios indeseables entre los rasgos. Además los nombres con alcance de rasgo hacen que un rasgo termine siendo independientemente encapsulado. Otro beneficio de utilizar este alcance es que no solo los nombres utilizado por este rasgo no colisionaran con otros rasgo; sino que también evita que un rasgo colisione con el mismo si es utilizado en varias ocasiones en una misma composición. El limitar nombres mediante el uso de rasgo es la única manera mediante la cual se puede, en *Pynion* diseñar rasgos que se puedan utilizar múltiples veces dentro de una misma composición.

El alcance de nombres a nivel rasgo se puede utilizar para nombrar variables, funciones, clases, etc. y restringirlos en su alcance. Además un nombre con alcance de rasgo puede ser utilizado como un identificador único como los útilizados en XML. En la figura 24 vemos un extracto de archivo *File/Update/client.mako.* Este archivo es parte del rasgo que permite que

```
 1 <%def name="repos()">\
 2  ${b.repos|i}\
 3 #Función referenciable solo por el código del rasgo
 4  def ${"transfer"|nm}(self,image):
 5      ${b_transfer|i}
 6  </%def>
 7
 8 <%def name="init()">\
 9  ${b_init|i}
10 #Referenciables solo por código en el rasgo
11  self.${"count"|nm} = 0
12  self.${"limit"|nm} = self.config.getint('${"file"|nm}','limit')
13  self.${"path"|nm} = self.config.get('${"file"|nm}','path')
14  </%def>
```

Figura 24: Extracto del archivo File/Update/client.mako

Mako almacena las imágenes en un archivo. En la plantilla llamada *repos* (en la línea 1) se nombra una función llamada "transfer" que sólo puede ser accedida desde este rasgo (en la línea 3). En la plantilla llamada "init" se complementa el constructor de la clase con variables a nivel clase que están aún mías restringidas a usarse exclusivamente desde este rango (En las líneas 11 a la 13). De este extracto podemos ver que todas las variables usadas por estos métodos son de alcance rasgo. Esto permite que este rasgo pueda ser reutilizado en repetidas dentro de una misma composición. Gracias a esto es posible que las imágenes se almacenen en dos o mías diferentes ubicaciones al mismo tiempo (que pueden ser locales o remotas a través del sistema de compartición de archivos de Windows) como se puede hacer por medio del rasgo *File.* Un extracto de el archivo *client.mako* se muestra en la figura 24. Podemos ver que las variables miembro de la clase *count, limit, path* (líneas 11, 12 y 13) y el método de la clase *transfer* (línea 4) son encapsulados por el filtro nm. El alcance de las variables del tipo **${"nombre"|nm}** es restringido a ser accedido solo por el código fuente en el rasgo actual (en este caso *file*).

En la figura 25 vemos el funcionamiento del alcance a nivel rasgo. En el rasgo 1 se aplica el alcance rasgo a una variable local, en el rasgo dos se aplica a una variable de una clase y en el rasgo 3 a una variable de alcance global. En la esquema podemos ver que el utilizar nombres de alcance rasgo logra limitar a un más los alcances existentes para limitarlos también a estar en un solo rasgo. Así el alcance rasgo siempre acaba poniendo limites más pequeños a los nombres de variables y funciones en el código de los rasgos desarrollados en *Pynion.*

El alcance a nivel de rasgo es de vital importancia en el desarrollo de líneas de productos de software. Este alcance permite que los rasgos puedan ser encapsulados, reutilizados y man¬tenerse independientes de los demás rasgos y del resto de la aplicación. Así el método *named* y el filtro *nm* en *Mako* son parte esencial del juego de herramientas que provee *Pynion* como herramienta para mecanismos de variación pues habilitan la creación de nombres acotados para ser válidos en un sólo rasgo.

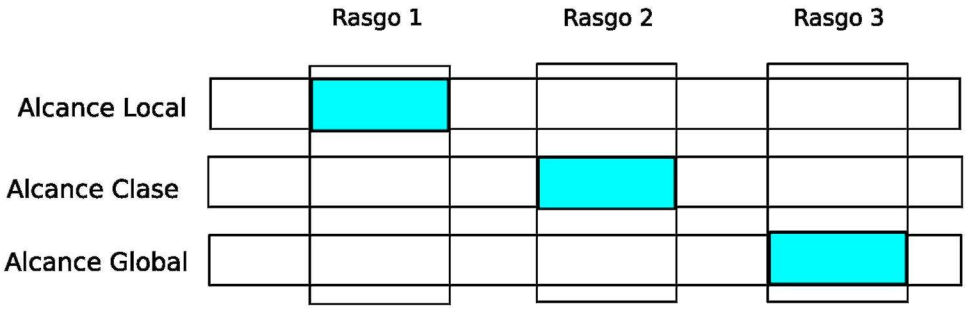


Figura 25: Diagrama del alcance interrasgo

4.4.2. Alcance de nombres: Interrasgo

En la subsección anterior analizamos el alcance a nivel de rasgo de nombres. Este alcance nos permite independizar los nombres de las variables y funciones de cada rasgo del resto de ellos. Esta situación es, por mucho, la mías deseable mientras se desarrolla una rasgo de una línea de productos de software. Sin embargo, existen ocasiones en que queremos que varios rasgos interactúen directamente unos con otros. Regularmente, esta interacción se puede hacer mediante la refinación de funciones (como podemos ver en la sección 4.4.8) pero en ocasiones se necesita una interaccion más cercana entre varios rasgos. Cuando se requiere hacer esto es que utilizamos el alcance a nivel interrasgo.

El uso del alcance interrasgo dentro de una línea de productos de software debe ser limitado pues rompe con la independencia entre los rasgos y su abuso puede ser perjudicial a la larga la calidad de la aplicación de salida. No existe ninguna nomenclatura particular para compartir nombres entre rasgos, pues este nivel de alcance es solamente lógico y no real. Para utilizar este alcance es necesario seleccionar los nombres que se desean compartir y usarlos cuidando que no colisionen de manera indeseada con nombres en otros rasgos. Una de las razones más típicas para usar nombres interrasgos es mejorar la eficiencia del código fuente de alguna aplicación. Las variables y nombres interrasgos te permite reutilizar código fuente(no se tiene que declarar funciones similares 2 veces) o recursos entre varios rasgos sin necesidad de obtenerlos en repetidas ocasiones. En el caso de la línea de productos de software de vigilancia remota el recurso que se reutilizó es la conexión http entre la aplicación de la *cámara remota* y la aplicación *Visor*. La conexión de web es utilizado por el rasgo *Http* y es reutilizado por el rasgo *Move* para transferir información sobre la dirección hacia donde se desea mover la lente de la cámara. El rasgo *Move* básicamente se monta de "caballito" para transferir la información de movimiento desde el *Visor* a la aplicación de la *cámara remota.* En la figura 26 podemos ver que los dos rasgos comparten información a través de la variable *self.action.* En el extracto del rasgo *Http* lo vemos referido en la línea 8, mientras que en el rasgo move lo vemos utilizado en las líneas 4, 5.

El utilizar variables entre rasgos no es recomendado en *Pynion.* Sin embargo, cuando varios rasgos requieren compartir información directamente es posible hacerlo sin perjudicar excesivamente a la línea de productos de software. Una de las razones para hacer esto es la posibilidad reutilizar recursos costosos de obtener que ya utiliza un rasgo en otros rasgos. En *Pynion* el alcance entre rasgos es sólo lógico y las variables se declaran como cualquier otra variable se declararía en el lenguaje de programación original.

Extracto del client.mako en el rasgo "Http"

```
1 <%def name="transfer()">\
2 self.${"transfer"|nm}(image)
3 image.save(self.${"name"|nm}, "JPEG")
4 params = {"file": open(self.${"name"|nm}, "rb") }
5 %if 'Move' in c.topLayers:
6 data = self.${"opener"|nm}.open("http://%s%s" %(self.${"server"|nm
    },self.${"path"|nm}), params)
7 headers = data.info()
8 self.action = headers['x-action']
9 %else:
10 self.${"opener"|nm}.open("http://%s%s" %(self.${"server"|nm}, self.$
    {"path"|nm}), params)
11 %endif
12 </%def>
```

Extracto del client.mako en el rasgo "Move"

```
1 <%def name="transfer()">\
2 self.${"transfer"|nm}(image)
3 if self.action != 'none':
4     print self.action}
5     self.${"opener"|nm}.open(self.${"move"|nm}[self.action])
6 </%def>
```

Figura 26: Ejemplo de alcance interrasgo

```
1       def  capture(self):
2           #[[[Py:capture ]]]
3           return  none
4           #[[[End ]]]
5
6       def  pause(self):
7           #[[[Py:pause ]]]
8           return  0.0
9           #[[[End ]]]
10
11      def  transfer(self, image):
12          #[[[Py:transfer ]]]
13          pass
14          #[[[End]]]
```

Figura 27: Extracto del archivo *Base/Add/RemoteCam/cliente.py*

4.4.3. Refactorización en funciones

Pynion es un herramienta para mecanismos de variación que permite la creación de líneas de productos de software a partir de productos legados. Así *Pynion* permite una gran reutilización de recursos e incluso reutilizar todo el código de aplicaciones completas (si estas forman parte un rasgo independiente). Sin embargo, para sacar el mejor provecho de una línea de productos de software es necesario refactorizar el código fuente. La refactorización, cuando realizado correctamente, crea una línea de productos de software mejor estructurada y permite el desarrollo más veloz de nuevos rasgos. *Pynion* permite que los puntos de variación puedan ser en cualquier lugar del código mientras se pueda encapsular entre dos líneas marcadas escondidas entre comentarios. Sin embargo, para mejor la estructura del código fuente es mejor que estos puntos de variación coincidan con la posición funciones dentro del código fuente idealmente encapsulando todo el contenido de la función. Esto permite que se pueda utilizar el proceso de refinación de funciones (ver subsección 4.4.8 para más información) en los rasgos.

