PROYECTO DE GRADO

Presentado ante la ilustre UNIVERSIDAD DE LOS ANDES como requisito final para obtener el Título de INGENIERO DE SISTEMAS

DESARROLLO DE MEJORAS AL COMPONENTE DE INTERNACIONALIZACIÓN Y DESCRIPCIÓN DE PROPIEDADES DEL FRAMEWORK CLEDA: SOPORTE PARA ECLIPSE Y MAVEN

Por

Br. Karly Paola Toloza González
Tutor: Prof. Gabriel Demián Gutiérrez Pinzone

Marzo 2015

Desarrollo de mejoras al Componente de Internacionalización y Descripción de Propiedades del Framework CLEDA: soporte para Eclipse y Maven

Br. Karly Paola Toloza González

Proyecto de Grado — Sistemas de Control, 120 páginas

Resumen: CLEDA (*Create, List, Edit, Delete Architecture*) es un *framework* (marco de trabajo) para el desarrollo de sistemas de información y aplicaciones web con bajo costo en tiempos de desarrollos razonables. Este *framework* contiene un componente para la internacionalización de cadenas, formatos de fecha y números, y descripción de propiedades de JavaBeans.

Este componente estaba implementado haciendo uso de un generador de código, bajo una arquitectura deficiente, siendo necesaria la generación de los archivos de internacionalización, uno por uno, ya que no existía una manera global de generar las clases de internacionalización. Otro inconveniente que se presentaba era que los modelos de Hibernate que usa CLEDA, en sus métodos de acceso, tienen anotaciones que utilizan constantes de sí mismo o de otros JavaBean. Si se eliminaba el descriptor de propiedades de algún JavaBean, ocurría un error en la compilación. Si se modificaba algún JavaBean, y no se modificaba el descriptor de propiedades de este, ocurría un error en tiempo de ejecución. La solución era identificar el descriptor que estaba arrojando errores y editarlo, siendo un procedimiento sujeto a errores del programador.

Con este proyecto, se mejoró el componente de internacionalización de cadenas, formatos de fecha y números, y descripción de propiedades de JavaBeans del *framework* CLEDA, aprovechando al máximo la técnica de generación de código, y de renombramiento de variables y objetos (entre otros) que tienen los Entornos de Desarrollo Integrados que tanto se usan hoy en día, brindando soporte al proceso de compilación de Maven.

Palabras clave: CLEDA, Internacionalización (i18n), JavaBeans, Descriptores de Propiedades, Maven, Generador de Código.

A mi padre (+) y mi madre

Por siempre.

Índice

Índice iv
Índice de Tablas viii
Índice de Figuras ix
Agradecimientos xii
Capítulo 1 Introducción 1
1.1 Antecedentes 1
1.1.1 Internacionalización (i18n) 1
1.1.1.1 Localización (l10n) 2
1.1.2 *Framework* CLEDA 3
1.1.3 Componente de internacionalización y descripción de propiedades CledaI18N 4
1.2 Definición del problema 5
1.3 Justificación 6
1.4 Objetivos 6
1.4.1 Objetivo general 6
1.4.2 Objetivos específicos 6
1.5 Alcance 7
1.6 Método 7
1.7 Estructura del Documento 8
Capítulo 2 Marco Teórico 10
2.1 Estrategias utilizadas para hacer Internacionalización (i18n) 10
2.1.1 Estrategia estándar de Java 10
Ejemplo de internacionalización en Java 11
2.1.2 Estrategia estándar de XML: ITS (*Internacionalization Tag Set*) 14
2.1.3 Biblioteca GNU de internacionalización *Gettext* 15
2.1.3.1 Programador 16
2.1.3.2 Traductor 18
2.1.3.3 Usuario 18
2.1.4 Extensión Gettext PHP 18

2.2 *Framework* CLEDA 21
2.2.1 Arquitectura del Framework CLEDA 21
2.3 Estrategia de Internacionalización de CledaI18N 22
2.4 Estrategia de Internacionalización estándar de Java vs CledaI18N 23
2.5 Reflexión y Meta-programación 24
Capítulo 3 Análisis y Requerimientos 29
3.1 Análisis de CledaI18N actual 29
3.1.1 Generación de las clases de internacionalización – i18n 29
3.1.2 Generación de los descriptores de propiedades de los JavaBeans 30
3.2 Requerimientos del nuevo prototipo 32
3.2.1 Casos de uso 33
3.2.1.1 Generar clases de internacionalización (i18n) CU-01 34
3.2.1.2 Generar descriptores de propiedades CU-02 35
3.2.1.3 Analizar archivos de claves *.properties* CU-03 36
3.2.1.4 Borrar código generado anteriormente CU-04 37
3.2.1.5 Identificar JavaBeans CU-05 38
3.2.1.6 Encontrar métodos *get* y *set* CU-06 39
Capítulo 4 Arquitectura y Desarrollo de CledaI18N y CledaProp 40
4.1 Arquitectura de CledaI18N 41
4.2 Arquitectura de CledaProp 43
4.3 Componente Maven 46
4.3.1 Mojo 46
4.3.2 POM del Mojo 49
4.3.3 Plugin en Maven 52
4.4 Scanner 53
4.4.1 Interfaz IsTraversable 56
4.4.2 Interfaz IsProcessable 57
4.4.3 Interfaz Visitor 59
4.5 Eliminación del código generado previamente 60
4.6 Verificación de los directorios 61
4.7 Verificación de los archivos de claves para la internacionalización 62

4.8 Identificación de los JavaBeans 63
4.9 Localización de los métodos *get* y *set* de las propiedades de los JavaBeans 65
4.10 Generador de código 67
4.10.1 Generador de código de internacionalización – CledaI18N 67
4.10.2 Generador de código de los descriptores de propiedades de los JavaBeans - CledaProp 72
4.11 Ejecución del *plugin* de Maven 74
Capítulo 5 Pruebas 76
5.1 Pruebas unitarias 76
5.2 Pruebas cajas negras 77
5.2.1 Generación de las clases de internacionalización 77
5.2.2 Generación de los descriptores de propiedades de los JavaBeans 83
5.3 Ejemplo de la implementación de un formulario internacionalizado con el componente CledaI18N 85
Capítulo 6 Conclusiones y Recomendaciones 89
6.1 Conclusiones 89
6.2 Recomendaciones 90
Bibliografía 91
Apéndice A 94
Plantillas usadas para la generación de código 94
Expresión Regular usada para la validación de los nombres de archivos de los *.properties* 98
Anotación usada para indicar que una clase es un JavaBean de CLEDA al que se le desea generar un descriptor de propiedades 98
Apéndice B 99
Componente teórico adicional 99
Entorno de Desarrollo Integrado Eclipse 99
Herramienta de desarrollo de software Maven 99
Ciclo de vida de un proyecto Maven 100
Mojo (Maven plain Old Java Object) 101
Definición de Plugin de Maven 103
Generación de Código 104
JavaBeans 104

Java Compiler API 105
Patrón de diseño Visitor 106

Índice de Tablas

Tabla 1 CU-01 34
Tabla 2 CU-02 35
Tabla 3 CU-03 36
Tabla 4 CU-04 37
Tabla 5 CU-05 38
Tabla 6 CU-06 39
Tabla 7 Métodos de la figura 32 54
Tabla 8 Métodos de la figura 33 57
Tabla 9 Métodos de la figura 34 58
Tabla 10 Métodos de la figura 35 59
Tabla 11Métodos de la figura 42 68
Tabla 12 Métodos de la figura 44 72
Tabla 13Resultados de las pruebas unitarias de CledaI18N y CledaProp 76
Tabla 14 Formatos de fecha y hora en Ingles para Estados Unidos y el Reino Unido (Gran Bretaña) ... 78
Tabla 15 Formatos de fecha y hora en Español, Alemán y Frances para Francia. 78
Tabla 16 Formatos numéricos en inglés, para Estados Unidos y el Reino Unido (Gran Bretaña) 79
Tabla 17 Formatos numéricos en español, alemán y francés para Francia 79
Tabla 18 Claves en inglés (para US y GB), español, alemán y francés (FR) 79
Tabla 19 Ciclo de vida de un proyecto Maven 101
Tabla 20 Descriptores para la definición de un *plugin* en Maven 103

Índice de Figuras

Figura 1 Modelo en Espiral de Boehm para el proceso del software (tomado de Sommerville - 2005) .. 8
Figura 2 Ejemplo de un programa que muestra mensajes sin hacer uso de i18n 11
Figura 3 Programa internacionalizado del ejemplo de la figura 2 (tomado de ORACLE , 2015a)..... 12
Figura 4 Workflow de ejecución de *GNU gettex* para un programa en lenguaje C 16
Figura 5 Estructura de los directorios de un proyecto PHP donde se hace uso de Gettext PHP..... 19
Figura 6 Ejemplo de archivo *.po* 20
Figura 7 Forma de obtener las cadenas traducidas con *gettext* 20
Figura 8 Arquitectura de CLEDA (Adaptado de Gutiérrez, D. et al -2007)..... 21
Figura 9 Estrategia general de CledaI18N..... 22
Figura 10 Verificación de los metadatos de una clase, usando la *Reflection API* de Java..... 28
Figura 11 Ejemplo de una archivo _CledaI18N.java que genera las clases con los componentes de internacionalización, para los archivos de claves *TestI18NFoo.properties*, *TestNumbFoo.properties* y *TestDateFoo.properties*. 30
Figura 12 Ejemplo de una archivo *_CledaI18N.java* que genera los descriptores de propiedades del JavaBean *FooBean.java*. 31
Figura 13 Casos de uso del sistema 33
Figura 14 Diagrama de actividades CU-01..... 34
Figura 15 Diagrama de actividades CU-02..... 35
Figura 16 Diagrama de actividades CU-03..... 36
Figura 17 Diagrama de actividades CU-04..... 37
Figura 18 Diagrama de actividades CU-05..... 38
Figura 19 Diagrama de actividades CU-06..... 39
Figura 20 Arquitectura de CledaI18N 42
Figura 21 Componentes de CledaI18N..... 42
Figura 22 Diagrama de Componentes de CledaI18N 43
Figura 23 Arquitectura de CledaProp 44
Figura 24 Componentes de CledaProp 44
Figura 25 Diagrama de Componentes para CledaProp 45

Figura 26 Implementación de *MojoI18N.java* donde se indica la ejecución del scanner para la generación de las clases de internacionalización, formatos de fecha y números. 47
Figura 27 Implementación de *MojoProp.java* donde se indica la ejecución del scanner para la generación de los descriptores de propiedades de los JavaBeans. 49
Figura 28 Archivo pom.xml del MojoI18N.java 50
Figura 29 Archivo pom.xml del MojoProp.java 52
Figura 30 Archivo pom.xml del *plugin* que ejecuta ambos Mojos 53
Figura 31 Diagrama de clases del *Scanner* 54
Figura 32 Funciones que se encargan de invocar los recorridos de los directorios, revisión y verificación de archivos, e invocación de los métodos para la generación de código. 56
Figura 33 Diagrama de clases que implementan la interfaz *IsTraversable* 56
Figura 34 Diagrama de clases que implementan la interfaz *IsProcessable* 57
Figura 35 Diagrama de clases que implementan la interfaz *Visitor* 59
Figura 36 Porción de código donde se elimina el código generado anteriormente 60
Figura 37 Funciones de la clase *RegExpFileIgnorer.java* que validan por medio de expresiones regulares si se ignora o no un directorio 62
Figura 38 Funciones de la clase *IsI18NProcessable.java* que validan por medio de expresiones regulares si se ignora o no un archivo 63
Figura 39 Función *visitAnnotation* de la clase *BeanIsProcessable.java*, que revisa que una clase java contenga la anotación predefinida para los JavaBeans *@GeneratePropertyDescriptor* 65
Figura 40 Función *process*, que pertenece a la clase *AbstractProcessor* de la *Java Compiler API*, en el cual se procesan las anotaciones 65
Figura 41 Método *failsafeVisit* de la clase *BeanVisitor.java* 67
Figura 42 Diagrama de clases que intervienen en la generación de código de i18n 67
Figura 43 Método *run* del generador de código de i18n, donde se ejecutan las sentencias involucradas en la generación de las clases de internacionalización 71
Figura 44 Diagrama de clases que intervienen en la generación de los descriptores de propiedades de JavaBeans 72
Figura 45 Método *run* del generador de código de los descriptores de propiedades de JavaBeans 74
Figura 46 Clase de i18n generada para la internacionalización de cadenas de texto. Solo se incluyeron dos métodos para obtener las cadenas traducidas. *_I18NI18NFormat.java* 81

