

Generador de Código Utilizando el Paradigma de Líneas de Producto Software
Code Generator Using the Software Product Lines Paradigm

Jaime P. Sayago Heredia

Pontificia Universidad Católica del Ecuador, Sede Esmeraldas.

La correspondencia sobre este artículo debe ser dirigida a Jaime P. Sayago Heredia.

Email: jaime.sayago@pucese.edu.ec

Fecha de recepción: 1 de marzo de 2018.

Fecha de aceptación: 19 de abril de 2018.

¿Cómo citar este artículo? (Normas APA): Sayago Heredia, J.P. (2018). Generador de Código Utilizando el Paradigma de Líneas de Producto Software. Revista Científica

Hallazgos21 ,3,(2), 190- 212. Recuperado de <http://revistas.pucese.edu.ec/hallazgos21/>

Revista Científica Hallazgos21. **ISSN 2528-7915. Indexada en Latindex.**

Periodicidad: cuatrimestral (marzo, julio, noviembre).

Director: José Suárez Lezcano. Teléfono: (593)(6) 2721459, extensión: 163.

Pontificia Universidad Católica del Ecuador, Sede Esmeraldas. Calle Espejo, Subida a Santa Cruz, Esmeraldas. CP 08 01 00 65. Email: revista.hallazgos21@pucese.edu.ec.

<http://revistas.pucese.edu.ec/hallazgos21/>

Resumen

En esta investigación se elaboró un generador de código utilizando el paradigma de líneas de producto software junto con otras metodologías de desarrollo por analogía. En la actualidad las aplicaciones de desarrollo se enfocan a un ambiente web y requieren de seguridad, mensajería, control de usuarios, etc., por lo que cada día se vuelven más complejas. Una línea de producto software ofrece una alternativa para hacer frente a esa complejidad proporcionando herramientas que reutilicen el código y se mejore el desempeño, la calidad y el tiempo en el desarrollo de un producto software. El objetivo principal de este trabajo es dar a conocer las ventajas y utilidad potencial de la línea de productos software construido por analogía y la mejora en el desarrollo de una aplicación web Java EE. El generador pretende automatizar el proceso de codificación de las anotaciones de persistencia e implementación de los métodos con Java Persistence API (JPA) del Data Access Object (DAO) de la Entidad para el acceso a la base de datos. El prototipo fue desarrollado en lenguaje de programación Ruby, librerías Gtk2 para el entorno gráfico, ERB para el manejo de archivos de texto a través de un lenguaje de plantillas. Esta herramienta aumenta el desempeño de desarrollo y una disminución del tiempo de codificación comprobado por la estimación por intervalo que se realizó a un grupo de programadores.

Palabras clave: ingeniería de software; líneas de producto software; generación código; lenguaje de programación ruby; lenguajes específicos del dominio.

Abstract

In this research, a code generator was developed using the software product lines paradigm along with other development

methodologies by analogy. Currently, development applications are focused on a web environment and require security, messaging, user control, etc., so each day they become more complex. A software product line offers an alternative to address that complexity by providing tools that reuse code and improve performance, quality, and time in software product development. The main objective of this work is to present the advantages and potential usefulness of the software product line built by analogy and the improvement in the development of a Java EE web application. The generator intends to automate the process of encoding persistence annotations and implementation of methods with the Java Persistence API (JPA) of the Data Access Object (DAO) of the Entity for access to the database. The prototype was developed in Ruby programming language, Gtk2 libraries for the graphical environment, ERB for handling text files through a template language. This tool increases the development performance and a reduction of the coding time verified by the interval estimate that was made to a group of programmers.

Keywords: software engineering; software product lines; generation code; ruby programming language; domain specific language.

Generador de Código Utilizando el Paradigma de Líneas de Producto Software

Las líneas de productos software (LPS) buscan explotar los puntos comunes entre los sistemas de un determinado dominio de problema mientras que maneja las variabilidades entre ellos de una manera sistemática (Pohl, Böckle, & Linden, 2005). Las LPS hacen referencia a la reutilización del código para mejorar la calidad de los sistemas y reducir los costos de desarrollo y

mantenimiento (Heradio, 2007), al aplicar en la fabricación de software principios de economía de escala en la producción de software, como en la producción de bienes físicos, cuando múltiples copias de una implementación inicial se producen mecánicamente a partir de prototipos desarrollados por ingenieros (Greenfield & Short, 2003). Y las economías de alcance que surgen cuando se usan los mismos estilos, patrones y procesos para desarrollar múltiples diseños relacionados que permite utilizar los mismos lenguajes, bibliotecas y herramientas para desarrollar las nuevas implementaciones a partir de los creados anteriormente (K. Czarnecki & Eisenecker, 1999). Para la utilización de una línea de producto software, es necesario implementar métodos como la ingeniería de dominio y la gestión de la variabilidad mediante modelos de características (Laguna, González-Baixauli, & Marqués, 2007), siendo el resultado de su aplicación, el poder satisfacer los respectivos requerimientos para el desarrollo del producto.

En conjunto con el paradigma de LPS, haremos uso de otra metodología para el desarrollo del producto Ejemplar Driven Development (EDD) (Ruben Heradio, Fernandez-Amoros, de la Torre, & Abad, 2012). El punto de partida de EDD es cualquier producto de dominio construido utilizando la ingeniería de software convencional. Implícitamente, este ejemplar implementa la intersección de todos los requisitos del producto del dominio. A continuación, el ejemplo se flexibiliza para satisfacer los requisitos de variables de dominio que están fuera de la intersección, es decir, una relación de analogía se define de una manera formal para derivar productos automáticamente a partir del molde o ejemplar que lo llamaremos.

Para facilitar la personalización en masa, los artefactos utilizados en diferentes productos tienen que ser suficientemente adaptables para que sean capaces de adaptarse a los diferentes sistemas creados en la línea de productos. Esto significa que a lo largo de todo el proceso de desarrollo tenemos que identificar y describir dónde los productos de la línea de productos pueden diferir en cuanto a las características que proporcionan, los requisitos que cumplen, o incluso en cuanto a la arquitectura subyacente, etc. Por lo tanto, tenemos que ofrecer flexibilidad en todos esos artefactos para apoyar la personalización masiva. Por ejemplo: diferentes automóviles de la misma línea de productos pueden tener diferentes limpiaparabrisas y lavaparabrisas. Diseñamos los coches de forma que permitan soportar los diferentes limpiaparabrisas, sus diferentes tamaños, etc. Esta flexibilidad conlleva una serie de limitaciones. Si usted conduce un descapotable, no querría que un limpiaparabrisas en la parte trasera. Por lo tanto, la selección de un coche descapotable significa la flexibilidad de que la disponibilidad de los limpiaparabrisas y las arandelas del parabrisas está restringida (Pohl et al., 2005).

La flexibilización en una línea de productos software se descompone en el análisis de los requisitos productos, la definición de una interfaz que facilite la especificación abstracta de los productos y la implementación de la flexibilización, que a partir de las especificaciones abstractas obtenga los productos finales correspondientes (Ruben Heradio et al., 2012).

El resultado de la flexibilización es un compilador de lenguaje específico de dominio (DSL) que se utiliza para obtener los productos automáticamente. Por lo

tanto, la motivación es demostrar el beneficio que obtenemos al aplicar una LPS, la mejora en el desarrollo en un Data Access Object (DAO) al automatizar y reutilizar el código para implementar Entities Java Beans (EJB). Al aplicar el proceso de EDD el resultado será un DSL compilador que genera automáticamente productos. Se propone una herramienta desarrollada en código abierto para que pueda tener participación de otros programadores y contribuir en su mejoramiento. Nos referiremos a este producto inicial como el dominio ejemplar (Heradio, 2007). La Figura 1 ilustra este enfoque, donde el generador de un compilador DSL es otro compilador que se utiliza para adaptar un ejemplo según las especificaciones de la fuente DSL. La figura también representa una posible descomposición de este subcompilador en subgeneradores responsables de diferentes tipos de variabilidad.

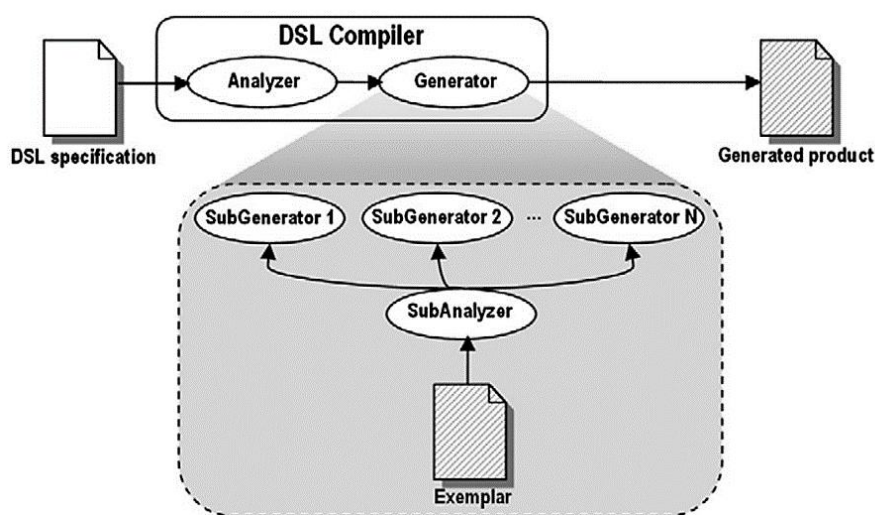


Figura 1. DSL compilador basado en un ejemplar de dominio.