En la línea de productos de software de vigilancia remota se tomaron las clases originales y se refactorizaron con dos propósitos: extraer el código que pertenezca a los diversos rasgos y cambiar el código de modo que los puntos de variación coincidan con la localización de funciones. En la figura 27 podemos ver que la base de la aplicación cliente fue refactorizada para que los puntos de variación *capture* (en la línea 2), *pause* (en la línea 7) y *transfer* (en la línea 12) coincidieran con el contenido de las funciones con el mismo nombre (líneas 1, 6 y 12 respectivamente).

La refactorización es un proceso necesario cuando se desarrollan líneas de productos de software a partir de productos legados. La refactorización inicial es un buen momento para hacer que los puntos de variación coincidan con funciones. Esta refactorización permitiría un desarrollo mías veloz de nuevos rasgos al darle una estructura mías legible al rasgo base y los demás rasgos que trabajen con este.

```
1 <%def name="sections()">\
2 ${b_sections|i}
3 [timer]
4 duration=${c.config['timer']['duration']}
5 </%def>
```

Figura 28: Código completo del archivo *Timer/Update/config.mako*

4.4.4. Refactorización de configuraciones

En la sección anterior se discutió como es recomendable aplicar la refactorización al código fuente de las aplicaciones legados para prepararlas para trabajar en una línea de productos de software. La refactorización también puede ser aplicada a los archivos de configuración de una aplicación. Tradicionalmente, los formatos de archivos de configuración son almacenados en diferentes secciones, ya sea que este sean archivos *.ini, xml* u otros. En las líneas de productos se puede aprovechar esta disposición natural de los archivos de configuración en secciones para que estas secciones representen uno a uno los rasgos. Dado que las aplicaciones son composiciones de rasgos, es un método natural que las secciones también estén organizadas por rasgos.

La organización de la configuración por rasgos es siempre sugerida en el desarrollo de las líneas de productos de software. Sin embargo esta organización no siempre es factible pues a veces la configuración de una aplicación esta ligada a plataformas sobre las cuales están construida la línea de productos de software o aplicaciones o componentes que esta usa. En la línea de productos de software para vigilancia remota la configuración de la aplicación de la *cámara remota* fue subdivida en rasgos, mientras que la configuración en el *Visor* no lo fueron. La razón de que el *Visor* no fuera subdivido así, es que las secciones están restringidas por la plataforma (en este caso el formato de *xml* particular usado por *asp.net*). En la figura 28 podemos ver el contenido del archivo *config.mako* del rasgo *Timer*. Esta plantilla modifica el archivo *client.ini* con formato *.ini* que contiene la configuración de la aplicación de la *cámara remota*. En este archivo podemos ver que se copian las secciones anteriores en la instrucción **`${b_sections|i}`** (en la línea 2), para despues agregar una sección nueva llamada *Timer* (en la línea 3). Esta nueva sección de la configuración controla el retraso entre el momento que se toma una imagen de la cámara y la siguiente vez que se hace.

La refactorización de la configuración es una buena medida para mantener la configuración de cada rasgo de manera independiente. Si bien este es un buen objetivo siempre deseable; no siempre se puede realizar pues a veces la configuración, el formato lo impide o la plataforma de desarrollo lo impide. La organización por rasgos es una modo lógico de organizar los archivos de configuración de una aplicación.

4.4.5. Referencias no repetirles

La gran mayoría de los lenguajes de programación permiten referenciar otras librerías o espacios de nombre dentro del código fuente de los archivos. Importar las librerías o espacios de nombres, regularmente, sólo se debe hacer en una ocasión en un sólo archivo o podría causar un error. Esto puede complicar los rasgos de una línea de productos de software pues un rasgo podría tratar de importar una librería que alguno otro rasgo ya había

importado. Existen varios métodos mediante un rasgo puede evitar repetir referencias. El mías sencillo y recomendado es verificar si ya se ha importado la referencia anteriormente, si ya se importo no volverla a importar dentro del código fuente. El evitar repetir referencias hace que más combinaciones entre rasgos sean posibles pues evita que interfieran unos con otros negativamente.

El evitar que las referencias se repitan se debe de utilizar para todos los archivos y rasgos que requieran importar o usar librerías. Esta situación se presenta en la gran mayoría de las aplicaciones y dentro gran mayoría del código fuente de los archivos de los rasgos. El importar librerías garantizando que no se repetirían permite que los rasgos se mantengan encapsulados e independiente uno de otro. En la línea de productos de software para vigilancia remota se utilizaron las referencias no repetibles en la gran mayoría de los archivos de código fuente tanto en *Python* como en *C#*. Para poder usar consistentemente este patrón de diseño es necesario mantener, en toda la aplicación, la importación de una sola librería por línea de código fuente. Además todo el código fuente para importar o usar las librerías tiene que estar encapsulado en un punto de variación (En toda la línea de productos de software se utilizó *imports* como el nombre del punto de variación donde se efectuaba la importación de librerías y espacio de nombres). En la figura 29 vemos un ejemplo de como se aplica este patrón de diseño con ejemplos directamente extraidos de la línea de productos de software para la vigilancia remota. La primera parte del patrón es incluir las referencias anteriores ya declaradas (**`${b_imports| i}`** en la línea 2 para ambos extractos) y despues verificar si no se había importado esa librería antes **`(%if b_imports.count('importar libreria') ==0:`** en la línea 3 para ambos extractos) antes de agregar la librería dentro del *if* (en la línea 4 para ambos extractos).

El patrón de diseño para no repetir referencias se usa primordialmente para importar o usar librerías o módulos en diferentes lenguajes de programación. Sin embargo, existen mías usos para el patrón visto en esta sección como puede ser declarar o revisar la existencia de variables o clases dentro de la composición de rasgos de la aplicación de salida de la línea de productos de software para evitar declaraciones múltiples. El método visto en este patrón es el método sugerido para evitar repeticiones indeseadas de importación de librerías mientras se desarrollan líneas de productos en *Pynion* y es un gran beneficio para la encapsulación de rasgos.

4.4.6. Dependencia entre rasgos

En la sección 2.2 vimos la existencia del tipo de variación derivativa. En la variación derivativa de acuerdo a [25] los cambios que realiza un rasgo depende de los otros rasgos que estén presentes dentro de la composición. Así un rasgo aplica los cambios que le pertenecen mías los cambios que hace cuando otros rasgos estén o no presentes. En la sección 4.4.2 vimos que se pueden declarar variables y funciones que pueden interactuar entre diversos rasgos. Estas variables son cruciales para desarrollar la dependencia entre rasgos. Para poder generar código fuente que dependa de la presencia o ausencia de otros rasgos se deben utilizar los arreglos *topLayers* y *bottomLayers* de la instancia *c* de la clase *FileContext.* Estos arreglos contienen los nombres de los rasgos declarados antes en la composición *(bottomLayers)* y los nombres de los rasgos declarados después en la composición *(topLayers).* El uso de la interdependencia entre rasgos no debe ser abusado pues puede afectar la calidad del desarrollo pero si es una herramienta más en el arsenal del diseñador de líneas de productos de software.

Este patrón de diseño debe ser usado cuando se tiene un rasgo cuyo código fuente depen-

Importar referencias sin repetir en Python

```
1 <%def name="imports()">\
2 ${b.imports|i}\
3 %i b_imports.count('import os')==0:
4 import os
5 %ndif
```

Importar referencias sin repetir en C#

```
1 <%def name="imports()">\
2 ${b.imports|i}\
3 %i b_imports.count("using System.Web.Caching")=0:
4 using System.Web.Caching;
5 %ndif
6 </%def>
```

Figura 29: Ejemplos de importar referencias sin repetir

da de la presencia de otros rasgos. Ejemplos de esto podría ser un módulo que implementa seguridad básica para una sección pero que se deshabilite cuando otro rasgo con seguridad más avanzada sea incluido en la composición. Ademas, este patrón puede ser útilizado por los mismos motivos que el alcance interrasgo: reutilizar recursos creados por otros rasgos en la composición si están disponibles. En la figura 30 podemos ver un ejemplo de la dependencia entre rasgos en la línea de productos de software de vigilancia remota. En este ejemplo se extrae información de los datos del encabezado de la respuesta web, sólo cuando se allá agregado el rasgo *Move* se encuentre en los rasgos siguientes (`%if 'Move' in c.topLayers` en la línea 1). Las líneas 2 a la 4 es el contenido del rasgo si se ha agregado el rasgo *Move* a la composición despues del rasgo *Http* y la línea 6 es el contenido si no.