Figura 47 Clase de i18n generada para la internacionalización de los formatos de números. Solo se incluyeron tres métodos para obtener los formatos localizados. *_NumbNumbFormat.java* 82
Figura 48 Clase de i18n generada para la internacionalización de los formatos de fechas. Solo se incluyeron tres métodos para obtener las formatos localizados. *_DateDateFormat.java* 83
Figura 49 Segmento de código del JavaBean *MBase.java* de CLEDA. Se aprecian las propiedades y los métodos de acceso a estas. 84
Figura 50 Descriptor de propiedades del JavaBean *MBase.java - _PropMBase.java* 85
Figura 51 Panel de selección de idioma 85
Figura 52 Uso de los métodos estáticos que acceden a las cadenas internacionalizadas 86
Figura 53 Formulario con los componentes localizados en lenguaje Inglés 87
Figura 54 Formulario con los componentes localizados en lenguaje Alemán 87
Figura 55 Formulario con los componentes localizados en lenguaje Español 88
Figura 56 Formulario con los componenten localizados en lenguaje Francés 88
Figura 57 Plantilla base para los descriptores de propiedades *class-prop.ftl* 94
Figura 58 Plantilla base para las clases de internacionalización de cadenas de texto *class-i18n.ftl* 95
Figura 59 Plantilla base para las clases de internacionalización de formatos de fecha *class-date.ftl* 96
Figura 60 Plantilla base para las clases de internacionalización de formatos de números *class-numb.ftl*.. 97
Figura 61 Ejemplo de Mojo simple. (Apache, 2014). 101
Figura 62 Dependencias dela API Maven Plugin Tools. (Tomado de Apache, 2014b). 102
Figura 63 Configuración del pom.xml que describe un *plugin* de Maven. (Apache, 2014). 103
Figura 64 Ejemplo de una clase en Java que corresponde a un JavaBean 105
Figura 65 Estructura del Patrón de Diseño Visitor (Tomado de Gamma, E. et al , 1995). 107
Figura 66 Diagrama de Colaboración del Patrón de Diseño Visitor (Tomado de Gamma, E. et al, 1995). 108

Agradecimientos

A mi madre, quien me acompañó en esta travesía con su amor y apoyo incondicional. Me ayudaste de infinitas maneras a alcanzar esta meta. Gracias Mamá.

A mi padre, que aunque partió muy temprano de nuestro lado, siempre esperó de mí lo mejor.

A Manuel, gracias por estar ahí. Gracias por ayudarme, impulsarme cuando no tenía ánimos y ser una luz en los momentos más oscuros.

A mis amigos: Carlos, Rosa, Rossy, Sofía, Julio, José Luis, José Miguel, Fernando, Antonio, Luis Felipe, Alonso, José Ángel, Leonardo, William, y todos con los que compartí gratos momentos.

A mi tutor, el profesor Demián Gutiérrez, por su gran contribución en mi formación profesional y personal. Eres un gran mentor y una gran persona. Gracias.

A todos los profesores que ayudaron en mi formación, especialmente a Domingo Hernández y Mario Spinetti. Gracias.

A la Universidad de Los Andes, por abrirme sus puertas y permitirme ser un miembro más de ella.

A la Escuela de Ingeniería de Sistemas, en especial a Judith Miranda, Rosmary Ramírez y René Izarra.

Capítulo 1

Introducción

Vivimos en un mundo globalizado, donde cualquier producto o servicio debe responder a la necesidad de un grupo de consumidores. La importancia de encontrar espacio en el mercado para un producto que responde a la necesidad de un grupo particular de usuarios, representa un problema de escala comercial.

La internacionalización de cualquier producto o servicio permite que éste llegue a más usuarios, expandiendo su red de beneficiarios y consumidores. La especialización de un producto o servicio para un grupo determinado de usuarios no es ideal si se desea que dicho producto sea usado en masa.

La Internet demanda software global, es decir, el software se puede desarrollar independientemente de los países o idiomas de sus usuarios, y luego ser traducido o adaptado para ciertos países o regiones. Es en esta instancia que cobra importancia la internacionalización (i18n[1]) del software. La internacionalización permite la adaptación de aplicaciones de software a diferentes idiomas y regiones, sin la necesidad de realizar modificaciones en el código fuente.

1.1 Antecedentes

1.1.1 Internacionalización (i18n)

El mercado de software se está convirtiendo realmente en un mercado global. El advenimiento de la Internet ha eliminado algunas de las barreras tecnológicas en el uso generalizado de productos de software, y ha simplificado la distribución de productos de software alrededor del mundo. Sin embargo,

[1] i18n: abreviación de la palabra internacionalización, debido a que entre la primera "i" y la última "n" de la palabra, hay 18 letras. Término acuñado por la compañía *Digital Equipment Corporation* en la década de los 80. http://www.w3.org/2001/12/Glossary#I18N

la disponibilidad de un producto de software para el uso en lenguajes locales de diferentes países, influenciará significativamente su nivel de adopción. Así, hay un paradójico y razonable requerimiento: que el software de soporte a distintos lenguajes y siga sus convenciones. Construir software para el mercado internacional, que soporte diferentes lenguajes, a un costo y tiempos razonables, solo se consigue si se hace uso de los estándares para la internacionalización.

Según Oliver, A. et al (2011, p 171):

"La internacionalización es el proceso de diseñar e implementar un producto que sea tan neutro como sea posible desde el punto de vista cultural y técnico que , por lo tanto, sea fácilmente trasladable a una o diversas culturas específicas; es decir, que sea fácilmente localizable".

Un programa internacionalizado tiene las siguientes características:

- Con la adición de los datos localizados, el mismo ejecutable puede funcionar en todo el mundo.
- Los elementos textuales, tales como mensajes de estados o etiquetas de los componentes de la interfaz gráfica de usuario GUI, no están codificados en el programa. En su lugar, se almacenan fuera del código fuente y se recuperan de forma dinámica.
- Soporte a nuevos idiomas no requiere re-compilación.
- Datos que dependen de la cultura, como moneda y fechas, aparecen en formatos que se ajustan a la región y al idioma del usuario final.
- Puede ser localizada rápidamente.

1.1.1.1 Localización (l10n[2])

La localización o regionalización según la LISA (*Localization Industry Standards Association*)[3] es el proceso mediante el cual un producto internacionalizado (a menudo, pero no siempre, un programa informático) se adapta o configura para satisfacer los requisitos lingüísticos, idiomáticos, culturales y de otro tipo aplicables a un entorno, país, zona geográfica o mercado específico, aprovechando las opciones que la internacionalización previa de este producto ha permitido.

[2] l10n: abreviación de la palabra localización, debido a que entre la primera "l" y la última "n" hay 10 letras.
[3] Localization Industry Standards Association - LISA: Asociación para la Estandarización de la Industria de la Localización. http://web.archive.org/web/20110101184308/http://www.lisa.org/

1.1.2 *Framework* CLEDA

Un *framework* (marco de trabajo) desde el punto de vista del desarrollo de software, es definido según Wikipedia (2015b) como "una estructura conceptual y tecnológica de soporte definido, normalmente con artefactos o módulos de software concretos, que puede servir de base para la organización y desarrollo de software". Su uso se ha incrementado debido a la reducción en los tiempos de desarrollo, y también a la reducción del trabajo al desarrollar aplicaciones.

En el proceso de desarrollo de software, es común que los desarrolladores involucrados, deliberen y disciernan sobre las herramientas a utilizar: el lenguaje de programación a usar, el sistema de gestión de bases de datos, entre otras cosas.

No es un proceso eficiente, porque a pesar de que permite el acuerdo grupal de las tecnologías a utilizar, cada decisión errada hace que los proyectos se demoren más, acarreando retrasos en cada una de las etapas de desarrollo. Adicionalmente, si se decide cambiar alguna de las herramientas usadas, el problema se complica

A raíz de esto surge un nuevo concepto en el desarrollo de software, denominado Marco de trabajo, que funciona como soporte definido para proyectos de software, donde la principal característica de éste es la de organizar el proceso de desarrollo.

Según Gutiérrez, D. et al. (2007, p. 1):

"CLEDA es un *framework* de software libre implementado en Java, diseñado para desarrollar Sistemas de Información y Aplicaciones Empresariales de alta calidad a bajo costo y con tiempos de desarrollo razonables. La palabra CLEDA es acrónimo de "*Create, List, Edit, Delete Architecture*", usado para referirse a las cuatro funciones básicas de bases de datos e interfaces de usuario".

El *framework* CLEDA, fue creado en la compañía de desarrollo de software Minotauro C. A.[4] cuando comenzaron a desarrollar aplicaciones web y notaron la diversidad de herramientas necesarias para cada proyecto, lo cual motivó a la empresa a implementar una serie de clases, plantillas,

[4] Minotauro C. A: Empresa asociada al sector de las tecnologías de información.

metodologías y estándares para facilitar el desarrollo de ese tipo de aplicaciones, incrementado así la productividad de la empresa. Luego este proyecto se convirtió en un proyecto de código abierto con el fin de que otros pudieran beneficiarse de él y también permitir la retroalimentación con otros desarrolladores en pro de la mejora del mismo.

Debido a que al desarrollar un software a la medida, el 80% de los requerimientos entran en patrones bien conocidos (crear, ver, editar, modificar objetos, configuración y generación de reportes, autenticación de usuarios, seguimiento de flujos de trabajo, inicio, atención de tareas, entre otros), CLEDA utiliza una estrategia basada en patrones de requerimientos, patrones de casos de uso y flujos de trabajo (*workflow*) para la realización de tareas, manejando plantillas, servicios, componentes y estándares bien definidos los cuales facilitan el desarrollo de éste 80%, permitiendo así enfocarse más en el 20% restante que corresponde a los casos más particulares del sistema en desarrollo.

El uso de CLEDA hace posible reducir los tiempos de desarrollo, hacer software de calidad y proveer menores costos para los clientes, aumentando la competitividad de las empresas en desarrollo de software.

1.1.3 Componente de internacionalización y descripción de propiedades CledaI18N

Actualmente CLEDA posee un componente de internacionalización y descripción de propiedades llamado CledaI18N. Éste componente ofrece un enfoque alternativo de internacionalización de aplicaciones de software desarrolladas en Java. CledaI18N propone una estrategia de internacionalización basada en la generación automática de código para lenguajes de programación compilados y estáticamente tipados[5].

CledaI18N almacena las cadenas de texto traducidas, en unos archivos de extensión "*.properties*". Cada lenguaje tiene su propio archivo con las claves correspondiente. Para generar las clases de internacionalización que manipularan estas cadenas traducidas, se deben ejecutar una serie de archivos

[5] Se dice que un lenguaje es estáticamente tipados cuando la comprobación de la tipificación se realiza en tiempo de compilación.

llamados "*_CledaI18N.java*". CledaI18N no tiene un mecanismo que automatice la generación de este código, de modo que se deben ejecutar uno a uno todos los archivos "*_CledaI18N.java*" que tenga el proyecto que se desea internacionalizar.

CledaI18N alberga los archivos fuentes (*.properties*) en el mismo directorio donde se generan las clases de internacionalización. Esta característica de CledaI18N contrasta con Maven[6] (herramienta de construcción de proyectos Java). Maven funciona con una estructura de directorios estandarizada el cual establece un directorio de recursos, para alojar los archivos con las claves traducidas (*.properties*) y un directorio fuente para alojar las clases de internacionalización que manipulan las cadenas traducidas.