Fuente: <http://e-spacio.uned.es/fez/eserv/tesisuned:IngInf-Rheradio/Documento.pdf>

Este artículo se presenta de la siguiente manera: en la sección 2 describe de desarrollo teórico, los materiales y métodos utilizados para la construcción del

generador, su uso concreto y respectivo resultado, finalmente, la sección 3 puntualiza las conclusiones obtenidas del presente estudio y trabajos a futuro.

Método

Historia Líneas de Producto Software

La programación generativa (GP, Generative Programming) es un paradigma de ingeniería del software basado en el desarrollo de familias de sistemas. El producto final de la GP es un modelo generativo capaz de sintetizar todos los programas de una familia a partir de especificaciones de alto nivel de abstracción (D.-I. K. Czarnecki, 1999). El resultado de un modelo generativo no tiene por qué ser un programa completo. Puede ser un programa parcial, un componente de otros programas o cualquier software (Jarzabek & Seviara, 2000). Apoyándose en la ingeniería de dominio K. Czarnecki y U. Eisenecker (D.-

I. K. Czarnecki, 1999) proponen un proceso de desarrollo de modelos generativos organizado en las siguientes etapas: análisis del dominio del modelo, diseño del dominio e implementación del dominio. Existen varias metodologías para analizar dominios: FAST (*Family-Oriented Abstraction, Specification and Translation*), ODM (*Organization Domain Modeling*), FODA (*Feature-Oriented Domain Analysis*) (Carneiro Roos, s.f.).

Las factorías de software. Son fábricas o complejos industriales que producen artículos en masa y suelen caracterizarse por disminuir los gastos de transporte

centralizando la producción, descomponer el proceso de fabricación en tareas elementales estandarizadas, disponer de mecanismos de control para asegurar la calidad de los productos, contar con trabajadores muy especializados y aplicar un alto grado de automatización del trabajo y de reutilización de artefactos (Garzías & Piattini, 2007). Los elementos fundamentales de una factoría de software son el esquema de la factoría que es un documento que recopila de forma estructurada los productos que debe producir la línea (archivos de código fuente, documentos XML, modelos, ficheros de configuración, archivos SQL). Los elementos constituyentes de los productos (componentes software, patrones de diseño, marcos de trabajo, DSLs, herramientas y lenguajes para desarrollar los distintos artefactos). Procesos sobre cómo desarrollar, configurar y reutilizar los elementos constituyentes y cómo combinarlos para crear productos (Greenfield & Short, 2003). Los procesos fundamentales para la construcción de una factoría de software son el desarrollo de una línea de productos y el desarrollo de un producto concreto de una línea.

La propuesta del desarrollo dirigido por modelos (MDA) la promueve el Object Management Group (OMG), un consorcio de empresas (IBM, Borland, Hewlett-Packard o Boeing entre otras) que produce y mantiene una serie de especificaciones para permitir la interoperabilidad entre aplicaciones software. Según la guía de MDA, MDA is an approach to using models in software development (Miller, 2007). La principal característica diferenciadora de MDA respecto a los enfoques tradicionales para el desarrollo de software se encuentra en el uso de los modelos como el recurso principal en el proceso de desarrollo. MDA propone

que los sistemas software sean generados directamente a partir de modelos de dicho sistema software. OMG propone y promueve el uso de diversos lenguajes relacionados con la creación y gestión de modelos (UML, MOF, CWM y QVT) como mecanismos básicos para soportar esta estrategia (Barbosa, Contreras, & Rodriguez, 2005). Como comenta Muñoz (Muñoz & Pelechano, 2004) separan por un lado el enfoque de Desarrollo Dirigido por Modelos, y por otro la propuesta MDA de OMG. Siguiendo este criterio podemos diferenciar entre: Model Driven Development (MDD) que es una aproximación al desarrollo de software basado en el modelado del sistema software y su generación a partir de los modelos. Al ser únicamente una aproximación, sólo proporciona una estrategia general a seguir en el desarrollo de software, pero no define ni técnicas a utilizar, ni fases del proceso, ni ningún tipo de guía metodológica. Model Driven Architecture (MDA) que es un estándar de OMG que promueve el MDD y agrupa varios lenguajes que pueden usarse para seguir este enfoque. MDA posee el valor añadido de proporcionar lenguajes con los que definir métodos que sigan MDD. Por lo tanto, MDA tampoco define técnicas, etapas, artefactos, etc. MDA sólo proporciona la infraestructura tecnológica y conceptual con la que construir estos métodos MDD. En resumen, podemos afirmar que la propuesta MDA proporciona una infraestructura para la construcción de métodos de desarrollo de software que sigan el MDD (Muñoz & Pelechano, 2005). En resumen la GP, las factorías del software y el MDD persiguen la reutilización sistemática de software y el aumento de la abstracción de los lenguajes formales de desarrollo abordando la resolución colectiva de programas en dominios específicos y

realizando un uso intensivo de generadores de código (Heradio, 2007).

Líneas de Producto Software

Una línea de productos de software es un grupo de productos que comparten un conjunto de características comunes y administradas. Los productos satisfacen las necesidades específicas de un mercado o misión en particular, y se desarrollan a partir de un conjunto común de activos básicos de una manera prescrita. No es únicamente una simple reutilización o de una estrategia de desarrollo basada en componentes, una línea de productos de software permite a una organización gestionar y evolucionar su familia de productos de forma holística, como una entidad única y unificada. La ingeniería de línea de productos es una subdisciplina de ingeniería de software en crecimiento, y muchas organizaciones incluyendo Philips, Hewlett-Packard, Raytheon, y Cummins, lo están usando para lograr ganancias extraordinarias en productividad, tiempo de comercialización y calidad del producto (P. C. Clements, Jones, Northrop, & McGregor, 2005)

Clements define dos actividades para el desarrollo de una Línea de Producto Software (P. Clements & Northrop, 2015): La primera se encarga de la Ingeniería de Dominio, el cual es definido como core asset

development. Este equipo es responsable de desarrollar los

elementos comunes al dominio: estudiar el dominio, definir su alcance (requisitos) dentro

del mercado objetivo de la LPS, definir las features, implementar los core assets reutilizables y su mecanismo de variabilidad, y establecer cómo es el plan de producción. Y la segunda que se encarga de la Ingeniería de Producto definido como product development. Sus cometidos incluyen desarrollar los productos para clientes concretos, a partir de los recursos basados no en los requisitos del dominio, sino en requisitos concretos de clientes. Para ello, este segundo equipo utiliza los recursos creados por el equipo anterior. La Figura 2 muestra gráficamente el proceso de ingeniería de producto y su relación con la ingeniería de dominio.

Este modelo considera la reutilización desde las perspectivas de las dos ingenierías:

- Desarrollo de software para la reutilización: el propósito es producir componentes de software

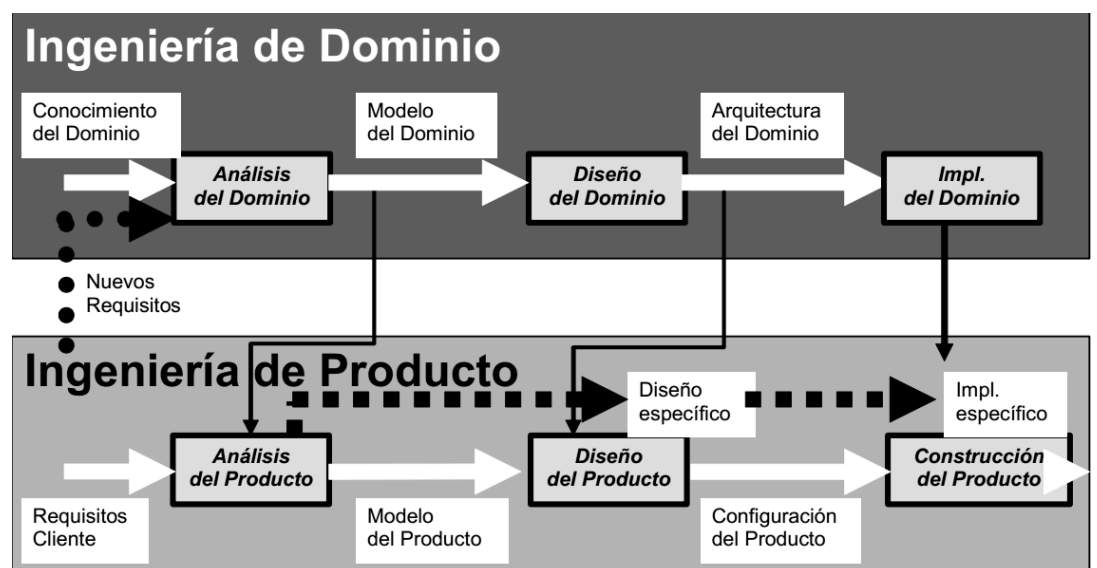


Figura 2. Procesos en Líneas de Producto

Fuente: Clements, P. C., Jones, L. G., Northrop, L. M., & McGregor, J. D. (2005). Project management in a software product line organization. *IEEE Software*, 22(5), 54–62.

reutilizables. A este proceso se le denomina Ingeniería de Dominio.

- Desarrollo de software con reutilización: su propósito es desarrollar software reutilizando componentes existentes. Este proceso se llama Ingeniería de Aplicación.

Modelo de Características para Tratar la Variabilidad

La variabilidad se define, según Svahnberg (Van Gorp, Bosch, & Svahnberg, 2000) como la habilidad de cambio o de personalización de un sistema. Para la definición de requisitos de una línea de productos, hay que prestar especial atención al análisis de la parte común y la parte variable, estableciendo las dependencias que existen entre ellas.