El poder modificar el código fuente resultante de un rasgo es necesario para poder realizar variaciones derivativas sobre la línea de productos de software. Como todo los patrones que involucren varios rasgos, este patrón no es recomendado para un uso extensivo en el desarrollo de un proyecto. Los arreglos *topLayers* y *bottomLayers* concentran la información sobre la composición actual y pueden ser utilizados por el rasgo para decidir que código fuente se debe incluir o no incluir en su aplicación. La dependencia entre rasgos, es entonces, una herramienta poderosa para crear rasgos avanzados que interactúan entre sí.

4.4.7. Repositorio de campos y funciones

El repositorio de funciones es un patrón de amplia utilidad en el desarrollo de líneas de productos de software con *Pynion*. En este patrón se declara un punto de variación vacío dentro de la clase en el rasgo base (o el primer rasgo en agregar el documento). A partir de ahí cada rasgo puede agregar métodos o campos a la clase designada en el punto de variación donde se encuentra el repositorio. Se recomienda incluir un punto de variación de repositorio en todas las clases que incluyan al menos un punto de variación. El repositorio es el lugar ideal para crear métodos y campos auxiliares para ser utilizados en la labor principal del rasgo.

El patrón de repositorio nos da un lugar libre donde poder modificar la clase al incluir nuevos campos y métodos. Estos campos y métodos generalmente son auxiliares a la función

```
1  %if 'Move' in c.topLayers:
2  data = self.${"opener"|nm}.open(" http://%s%s" %(self.${"server"|nm
       },self.${"path"|nm}), params)
3  headers = data.info()
4  self.action = headers['x—action']
5  %else:
6  self.${"opener"|nm}.open(" http://%s%s" %(self.${"server"|nm},self.$
       {"path"|nm}), params)
7  %endif
```

Figura 30: Extracto del archivo *Http/Update/Client.mako*

```
1  <%def name="repos()">\
2  ${b.repos|i}
3  public Dictionary<string ,Bitmap> Images
4 {
5       get
6   {
7           Cache cache = HttpContext.Current.Cache;
8           if(cache["Images"] == null)
9      {
10              cache["Images"] = new Dictionary<string, Bitmap>();
11          }
12          return (Dictionary<string, Bitmap>)cache["Images"];
13       }
14 }
15 </%def>
```

Figura 31: Extracto del archivo *MemoryStore/Update/image.mako*

principal de los rasgos. El gran beneficio del patrón de repositorio de funciones y campos es su versatilidad. El uso de este patrón de diseño fue extendido en la línea de productos de software de vigilancia remota. En la figura 31 vemos un ejemplo de como se utilizo esto en la línea de productos de software para la vigilancia remota. En este caso el rasgo *MemoryStore* primero mantiene los contenidos originales del repositorio intactos (en la línea 2). Y despues agrega una función con un diccionario (o tabla de *Hash)* llamada *Images*(desde la línea 3 a la 14) que almacena en memoria cache las imágenes que han generado las cámaras. Es importante que en todos los repositorios se conserve siempre la totalidad del código producido por los rasgos anteriores.

El patrón de repositorio es uno de los patrones mías versátiles y esenciales en el desarrollo de línea de productos de software con *Pynion.* Una de sus funciones es agregar nuevos campos y métodos a clases ya existentes que pueden servir como auxiliares en el proceso de generar el código fuente del rasgo. En la subsección 4.4.8 veremos como el patrón de repositorio es parte esencial del patrón de refinación de funciones.

4.4.8. Refinación a pasos de funciones

La *refinación a pasos* es definida por Batory en [56] como: el derivar programas complejos a partir de programas sencillos al agregar progresivamente nuevos rasgos. Para crear líneas de productos de software a través de la composición se necesita poder agregar refinamientos de las clases poco a poco a través de varias capas. *Pynion* puede realizar este refinación a través de llamadas subsecuentes de funciones. La estrategia de este patrón es la estrategia recomendada en el proceso de crear líneas de productos de software.

El proceso de refinación permite encadenar funciones de modo que cada rasgo sea capaz de agregar una nueva función a la cadena. Cada función en la cadena debe poder llamar a la función anterior en la cadena de funciones. Finalmente, toda esta cadena de funciones debe de poder ser llamada por el nombre inicial de la aplicación y con los parametros y valores de retorno originales. Este método es el modo de agregar nueva funcionalidad a la línea de productos de software sin sustituir la anterior. Para poder utilizar este patrón es necesario primero crear una función con nombre único (usando el método *named* o el filtro *nm*) que contenga el código de la función original y este contenida en el repositorio (como visto en la subsección 4.4.7). Una vez hecho esto se sustituye el contenido de la función original por el nuevo código que se quiera agregar. Este código que se agregó puede hacer llamar a la función padre que se creo en el repositorio para encadenar las dos funciones. Estos nos crea una estructura de composición por medio de una cadena de funciones necesaria para realizar la refinación a pasos.

Pocos patrones tuvieron un uso tan extenso en la línea de productos de software de vigilancia remota como este. Esta fue la base de como operan los diferentes rasgos de trans¬ ferencia de imágenes (a archivos, *ftp* y *http*) y la razón por la cuál se pueden usar varios en la mismas combinación y en repetidas ocasiones. En la figura 32 vemos el listado parcial de el archivo *client.mako* del rasgo *Ftp* donde se utiliza este patrón. Este rasgo se encarga de trans¬ ferir las imágenes a un servidor *ftp* permitiendo que se almacene las imágenes con números secuenciales que se repiten. En la línea 2 vemos que se recupera el repositorio ya existente. En las líneas 2 y 3 es donde se crea una función *transfer* con alcance a nivel rasgo que contiene la función original. En la plantilla *transfer* (líneas 7 a 18) se sustituye el contenido original de la función con el nuevo funcionamientos que se desea agregar. Esta nuevo código hace la llamada a la función *transfer* original (declarada en la línea 3) en la línea 8.

El proceso de refinación a pasos es parte esencial de las líneas de productos de software y la composición, este patrón permite usar muchas de las estrategias conocidas. Siempre que sea factible se debe procurar utilizar este patrón de diseño, pues permite una mayor flexibilidad y reuso en los rasgos que pertenecen a la línea de productos de software.

4.4.9. Jerarquía de puntos variaciones

Una manera de visualizar los puntos de variación es como si fueran un árbol jerárquico. Este tipo de organización permitirá un control más fino o más áspero de como se desea modificar un documento de texto. Así, se podría decidir si se desea modificar una fracción del punto de variación o se desea modificarlo por completo. Esto podría hacer que un documento básicamente ofreciera servicios a los rasgos de modo que rasgo siempre tenga que pueda modificar las partes más pequeñas posibles de el; pero si un rasgo así lo requiriese podría modificar una sección más grande del código fuente. Esto permite mayor control sobre la modificación a hacerse a cada rasgo permitiendo que este seleccione su alcance.

```
 1 <%def  name="repos ()">\
 2  ${b.repos|i}
 3  def  ${"transfer"|nm}(self ,image) :
 4       ${b_transfer|i}
 5  </%def>
 6
 7 <%def  name="transfer ()">\
 8  self.${"transfer"|nm}(image)
 9  self.${"count"|nm} = self.${"count"|nm}+ 1
10  ${"localName"|nm} = self.${"name"|nm}
11  if  self.${"name"|nm}.count('%i') == 1:
12       ${"localName"|nm} = self.${"name"|nm}  %self.${"count"|nm}
13  ${"tmpFile"|nm} = StringIO.StringIO()
14  image.save(${"tmpFile"|nm}, "JPEG")
15  ${"srcFile"|nm} = StringIO.StringIO(${"tmpFile"|nm}.getvalue())
16  self.${"ftp"|nm}.storbinary('STOR %'  %${"localName"|nm}, ${"
       srcFile"|nm})
17  self.${"count"|nm} = self.${"count"|nm}  %self.${"limit"|nm}
18  </%def>
```

Figura 32: Extracto del archivo *Ftp/Update/client.mako*

Este tipo de jerarquías de puntos de variación en *Pynion* no es factible directamente, pues *Pynion* no soporta marcar un punto de variación dentro de otros directamente. Sin embargo, esta jerarquía se puede simular declarando una función padre que llame a varias funciones hijos y los contenidos de cada una de estas deben ser marcados como puntos de variación. Esto permitiría que un rasgo sustituya alguno o varios de los hijos de la función o directamente la función padre según le sea conveniente creando así, efectivamente, una jerarquía de puntos de variación.