Por otra parte, CLEDA hace uso de un elemento llamado descriptor de propiedades de JavaBeans. Se puede decir que el descriptor de propiedades de un JavaBean no es más que una clase con constantes, donde estas constantes son los nombres de las propiedades de dicho JavaBean. CledaI18N genera estos descriptores, sin embargo, en los modelos de Hibernate[7] (herramienta de mapeo Objeto-Relación) de CLEDA, muchos de los métodos modificadores y consultores (*set* y *get*) tienen anotaciones que utilizan constantes de sí mismo o de otros JavaBeans, dando como resultado que si se eliminan los descriptores de propiedades el código no compila, y al no compilar el generador de código no se pueden obtener los descriptores de propiedades en vista de que el generador usa reflexión[8]; es decir si se elimina algún descriptor de propiedades, no hay manera alguna de volver a generarlo. De igual manera, si se modifica algún JavaBean, se debe editar su descriptor de propiedades.

1.2 Definición del problema

CLEDA tiene un componente para la internacionalización de aplicaciones de software llamado CledaI18N. Este componente soluciona el problema de internacionalización, y de generación de los descriptores de propiedades de JavaBeans; sin embargo, aunque cumple los objetivos no lo hace de manera óptima. CledaI18N hace uso de la generación de código, y éste a pesar de ser muy efectivo, puede presentar problemas al ser un proceso engorroso y poco sustentable para el mantenimiento.

[6] Herramienta de desarrollo de software Maven, ver Apéndice B. Pág. 99.
[7] Hibernate: herramienta de mapeo objeto-relacional para lenguaje de programación Java. http://hibernate.org/
[8] Reflexión y Meta programación, ver Capítulo 2. Pág. 24.

Incluido a esto, el empleo de nuevas tecnologías en el proceso de desarrollo de aplicaciones de software, tales como Maven, exhorta a que CledaI18N pueda usarse desde estas herramientas de manera fácil y productiva.

1.3 Justificación

CledaI18N ha dado hasta ahora soporte a la internacionalización de las aplicaciones desarrolladas en CLEDA, pero las fallas al usarla en distintos entornos y la no sustentabilidad en el mantenimiento del código observada por los desarrolladores involucrados en el proyecto CLEDA, ha incentivado a la modificación y mejora de la herramienta CledaI18N.

1.4 Objetivos

1.4.1 Objetivo general

Mejorar la arquitectura de generación de código del Componente de Internacionalización del Framework CLEDA, para brindar soporte desde el proceso de compilación de Eclipse[9] y/o Maven.

1.4.2 Objetivos específicos

- Estudiar las distintas alternativas de internacionalización (i18n).
- Estudiar la arquitectura del Framework de Internacionalización de CLEDA.
- Mejorar la arquitectura del componente de Internacionalización y descripción de propiedades del Framework CLEDA.
- Desarrollar un *plugin* en Maven que permita usar el componente de Internacionalización y descripción de propiedades del Framework CLEDA.
- Desarrollar un *plugin* para Eclipse que permita usar el componente de Internacionalización y descripción de propiedades del Framework CLEDA sin la necesidad de usar la consola.

[9] Entorno de Desarrollo Integrado Eclipse, ver Apéndice B. Pág. 99.

1.5 Alcance

Se realizará la automatización de la generación de las clases de internacionalización y descripción de propiedades, integrando este proceso a la herramienta de desarrollo de software Maven.

1.6 Método

El método que se seguirá para el desarrollo del proyecto implementa un enfoque incremental, específicamente se usó el Modelo en Espiral de Boehm[10]. Según Sommerville (2005), dicho modelo básicamente expresa que las actividades del desarrollo de software se organizan en una espiral en la que cada ciclo de la misma representa una fase del mismo. Cada uno de los ciclos tiene cuatro regiones bien definidas que se recorren secuencialmente en el siguiente orden:

- Definición de objetivos: se definen los objetivos a alcanzar en el ciclo, se definen las restricciones del proceso y del producto, se identifican los riesgos y se elabora un plan detallado de las estrategias a utilizar.
- Evaluación y reducción de riesgos: se evalúan las distintas alternativas a seguir en la estrategia, se analiza cada uno de los riesgos identificados para el proyecto, y se definen los pasos para reducirlos.
- Desarrollo y validación: se elige un modelo para el desarrollo del sistema correspondiente a la fase (ciclo) actual, se verifica y se valida.
- Planificación: se evalúa el trabajo hecho hasta ahora en el ciclo, y en caso de requerirse otro ciclo, se planifica el mismo.

En resumen, según Sommerville (2005):

Un ciclo de la espiral empieza con la elaboración de objetivos. Luego se enumeran formas alternativas de alcanzar estos objetivos y las restricciones impuestas en cada una de ellas. Cada alternativa se evalúa contra cada objetivo y se identifican los riesgos del proyecto. El siguiente paso es resolver estos riesgos mediante actividades de recopilación de información. Una vez que se han

[10] Modelo en Espiral de Boehm: modelo de ciclo de vida de software expuesto por Barry Boehm en su trabajo: *A Spiral Model of Software Development and Enhancement*

evaluado los riesgos, se lleva a cabo cierto desarrollo, seguido de una actividad de planificación para el siguiente ciclo.

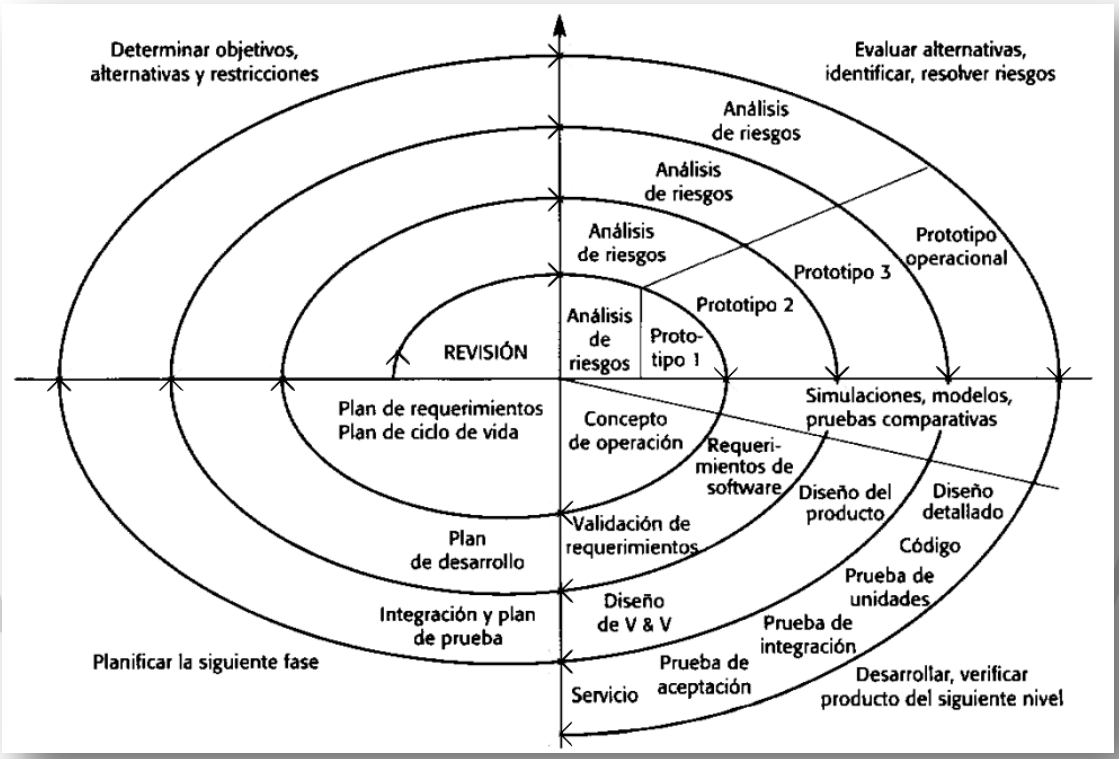

Figura 1 Modelo en Espiral de Boehm para el proceso del software (tomado de Sommerville - 2005)

1.7 Estructura del Documento

Capítulo 2, Marco teórico. Contiene los fundamentos teóricos necesarios para el entendimiento y comprensión del proyecto, entre los cuales se describe detalladamente algunas estrategias de internacionalización, así como la técnica de internacionalización de CledaI18N.

Capítulo 3, Análisis y requerimientos. Comprende el análisis de la herramienta de internacionalización y de descripción de propiedades actual, el estudio de los requerimientos de la nueva herramienta, utilizando casos de uso y diagramas de actividades de UML

Capítulo 4, Arquitectura y Desarrollo. En este capítulo se da una breve explicación de las clases más importantes involucradas en el proceso de internacionalización y descripción de propiedades de JavaBeans, así como la implementación de estas.

Capítulo 5, Pruebas. Se exponen las pruebas realizadas a las herramientas.

Capítulo 6, Conclusiones y recomendaciones. Se describen las conclusiones generales del trabajo realizado y las recomendaciones para trabajos futuros.

Capítulo 2

Marco Teórico

En este capítulo, se describen los fundamentos teóricos básicos necesarios para el entendimiento y comprensión del proyecto. Las herramientas utilizadas y/o estudiadas durante la realización del proyecto se explicaran en los siguientes capítulos si es necesario.

2.1 Estrategias utilizadas para hacer Internacionalización (i18n)

Hoy en día se cuenta con distintas técnicas para internacionalización según la tecnología que se usa para el desarrollo del software que se desea internacionalizar.

2.1.1 Estrategia estándar de Java

ORACLE (a, 2014), indica que particularmente el lenguaje de desarrollo Java, contiene un amplio conjunto de APIs (*Application Programming Interface* – Interfaz de Programación de Aplicaciones) para el desarrollo de aplicaciones globales. Estas APIs de internacionalización se basan en el estándar *Unicode*[11], e incluyen la capacidad de adaptación de texto, números, fechas, moneda y objetos definidos por el usuario a las convenciones de cualquier país.

En la edición estándar de la plataforma Java (*Java SE Plataform*), el soporte de internacionalización está totalmente integrado con las clases y paquetes que proporcionan funcionalidad al lenguaje o dependencias culturales.

[11] *Unicode*: Estándar para la codificación de caracteres. http://www.unicode.org/

El núcleo de Java provee las bases para la internacionalización de aplicaciones tanto de escritorio como de servidor.

Ejemplo de internacionalización en Java

A continuación se muestra un pequeño ejemplo sobre la internacionalización en Java, tomado de la página web oficial de ORACLE[12].

Si se deseara escribir un pequeño programa en Java que solo imprimiera por pantalla algunos saludos, como "*Hello*", "*How are you?*" y "*Good bye*", sería algo similar al ejemplo de la siguiente figura:

```
public class NotI18N {
    static public void main (String[] args) {

        System.out.println("Hello.");
        System.out.println("How are you?");
        System.out.println("Goodbye");
    }
}
```

Figura 2 Ejemplo de un programa que muestra mensajes sin hacer uso de i18n (tomado de ORACLE, 2015a)

Sin embargo, si queremos que estos mensajes se muestren a usuarios de otras regiones con otros idiomas, debemos internacionalizar el programa, y según la locación en particular, traducirlo. Un ejemplo de esto se puede visualizar en la siguiente figura:

```
public class I18NSample {
    static public void main (String [] args) {
        String language;
        String country;

        if(args.length != 2) {
            language = new String("en");
```

[12] http://docs.oracle.com/javase/tutorial/i18n/intro/quick.html

```
            country = new String("US");
        } else {
            language = new String(args[0]);
            country = new String(args[1]);
        }

      Locale currentLocale = new Locale(language, country);
      ResourceBundle messages;

      messages = ResourceBundle.getBundle("MessageBundle", currentLocale);

      System.out.println(messages.getString("greetigns"));
      System.out.println(messages.getString("inquiry"));
      System.out.println(messages.getString("farewell"));
    }
}
```

Figura 3 Programa internacionalizado del ejemplo de la figura 2 (tomado de ORACLE , 2015a)

Al estudiar el código fuente del programa internacionalizado, se observa que las cadenas de texto con los saludos, han sido eliminadas. Debido a que los saludos ya no son especificados en el código, y el idioma es especificado en tiempo de ejecución, el mismo ejecutable puede ser distribuido en todo el mundo. No es necesario re-compilar el programa para la localización.