El modelado de características (Feature modeling) está integrado en la metodología FODA y ha sido aplicado con éxito en el desarrollo de sistemas de telecomunicaciones, librerías de plantillas, protocolos de red, sistemas embebidos (D.-I. K. Czarnecki, 1999). El elemento fundamental de un modelo de características es un diagrama arbóreo cuya raíz representa un concepto que tiene como descendientes sus características asociadas. Una característica es una propiedad relevante para el cliente. Suele utilizarse para detectar los puntos de variación de una familia de programas. Existen diversas notaciones para los

diagramas de características (Laguna, 2009). Y este método es con el que vamos a trabajar para definir y modelar nuestra línea de productos software.

Concepto de Paquetes. "Package Merge".

Para la correcta gestión de la variabilidad, el grupo GIRO aboga por expresar la variabilidad en los modelos UML utilizando el concepto de combinación de paquetes (o "package merge"), presente en el metamodelo de infraestructura de UML 2 y utilizado de forma exhaustiva en la definición misma de UML 2. El mecanismo "package merge" consiste fundamentalmente en añadir detalles de forma incremental, como se observa en la Figura 3, extraída del documento oficial de UML 2) (Laguna, 2009).

Lo fundamental para las LPS es

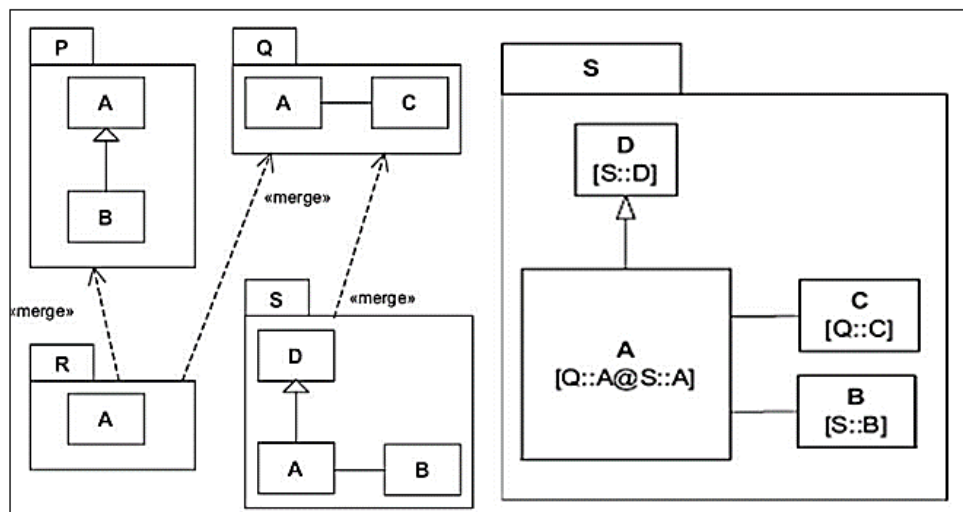


Figura 3. Ejemplo del mecanismo de "package merge"

Fuente: Laguna, M. A. (2009). Desarrollo de Líneas de Productos: un Caso de Estudio en Comercio Electrónico. 2009. Retrieved from http://www.laccei.org/LACCEI2009-Venezuela/Papers/IT155_Laguna.pdf

desarrollar un set de productos basados en elementos comunes, gestionando las diferencias para cada uno de los casos. El desarrollo de LPS ha traído beneficios como reducción en los ciclos de desarrollo y costos asociados, incrementos de productividad y

mejoras en la calidad de los productos (Sepúlveda, Cachero, & Cares, 2012) . La generación de código es una herramienta extremadamente valiosa que puede tener un importante impacto en la productividad y calidad en proyectos de ingeniería de software(Herrington, 2003). Por lo que una LPS está orientada a la selección e integración de elementos de software que han sido planificados, diseñados y desarrollados para ser reutilizados como base para elaborar nuevos productos (Bergey, Cohen, Donohoe, & Jones, 2005). Una LPS para su ejecución requiere de diversos procesos como por ejemplo el análisis del dominio, modelo de características, variabilidad, arquitectura, etc. Para esto se utiliza la metodología Exemplar Driven Development (EDD) que en esta sección se describe.

Los Lenguajes Específicos del Dominio (DSL) son mini-idiomas adaptados para un dominio específico, lo que puede ofrecer ventajas significativas sobre los GPL (Lenguajes de Propósito General) como Java(Van Deursen, Klint, Visser, & others, 2000). Cuando se desarrollan sistemas de software en una GPL, a menudo se encuentran situaciones en las que un problema no es expresable de forma natural en la GPL elegida. Tradicionalmente se recurre a encontrar un marco de trabajo adecuado (dentro del marco proporcionado por la GPL) para codificar la solución. Uno de los inconvenientes del uso de tal

solución es que el programa puede llegar a ser complejo, lo que lo hace mucho menos comprensible de lo que el desarrollador había deseado. La falta de expresividad en una GPL puede ser superada mediante el uso de DSL. Las herramientas DSL permiten implementar programas a nivel de abstracción del dominio de aplicación, lo que permite un desarrollo rápido y eficaz de los sistemas de software (Vasudevan & Tratt, 2011). La generación de código es una técnica cada vez más popular para la implementación de LPS (Pohl et al., 2005)) que produce código de especificaciones abstractas escritas en DSL (Herrington, 2003). La siguiente paradoja suele aparecer cuando se desarrolla un compilador DSL. Una DSL es un lenguaje especializado, orientado a problemas. Para el usuario DSL, es interesante que el DSL sea tan abstracto como sea posible (soportando la

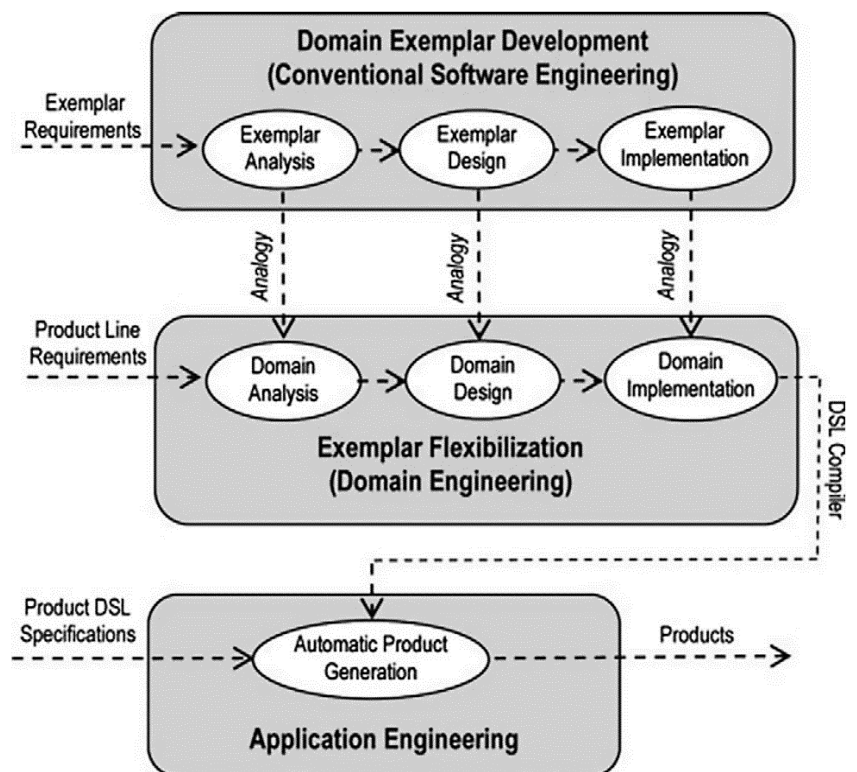


Figura 4. Esquema general del proceso EDD
Fuente: <http://e-spacio.uned.es/fez/eserv/tesisuned:IngInf-Rheradio/Documento.pdf>

terminología del dominio y eliminando los detalles de implementación de bajo nivel). Por otro lado, desde el punto de vista del desarrollador del compilador, la abstracción DSL dificulta la compilación del compilador. Es decir, las especificaciones DSL adicionales son del código final, más difícil es transformarlas en código final (R. Heradio, Cerrada, Lopez, & Coz, 2009) .

Exemplar Driven Development (EDD). Este nuevo proceso de desarrollo propuesto (Heradio Gil, 2007) para construir una familia de sistemas usando un enfoque de LPS se toma generalmente cuando el trabajo repetitivo se detecta en un dominio o cuando las oportunidades de negocio se identifican en la extensión de un producto exitoso. Por lo tanto, cuando se inicia el desarrollo de LPS, a menudo se dispone de un ejemplar del dominio.

La Figura 4 muestra un esquema general del proceso EDD, que trata de maximizar la reutilización de este ejemplar aplicando intensivamente la idea de analogía en todas las actividades de ingeniería de dominio.

1. *Análisis de dominios.* El análisis del dominio EDD se basa en el análisis del ejemplar. Todas las características obligatorias, comunes a todos los productos de dominio, son implementadas por el ejemplo, el análisis de dominio se centra en identificar las características variables, buscando las diferencias entre los requisitos ejemplares y los requisitos de los productos restantes.

2. *Diseño del dominio.* EDD deriva la arquitectura LPS de la arquitectura ejemplar. El diseño del dominio especifica qué adaptaciones del diseño ejemplar son necesarias para transformarlo en cualquier otro diseño del producto.

3. *Implementación del dominio.* El ejemplar está flexibilizado para proporcionar su adaptación automática para satisfacer las especificaciones DSL de entrada. Las técnicas de implementación para flexibilizar el ejemplo deben apoyar las siguientes capacidades deseables:

- No invasividad. La Figura 5 muestra cómo EDD se integra con el modelo de ciclo de vida espiral de Boehm (Boehm, 1988). En el primer ciclo de desarrollo, un ejemplo se construye utilizando la ingeniería de software convencional. En los siguientes ciclos sucesivos, se añaden módulos de flexibilización al ejemplo para introducir la variabilidad del dominio. Cuando los módulos de flexibilización no son invasivos al ejemplar, no hay modificación manual del mismo, facilitando la evolución del SPL.