En la línea de productos para software de vigilancia remota este método fue utilizado en el rasgo *Visor*. El documento *image.ashx* del rasgo *Visor* se puede encargar de enviar una imagen escogida como respuesta de la página web o el rasgo se puede hacer cargo de escoger y procesar la respuesta por si mismo. El rasgo *MemoryStore* sustituye el punto de variación hijo *getImage* pues el rasgo sólo obtiene la imagen correcta y la regresa para que la aplicación original la despliegue al usuario. El rasgo *FileStore* sustituye en cambio el punto de variación *process* así que se encarga tanto de escoger la imagen como enviar la imagen a la respuesta del servidor web. En la figura 33 vemos el archivo *image.ashx* del rasgo *Visor* que hace uso de la marcación de los puntos de variación siguiendo el patrón de jerarquía de puntos de variación. En las líneas 3 a la 6 podemos ver el punto de variación *getImage* de la función hija, mientras que en las líneas 9 a la 20 podemos ver el punto de variación *process* que representa la función padre. Es necesario que en el punto de variación padre (en este caso *process)* haga un llamado a la función que contiene el punto de variación hijo, este llamado se hace en la línea 13.

El patrón de jerarquías de punto de variación es un patrón avanzado que esta adaptado específicamente para *Pynion*. Esta herramienta para mecanismos suple el echo de que no se pueden declarar puntos de variación dentro de otros. Las jerarquías de puntos de variación permite que un rasgo escoja que tan grande o pequeña quiere hacer la modificación que se

```
1  public Bitmap GetImage(string camera)
2  {
3       //[[[Py:getimage]]]
4       return null;
5       //[[[End]]]
6  }
7
8  public void ProcessRequest (HttpContext context) {
9       //[[[Py:process ]]]
10      context.Response.ContentType = "image/jpg";
11      context.Response.Cache.SetCacheability(HttpCacheability.NoCache
            );
12      string camera = context.Request["Camera"];
13      using (Bitmap b = GetImage(camera))
14      {
15          if (b != null)
16          {
17              b.Save(context.Response.OutputStream, System.Drawing.
                    Imaging.ImageFormat.Jpeg);
18          }
19       }
20      //[[[End]]]
21
22 }
```

Figura 33: Extracto del archivo *image.ashx* del rasgo *Visor*

vaya a realizar sobre el código fuente. Generalmente, se desea que las modificaciones se hagan en los puntos de variación hijo y sólo se hagan en el punto de variación padre en caso de que sea estrictamente necesario hacerlo. En el caso del rasgo *FileStore* este sustituyo el punto de variación padre para evitar el uso redundante de memoria que no hacerlo hubiese existido.

4.4.10. Centralización de la configuración

Un buen porcentaje de los proyectos de desarrollo de software son en realidad sistemas de software. Los sistemas se componen de varias aplicaciones que interactúan unas con otras. Cada uno de estas aplicaciones, regularmente, contiene su propia configuración independiente. El tener varios archivos de configuración implica que cuando se requiera cambiar la configuración del sistema se deben realizar modificaciones en varios archivos. *Pynion* permite unir todas estas configuraciones en un sólo archivo *Pynion.ini*. El proceso de reunir todos los archivos de configuración en uno solo se le conoce como el patrón de *centralización de la configuración*.

En *Pynion* la configuración de todas las aplicaciones de un sistema de productos se pueden almacenar en un solo archivo de configuración central. Este archivo es el archivo *Pynion.ini* que se debe encontrar en la raíz de la línea de productos de software (más información sobre el archivo *Pynion.ini* en la sección 3.3.4). En la figura 34 vemos la implementación del patrón de la centralización de la configuración. El archivo *Pynion.ini* contiene la configuración

Extracto del archivo "Pynion.ini"

```
1  [file]
2  path=images/camera.%.jpg
3  limit=5
4  [usb]
5  device=0
6  [timer]
7  duration=3.0
```

Contenido del archivo "Timer/Update/config.mako"

```
1 <%def name="sections()">\
2  ${b_sections|i}
3  [timer]
4  duration=${c.config['timer']['duration']}
5  </%def>
```

Figura 34: Ejemplo del patrón de *Centralización de configuración*

completa de las diversas aplicaciones que puede contener el resultado de la composición de la línea de productos de software. En la línea 6 define por ejemplo la sección de la configuración de la *cámara remota* del rasgo *Timer*. En la línea 2 del archivo *config.mako* del rasgo *Timer* se conservan todas las secciones de configuración anteriores. En la línea 4 se obtiene el valor del archivo *Pynion.ini*(**`${c.config['timer']['duration']}`** que en este caso tiene un valor de 3). La configuración contenida en el archivo *pynion.ini* puede ser accedida por medio de la variable *c.config* como podemos ver en la línea 3. Esto permite que desde un solo archivo se pueda controlar la configuración por defecto de todas las aplicaciones.

En la figura 35 vemos un esquema de como funciona el patrón de la configuración centralizada. La configuración del archivo *Pynion.ini* es divida entre todos los rasgos. Los rasgos están diseñados para tomar la configuración de *Pynion.ini* y pasarla al archivo de configuración final de la aplicación miembro de la línea de productos de software. Una vez que se selecciona la composición (rasgo 2 y 4 en este caso) solo la configuración de esos rasgos es incluida en los archivos de configuración de la aplicación final. Con esta metodología sólo el archivo *Pynion.ini* se necesita modificar para cambiar los valores de cualquier parametro de configuración en la línea de productos de software.

La *centralización de la configuración* es un patrón que se puede utilizar para simplificar la administración de una línea de productos de software. Este patrón es exclusivo de *Pynion* y no es general de todos los mecanismos de variación disponibles, muchos de los cuales no consideran la configuración como parte de una línea de productos de software. Con la centralización de la configuración un mismo archivo de configuración se puede transformar en varios archivos de configuración independientes, para hacer que cambios en la configuración de un sistema sea mías sencillo.

4.4.11. Inclusión mutua de rasgos

Como visto en la sección 3.3.3 no todas las composiciones de una línea de productos de software son válidas. La restricciones de las combinaciones sirven para evitar composiciones

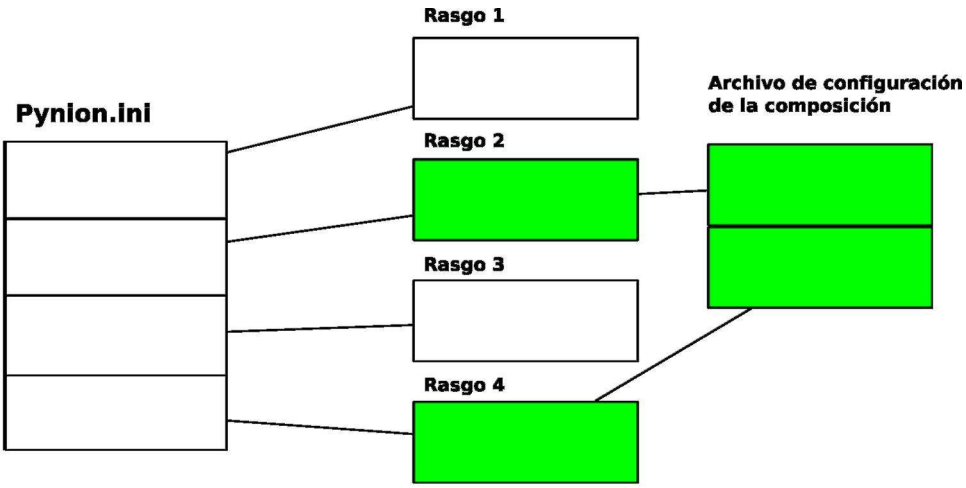


Figura 35: Diagrama del alcance interrasgo

que no podrían ejecutarse o que lógicamente no tendrían ningún sentido. Una de las restricciones más comunes es la dependencia entre rasgos. En esta restricción un rasgo solo puede agregarse a la composición si un otro o varios otros rasgos están presentes en la composición ya sea antes o despues(esto a aplica para dependencias diferentes del rasgo *base*, que se incluye en todas las composiciones de cualquier modo).