El archivo de propiedades almacena el par clave – valor, que representa texto traducible de los mensajes que se mostrarán, donde la clave es la referencia a un texto que es traducible, y el valor es la traducción de ese texto. El archivo de propiedades por defecto que es llamado *MessagesBundle.properties*, contiene las siguientes líneas.

```
greetings = Hello
farewell = Goodbye
inquiry = How are you?
```

Una vez que los mensajes están en un archivo de propiedades, pueden ser traducidos a varios idiomas. No se requieren cambios en el código fuente. Se tendrán entonces varios archivos con las traducciones en varios idiomas. El archivo con las cadenas de texto en español, es llamado *MessagesBundle_es_ES.properties*, y contiene las siguientes líneas:

```
greetings = Hola.
farewell = Adios.
inquiry = Cómo estás?
```

Los valores al lado derecho del signo igual han sido traducidos, pero la clave al lado izquierdo no ha cambiado. Estas claves no deben cambiar, porque serán las referencias cuando el programa desee obtener el texto traducido.

El objeto *Locale* identifica un lenguaje y un país en particular. La siguiente sentencia define un *Locale* para el cual el lenguaje es Ingles y el país es Estados Unidos.

```
Locale aLocale = new Locale ("en", "US");
```

Los objetos *ResourceBundle* contienen objetos especificados localmente. Estos objetos son usados para aislar datos que son sensibles a la localidad, tales como texto traducible. En el ejemplo, el *ResourceBundle* está respaldado por el archivo de propiedades que contiene los mensajes de textos que se desean mostrar.

El *ResourceBundle* es creado como se muestra a continuación:


```
messages = ResourceBundle.getBundle("MessagesBundle", currentLocale);
```

Los argumentos que se pasan al método *getBundle* identifican que archivo de propiedades se accederá. El primer argumento, *MessagesBundle* se refiere a la siguiente familia de archivos de propiedades:

```
MessagesBundle_en_US.properties
MessagesBundle_es_ES.properties
MessagesBundle_de_DE.properties
```

El *Locale*, que es el segundo argumento de *getBundle*, especifica cuál de los archivos *MessageBundle* se elige. Cuando se creó el objeto *Locale*, el código del lenguaje y el código del país fueron pasados por el constructor del *Locale*. Ahora lo único que falta por hacer es conseguir los mensajes traducidos desde el *ResourceBundle*.

Para recuperar el mensaje identificado por la clave *greetings*, se debe invocar *getString* de la siguiente manera:

```
String msg1 = messages.getString("greetings");
```

La clave está codificada en el programa y debe estar presente en los archivos de propiedades. Si las claves en los archivos de propiedades son modificadas accidentalmente, *getString* no será capaz de encontrar los mensajes.

2.1.2 Estrategia estándar de XML: ITS (*Internacionalization Tag Set*)

Otra estrategia conocida es la utilizada en el lenguaje XML, denominada ITS (*Internacionalization Tag Set*). ITS es un conjunto de atributos y elementos diseñados para proveer soporte de internacionalización y localización en documentos XML[13]. (World Wide Web Consortium - W3C, 2007).

La especificación ITS identifica conceptos que son importantes para la internacionalización y localización. También define implementaciones de estos conceptos a través de un conjunto de elementos y atributos agrupados en el espacio de nombres ITS. Desarrolladores de XML pueden usar este espacio de nombres para integrar características de internacionalización directamente en sus propios esquemas y documentos XML. (Wikipedia, 2014a).

La versión 1.0 de ITS incluye siete categorías de datos:

- Traducir: define que partes de un documento son traducibles o no.
- Nota de localización: proporciona alertas, consejos, instrucciones y otra información para ayudar a los localizadores o traductores.
- Terminología: indica partes de los documentos que son términos y opcionalmente punteros a información acerca de esos términos.
- Direccionalidad: indica el tipo de pantalla de direccionalidad debe ser aplicada a partes del documento.
- Ruby: indica que partes del documento se deben mostrar como texto ruby. (Ruby es una pequeña ejecución de texto junto a un texto base, que se utiliza normalmente en los

[13] XML: siglas en inglés para Lenguaje de Marcas Extensibles (*e**X**tensible **M**arkup **L**anguaje*). http://www.w3.org/XML/

documentos de Asia Oriental para indicar la pronunciación o para proporcionar una breve anotación).

- Información de idioma: identifica el idioma de las diferentes partes del documento.
- Elementos dentro del texto: indica cómo los elementos deben ser tratados con respecto a la segmentación lingüística.

El vocabulario está diseñado para trabajar en dos frentes diferentes: primero, proporcionando un marcado utilizable directamente en los documentos XM; segundo, ofreciendo una manera para indicar si hay partes de un margen determinado que corresponden a algunas de las categorías de los datos y deben ser tratados como tales por sus procesadores. (Wikipedia, 2014a).

Con la ITS se pueden especificar el uso de reglas globales y locales:

- Las reglas globales se expresan en cualquier parte del documento (reglas globales incrustadas), o incluso fueran del documento (reglas globales externas), usando el elemento `its:rules`
- Las reglas locales son expresadas por atributos especializados (y algunas veces elementos) especificados dentro de la instancia del documento, en el lugar donde se aplican.

2.1.3 Biblioteca GNU de internacionalización *Gettext*

GNU gettex es un importante avance en el *GNU Translation Project* (Proyecto de Traducción GNU). Este paquete ofrece a programadores, traductores y usuarios, un conjunto de herramientas integradas y su documentación. Específicamente, las utilidades de *GNU gettex* son un conjunto de herramientas que proveen un marco de trabajo para ayudar a otros paquetes GNU a producir mensaje en múltiples lenguajes. Estas herramientas incluyen un conjunto de convenciones acerca de cómo los programas deben ser escrito para soportar los catálogos de mensajes, una organización de nombres de directorios y archivos para los catálogos de mensajes, y una biblioteca en tiempo de ejecución de apoyo a la recuperación de mensajes traducidos. (GNU Operating System, 2014).

Además de las implementaciones en C, *GNU gettex* tiene implementaciones en C++, Objetive C, sh script, bash script, Python, GNU CLISP, Emacs Lisp, librep, GNU Smaltalk, Java, GNU awk, Pascal, Tcl, Perl, PHP y Pike.

Originalmente, *gettex* fue escrito por *Sun Microsystems* al inicio de la década de los 90. El Proyecto GNU lanzó *GNU gettext* una implementación de software libre del sistema, en 1995.

En la figura 4, se resume la relación entre los archivos manejados por *GNU gettex* y las herramientas que actúan sobre estos archivos para la internacionalización de un programa en lenguaje C. Tener una comprensión clara de estas interrelaciones, ayudará a los programadores, traductores y usuarios de la herramienta *GNU gettext*.

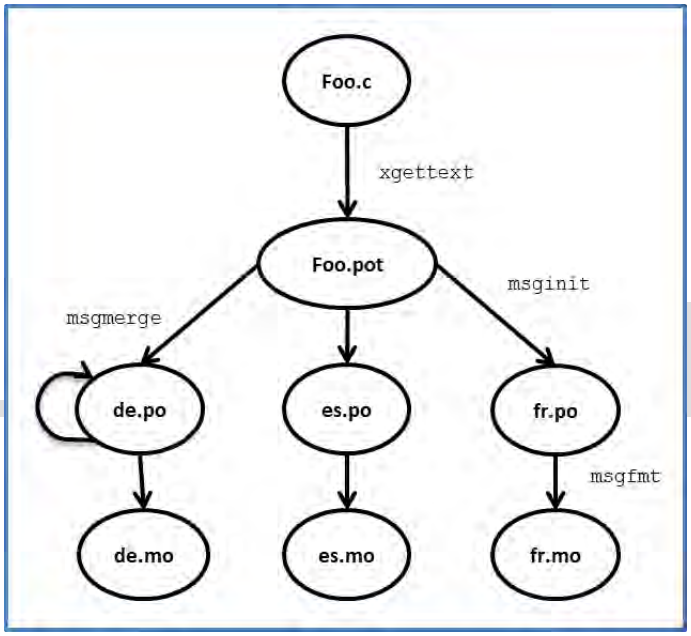

Figura 4 Workflow de ejecución de *GNU gettex* para un programa en lenguaje C (tomado de Wikipedia – 2015a)

A continuación se explica el procedimiento de internacionalización con *GNU gettext* según los involucrados en el proceso.

2.1.3.1 Programador

Como programador, el primer paso para usar *GNU gettext* en un proyecto es identificar, correctamente en los archivos fuentes, aquellas cadenas que se desean traducir, y aquellas que no serán

traducidas. Se modifica el código fuente para poder hacer las llamadas de *GNU gettext*. Se debe usar como parámetros de la función *gettext*, las cadenas de texto que se desean ver. Ejemplo:

```
printf(gettext(My name is %s. \n), mi_nombre);
```

Comentarios (que comiencen con ///) ubicados antes de cadenas de caracteres deberán ser marcados y ponerlos a disposición como sugerencias para el traductor y programas de ayuda.

gettext usa las cadenas de texto proporcionadas como claves para buscar traducciones alternativas, y se devuelve la cadena original si la traducción no está disponible.

xgettex se ejecuta sobre el código fuente para producir un archivo plantilla de extensión *.pot* (*Portable Object Template),* cuando contiene una lista de todas las cadenas de texto traducibles extraídas del código fuente.

Por ejemplo, un archivo de entrada puede lucir como:

```
/// TRANSLATORS: Please leave %s as it is, because it is needed by /// the
program.
/// Thank you for contributing to this project.
printf(_("My name is %s.\n"), my_name);
```

xgettex es ejecutado usando el comando: `xgettext --add-comments=/`

El archivo *.pot* resultante se puede ver como:

```
#. TRANSLATORS: Please leave %s as it is, because it is needed by
#. the program.
#. Thank you for contributing to this project.
#: src/name.c:36
msgid "My name is %s.\n"
msgstr ""
```

2.1.3.2 Traductor

El traductor genera un archivo *.po* (*Portable Object*) desde la plantilla usando el programa *msginit* y luego lo traduce. *msginit* inicializa el archivo con las traducciones, por lo tanto, si deseamos crear una traducción al español, se ejecutaría de la siguiente manera:

```
msginit --locale=es --input=name.pot
```

Esto generaría un archivo *es.po*. El traductor debe editar el archivo final. Una entrada sencilla se verá como la siguiente:

```
#: src/name.c:36
msgid "My name is %s.\n"
msgstr "Mi nombre es %s.\n"
```

Finalmente los archivos .po son compilados en un archivo binario *.mo* (*Machine Object*) con *msgfmt*. De forma que queden listos para ser distribuidos con el paquete de software.

2.1.3.3 Usuario

El usuario, sobre sistemas tipo UNIX, debe establecer la variable de entorno *LANGUAGE*, y el programa mostrará las cadenas en el idioma seleccionado, si es que hay un archivo *.mo* para ello.

2.1.4 Extensión Gettext PHP

La extensión *Gettext PHP* permite traducir dinámicamente cadenas de texto presentes en código PHP, usando la función *gettext* para obtener apropiadamente la cadena traducida. Si una cadena no ha sido traducida previamente, se usa la original. (PHP, 2005).

Gettext PHP requiere una estructura específica de directorios para funcionar correctamente.