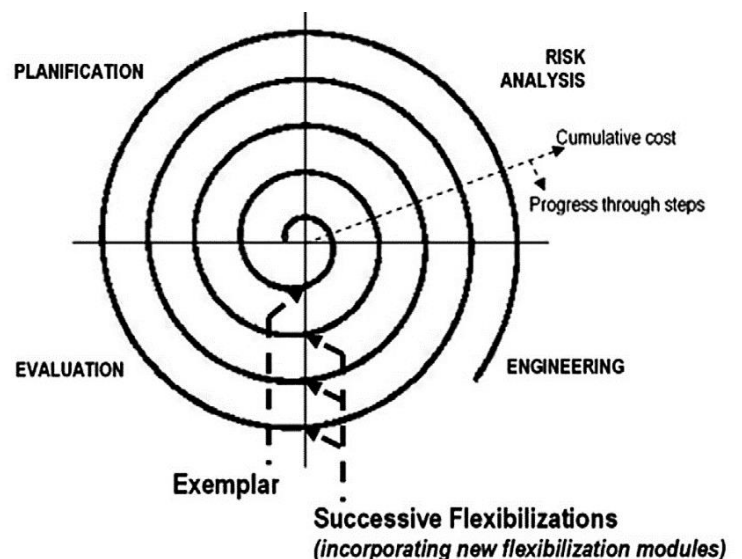


Figura 5. Integración con el modelo de ciclo de vida en espiral de Boehm. Fuente: <https://desarrollo-en-espiral.jimdo.com/historia-del-desarrollo-en-espiral/>

- Flexibilizaciones transversales; Como argumenta Voelter (Voelter & Groher, 2007), alguna variabilidad del dominio debería ser implementada como flexibilizaciones

transversales. En el contexto EDD, en el caso de que un modulo de flexibilización aplique a varios módulos, debería soportarlo el ejemplar implementado.

- Aplicable a cualquier tipo de artefacto de software; Debe apoyarse la flexibilización de la documentación ejemplar, casos de prueba, etc.
- Gestión eficiente de la variabilidad en tiempo de ejecución; Por ejemplo, proporcionando la parametrización de la variabilidad inter-producto antes del tiempo de ejecución de los productos. Falta complementar la idea

Con el fin de implementar la flexibilización ejemplar, se pueden utilizar diferentes técnicas comúnmente usadas para generalizar código. Tales técnicas pueden ser clasificadas como internas y externas.

- Con las técnicas internas, la flexibilización se implementa utilizando los mecanismos disponibles en el idioma donde se escribe el ejemplo (por ejemplo, usando herencia, genericidad, aspectos, etc.).
- Con técnicas externas la flexibilización se implementa utilizando un lenguaje o herramienta diferente.

El código es el producto del ciclo de vida con mayor tasa de reutilización. Por esta razón, las técnicas que comúnmente se emplean para generalizar código y posibilitar su reutilización parecen buenas candidatas para la flexibilización de un ejemplar. Debe satisfacer las condiciones dadas por Integridad conceptual y manejabilidad (Greenfield & Short, 2003). En la siguiente Sección, se realiza la implementación de una herramienta de generación de código utilizando esta

metodología junto con las descritas en la sección.

Métodos

Análisis del Sistema

En concreto se va a determinar los objetivos y límites del sistema, así como de caracterizar su estructura y funcionamiento, las necesidades del usuario para lo cual se establecen una serie de requisitos. Esta primera fase del proyecto es de vital importancia, ya que un error en la identificación de los requisitos que debe cumplir el sistema puede acarrear problemas graves a la hora del desarrollo y de la utilización del producto fina

Análisis y Modelado de Dominios

El primer paso es el análisis del dominio para el prototipo y que esto nos darán las necesidades del usuario que el producto debe satisfacer (Laguna, 2009). El factor de variabilidad de un requisito es transcendental y un concepto muy importante en un LPS porque determina los modelos de una familia de productos (Pérez, 2007). El concepto de características en LPS fue un éxito debido a su forma natural e intuitiva de representar un producto, y facilitó la comunicación entre los participantes. Un modelo de característica representa gráficamente una línea de producto, a través de las posibles combinaciones entre las características (Roos-Frantz & Segura, 2008). Lo que se realizó fue obtener los requisitos y convertir a un modelo de características. El elemento fundamental de un modelo de características es un diagrama arbóreo cuya raíz representa un concepto que tiene como descendientes sus características asociadas (D.-I. K. Czarnecki, 1999). En la Figura 6 se presenta el modelo de características del prototipo utilizado en este trabajo; un generador de código para la implementación de las Entities Java Beans de una

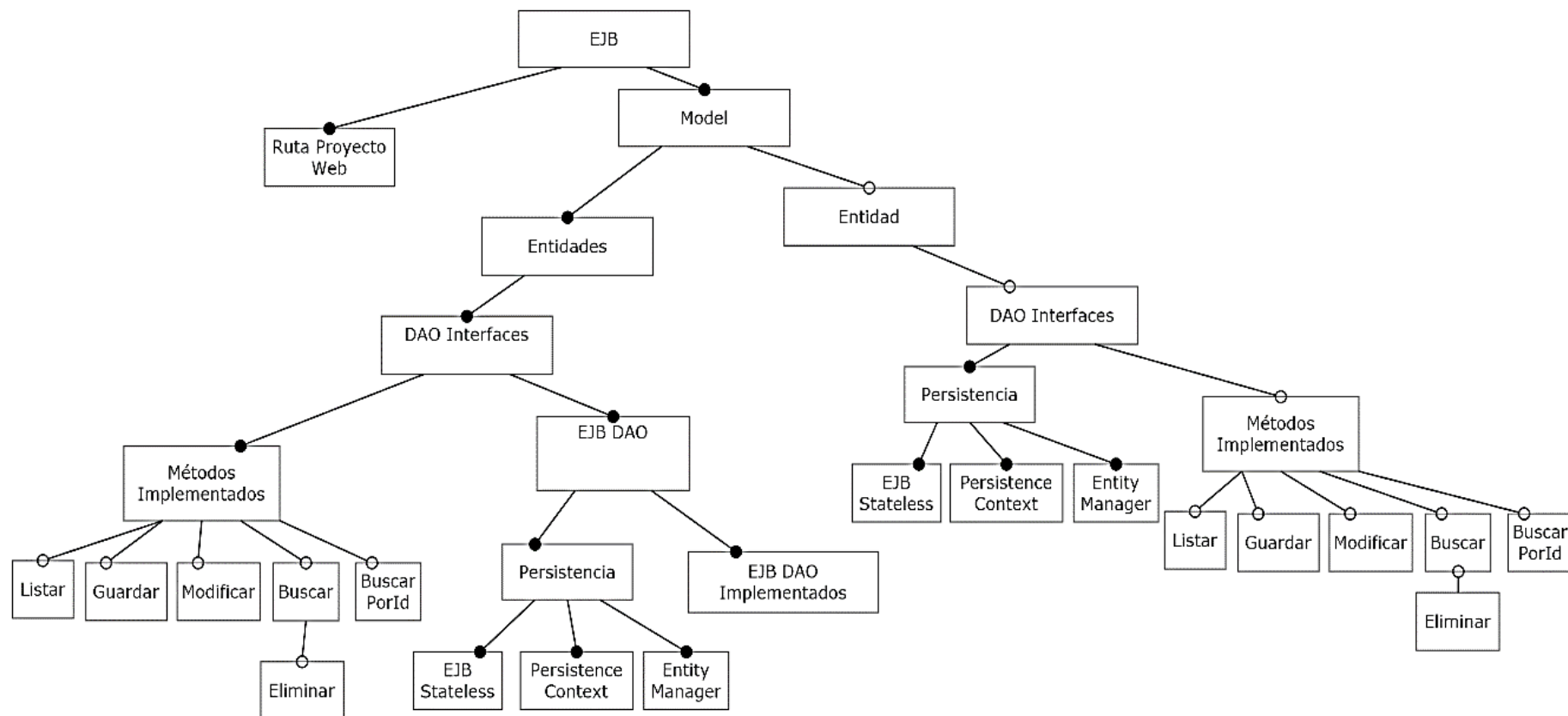


Figura 6. Modelo de características EJB.

Fuente: Elaboración propia.

aplicación web Java EE. Es posible notar dos relaciones entre cada característica:

En la Figura 6 se observan dos características detalladas:

Obligatoria: En el caso de las subcaracterísticas son obligatorias debido a que se debe seleccionar algunas para que otras de por efecto la generación de un EJB.

Opcional: Varias subcaracterística son opcionales para manejar la variabilidad con que se genera un EJB, puede o no ser incluida en el producto final.

Catálogo de Requisitos

La especificación de los requisitos debe ser completa, no tener definiciones contradictorias, y debe ser expresada con exactitud, ya que si hay ambigüedad se corre el riesgo de que sean interpretados de forma diferente por los usuarios y por los desarrolladores.

Requisitos funcionales.

Describen la funcionalidad del sistema y de qué forma va a utilizarlo el usuario. Como se trata de una Línea de Producto, se establece los requisitos por paquetes.

Requisitos no funcionales.

Especifican las propiedades del sistema que tienen que ver con el rendimiento, velocidad, uso de memoria, fiabilidad, etc. Imponen condiciones a los requisitos funcionales.

Junto con estos análisis se realiza la Especificación de Casos de Uso, Diagramas de Estado y de Secuencia para obtener el modelo del dominio. La Figura 7 muestra el Modelo de Dominio con los paquetes que se incluyen y las clases que contiene cada paquete, especificando también sus atributos utilizando la herramienta UML

Designer que es un software como plugin que lo instalamos en el entorno de desarrollo de Eclipse (UMLDesigner, 2014).