La inclusión mutua es una de las restricciones más comunes en las líneas de productos de software. En el caso de la línea de productos de software para la vigilancia remota esta restricción se utilizó en varias ocasiones. Una de ellas es la inclusión de la aplicación *Visor* a la composición solo se puede hacer si la cámara remota transfiera las imágenes a través de *http;* esto quiere decir que el rasgo *http* esta dentro de la composición. En la figura 36 vemos los archivos *root.py* de tanto el rasgo *http* como del rasgo *visor*. En las líneas 6 y 7 vemos que el rasgo *http* publica los valores *http* y *transfer*. El valor *http* es el valor típico que exponen todos los rasgos (esto es su nombre), mientras que *transfer* se refiere al tipo de rasgo que es (transfiere las imágenes a de la cámara remota a otro lado donde serán almacenadas). En el rasgo *Visor* en la línea 11, se verifica que este el valor *http* en los prevalores o los postvalores; esto es en cualquier parte de la composición. Si no esta el valor en la composición se genera un error y se agrega a la lista de errores (línea 12). En la figura vemos que la dependencia se establece usando la instrucción de *not in* (que verifica la pertenencia del valor en la lista de prevalores o postvalores) buscando los valores de los rasgos de los que se depende en la composición.

La inclusión mutua nos permite restringir las composiciones válidas en una línea de productos de software. La inclusión mutua permite generar una dependencia entre dos o mías rasgos dentro de una composición. En *Pynion* este patrón se implementa por medio de los archivos *root.py* que se encargan de la validación de las composiciones.

4.4.12. Rasgos no repetibles

Una de las características de la gran mayoría de los rasgos es que estos no pueden ser utilizados dos veces en una misma composición. Una vez que un rasgo agrega una función muchas veces agregarla una segunda vez no tiene ningún sentido pues ya se tiene la funciona-

Archivo "root.py" de "http"

```
1  import sys
2  sys.path.append("..")
3  from Controller import RootController
4
5  class Root(RootController):
6      def publish(self):
7          return ("http", "transfer")
8
9      def validate(self, prevalues, postvalues):
10         errors = []
11         return errors
```

Archivo "root.py" de "Visor"

```
1  import sys
2  sys.path.append("..")
3  from Controller import RootController
4
5  class Root(RootController):
6      def publish(self):
7          return ("visor",)
8
9      def validate(self, prevalues, postvalues):
10         errors = []
11         if 'http' not in prevalues and 'http' not in postvalues:
12             errors.append('No se puede usar el visor sin la
                   transferencia "http"')
13         return errors
```

Figura 36: Ejemplo de inclusión mutua de rasgos

```
1  import sys
2  sys.path.append("..")
3  from Controller import RootController
4
5  class Root(RootController):
6      def publish(self):
7          return ("timer", "delay")
8
9      def validate(self, prevalues, postvalues):
10         errors = []
11         if 'timer' in prevalues or 'timer' on postvalues:
12             errors.append('No se puede agregar el rasgo "timer" en
                   varias ocasiones')
13         return errors
```

Figura 37: Archivo *root.py* en el rasgo *Timer*

lidad deseada. Así es muchas veces es necesario evitar que un mismo rasgo se incluya varias veces en la misma composición. En *Pynion* esto se hace a través de las funciones *publish* y *validate* del archivo *root.py* de cada rasgo.

En la línea de productos de software para la vigilancia remota en repetidas ocasiones se impidió que un mismo rasgo se reutilizara en varias ocasiones dentro de una misma composición. Una de estas ocasiones fue con el rasgo *Timer* que no tiene sentido que se ejecuten varia veces en una misma aplicación (en ese caso seria más sencillo aumentar el tiempo de espera del rasgo *Timer*). En la figura 37 vemos el código del archivo *root.py* en el rasgo *Timer*. En el archivo vemos que el mismo valor que se publica *(timer* en la línea 7) se verifica que no este presente en la composición (en la línea 11) ni como prevalor ni como postvalor. Este patríon nos permitió evitar que el rasgo pueda ser repetido dentro de una composición de la línea de productos de software ayudando a garantizar que las composiciones sean correctas y no tenga funcionalidad superflua.

4.4.13. Exclusión muta de rasgos

En los rasgos es común que se encuentre un grupo de rasgos en los cuales sólo se puede seleccionar uno. Este patrón de diseño es conocido como la exclusión mutua. Este patrón es tan común que cuenta con su propio símbolo (el arco no relleno) en los diagramas de rasgos. La exclusión mutua impide que varios rasgos implementen la misma funcionalidad y entren en conflicto uno con otros dentro de la misma composición. Así en muchas ocasiones es deseable implementar la exclusión mutua para que cada aplicación sólo tenga un modo de obtener una cierta funcionalidad. Es importante así verificar que todos los rasgos que formen parte de un grupo de exclusión muta implementen la misma funcionalidad pero de modos diferentes.

En la línea de productos de software para la vigilancia remota hubo ocasiones en que se necesito la exclusión muta. La aplicación de la cámara remota sólo puede capturar imágenes de una sola fuente como puede ser una *webCam* o una *cámara ip*. En este caso los rasgos que corresponden a las *webCam* (el rasgo *USB*) y *cámara ip* (el rasgo *IP*) no pueden ser utilizados al mismo tiempo. En la figura 38 vemos el archivo *root.py* del rasgo USB. En la línea 6, vemos

```
1  import sys
2  sys.path.append("..")
3  from Controller import RootController
4
5  class Root(RootController):
6      def publish(self):
7          return ("usb", "capture")
8
9      def validate(self, prevalues, postvalues):
10         errors = []
11         if 'capture' in prevalues or 'capture' in postvalues:
12             errors.append('No se puede usar las camaras USB con
                   otro tipo de captura')
13         return errors
```

Figura 38: Archivo "root.py" del rasgo "USB"

que se publican dos valores *USB* y *capture*. El valor *USB* corresponde simplemente al rasgo, en cambio el valor *capture* tiene que ser publicado por todos los miembros del grupo de exclusión mutua. En la línea 11 se garantiza que no se encuentre el valor *capture* que identifica a todos los miembros del grupo de exclusión (a diferencia de la no repetición de un rasgo en donde se utilizar el valor que identifica el rasgo).

El patrón de diseño de exclusión mutua es útil para evitar que dos patrones implementen la misma funcionalidad. Los grupos de exclusión mutua son grupos de rasgos que implementan la misma funcionalidad de manera diferente. Los grupos de exclusión mutua deben ser nombrados con un valor. El valor que identifica del grupo de exclusión mutua no debe ser repetido en la composición. Este patríon de diseño es uno de las restricciones de composición mías útiles en las líneas de productos de software.

4.5. Desarrollo futuro de la línea de productos de software para la vigilancia remota

En este capítulo hemos visto como fue desarrollado la línea de productos de software para la vigilancia remota y fue descrita a gran detalle. La línea de productos de software creada tiene una gran flexibilidad y se adapta bastante bien a las diferentes necesidades de las aplicaciones del dominio. Sin embargo, al igual que el resto de las líneas de productos de software, es posible seguir expandiendo su funcionalidad agregando nuevos rasgos a la línea de productos de software. Uno de los rasgos posibles a agregar es la implementación de nuevos protocolos de transferencia, como podría ser el uso del correo electrónico para enviar las imágenes. Otro rasgo que se podría implementar sería adaptar la línea de productos de software para la vigilancia remota para aceptar cámaras con detección de movimiento de modo que se envíen imágenes solamente cuando se ha detectado algún cambio. También podría ser deseable compartir estas imágenes con otras aplicaciones, un modo de hacer esto podría ser por medio de servicios web. Finalmente podrían haber nuevos rasgos que se adapten a las necesidades de un cliente específico que no se habían considerado inicialmente en la línea de productos de software. La línea de productos de software para la vigilancia remota tiene un

gran potencial para ser expandida para aumentar la gama de aplicaciones que sean miembros de ella para adaptarse a las necesidades de más clientes potenciales.

5. Resultados

En este capítulo analizaremos los resultados finales del desarrollo de *Pynion* en esta tesis. Se analizara los resultados de obtenidos al desarrollar la línea de productos de software para vigilancia remota usando *Pynion* como una herramienta para mecanismo de variación. En este análisis destacaremos las fortalezas y debilidades de *Pynion* y a partir de estas decidiremos si es adecuado para su propósito principal: el desarrollo de líneas de productos de software en ambientes heterogéneos y legados. Además de esta evaluación, este capítulo incluye una breve descripción de que le podría reservar el futuro a *Pynion.*

5.1. Evaluación de Pynion

Cuando se decidió generar una línea de productos de software para la vigilancia remota en *Pynion* se busco lograr un objetivo principal: evaluar el funcionamiento de *Pynion* como herramienta para mecanismos de variación. Existen varias cualidades que se evaluaron durante el desarrollo de la línea de productos de software para saber si el diseño de *Pynion* era satisfactorio. Las cualidades que se buscaron lograr fueron:

1. Soporte para ambientes heterogéneos
2. Poca invasividad
3. Soporte para modificaciones de diferentes granularidades
4. Soporte para diferentes estrategias de desarrollo
5. Soporte para diferentes tipos de variabilidad
6. Código legible
7. Código Rastreable

Durante el desarrollo de la línea de productos se busco apreciar como se comporto la herra¬ mienta ante estas demandas y ver que funcionalidad ayudo con estas características y cuales le perjudicaron. Una vez hecho este análisis podemos evaluar si *Pynion,* como herramienta para mecanismos de variación, cumplió con sus objetivos establecidos.