Por cada proyecto, se tendrá un directorio llamado *Locale* (nombre estándar, puede variar), donde estarán alojados los diferentes archivos de traducción para los diferentes lenguajes. Se tendrá un

directorio para cada lenguaje, cuyo nombre debe corresponder con la norma ISO 639[14] y dentro de esto, otro directorio llamado LC_MESSAGES que contendrá los mensajes traducidos reales. Se puede observar un ejemplo en la siguiente figura:

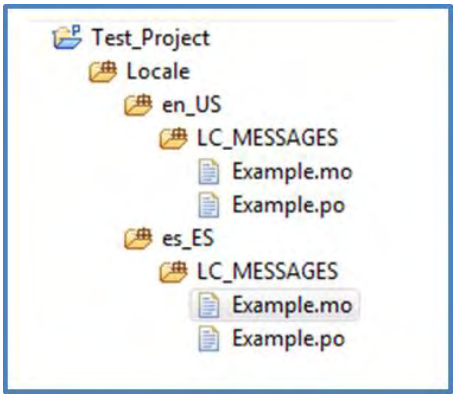

Figura 5 Estructura de los directorios de un proyecto PHP donde se hace uso de Gettext PHP

Los archivos de extensión *.po* (*Portable Object*) almacenaran las cadenas traducidas. Se tendrá un archivo *.po* por cada lenguaje. Los archivos *.mo* son los archivos en formato binario leídos por PHP para las traducciones. Estos archivos *.po* deben crearse apropiadamente. Se debe usar un editor como el *Poedit*, puesto que deben ser llenados ciertos campos para la configuración del proyecto, tales como el lenguaje, el país, el set de caracteres y la ruta donde está alojado la extensión *PHP Gettext*. *Poedit* genera automáticamente los archivos *.mo*.

El archivo *.po* debe lucir de la siguiente manera:

[14]ISO 639: Estándar internacional para códigos de lenguajes.
http://www.iso.org/iso/ES/home/standards/language_codes.htm

Example.po

```
1  msgid ""
2  msgstr ""
3  "Project-Id-Version: \n"
4  "POT-Creation-Date: 2015-01-01 10:10+0530\n"
5  "POT-Revision-Date: 2015-01-01 12:10+0530\n"
6  "Last-Translator: \n"
7  "Language-Team: \n"
8  "MIME-Version: 1.0\n"
9  "Content-Type: text/plain; charset=UTF-8\n"
10 "Content-Transfer-Encoding: 8bit\n"
11 "X-Generator: Poedit 1.6.5\n"
12 "X-Poedit-Basepath: .\n"
13 "Plural-Forms: nplurals=2; plural=(n != 1);\n"
14 "Language: es_ES\n"
15
16 #Test
17 msgid "This is a text string"
18 msgstr "Esta es una cadena de texto"
```

Figura 6 Ejemplo de archivo *.po*

De la línea 1 a la 14, están las configuraciones del archivo *.po*. *msgid* identifica una cadena de texto que debe ser traducida, y *msgstr* es la cadena de texto que reemplazará la de *msgid*.

Si deseáramos probar la traducción de esta cadena, se crearía un pequeño archivo PHP y la manera en la que obtenemos las cadenas traducidas es por medio de la función *gettext*. De esta manera, la salida sería la cadena de texto "*Esta es una cadena de texto*", puesto que corresponde a la traducción en el lenguaje español (es_ES) de la cadena "*This is a text string*". (Ver Figura 7)

test_project.php

```php
<?php
//I18N support information here
$language = "es_ES";
putenv("LANG=".$language);
setlocale(LC_ALL, $language);

//Set the text domain as "messages"
$domain = "messages";
bindtextdomain($domain, "Locale");
textdomain($domain);

echo gettext("This is a text string");
?>
```

Figura 7 Forma de obtener las cadenas traducidas con *gettext*

2.2 *Framework* CLEDA

CLEDA es un *framework* (marco de trabajo) de software libre implementado en Java desarrollado por G. Gutiérrez, A. Salas y A. Pérez. (Gutiérrez, D. et al., 2007). Fue diseñado para desarrollar Sistemas de Información y Aplicaciones Empresariales.

2.2.1 Arquitectura del Framework CLEDA

Gutiérrez, D. et al. (2007), indica que para la implementación de CLEDA se utilizó la arquitectura MVC (Modelo-Vista-Controlador)[15], por su diseño para reducir el esfuerzo de programación, la reutilización de código y la facilidad de mantenimiento de los sistemas desarrollados.

En la figura 8 podemos visualizar la arquitectura del *framework* CLEDA.

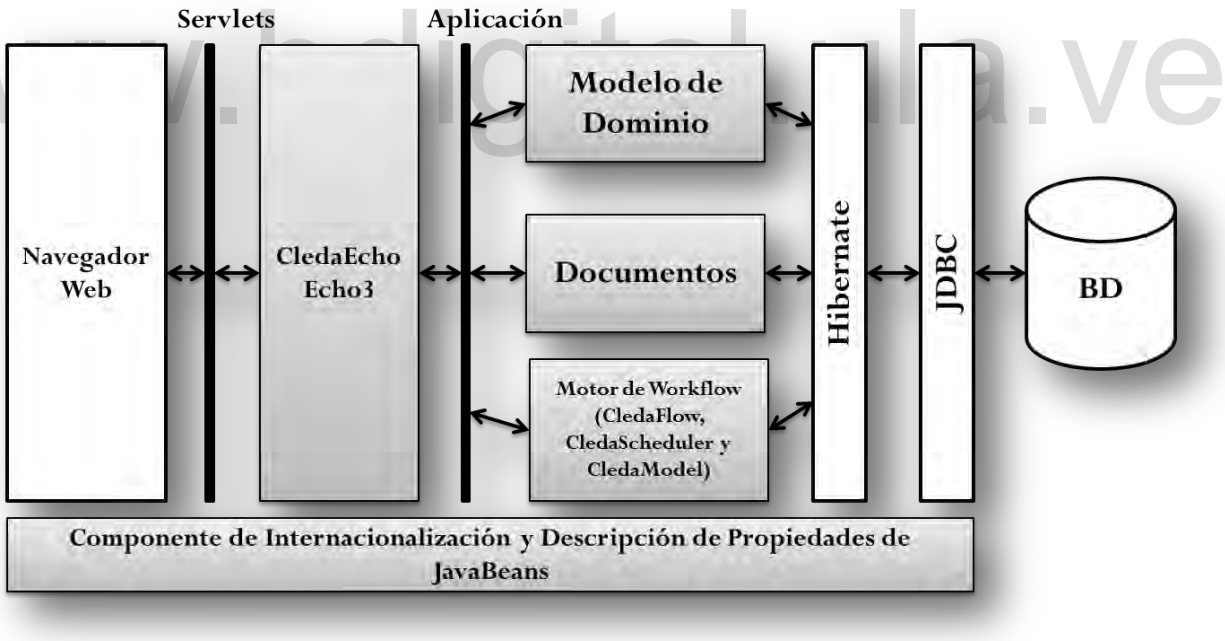

Figura 8 Arquitectura de CLEDA (Adaptado de Gutiérrez, D. et al -2007)

[15] Modelo-Vista-Controlador: patrón de arquitectura de software que separa los datos y la lógica de negocio de una aplicación, de la interfaz de usuario y el módulo encargado de gestionar los eventos y las comunicaciones: http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html

2.3 Estrategia de Internacionalización de CledaI18N

CledaI18N es el componente de internacionalización y descripción de propiedades del *framework* CLEDA. La estrategia de localización utiliza archivos de propiedades (*.properties*). Estos archivos tienen un nombre base y un sufijo que define el idioma.

Se tiene una instrucción que selecciona y carga el archivo correcto según la localización actual del sistema, y otro para obtener la cadena correcta del archivo. Si una cadena se utiliza en el código pero ésta no existe en el archivo *.properties* correspondiente, se genera un error en tiempo de compilación.

Es posible cambiar el nombre de una clave utilizando herramientas de renombramiento existentes. Se elimina totalmente el uso de cadenas de caracteres de código. No es necesario especificar las claves de los mensajes requeridos o el nombre base del archivo de recursos en forma de cadenas de caracteres.

La figura 9 muestra la estrategia general de CledaI18N para la internacionalización de cadenas.

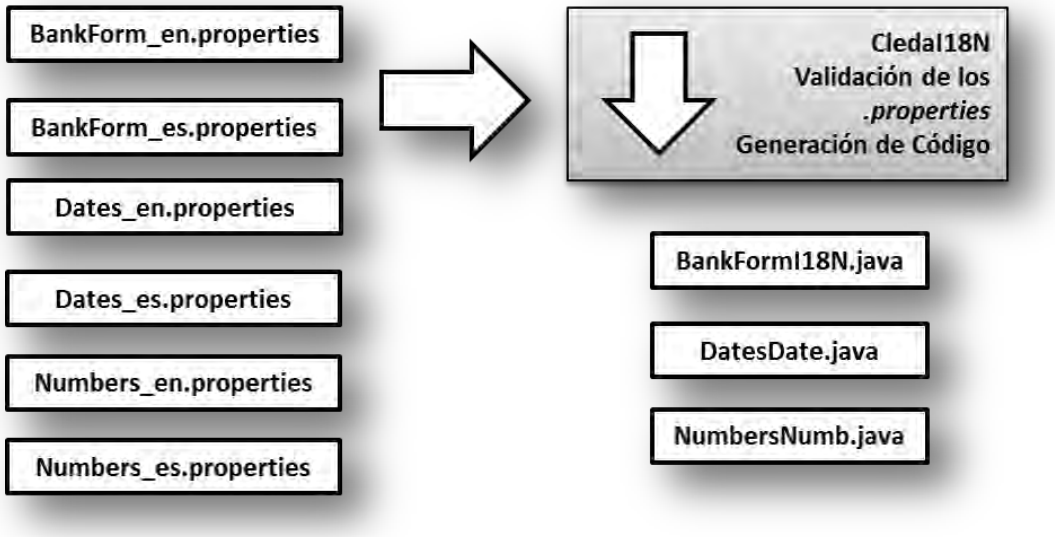

Figura 9 Estrategia general de CledaI18N

2.4 Estrategia de Internacionalización estándar de Java vs CledaI18N

La estrategia de internacionalización de Java nos permite eliminar las cadenas de textos en los mensajes de nuestras aplicaciones, sin embargo, estas cadenas de texto en realidad son sustituidas por las claves de los archivos de propiedades y estas claves son literales de cadenas.

En la figura 2 se muestra un ejemplo de un pequeño programa Java que desea imprimir saludos en la salida estándar por pantalla. Particularmente una de las sentencias es:

```
System.out.println("Hello.");
```

Una vez internacionalizado el programa, tenemos que esta sentencia se convierte en

```
System.out.println(messages.getString("greetigns"));
```


Donde "*greetigns*" es la clave para buscar el texto correspondiente a la traducción.

A pesar de que la aplicación está internacionalizada, aún hay cadenas de texto en el código (las claves), solo se ha reemplazado una cadena por otra. El enfoque de CledaI18N es eliminar o reducir al máximo estas claves.

Uno de los motivos para esto es que el exceso de literales de tipo cadena usualmente resta legibilidad al código. Estos no suelen ser verificados por los compiladores para determinar la validez de su contenido; por el contrario, si su contenido no es válido, el error es detectado en tiempo de ejecución. Adicionalmente, los literales de tipo cadena (bien sean claves de internacionalización o mensajes de usuario o de otro tipo) no pueden ser manejados por herramientas que permiten hacer renombramiento del código.

De este último de los motivos expuesto, podemos agregar que entornos de desarrollo integrados (IDEs) tales como Eclipse, NetBeans, BlueJ, entre otros; permiten renombrar variables, clases, interfaces, métodos, etcétera, de forma consistente, tal que todos los cambios necesarios en el

resto del código fuente se hacen automáticamente, y es una ventaja que se desea agregar al componente de internacionalización y descripción de propiedades de CLEDA.

CledaI18N propone la creación por medio de generación de código, de clases con métodos estáticos para acceder a los archivos de propiedades y que devuelvan los valores de las claves de estos archivos.