En la fase de diseño del generador, se transforma el modelo de dominio, creado durante el análisis, en las estructuras de datos necesarios para implementar el prototipo.

Materiales

El lenguaje de programación utilizado para el desarrollo es Ruby que resulta idóneo para la construcción de pequeños DSL (Thomas, Fowler, & Hunt, 2005a), que forma parte de la flexibilización; el cual, se ejecutará en el generador obteniendo el producto final. El desarrollo del prototipo de generador de código se realizó en el siguiente entorno tecnológico utilizando únicamente herramientas de código abierto: Eclipse 4.6 (Neon) como entorno de desarrollo integrado (IDE Integrated Development Environment), utilizado para el desarrollo de aplicaciones en Java y otros lenguajes de programación que está compuesta de varios complementos y está diseñada para ser extensible a través de plug-ins (Vogel, 2014). Ruby es el lenguaje de programación seleccionado por ser un lenguaje que combina una importante flexibilidad con alta productividad y por ser orientado a objetos (Baird, 2007), y otras mencionadas anteriormente. Se requirió la instalación del plugin Dynamic Languages Toolkit (DLTK) v5.7 que es una herramienta que se compone de un conjunto de marcos extensibles para reducir la construcción y configuración de entornos de desarrollo (Ford, 2008), en este caso para Ruby 2.3.

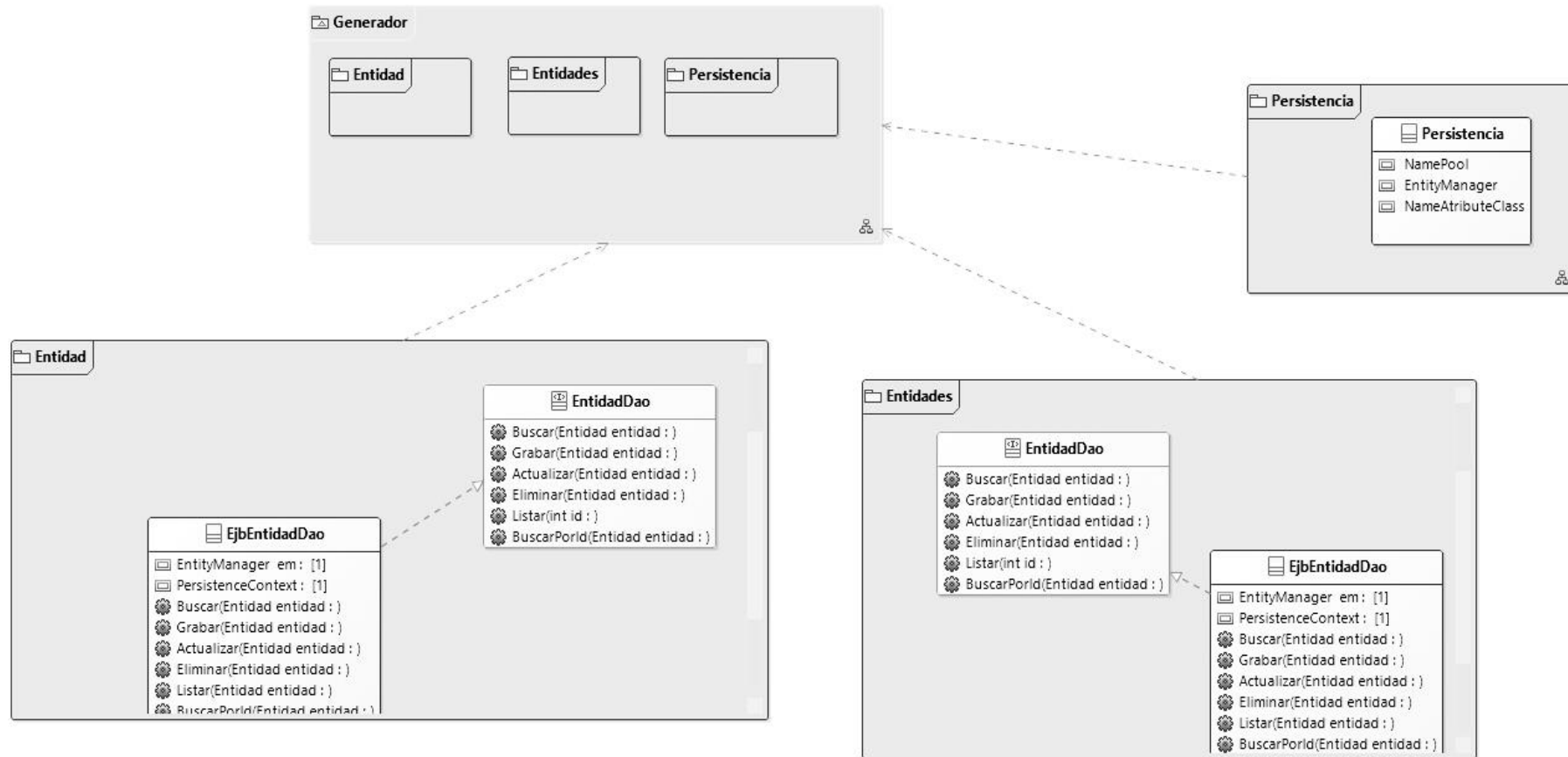


Figura 7. Modelo de Dominio.

Fuente: Elaboración Propia.

Revista Científica Hallazgos21. ISSN 2528-7915. Indexada en Latindex.

Periodicidad: cuatrimestral (marzo, julio, noviembre).

Director: José Suárez Lezcano. Teléfono: (593)(6) 2721459, extensión: 163.

Pontificia Universidad Católica del Ecuador, Sede Esmeraldas. Calle Espejo, Subida a Santa Cruz, Esmeraldas. CP 08 01 00 65. Email: revista.hallazgos21@puces.edu.ec. <http://revistas.puces.edu.ec/hallazgos21/>

Además se configuro GTK2 o Gimp Tool Kit v3.1.1 que son librerías multiplataforma para el desarrollo de entornos gráficos a través de ventanas, botones, menús, etiquetas, deslizadores, pestañas, etc., Pango v3.1.1 para el diseño y renderizado de texto, Cairo v3.1.1 para el renderizado de controles de aplicación y otras librerías dependientes (Pennington & others, 1999). Y por último la librería ERB (Embedded RuBy) que es una característica de Ruby que permite generar fácilmente cualquier tipo de texto, en cualquier cantidad, a partir de plantillas, que se combinan el texto sin formato con el código Ruby para la sustitución de variables y el control de flujo, haciéndolos fáciles de escribir y mantener (Thomas, Fowler, & Hunt, 2005b). Se

presenta a continuación el proceso de desarrollo del proyecto. El ambiente desarrollo que fue utilizado es apreciable el entorno del proyecto y su estructura de carpetas y archivos, son los mostrados en la Figura 8:

Ruta: Este directorio almacena los DAO's del modelo implementados con el generador de código.

Dao: Este directorio almacena las interfaces DAO's seleccionadas y sus respectivos EJB's generados de las entidades de la aplicación web al utilizar el generador de código.

Archivo.java: Es una clase de implementación de los EJB's de una entidad.

Analizador.rb: Dentro de este script de java se localiza el generador de código, que

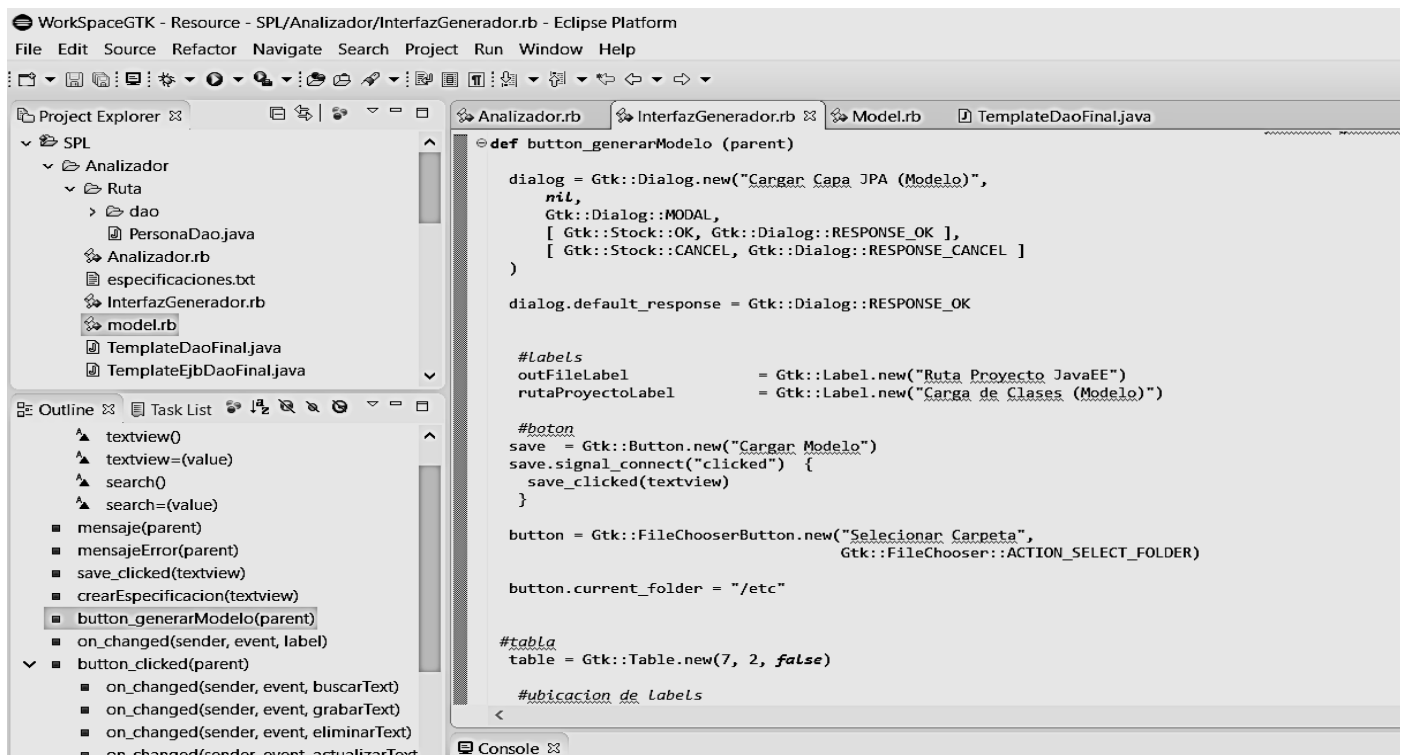


Figura 8. Entorno de desarrollo del proyecto y su estructura.
Fuente: Elaboración propia.