5.1.1. Soporte para ambientes heterogéneos

El objetivo principal de *Pynion* es el soporte de ambientes heterogéneos de desarrollo para líneas de productos de software. Uno de los requerimientos importantes es el soportar diferentes lenguajes de programación dentro de una misma línea de productos de software. En el caso de la línea de productos de software para la vigilancia remota se usaron diversos len¬ guajes de programación y formatos de documentos. Los lenguajes de programación utilizados fueron *C#, Python, Javascript,* ademas de las plantillas de *asp.net.* Dentro de los formatos de documentos incluídos en esta línea de productos de software se conto con *xml, css* y ar- chivos de configuración *.ini.* Esta gran diversidad de lenguajes de programación y formatos de archivos es típico en la gran mayoría de sistemas de software modernos y el haber sido incluídos sin problemas en *Pynion* nos indica que se cumplio con este propósito esencial sin problemas.

5.1.2. Poca invasividad

Una de las características importantes para *Pynion* era que esta fuera una herramienta poco invasiva para poder reutilizar los recursos ya existentes. En este caso existen varias pruebas directas e indirectas que este objetivo se logro. Una de las características que hace de *Pynion* una herramienta poco invasiva es la capacidad de esconder la identificación de los puntos de variación detrías del soporte para comentarios de cada lenguaje. Esto permite preparar los archivos para ser alterados por otros rasgos sin modificarlos semánticamente. Las plantillas de Mako fueron escogidas sobre otras por tener una sintaxis poco invasiva comparado con otros sistemas de plantillas. Además también existen pruebas indirectas de esta poca invasividad: el desarrollo de la línea de productos de software para la vigilancia remota fue muy rápido. En el transcurso de dos semanas, se logró pasar del diseño a la implementación de la línea de productos de software completa.

5.1.3. Soporte para modificaciones de diferentes granularidades

El soporte para modificaciones de diferentes granularidades era importante para *Pynion* pues permite mayor control sobre el reuso pues se pueden utilizar fracciones mías grandes del código cuando es posible y más pequeñas cuando no. *Pynion* tiene soporte para agregar, qui¬ tar, o remplazar archivos completos, ya sean binarios o de texto. Esto permite incluir librerias, módulos, clases y demás artefactos completos con la inclusión de los rasgos. Ademas *Pynion* permite incluir o no, los diferentes archivos en la carpeta *add* esto nos da un control mías preciso sobre que archivos se incluyen en una aplicación miembro. Ademaos *Pynion* permite modificaciones a través de plantillas lo cuál le da acceso a modificar bloques de código de un solo archivo. Finalmente, *Pynion* también puede ser modificaciones a través de funciones que dan control total sobre la salida. Toda esta funcionalidad es lo que permite que *Pynion* pueda reutilizar artefactos de cualquier tamaño en una línea de productos de software.

5.1.4. Soporte para varias estrategias de desarrollo

Una parte importante de *Pynion* como herramienta mecanismo de variación es el soporte de distintas estrategias de desarrollo de líneas de productos de software. Esto es particular¬ mente importante para *Pynion* pues este no fuerza una metodología en los desarrolladores. Es difícil juzgar el soporte de diferentes estrategias de desarrollo en *Pynion* pues sólo se realizó una sola línea de productos de software para esta tesis. Sin embargo, existen carac¬ terísticas que si nos permiten visualizar la posibilidad de que estas estrategias estén bien soportadas. El hecho de que se tomo como referencia un par de aplicaciones ya existentes es parte de una estrategia *extractiva.* La reestructuración del código inicial se realizo de tal modo que la estrategia de desarrollo subsecuente fuera aproximandamente *proactiva.* Finalmente, antes de crear el rasgo de *seguridad* hubo necesidad de reestructurar archivos ya marcados con puntos de variación, este tipo de refactorización constante es tipicamente parte de las es¬ trategias *reactivas.* Así, a pesar de que se ha realizado sólo una línea de productos de software en esta tesis, el de la vigilancia remota, hay buenas razones para pensar que *Pynion* puede soportar las distintas estrategias de desarrollo existentes.

5.1.5. Soporte para diferentes tipos de variabilidad

La flexibilidad en *Pynion* es una de las características más deseables. Entre más flexibilidad permita una herramienta para mecanismos de variación sin sacrificar la facilidad de desarrollo, mías poderoso es este. En la sección 4.3 vimos los distintos tipos de variabilidad que se ha utilizado en esta línea de productos de software. Ademas de esta variabilidad, la variabilidad negativa también es soportada directamente por *Pynion* pudiendo eliminar tanto archivos completos, como bloques de texto de los puntos de variación con facilidad. *Pynion* tiene un extenso soporte para variabilidad. Para ayudar en el soporte de los diferentes tipos de variabilidad, *Pynion* cuenta con un amplio juego de herramientas que permiten control total del proceso de remplazo del contenido de los puntos de variación. Estas herramientas permiten a los rasgos el acceso completo a su contexto (que incluye información de otros rasgos en la composición y su configuración). Este acceso al contexto del rasgo es lo que hace factible la variación derivativa. *Pynion* es una herramienta para mecanismo de variación bastante flexible y permite el uso completo del poder de *Python* y sus las librerías haciendo el control total del código generado sea posible. Mucho del desarrollo de *Pynion* se dedico al soporte de variabilidad, convirtiendo a *Pynion* en una herramienta para mecanismos de variación de alta flexibilidad.

5.1.6. Código Legible y Rastreable

Ninguna aplicación o sistema de software se puede desarrollar a la primera sin defectos, incluso es cierto que sólo las aplicaciones de software mías sencillas están libres de defecto. Esta es la razón por la cuál el poder depurar una línea de productos de software es tan importante. Cuando consideramos que *Pynion* es una herramienta para mecanismos de variación que genera código fuente, es importante saber de donde es que surgió este código, esto es poder rastrear el código generado a su fuente. Para ayudar con la depuración del código fuente de las aplicaciones miembro de la línea de productos, *Pynion* permite mantener la indentación en el código generado de dos maneras. El primero es que el código generado se reindenta con la misma indentación que el bloque de código que se extrajo originalmente. El segundo, es que los bloques que se integran en el resultado de la plantilla pueden mantener la indentación de la plantilla por medio del filtro de Mako i. La posibilidad de mantener la indentación ayuda a que el código de las aplicaciones miembro de la línea de productos sean mías legibles y fáciles de seguir haciendo que la depuración del código sea mías sencilla que al utilizar otros mecanismos de variación por plantillas. En cuanto al rastreo, *Pynion* genera el archivo *output.txt* que contiene información de rastreo sobre los archivos de la aplicación resultado de la composición. Este rastreo es muy útil para identificar que rasgos se encargaron de crear, modificar y eliminar cada archivo al reportar cada acción que se tomo sobre el. Sin embargo, el rastreo solo se realiza a nivel archivo y no rastrea modificaciones sobre fragmentos del código fuente, esto lo hace tremendamente útil, pero aún así limitado. Podemos concluir que el soporte para depuración y rastreo en *Pynion* es bastante bueno.

5.1.7. Limites de Pynion

El desarrollo de la línea de productos de software para la vigilancia remota con *Pynion* no resulto ser sin dificultades. Alguno de estos problemas se debieron a los límites que tiene *Pynion* como herramienta para mecanismos de variación. Uno de los problemas de *Pynion* fue que ciertos archivos, como los de configuración de rasgo (el archivo *feature.ini*), la estructura

de plantillas y los archivos de validación de la composición (el archivo *root.py),* de rasgos que implementaban funcionalidad similar solían ser muy parecidos o idénticos. Esto es un indicio de quizás se pueda extraer y centralizar esta información o al menos es posible que estas estructuras puedan ser automatizadas. Otro punto débil de *Pynion* es el rastreo que no permite rastrear modificaciones en un solo punto de variación al archivo que los genero. Finalmente, y en una escala menor, *Pynion* no fuerza a modificar el estilo de programación en los archivos de texto, pero si favorece ciertas estructuras sobre otras (por ejemplo, importar una sola librería por línea de código fuente en las plantillas) que si bien no fue muy significativo tampoco es deseable que se necesite adaptar el código fuente de este modo. *Pynion* es una excelente herramienta para mecanismos de variación para líneas de productos de software en ambientes heterogéneos, sin embargo tiene algunos límites que es importante tomar en cuenta cuando se desee utilizar.

5.1.8. Resultados de la Evaluación

En las subsecciones anteriores se resumió como *Pynion* puede ser contrastado contra sus objetivos. *Pynion* logró cumplir, a pesar de algunas límitantes, con el propósito que fue creado. La principal demostración del potencial de *Pynion* fue el poder crear una línea de productos de software para la vigilancia remota en un ambiente heterogéneo. Ademas de este soporte también es importante remarcar que el desarrollo fue sencillo y rápido. Todo esto nos lleva a concluir que *Pynion* es una herramienta adecuada para mecanismos de variación en ambientes heterogéneos.