La generación de código permitirá que, si se realizan modificaciones en los archivos de propiedades (agregar, remover o modificar algún par clave-valor) que vuelvan a estos archivos inconsistentes, el error sea detectado en tiempo de compilación, puesto que los métodos usados para la obtención de las cadenas traducidas no existirían.

2.5 Reflexión y Meta-programación

La reflexión es una propiedad que permite representar el estado de los objetos de un sistema como entidades de primera clase, y por tanto poder observarlos, manipularlos y razonar sobre ellos como elementos básicos. (Ramos y Lozano, 2000).

La reflexión computacional se entiende como la habilidad de una entidad de software de conocer o modificar su estado. A la primera forma se le denomina reflexión estructural, y a la segunda reflexión de comportamiento. Es una actividad computacional que razona sobre su propia computación. Por lo general la reflexión es dinámica o en tiempo de ejecución, aunque algunos lenguajes de programación permiten reflexión estática o en tiempo de compilación.

Cuando el código fuente de un programa es compilado, se pierde la información sobre la estructura del programa. Si este permite reflexión, se preserva la estructura como metadatos del código generado.

Entre las características de los lenguajes de programación con reflexión, está:

- Descubrir y modificar construcciones de código fuentes como objetos de categoría superior en tiempo de ejecución.

- Convertir una cadena que corresponde al nombre de una clase o función en una referencia o invocación a esa clase o función.
- Evaluar una cadena como si fuera una sentencia de código fuente en tiempo de ejecución.

Se denomina Meta-programación a la programación que utiliza técnicas de reflexión Esta consiste en escribir programas que escriben o manipulan otros programas (o a sí mismo) como datos, o que hacen en tiempo de compilación parte del trabajo que se haría en tiempo de ejecución, permitiendo al programador ahorrar tiempo de desarrollo. (Ramos y Lozano, 2000).

Un compilador es la herramienta de meta-programación más común, la cual permite al programador escribir un programa relativamente sencillo en un lenguaje de alto nivel para luego escribir un programa equivalente en lenguaje ensamblador o lenguaje máquina.

Java es un lenguaje reflexivo. Los programas Java pueden reflejar su propia ejecución y estructura. Los meta-objetos del sistema se pueden reificar[16] como objetos ordinarios que pueden ser consultados e inspeccionados como cualquier otro objeto. Los meta-objetos de Java son: clases, métodos, atributos, constructores, modificadores y paquetes. Este procedimiento de reflexión también es conocido como introspección. (Vivona, 2011).

El lenguaje de programación Java tiene una interfaz de programación de aplicaciones llamada *Reflection API*, que contiene toda la funcionalidad para hacer reflexión en programas escritos en lenguaje Java. (ORACLE, 2014b).

Entre las operaciones que se pueden realizar con la *Reflection API* están: el recuperar objetos de una clase, examinar los modificadores de las clases y los tipos, y obtener los miembros de una clase.

A continuación se observa un ejemplo de reflexión en Java.

[16] reificar: tener un tipo de datos para una abstracción

Se tiene una interfaz C, donde solo se declara un método llamado operación. La implementación de este método se realizará en las clases que implementen esta interfaz.

```
public interface C {
    public int operación(int a, int b);
}
```

La clase A implementa la interfaz C, y adicionalmente contiene unos atributos y constructores.

```
public class A implements C {
    private int n1;
    private int n2;

    public A(){
        n1 = 0;
        n2 = 0;
    }

    public A (int n1, int n2){
        this.n1 = n1;
        this.n2 = n2;
    }

    @Override
    public int operacion(int a, int b){
        return a+b;
    }
}
```

Mediante reflexión, se observan los miembros de la clase A: los atributos, los constructores, el tipo de modificador de la clase, y otras características.

Reflection API contiene un conjunto de métodos y funciones que permiten la visualización de estos metadatos. A continuación se muestran algunas de estas funciones

```
…
    try {
        Class c = Class.forName("A");
        Class i[] = c.getInterfaces();

        System.out.println("Nombre de la clase: " + c.getName());
        int m = c.getModifiers();
        System.out.println("Modificadores: " + m);
```

```
        // Obtencion de atributos de clase o interfaz
            System.out.print("Interface: ");

        if (Modifier.isInterface(m))

            System.out.println("Es una interfaz");
        else {
            System.out.println("No es una interfaz");

            System.out.print("Static: ");
            if (Modifier.isStatic(m))
                System.out.println("Es clase estática");
            else
                System.out.println("No es clase estática");

            System.out.print("Final: ");
            if (Modifier.isFinal(m))
                System.out.println("Es clase final");
            else
                System.out.println("No es clase final");

            System.out.print("Public: ");
            if (Modifier.isPublic(m))
                System.out.println("Es clase publica");
            else
                System.out.println("No es clase publica");

            System.out.print("Private: ");
            if (Modifier.isPrivate(m))
                System.out.println("Es clase privada");
            else
                System.out.println("No es clase privada");

            System.out.print("Protected: ");
            if (Modifier.isProtected(m))
                System.out.println("Es clase protegida");
            else
                System.out.println("No es clase protegida");

            System.out.print("Abstract: ");
            if (Modifier.isAbstract(m))
                System.out.println("Es clase abstracta");
            else
                System.out.println("No es clase abstracta");

            System.out.println("Interfaces implementadas: ");
            for (int n =0; n<i.length; n++)
                System.out.println(i[n].getName());
        }
        // Obtencion de propiedades (variables publicas)
        System.out.println("Propiedades");
        Field[] f = c.getDeclaredFields();
        for (int n=0; n<f.length; n++) {
            System.out.print("Nombre: " + f[n].getName());
            System.out.println(", Tipo: " + f[n].getType().getName());
        }

        // Obtencion de constructores
```

```
        System.out.println("Constructores");
        Constructor[] co = c.getDeclaredConstructors();

        for (int n=0; n<co.length; n++) {
            System.out.print("Nombre: " + co[n].getName());
            Class pa[] = co[n].getParameterTypes();

            for (int y=0; y<pa.length; y++)
                System.out.print(", Parametro: " + pa[y].getName());
        }

        // Instanciacion dinamica
        C objeto = (C) c.newInstance();
        System.out.println("Resultado: " + objeto.operacion(5, 2));
    }
    catch (Exception e) {
        System.out.println("Error: " + e.toString());
    }
…
```

La salida generada se visualiza en la figura 10, donde se observa que los métodos de *Reflection API* han evaluado los componentes de la clase A. Se verifican los metadatos de la clase: qué tipo de modificadores tiene la clase, si implementa interfaces, las propiedades, los constructores y la instanciación dinámica de un objeto de la interfaz C:

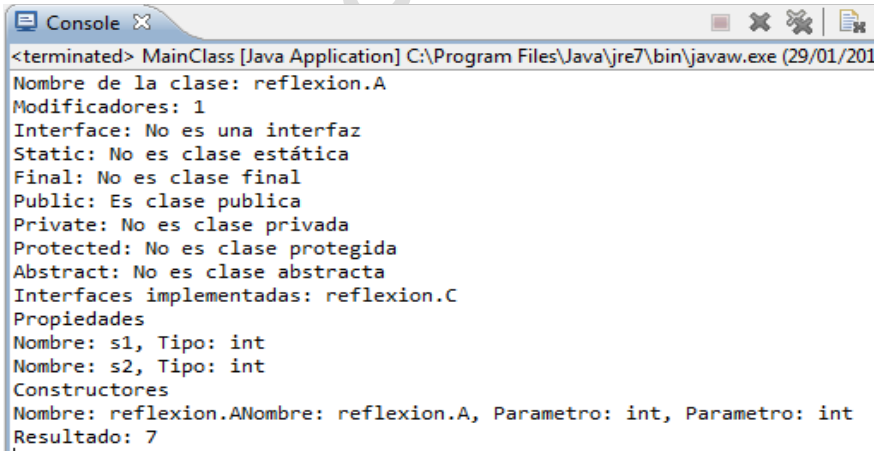

Figura 10 Verificación de los metadatos de una clase, usando la *Reflection API* de Java

Capítulo 3

Análisis y Requerimientos

En este capítulo se presenta el análisis de la herramienta de internacionalización de cadenas, formatos de fecha y números, y descriptores de propiedades CledaI18N existente, así como el análisis y diseño del prototipo a desarrollar.

3.1 Análisis de CledaI18N actual

Actualmente, CledaI18N está generando las clases de internacionalización de cadenas, formatos de fecha y de números; sin embargo, la manera en que se está llevando a cabo este procedimiento, es inadecuado, puesto que presenta deficiencias.

3.1.1 Generación de las clases de internacionalización – i18n

El principal problema al momento de generar las clases de internacionalización, es la ejecución de los archivos que contienen las claves de internacionalización. CledaI18N no tiene la característica de generar las clases de internacionalización para todos los archivos de claves, sino que debe ejecutarse uno a uno cada uno de estos archivos. Se deben ejecutar desde consola una serie de archivos "*_CledaI18N*" para generar las clases que contendrán los componentes de internacionalización específicos. En la siguiente figura, se puede observar un claro ejemplo de esto:

```
public class _CledaI18N extends BaseI18NMain {

    public static void main (String[] args) throws Exception {

        _CledaI18N m = new _CledaI18N();
        m.i18n("TestI18NFoo");
```

```
        m.numb("TestI18NFoo");
        m.date("TestI18NFoo");
    }
}
```

Figura 11 Ejemplo de una archivo _CledaI18N.java que genera las clases con los componentes de internacionalización, para los archivos de claves *TestI18NFoo.properties*, *TestNumbFoo.properties* y *TestDateFoo.properties*.

Se observa el llamado de cada una de las funciones que se encargan de cada uno de los componentes de internacionalización (cadenas, fechas y números). Esto solo generará las clases con los componentes de internacionalización de esos tres tipos de archivos (*TestI18NFoo.properties, TestDateFoo.properties y TestNumbFoo.properties*). Pero, ¿qué sucede si se tienen varios (sino muchos) archivos de claves, para generar clases con sus componentes de internacionalización? En este caso es donde se vislumbra el problema. Tendría que realizarse varias (sino muchas) clases *_CledaI18N*.java para generar las clases de internacionalización.

A raíz de este inconveniente, se decide automatizar el proceso de generación de código. Se busca conseguir que sea necesaria una sola ejecución para generar las clases con los componentes de internacionalización de todos los archivos de claves existentes en el proyecto en que se está trabajando. Otro de los problemas que presenta CledaI18N, es que el código generado anteriormente permanece alojado en los directorios. Si estas clases generadas anteriormente, no corresponden a los archivos de claves actuales, se presentan errores de consistencia. Debido a este problema, se deben depurar los directorios de salida, antes de la generación de código nuevo. Adicionalmente, CledaI18N no tiene soporte desde ninguna herramienta de desarrollo de software o de construcción de proyectos, por esto se desea integrar la ejecución de CledaI18N al *workflow* de compilación de Eclipse y de Maven.

3.1.2 Generación de los descriptores de propiedades de los JavaBeans

Con respecto al generador de descriptores de propiedades de los JavaBeans, en CLEDA se hace uso intensivo de reflexión (*reflection programming*). Por ejemplo, si se desea describir un campo de un formulario, en lugar de tener algo como lo siguiente:

```
beanDatos.setNombre(txtNombre.getText());
```

Se puede indicar que a un campo en particular de un formulario le corresponde la propiedad "nombre", mediante una función que recorre los campos, y dado el nombre de la propiedad en el *beanDatos*, usando reflexión se le asigna el valor del campo al JavaBean.

Sin embargo, para poder realizar esto, se necesita el nombre de la propiedad en el código, pero el nombre es un literal de cadena, que puesto directo es como si fuera un número mágico (un literal del cual no se sabe nada). Entonces cuando se modifica el nombre de una propiedad en el JavaBean, si no se modifica el literal, el código compilará, pero fallará en la ejecución.

Esto se soluciona con los descriptores de propiedades. La idea es que en lugar de usar literales, el código use constantes definidas en los descriptores de las propiedades, de modo que si se cambia el nombre de una propiedad en el JavaBean, el nombre de la constante va a cambiar, y el código ya no compilará, por lo que el error se atrapa en tiempo de compilación, mas no en tiempo de ejecución.