Revista Científica Hallazgos21. ISSN 2528-7915. Indexada en Latindex.

Periodicidad: cuatrimestral (marzo, julio, noviembre).

Director: José Suárez Lezcano. Teléfono: (593)(6) 2721459, extensión: 163.

Pontificia Universidad Católica del Ecuador, Sede Esmeraldas. Calle Espejo, Subida a Santa Cruz, Esmeraldas. CP 08 01 00 65. Email: revista.hallazgos21@pucese.edu.ec.

<http://revistas.pucese.edu.ec/hallazgos21/>

contiene el análisis de las especificaciones DSL y la generación de la clase correspondiente en java, invocando el motor ERB con la plantilla.

Especificaciones.txt: Este archivo es que es generado a partir de las especificaciones ingresadas por el usuario a través de la interfaz.

TemplateEjbDaoFinal.java y TemplateEjbDaoFinal.java: Representan la plantilla de código de una flexibilización. Realizada con la tecnología de plantillas ERB. En la siguiente sección se realiza una implementación de las entidades con sus DAO's y EJB's específicos con sus distintos métodos y el resultado junto con la

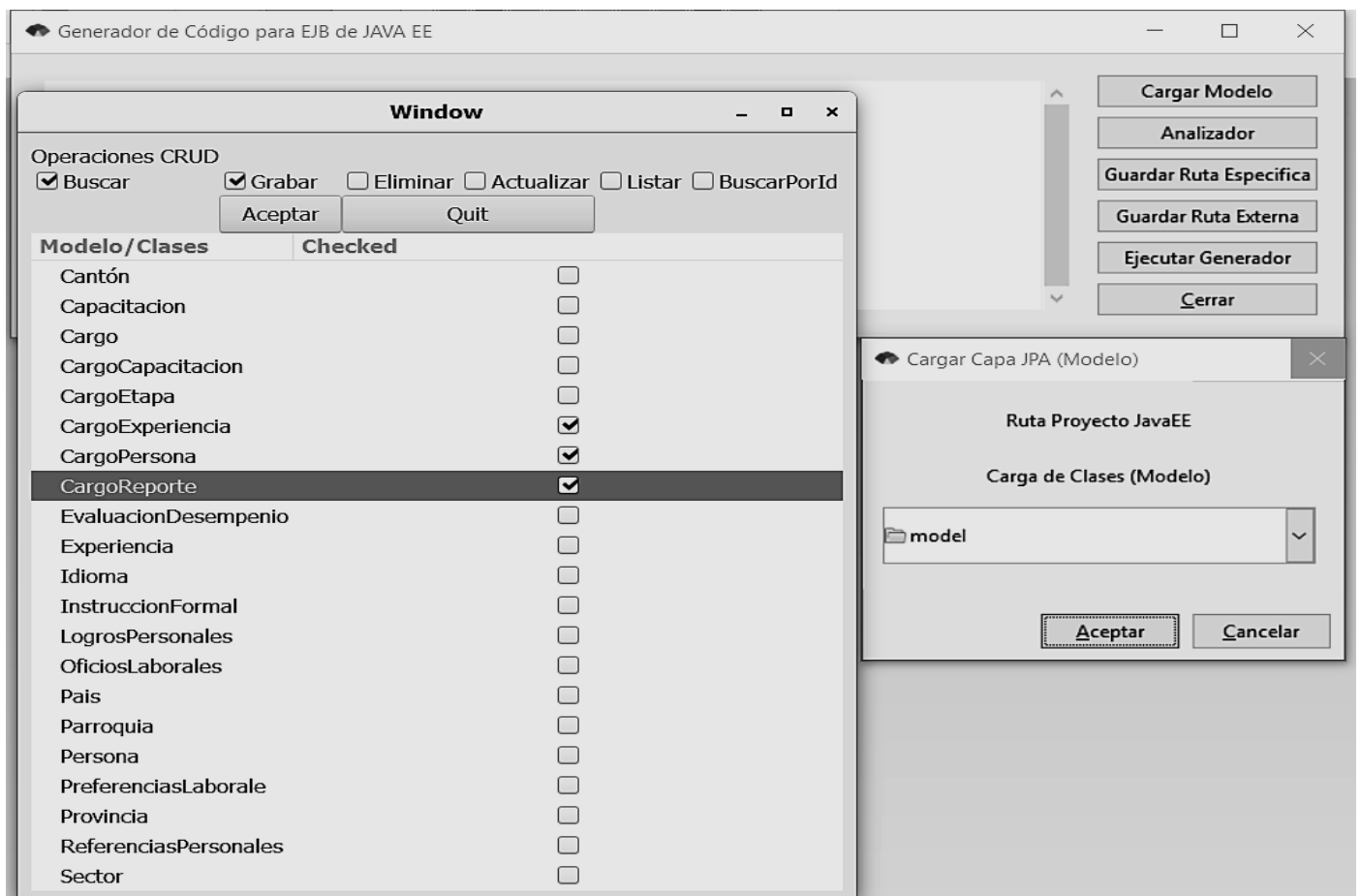


Figura 9. Interfaz carga de modelo y opciones de implementación.
Fuente: Elaboración propia.

InterfazGenerador.rb: Presenta una interfaz gráfica para el ingreso de las especificaciones DSL del usuario su correspondiente generación de código.

Model.rb: Aquí se genera la correspondiente lectura de la carpeta donde se ubican las entidades a seleccionar y su respectiva selección de métodos a implementar y entidades a seleccionar de la aplicación web.

implementación de los métodos CRUD.

1. Resultado.

El primer paso es cargar el modelo que es la ubicación de la carpeta donde se encuentra las entidades de la aplicación web, luego se escoge las opciones de variabilidad que pueden ser las entidades y métodos a generar que presenta la interfaz que generara las respectivas

especificaciones DSL. En la Figura 9, se observa dicha interfaz.

Otra forma de utilizar el generador de código es generar individualmente las entidades, a través del analizador, se requiere de llenar las especificaciones como son: la ruta del proyecto Java EE, el nombre la clase DAO a implementar, nombre de la persistencia utilizada, nombre de la entidad y los métodos a implementar del DAO como se ve en la Figura 10.

código que lee e inserta contenido del archivo con las especificaciones, para luego ejecutar el "generador", que implementa las especificaciones DSL invocando al motor ERB de plantillas "TemplateDaoFinal.java" y "TemplateEjbDaoFinal.java", para la generación automática del código del DAO y EJB. En la Figura 8 se puede apreciar el código que será reemplazado de acuerdo a las especificaciones ingresadas por el usuario a través de la plantilla ERB.