5.2. Relación de Pynion con otros mecanismos de variación

Pynion es una herramienta para mecanismos de variación que no esta ligado a ningún mecanismo de variación en particular. *Pynion* puede usar diferentes metologias para la realización de la línea de productos de software. En la tabla 1 vemos un listado de las características de cada mecanismo y como estas características pueden ser implementadas usando *Pynion*

5.3. Desarrollo futuro de Pynion

Pynion ha demostrado ser útil en la construcción de líneas de productos de software para ambientes heterogéneos y legados. Actualmente, *Pynion* es sólo un prototipo y su fun¬ cionalidad debe ser ampliada antes de ser usada como una herramienta para mecanismos de variación general. Parte del desarrollo futuro de *Pynion* se debe dar en áreas en que su funcio¬ nalidad se encontro más limitada. Otra parte del desarrollo debe incluir nueva funcionalidad que complemente la funcionalidad ya existente. Como todas las herramientas de software, *Pynion* tiene espacio para recibir mejoras corrigiendo sus límites y ampliando su alcance.

Una de las áreas en que *Pynion* podría recibir mejoras significativas es en el proceso de automatización. Como se había visto en la sección anterior existen un par de archivos que se suelen repetir entre rasgos que implementan la misma funcionalidad: el archivo *feature.ini* y *root.py.* En el futuro *Pynion* podría crear estos archivos para los grupos de rasgos con características similares. La creación de los archivos de plantillas *Mako* también se puede automatizar parcialmente a partir de los archivos originales marcados. Finalmente, *Pynion* podría incluir en el futuro soporte para crear la composición a partir de *Wizards,* simplificando el proceso de selección y evitando errores en las composiciones.

Plantillas C + +	
Característica del Mecanismo	*Característica de Pynion*
Opera en Capas	Los rasgos se utilizan como capas
Lenguaje secundario de plantillas	Plantillas Mako
Puede hacer cambios en varias clases a la vez	los rasgos operan en varias clases a la vez
Generadores	
Característica del Mecanismo	*Característica de Pynion*
Trabajo en capas	Los rasgos se utilizan como capas
Validación de composiciones	Sistema equivalente de validaciones
Genera el código final	Genera el código final
Refinación a pasos	Patrón de diseño de refinación a pasos
Aspectos	
Característica del Mecanismo	*Característica de Pynion*
Modificación sin alterar los archivos originales	Agregar o sustituye archivos completos
Utilizar el mismo cambio en varias puntos de variación	Usar la misma plantilla o función para varios puntos de variación
Organiza las modificaciones en aspectos	Organiza las modificaciones en rasgos
Frames	
Característica del Mecanismo	*Característica de Pynion*
Trabajo por medio de plantillas	Trabajo por medio de plantillas Mako
Parametrización de las composiciones	Parametrización a través de configuraciones
Aplicación de plantillas en orden jerárquico	Se puede llamar las funciones de modificación jerárquicamente

Cuadro 1: Relación de Pynion con otros mecanismos de variación

Otro punto de crecimiento de *Pynion* puede ser agregar nueva funcionalidad. La nueva funcionalidad de *Pynion* se debe centrar en el objetivo principal de la herramienta: soporte para ambientes heterogéneos. Una de las mejoras factibles es ampliar el soporte para otros lenguajes para plantillas ademas de Mako. Esto permitiría seleccionar el lenguaje para plantillas que mejor funcione para cada lenguaje de programación o formato de documento haciendo el desarrollo más fácil en una mayor cantidad de lenguajes de programación. También en el futuro, *Pynion* podría ser aún menos invasivos al expandir la funcionalidad de Mako para que el código fuente sea más parecido al código fuente original. Finalmente, *Pynion* podría agregar nueva funcionalidad que este adaptada en particular para algun lenguaje de programación específico. Esto permitiría que el desarrollo en ambientes heterogéneos sea aún más sencillo con la herramienta descrita en esta tesis.

El objetivo a largo plazo de *Pynion* es ser una herramienta que pueda utilizarse en un ambiente de desarrollo de software empresariales. Para poder utilizarse en ambientes de desarrollo es necesario que *Pynion* mejore en su capacidad para soportar el desarrollo de múltiples usuarios al mismo tiempo en un ambiente empresarial. Una de los cambios que podría ayudar con este objetivo es permitir que *Pynion* ejecute comandos antes y después de crear un miembro de la línea de productos de software. Esto permitiría que se pudieran ejecutar procesos de limpieza antes de mezclar el código fuente y de compilar los resultados después de este proceso. También, como mencionado con anterioridad, la mejora del rastreo del código fuente generado es importante para poder revisar el código fuente creado por la línea de productos de software.

Pynion en este momento es sólo un prototipo de una herramienta para mecanismos de variación. En su estado actual ha demostrado ser de gran utilidad en el desarrollo de líneas de productos de software en ambientes heterogéneos. Sin embargo, como en la mayoría de los sistemas de software, existen mejoras posibles para su funcionalidad. En *Pynion* estas mejoras se podrían sentar sobre la automatización del desarrollo, aminorar la invasividad sobre el código fuente original y la preparación para el desarrollo en ambientes de empresariales.

6. Conclusiones

Durante el transcurso de esta tesis, nos enfocamos en crear una herramienta para mecanismos de variación para líneas de productos de software en ambientes heterogéneos y legados. Para lograr esto nos aseguramos de que el mecanismo de variación tuviera ciertas características como: facilidad de desarrollo, no invasividad y flexibilidad. Esta herramienta puede hacer mías fácil la adopción de las estrategias de líneas de productos en empresas pues éstas, regularmente, lidian con sistemas de software complejos y heterogéneos. En respuesta a esto la tesis presentó un prototipo de mecanismo de variación para líneas de productos de software en ambientes heterogéneos y legado: *Pynion.*

En esta tesis definimos primero el contexto de las líneas de productos de software para entender mejor el dominio en que *Pynion* fue desarrollado. También se presentó otros mecanismo de variación ya existentes en la actualidad y que no son adecuados para el trabajo en ambientes heterogéneos. Después presentamos a *Pynion* como mecanismo de variación para líneas de productos de software en ambientes heterogéneos. La tesis describió a fondo los objetivos y el diseño de *Pynion* incluyendo una manual de usuarios y un compendio de herramientas que directa o indirectamente influyeron en su desarrollo. En esta tesis, *Pynion* se puso a prueba con el desarrollo de una línea de productos de software en el ambiente de la vigilancia remota. El dominio de las aplicaciones de la vigilancia remota resulto ser un dominio fuertemente heterogéneo que puso a prueba las capacidades y alcance del diseño de *Pynion.* Al desarrollar una línea de productos de software nos dimos cuenta de patrones de diseño que eran inherentes en la utilización de *Pynion.* Finalmente, analizamos los resultados de utilizar *Pynion* y esbozamos que dirección podría tomar este mecanismo de variación a futuro. Los resultados del trabajo resultaron alentadores y prueban que el mecanismo de variación aquí descrito podría ayudar rn la adopción de las estrategias de líneas de productos de software en ambientes heterogéneos y legados.

A n e x o s

A. Tabla de términos y su traduccion al ingles

El área de estudio de las líneas de productos de software es una área relativamente poco explorada de la ingeniería de software. Mientras los términos en el idioma ingles son casi universales, su traducción al español puede variar de autor a autor. Para ayudar a entender este trabajo y relacionarlos con otros en el mismo campo de estudio en ingles y español se provee una tabla con los términos utilizados en este escrito, la palabras en inglés que traducen y una lista de otros términos que podríamos encontrar en español si existiesen.

Termino en Español	Original en Inglés	Otras Opciones
Comunalidad	Commonality	Similitud
Conjunto	Set	Juego
Dominio	Domain	Campo
Forzoso	Mandatory	Obligatorio
Juego	Set	Conjunto
Linea de Productos	Product Line	Lineas de Producción
Marcaje	Marking	Marcación
Mecanismo de Variación	Variation Mechanism	
Núcleo	Core	Central
Núcleo	Kernel	Grano
Plantilla	Template	Patrón
Preocupación	Concern	Interés
Punto de Variación	Variation Point	
Rasgo	Feature	Característica, Propiedad
Recurso	Asset	activo
Refactorización	Refactorizatión	
Refinamiento	Refinement	
Reino	Realm	Dominio, esfera
Repositorio	Repository	Almacén, Depósito
Variabilidad	Variability	
Variación	Variation	
Variante	Variant	

Referencias

[1] H. Gomaa, *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures.* Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2004.