Sin embargo, la manera en que CledaI18N realiza esto es deficiente, debido a que es necesario conocer el nombre del JavaBean y tener un archivo *_CledaI18N.java* que se encargue de la llamada a las funciones que se encargan de la generación de los descriptores de propiedades. Si se tiene varios (sino muchos) JavaBeans, se tendrán varios (sino muchos) archivos *_CledaI18N.java* para generar los descriptores de propiedades. En la figura 12, se muestra un ejemplo de cómo lucen estos archivos *_CledaI18N.java*:


```
public class _CledaI18N extends BaseI18NMain {

    public static void main (String[] args) throws Exception {

        _CledaI18N m = new _CledaI18N();
        m.prop(FooBean.class);
    }
}
```

Figura 12 Ejemplo de una archivo *_CledaI18N.java* que genera los descriptores de propiedades del JavaBean *FooBean.java*.

Como en el caso de las clases con los componentes de internacionalización, también se decide automatizar el proceso de generación de código de los descriptores de propiedades. Se busca conseguir

que sea necesaria una sola ejecución, para generar los descriptores de propiedades de todos los JavaBeans con los que se cuente en un proyecto en particular.

3.2 Requerimientos del nuevo prototipo

Se establecen los requerimientos que deben tener las herramientas a desarrollar. En primer lugar se especifican los requerimientos necesarios para el desarrollo de las herramientas de internacionalización y de descripción de propiedades. En segundo lugar, se muestra los casos de uso del sistema, junto con la descripción de cada uno de ellos y su respectivo diagrama de actividades. Finalmente se procede al diseño de la arquitectura general de la herramienta y al diseño de los componentes de esta. Se establece que la herramienta se divida en dos componentes, uno dedicado a la internacionalización y el otro a la generación de los descriptores de propiedades de JavaBean. Por cuestiones de estandarización, llamaremos los componentes CledaI18N y CledaProp respectivamente. Se entiende que estos componentes están dirigidos a desarrolladores de software, por lo tanto se define un único actor. Este actor es el Desarrollador, y es el único que hace uso tanto de CledaI18N como de CledaProp. Los componentes deben tener las siguientes características:

Con respecto a los requerimientos de internacionalización - CledaI18N:

- Validar que los archivos de claves (*.properties*) cumplan con un patrón de nombre requerido, establecido por una expresión regular[17], que valida los nombres de los archivos.
- Verificar que los archivos con las claves (*.properties*), estén completos, sin claves repetidas o faltantes.
- El generador de código sea automático. Que no sea necesario indicar uno a uno los archivos con claves para generar las clases de internacionalización, sino que se le indique un directorio y a partir de este, se generen las clases para todos los archivos válidos.

Con respecto a los requerimientos del descriptor de propiedades - CledaProp:

[17] Expresión regular usada para validar los nombres de los archivos *.properties,* ver Apéndice A. Pág. 98

- Verificar que las clases que se desean generar sus descriptores de propiedades, sean de hecho JavaBeans. Esto se debe hacer verificando la existencia de una anotación[18] que indicaría que a la clase en particular se le desea generar un descriptor de propiedades.
- El generador de código sea automático. Que no sea necesario indicar una a una las clases que se desea generar su descriptor de propiedades, sino que se le indique un directorio y a partir de este se generen los descriptores de propiedades de las clases válidas.

3.2.1 Casos de uso

En la figura 13, se definen los requerimientos a través de casos de usos para mayor comprensión del lector. Estos casos de uso son necesarios para satisfacer las necesidades del Desarrollador, tomando en cuenta las características del componente, planteadas anteriormente.

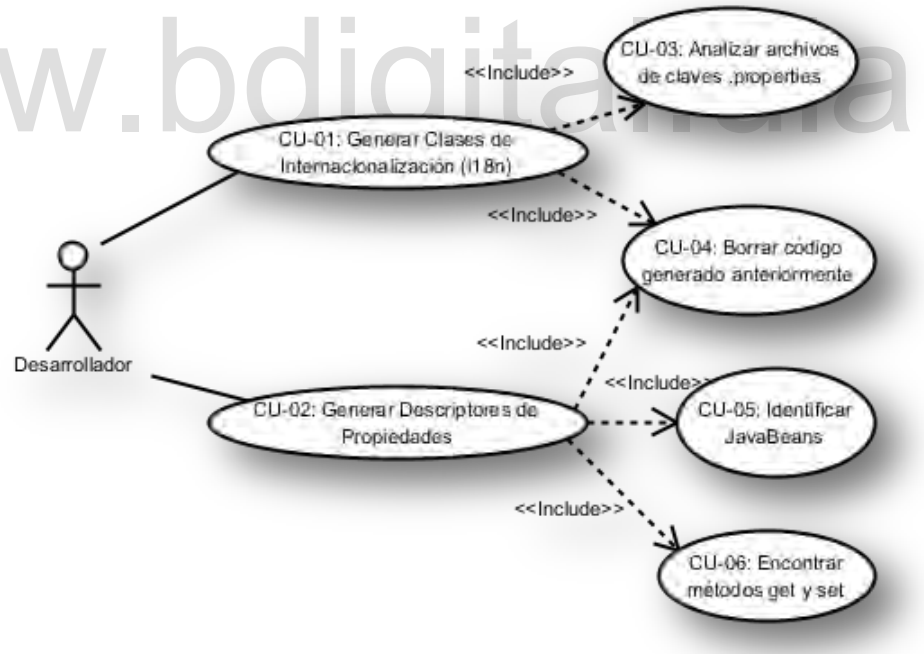

Figura 13 Casos de uso del sistema

[18] Anotación que indica que una clase es un JavaBean de CLEDA para la generación de su descriptor de propiedades, ver Apéndice A. Pág 98.

3.2.1.1 Generar clases de internacionalización (i18n) CU-01

Tabla 1 CU-01

Generar Clases de Internacionalización (i18n)	
Número	01
Descripción	Permite al desarrollador generar las clases de internacionalización (i18n) requeridas.
Flujo Normal	1. El desarrollador selecciona el directorio de entrada (donde están los archivos fuentes .properties) y el directorio de salida (donde irán las clases de i18n). 2. El Scanner verifica los directorios y los archivos de claves (.*properties*). 3. El generador de código genera las clases de i18n en el directorio de salida.
Flujo Alternativo	Si los archivos de entrada no son de tipo correcto o no cumple el patrón de nombre requerido, no se generan las clases de i18n.
Actores	Desarrollador

En la figura 14 se visualiza el diagrama de actividades para el caso de uso CU-01: Generar clases de internacionalización.

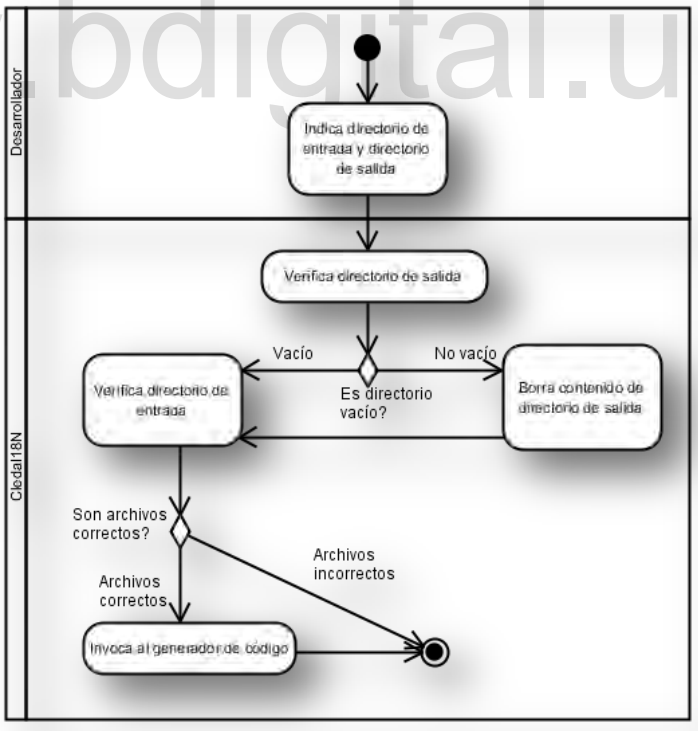

Figura 14 Diagrama de actividades CU-01

3.2.1.2 Generar descriptores de propiedades CU-02

Tabla 2 CU-02

Generar Descriptores de Propiedades	
Número	02
Descripción	Permite al desarrollador generar los descriptores de propiedades de las clases requeridas.
Flujo Normal	1. El usuario selecciona el directorio de entrada (donde están las clases fuentes) y el directorio de salida (donde irán los descriptores de propiedades). 2. El Scanner verifica los directorios y las clases. 3. El Generador de Código genera los descriptores de propiedades en el directorio de salida.
Flujo Alternativo	Si los archivos de entrada no son de tipo correcto, o no cumplen con el estándar de los JavaBeans, no se generan los descriptores de propiedades.
Actores	Desarrollador

En la figura 15 se visualiza el diagrama de actividades para el caso de uso CU-02: Generar descriptores de propiedades.

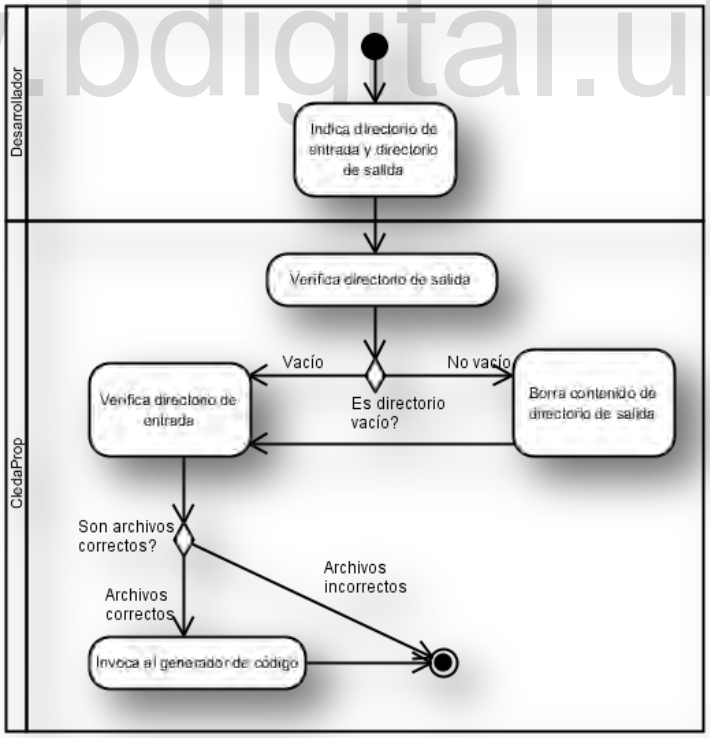

Figura 15 Diagrama de actividades CU-02

3.2.1.3 Analizar archivos de claves *.properties* CU-03

Tabla 3 CU-03

Analizar archivos de claves .properties	
Número	03
Descripción	Revisa si un archivo .properties es consistente con los requerimientos, para generar clases de internacionalización.
Flujo Normal	1. CledaI18N comprueba que el archivo es de extensión *.properties* 2. CledaI18N confirma que el nombre del archivo corresponde al patrón de nombres establecido para los archivos de claves de internacionalización.
Flujo Alternativo	1. El archivo no es de extensión .properties, entonces no es un archivo de claves. 2. El nombre del archivo no corresponde al patrón de nombre establecidos, por lo tanto no es archivo de claves de internacionalización
Actores	CledaI18N

En la figura 16 se visualiza el diagrama de actividades para el caso de uso CU-03: Analizar archivos de claves *.properties*.