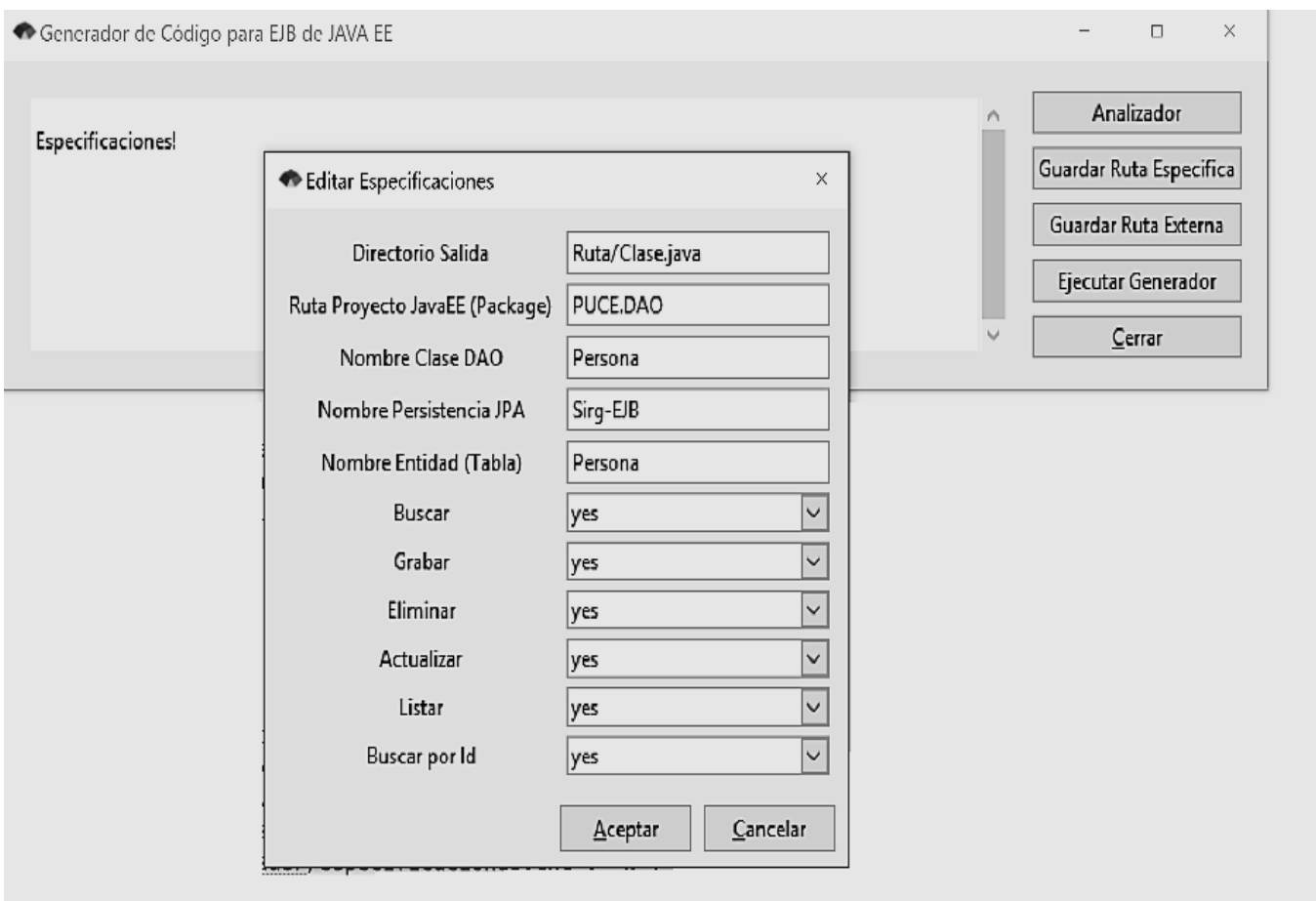


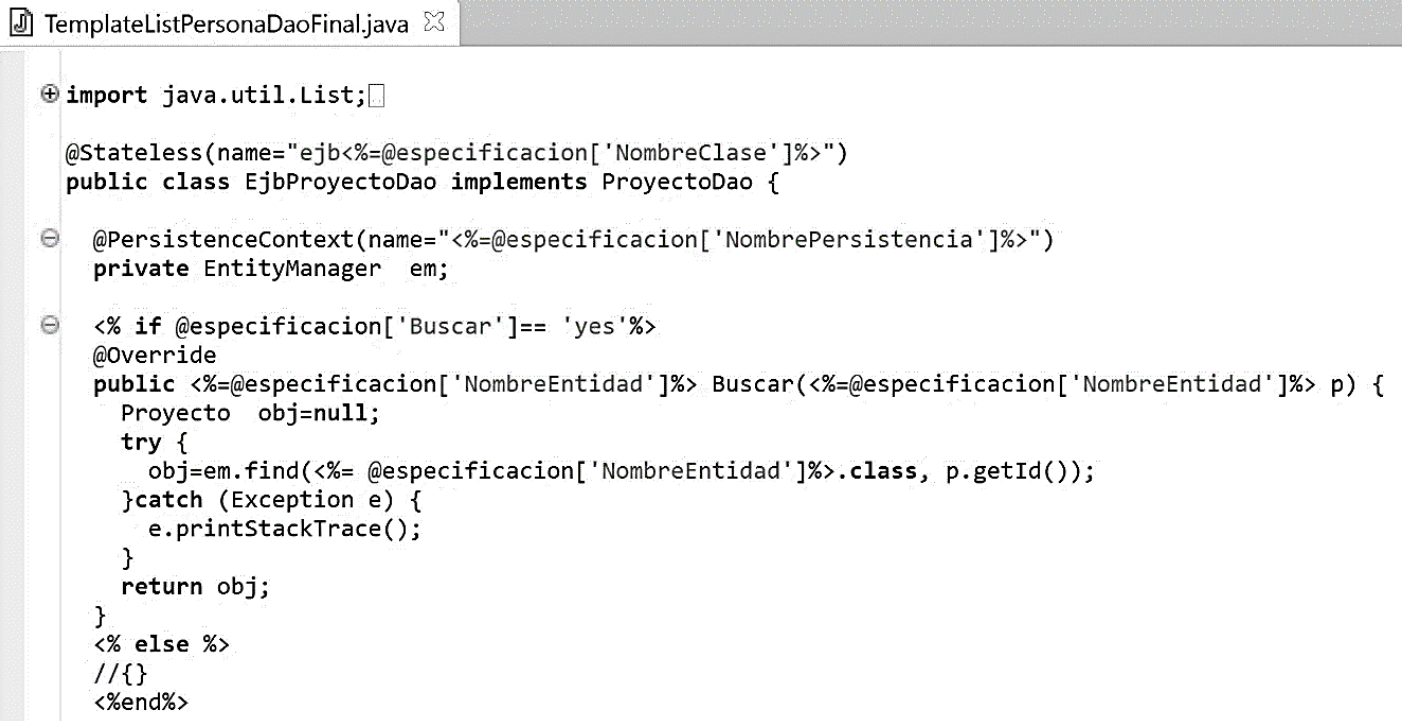
Figura 10. Interfaz de ingreso de analizador de entidad.
Fuente: Elaboración propia.

La interfaz del prototipo reduce y facilita el ingreso de las especificaciones, ya que las guarda en un archivo plano en una ruta específica o elegida por el usuario. El generador ejecuta las especificaciones en un archivo de texto. Este análisis está dado por

El resultado es la implementación de los DAO's y EJB's de una aplicación web, con sus anotaciones JPA, persistencia y métodos implementados con el código requerido para acceso de la base de datos de las entidades, para ser utilizada como parte de una

aplicación web Java EE, específicamente en su capa de negocio, la Figura 12 muestra lo descrito en el código resultante y las respectivas clases java creadas.

También es importante mencionar que se realizó una estimación por intervalo al patrón de desarrollo de un EJB a grupo de 6 programadores utilizado para medición del tiempo que se mejora al utilizar el generador

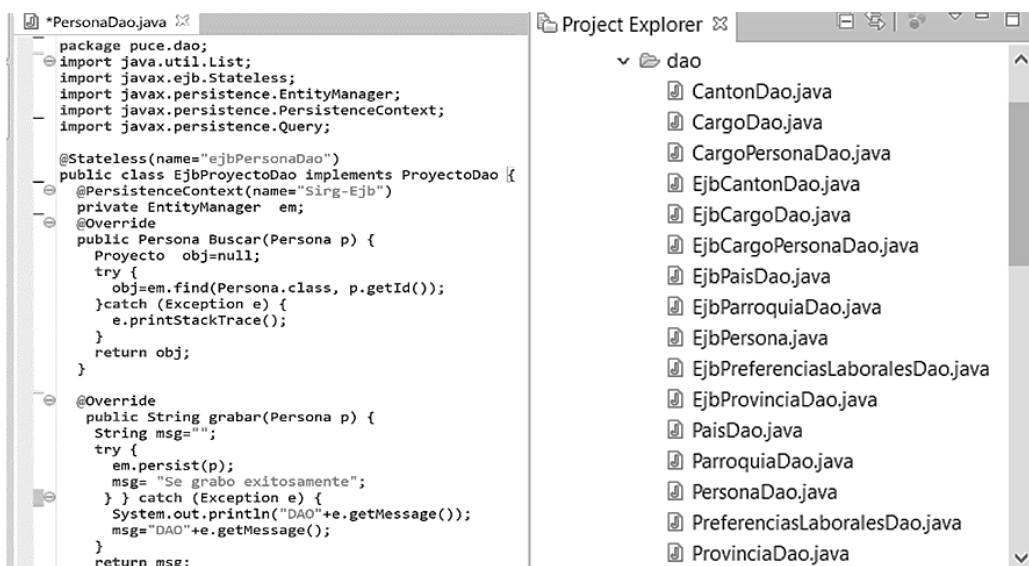


```

TemplateListPersonaDaoFinal.java
+ import java.util.List;
@Stateless(name="ejb<%=@especificacion['NombreClase']%>")
public class EjbProyectoDao implements ProyectoDao {
    @PersistenceContext(name="<%=@especificacion['NombrePersistencia']%>")
    private EntityManager em;
    <% if @especificacion['Buscar']== 'yes'%>
    @Override
    public <%=@especificacion['NombreEntidad']%> Buscar(<%=@especificacion['NombreEntidad']%> p) {
        Proyecto obj=null;
        try {
            obj=em.find(<%= @especificacion['NombreEntidad']%>.class, p.getId());
        }catch (Exception e) {
            e.printStackTrace();
        }
        return obj;
    }
    <% else %>
    //{}
    <%end%>

```

Figura 11. Implementación del ejemplar con plantilla ERB.
Fuente: Elaboración propia.



```

*PersonaDao.java
package puce.dao;
import java.util.List;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;

@Stateless(name="ejbPersonaDao")
public class EjbProyectoDao implements ProyectoDao {
    @PersistenceContext(name="Sing-Ejb")
    private EntityManager em;
    @Override
    public Persona Buscar(Persona p) {
        Proyecto obj=null;
        try {
            obj=em.find(Persona.class, p.getId());
        }catch (Exception e) {
            e.printStackTrace();
        }
        return obj;
    }
    @Override
    public String grabar(Persona p) {
        String msg="";
        try {
            em.persist(p);
            msg= "Se grabo exitosamente";
        } catch (Exception e) {
            System.out.println("DAO"+e.getMessage());
            msg="DAO"+e.getMessage();
        }
        return msg;
    }
}

```

Project Explorer

- dao
 - CantonDao.java
 - CargoDao.java
 - CargoPersonaDao.java
 - EjbCantonDao.java
 - EjbCargoDao.java
 - EjbCargoPersonaDao.java
 - EjbPaisDao.java
 - EjbParroquiaDao.java
 - EjbPersona.java
 - EjbPreferenciasLaboralesDao.java
 - EjbProvinciaDao.java
 - PaisDao.java
 - ParroquiaDao.java
 - PersonaDao.java
 - PreferenciasLaboralesDao.java
 - ProvinciaDao.java

Figura 12. Código y clases resultantes en java.
Fuente: Elaboración propia.

de código, el tiempo que se reduce es de 3.9 y 4.52 horas de un promedio de 5.58 horas, con una confiabilidad del 95% y con error de estimación de 0.3 horas, está distribuido normalmente según prueba Kolmogorov-Smirnov (Sig. Asintoticca bilateral= 0.20). Por lo tanto, esta estimación realizada es válida.