[2] E. Wohlstadter, S. Jackson, and P. Devanbu, "Dado: enhancing middleware to support crosscutting features in distributed, heterogeneous systems," *Software Engineering, 2003. Proceedings. 25th International Conference on,* pp. 174-186, 3-10 May 2003.

[3] R. LaRowe and T. Probert, "Heterogeneous by design: An environment for exploiting heterogeneity," *Heterogeneous Processing, 1993. WHP 93. Proceedings. Workshop on,* pp. 84-91, 13 Apr 1993.

[4] G. M. e. a. Scott Loveland, *Software Testing Techniques: Finding Defects that Matter.* Hingham, MA, USA: CharlesRiver Media, 2004.

[5] R. Couto Antunes da Rocha and M. Endler, "Middleware: Context management in heterogeneous, evolving ubiquitous environments," *Distributed Systems Online, IEEE,* vol. 7, no. 4, April 2006.

[6] D. Batory, J. Sarvela, and A. Rauschmayer, "Scaling step-wise refinement," *Software Engineering, IEEE Transactions on,* vol. 30, no. 6, pp. 355-371, June 2004.

[7] "Welcome to video surveillance." http://videosurveillance.com, 2008.

[8] "Video surveillance and monitoring." http://www.cs.cmu.edu/vsam, 2000.

[9] P. Clements and L.Northrop, *Software Product Lines: Practices and Patterns.* Addison-Wesley Professional, 2002.

[10] D. Weiss and R. Lai, *Software Product Line Engineering: A Family-Based Software De¬velopment Process.* Reading, MA: Addison-Wesley, 1999.

[11] K. Pohl, G. Bockle, and F. van der Linden, *Software Product Line Engineering: Foun¬dations, Principles and Techniques.* Springer, 2005.

[12] J. D. McGregor, L. M. Northrop, S. Jarrad, and K. Pohl, "Guest editors' introduction: Initiating software product lines," *IEEE Software,* vol. 19, no. 4, pp. 24-27, 2002.

[13] C. Krueger, "Eliminating the adoption barrier," *Software, IEEE,* vol. 19, pp. 29-31, Jul/Aug 2002.

[14] P. Clements, "Being proactive pays off," *Software, IEEE,* vol. 19, no. 4, pp. 28, 30-, Jul/Aug 2002.

[15] F. Bachmann and P. C. Clements, "Variability in software product lines," tech. rep., CMU/SEI, 2005.

[16] C. W. Krueger, "New methods in software product line practice," *Commun. ACM,* vol. 49, no. 12, pp. 37-40, 2006.

[17] C. W. Krueger, "Towards a taxonomy of software product lines," in *Proceedings of the 5th International Workshop on Product Family Engineering.*

[18] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (foda) feasibility study," tech. rep., Carnegie-Mellon University Software Engineering Institute, November 1990.

[19] C. Gacek and M. Anastasopoulos, "Implementing product line variabilities," in *SSR '01: Proceedings of the 2001 symposium on Software reusability,* (New York, NY, USA), pp. 109-117, ACM, 2001.

[20] K. Czarnecki and U. W. Eisenecker, "Components and generative programming (invited paper)," in *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering,* (London, UK), pp. 2-19, Springer-Verlag, 1999.

[21] D. C. Sharp, "Containing and facilitating change via object oriented tailoring techni¬ques," in *Proceedings of The First Software Product Line Conference,* 2000.

[22] J.-C. T. Pierre-Yves Schobbens, Patrick Heymans, "Feature diagrams: A survey and a formal semantics," *Requirements Engineering Conference, 2006. RE 2006. 14th IEEE International,* pp. 136-145, 2006.

[23] M. Antkiewicz and K. Czarnecki, "Featureplugin: feature modeling plug-in for eclipse," in *eclipse '04: Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange,* (New York, NY, USA), pp. 67-72, ACM, 2004.

[24] A. Metzger, K. Pohl, P. Heymans, P.-Y. Schobbens, and G. Saval, "Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis," *Requirements Engineering Conference, 2007. RE '07. 15th IEEE International,* pp. 243-253, 15-19 Oct. 2007.

[25] J. Liu, D. Batory, and C. Lengauer, "Feature oriented refactoring of legacy applications," in *ICSE '06: Proceeding of the 28th international conference on Software engineering,* (New York, NY, USA), pp. 112-121, ACM Press, 2006.

[26] D. Spinellis, "Notable design patterns for domain specific languages," *Journal of Systems and Software,* vol. 56, pp. 91-99, feb 2001.

[27] M. Fowler, "Language workbenches: The killer-app for domain specific languages?." http://www.martinfowler.com/articles/languageWorkbench.html, 2005.

[28] D. Batory, C. Johnson, B. MacDonald, and D. von Heeder, "Achieving extensibility through product-lines and domain-specific languages: a case study," *ACM Trans. Softw. Eng. Methodol.,* vol. 11, no. 2, pp. 191-214, 2002.

[29] M. VanHilst and D.Ñotkin, "Using role components in implement collaboration-based designs," *SIGPLAN Not.,* vol. 31, no. 10, pp. 359-369, 1996.

[30] Y. Smaragdakis and D. Batory, "Mixin layers: an object-oriented implementation tech¬nique for refinements and collaboration-based designs," *ACM Trans. Softw. Eng. Methodol.,* vol. 11, no. 2, pp. 215-255, 2002.

[31] D. Batory and S. O'Malley, "The design and implementation of hierarchical software systems with reusable components," *ACM Trans. Softw. Eng. Methodol.*, vol. 1, no. 4, pp. 355-398, 1992.

[32] B. Batory, D.; Geraci, "Validating component compositions in software system genera¬ tors," *Software Reuse, 1996., Proceedings Fourth International Conference on*, pp. 72-81, 23-26 April 1996.

[33] V. Singhal and D. Batory, "P++: A language for large-scale reusable software compo¬ nents," in *WISR*.

[34] S. Thaker, D. Batory, D. Kitchin, and W. Cook, "Safe composition of product lines," in *GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering*, (New York, NY, USA), pp. 95-104, ACM, 2007.

[35] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *Proceedings European Conference on Object-Oriented Programming* (M. Aksit and S. Matsuoka, eds.), vol. 1241, pp. 220-242, Berlin, Heidelberg, and New York: Springer-Verlag, 1997.

[36] M. Mezini and K. Ostermann, "Variability management with feature-oriented program¬ ming and aspects," in *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, (New York, NY, USA), pp. 127-136, ACM, 2004.

[37] C. Kastner, S. Apel, and D. Batory, "A case study implementing features using aspectj," *Software Product Line Conference, 2007. SPLC 2007. 11th International*, pp. 223-232, 10-14 Sept. 2007.

[38] S. M. Swe, H. Zhang, and S. Jarzabek, "Xvcl: a tutorial," in *SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering*, (New York, NY, USA), pp. 341-349, ACM, 2002.

[39] P. G. Bassett, *Framing software reuse: lessons from the real world*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1997.

[40] "Mako templates for python." http://makotemplates.org, 2008.

[41] "Python programming language: Official website." http://www.python.org, 2008.

[42] "Reading and writing config files." http://www.voidspace.org.uk/python/configobj.html, 2008.

[43] "Easyinstall-the peaks developer center." http://peak.telecommunity.com/DevCenter/EasyInstall, 2008.

[44] "Python package index." http://pypi.python.org/pypi, 2008.

[45] G. van Rossum, "Python tutorial." http://docs.python.org/tut/, 2008.

[46] G. van Rossum, "Python library reference." http://docs.python.org/lib/lib.html, 2008.

[47] "Mako documentation." http://www.makotemplates.org/docs/, 2008.

[48] M. Lutz, *Programming Python.* Sebastapol, CA, USA: O'Reilly, Inc.

[49] "Cheetah - the python powered template engine." http://www.cheetahtemplate.org/, 2008.

[50] "Genshi- generate output for the web." http://genshi.edgewall.org/, 2008.

[51] N. Batchelder, "Ned batchelder: Cog." http://nedbatchelder.com/code/cog/index.html, 2008.

[52] S. A. Hendrickson and A. van der Hoek, "Modeling product line architectures through change sets and relationships," in *ICSE '07: Proceedings of the 29th International Confe¬ rence on Software Engineering,* (Washington, DC, USA), pp. 189-198, IEEE Computer Society, 2007.

[53] M. Gritsch, "Videocapture." http://videocapture.sourceforge.net/, 2008.

[54] W. Holcomb, "Where will's projects come to die." http://odin.himinbi.org/, 2007.

[55] R. Helm, "Patterns in practice," in *OOPSLA '95: Proceedings of the tenth annual confe¬ rence on Object-oriented programming systems, languages, and applications,* (New York, NY, USA), pp. 337-341, ACM, 1995.

[56] D. Batory, J. Liu, and J.N. Sarvela, "Refinements and multi-dimensional separation of concerns," *SIGSOFT Softw. Eng. Notes,* vol. 28, no. 5, pp. 48-57, 2003.