Conclusiones

Durante la investigación se ha profundizado en el uso de las metodologías de LPS y EDD, además de herramientas como DSL y Ruby que, al ser aplicadas de manera correcta y planificada, permiten obtener una mejora en el proceso de desarrollo de un producto software, y se ha llegado a las siguientes conclusiones:

El generador de código cumple con los requerimientos iniciales para los que fue desarrollado, como son automatizar y generar el código para una parte de una aplicación Java EE, que es la capa de negocio y sus respectivos DAO's y EJB's.

La implementación representa un aumento en el desempeño de desarrollo y una disminución del tiempo de codificación en este tipo de aplicación, comprobado por la estimación por intervalo que se realizó a un grupo de programadores que probaron el prototipo.

El trabajo a futuro que se plantea es mejorar la herramienta agregando mayores funcionalidades para que, en mediano plazo se transforme en un marco para el desarrollo de aplicaciones web Java EE, que automatice y genere las distintas capas y codificación de un producto software completo.

Referencias

- Baird, K. C. (2007). *Ruby by example: concepts and code*. San Francisco: No Starch Press.
- Barbosa, P. A., Contreras, C. G., & Rodriguez, J. M. M. (2005). AspectMDA: Hacia un desarrollo incremental consistente integrando MDA y Orientación a Aspectos. *Actas Del II Taller Sobre Desarrollo de Software Dirigido Por Modelos, MDA Y Aplicaciones (DSDM 2005)*, 74.
- Bergey, J. K., Cohen, S., Donohoe, P., & Jones, L. G. (2005). Software Product Lines: Experiences from the Eighth DoD Software Product Line Workshop. Retrieved from <http://repository.cmu.edu/sei/426/>
- Boehm, B. W. (1988). A spiral model of software development and enhancement. *Computer*, 21(5), 61–72.
- Carneiro Roos, F. (n.d.). Análisis Automático de Líneas de Producto Software Usando Distintos Modelos de Variabilidad.
- Clements, P. C., Jones, L. G., Northrop, L. M., & McGregor, J. D. (2005). Project management in a software product line organization. *IEEE Software*, 22(5), 54–62.
- Clements, P., & Northrop, L. (2015). *Software Product Lines: Practices and Patterns: Practices and Patterns*. Addison-Wesley. Retrieved from <https://books.google.com.ec/books?id=ATFijgEACAAJ>

- Czarnecki, D.-I. K. (1999). *Generative programming*. TU Ilmenau, Germany. Retrieved from <http://www.issi.uned.es/doctorado/generative/Bibliografia/TesisCzarnecki.pdf>
- Czarnecki, K., & Eisenecker, U. W. (1999). Components and generative programming. In *ACM SIGSOFT Software Engineering Notes* (Vol. 24, pp. 2–19). Springer-Verlag. Retrieved from <http://dl.acm.org/citation.cfm?id=318779>
- Ford, N. (2008, July 24). Using the Ruby Development Tools plug-in for Eclipse. Retrieved May 17, 2017, from <http://www.ibm.com/developerworks/library/os-rubyeclipse/index.html>
- Garzás, J., & Piattini, M. (2007). Concepto y Evolución de las Fábricas de Software. *Kybele Consulting*. Retrieved from <https://pdfs.semanticscholar.org/32ec/f1537492e81315f57d1f1e413db67583cc16.pdf>
- Greenfield, J., & Short, K. (2003). Software factories: assembling applications with patterns, models, frameworks and tools. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (pp. 16–27). ACM. Retrieved from <http://dl.acm.org/citation.cfm?id=949348>
- Heradio Gil, R. (2007). Metodología de desarrollo de software basada en el paradigma generativo: realización mediante la transformación de ejemplares. Retrieved from <http://e-spacio.uned.es/fez/eserv/tesisuned:IngInf-Rheradio/Documento.pdf>
- Heradio, R., Cerrada, J. A., Lopez, J. C., & Coz, J. R. (2009). Code Generation with the Exemplar Flexibilization Language. *Electronic Notes in Theoretical Computer Science*, 238(2), 25–34. <https://doi.org/10.1016/j.entcs.2009.05.004>

- Heradio, R., Fernandez-Amoros, D., de la Torre, L., & Abad, I. (2012). Exemplar driven development of software product lines. *Expert Systems with Applications*, 39(17), 12885–12896. <https://doi.org/10.1016/j.eswa.2012.05.004>
- Herrington, J. (2003). *Code generation in action*. Greenwich, CT: Manning.
- Jarzabek, S., & Seviora, R. (2000). Engineering components for ease of customisation and evolution. *IEE Proceedings - Software*, 147(6), 237. <https://doi.org/10.1049/ip-sen:20000914>
- Laguna, M. A. (2009). Desarrollo de Líneas de Productos: un Caso de Estudio en Comercio Electrónico. 2009. Retrieved from http://www.laccei.org/LACCEI2009-Venezuela/Papers/IT155_Laguna.pdf
- Laguna, M. A., González-Baixauli, B., & Marqués, J. M. (2007). Seamless development of software product lines. In *Proceedings of the 6th international conference on Generative programming and component engineering* (pp. 85–94). ACM. Retrieved from <http://dl.acm.org/citation.cfm?id=1289988>
- Miller, J. (2007). mda. Retrieved from <http://facepress.net/pdf/300.pdf>
- Muñoz, J., & Pelechano, J. (2004). MDA a Debate. *Actas Del Primer Taller Sobre Desarrollo Dirigido Por Modelos, MDA Y Aplicaciones (DSDM'04)*. Retrieved from https://www.researchgate.net/profile/Vicente_Pelechano/publication/228871915_MDA_a_Debate/links/549403cc0cf295024eb465a8.pdf
- Muñoz, J., & Pelechano, V. (2005). MDA vs Factorías de Software. *Actas Del II Taller Sobre Desarrollo de Software Dirigido Por Modelos, MDA Y Aplicaciones (DSDM 2005)*, 1. Retrieved from

https://www.researchgate.net/profile/Jesus_Torres10/publication/220776002_Implicaciones_de_Transformaciones_Oblicuas_en_el_Desarrollo_de_un_Framework_Generador_de_Aplicaciones_Orientadas_a_Aspectos/links/02bfe50d171e0461ec000000.pdf#page=9

Pennington, H., & others. (1999). *GTK+/Gnome application development*. New Riders Indianapolis. Retrieved from <https://pdfs.semanticscholar.org/2366/f069aeae2ab4079b40f2e10572697e6c0f5f.pdf>

Pérez, I. M. (2007). An Open Source Framework Oriented to Modular Web-Solution Development based in Software Product Lines. *Este Libro Se Distribuye Bajo Licencia Creative Commons Reconocimiento CompartirIgual 2.5 Espa*, 133.

Pohl, K., Böckle, G., & Linden, F. van der. (2005). *Software product line engineering: foundations, principles, and techniques* (1st ed). New York, NY: Springer.

Roos-Frantz, F., & Segura, S. (2008). Automated Analysis of Orthogonal Variability Models. A First Step. In *SPLC (2)* (pp. 243–248). Citeseer. Retrieved from <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.505.4351&rep=rep1&type=pdf>

Sepúlveda, S., Cachero, C., & Cares, C. (2012). Modelado de Características para Líneas de Producto de Software: una propuesta. In *INTERNATIONAL WORKSHOP ON ADVANCED SOFTWARE ENGINEERING, Valparaíso*. Retrieved from http://www.academia.edu/download/45233646/Modelado_de_Caractersticas_para_Lneas_de20160430-12534-ftjbt3.pdf

Thomas, D., Fowler, C., & Hunt, A. (2005a). *Programming Ruby: the pragmatic programmers' guide* (2nd ed). Raleigh, N.C: Pragmatic Bookshelf.

Thomas, D., Fowler, C., & Hunt, A. (2005b). *Programming Ruby: the pragmatic programmers' guide* (2nd ed). Raleigh, N.C: Pragmatic Bookshelf.

UMLDesigner. (2014). Getting started. Retrieved March 1, 2018, from <http://www.uml designer.org/tutorials/tuto-getting-started.html>

Van Deursen, A., Klint, P., Visser, J., & others. (2000). Domain-specific languages: An annotated bibliography. *Sigplan Notices*, 35(6), 26–36.

Van Gorp, J., Bosch, J., & Svahnberg, M. (2000). On the notion of variability in software product lines. Reading, MA: Addison-Wesley.

Vasudevan, N., & Tratt, L. (2011). Comparative Study of DSL Tools. *Electronic Notes in Theoretical Computer Science*, 264(5), 103–121.
<https://doi.org/10.1016/j.entcs.2011.06.007>

Voelter, M., & Groher, I. (2007). Product Line Implementation using Aspect-Oriented and Model-Driven Software Development (pp. 233–242). IEEE.
<https://doi.org/10.1109/SPLINE.2007.23>

Vogel, L. (2014). *Eclipse IDE tutorial*. Vogella, <http://www.vogella.de/articles/Eclipse/article.html#overview>, accessed August. Retrieved from <http://www.draelshafee.net/spring2011/Eclipse-tutorial.pdf>