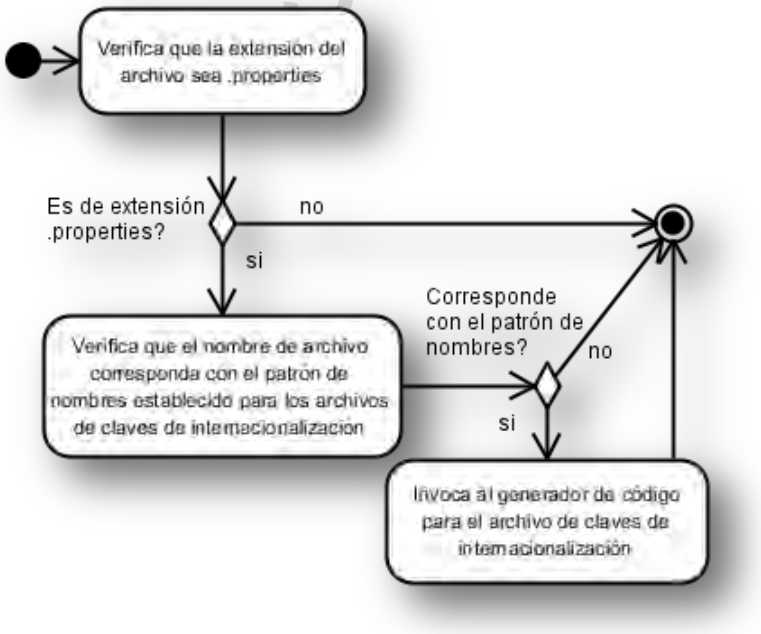

Figura 16 Diagrama de actividades CU-03

3.2.1.4 Borrar código generado anteriormente CU-04

Tabla 4 CU-04

Borrar código generado anteriormente	
Número	04
Descripción	Borra código generado anteriormente.
Flujo Normal	1. CledaI18N verifica que el directorio de salida esté vacío. 2. CledaI18N comprueba que el directorio de salida no es vacío. 3. CledaI18N borra el contenido del directorio de salida.
Flujo Alternativo	1. Como el directorio de salida es vacío, no se hace nada.
Actores	CledaI18N

En la figura 17 se visualiza el diagrama de actividades para el caso de uso CU-04: Borrar código generado anteriormente.

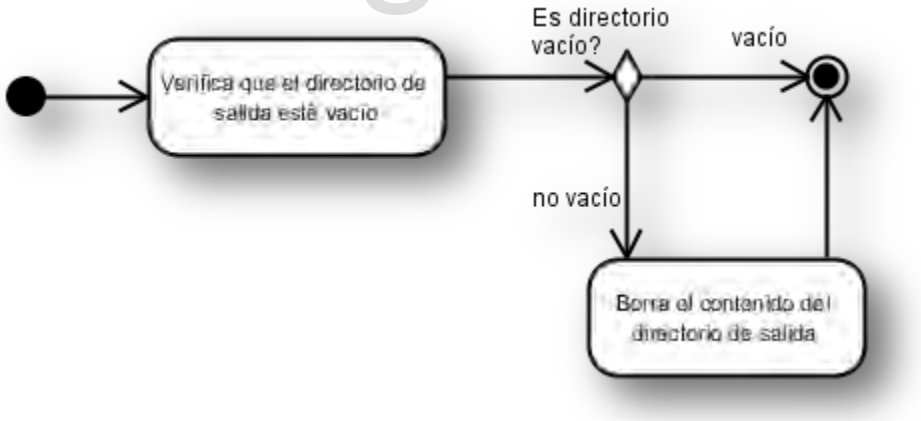

Figura 17 Diagrama de actividades CU-04

3.2.1.5 Identificar JavaBeans CU-05

Tabla 5 CU-05

Identificar JavaBeans	
Número	05
Descripción	Valida que un archivo sea un JavaBean
Flujo Normal	1. CledaI18N comprueba que la extensión del archivo sea *.java* 2. CledaI18N comprueba que el archivo contiene la anotación definida en Cleda para indicar que una clase es un JavaBean.
Flujo Alternativo	1. La extensión del archivo no es .java, entonces no es un JavaBean. 2. El archivo no contiene la anotación definida para indicar que es un Bean de Cleda, entonces no es un JavaBean.
Actores	CledaI18N

En la figura 18 se visualiza el diagrama de actividades para el caso de uso CU-05: Identificar JavaBeans.

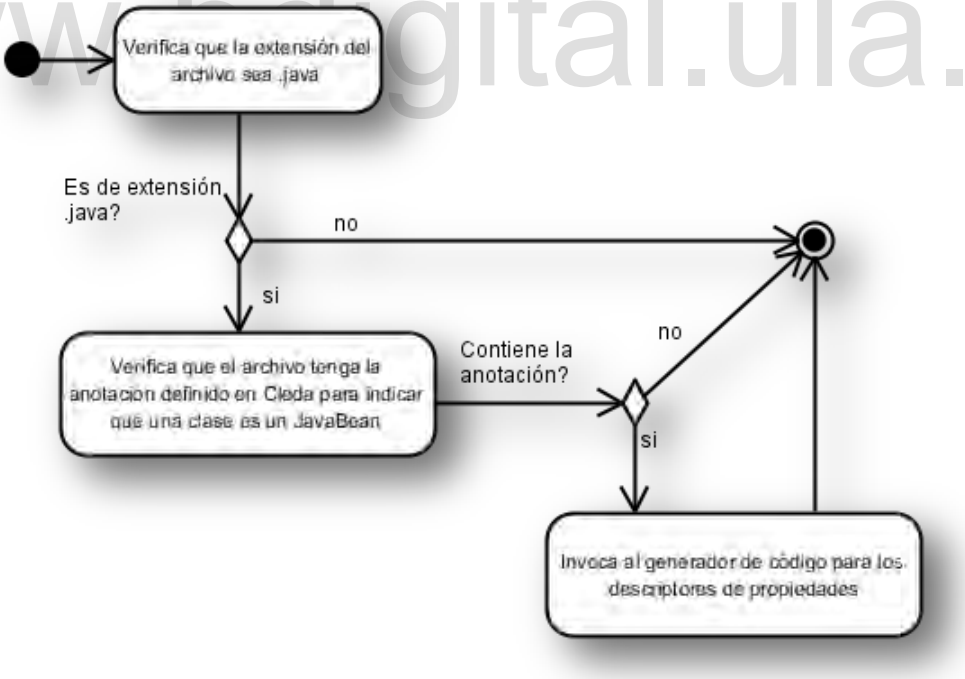

Figura 18 Diagrama de actividades CU-05

3.2.1.6 Encontrar métodos *get* y *set* CU-06

Tabla 6 CU-06

Encontrar métodos get y set	
Número	06
Descripción	Encuentra los métodos get y set de las propiedades del JavaBean
Flujo Normal	1. CledaI18N recorre el JavaBean buscando los métodos modificadores (con prefijos *set*, *is* o *get*). 2. Por cada método get o is, encuentra un set (y viceversa). 3. Construye el nombre de la propiedad removiendo el prefijo (*get*, *is* o *set*).
Flujo Alternativo	2. No se consigue ambos métodos modificadores que correspondan a una propiedad. Por lo tanto no se generará descriptor de esa propiedad.
Actores	CledaI18N

En la figura 19 se visualiza el diagrama de actividades del caso de uso CU-06 Encontrar métodos *get* y *set*

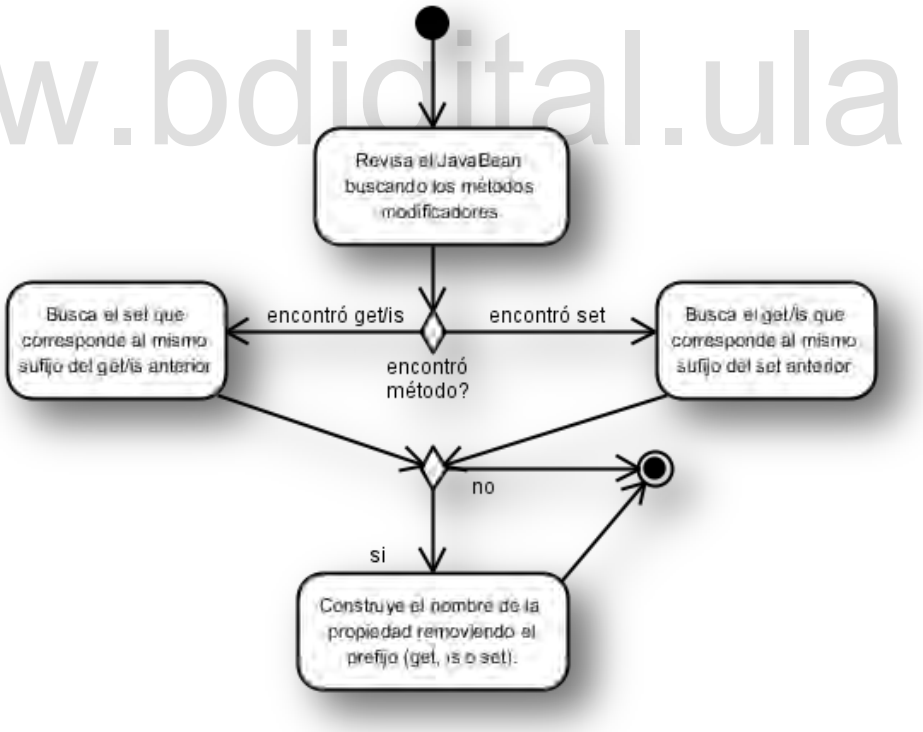

Figura 19 Diagrama de actividades CU-06

Capítulo 4

Arquitectura y Desarrollo de CledaI18N y CledaProp

En la aplicación de la metodología, cada ciclo de la espiral representa una fase del desarrollo del proyecto. Cada ciclo representa la ejecución de uno de los objetivos establecidos. Los objetivos planteados por cada ciclo son:

- Ciclo 1: Explorar distintas técnicas de internacionalización, enfocándose en la técnica de internacionalización usada en el lenguaje Java, en contraste con la técnica empleada por CledaI18N.
- Ciclo 2: Estudiar la arquitectura del componente de internacionalización y descripción de propiedades, analizando cada módulo que lo compone.
- Ciclo 3: Diseñar e implementar un componente que automatice el recorrido de directorios y validación de archivos, tanto para archivos de claves (,*properties*) como de JavaBeans (.*java*).
- Ciclo 4: Diseñar e implementar un analizador y procesador de anotaciones, haciendo uso de la *Java Compiler API*, para la identificación de los JavaBeans de CLEDA.
- Ciclo 5: Diseñar e implementar un analizador y procesador de clases, para obtener los métodos consultores y modificadores de los JavaBeans, haciendo uso de la *Java Compiler API*, para la construcción de los descriptores de propiedades de JavaBeans.
- Ciclo 6: Elaborar los objetivos ejecutables de Maven (*Mojos*) para CledaI18N y CledaProp, de modo que éstos se puedan insertar en el proceso de compilación de Maven.
- Ciclo 7: Elaborar un plugin en Maven para la ejecución de los *Mojos* de CledaI18N y CledaProp.

Se realizan dos prototipos, CledaI18N y CledaProp. CledaI18N es el componente de generación de clases de internacionalización, y CledaProp es el componente de generación de descriptores de propiedades de JavaBeans.

En general, la intención de este proyecto es definir una arquitectura versátil que permita adaptarse para la realización de las tareas de internacionalización y descripción de propiedades de JavaBeans.

El componente que realiza el proceso de recorrido de directorios y revisión de archivos se llama Scanner. Este Scanner se basa en el patrón de diseño Visitor[19], el cual permite definir operaciones distintas para cada implementación de las interfaces que opera el Scanner.

Por ejemplo, en el proceso de generación de las clases de internacionalización, se verifica que los archivos fuentes para la generación de código son de extensión .properties; en el proceso de generación de los descriptores de propiedades de JavaBeans se verifica que los archivos fuentes son de extensión .java. Es un procedimiento similar, pero el criterio es distinto. De esta manera, la clase que implementa la interfaz de validación de archivos (*IsProcessable*) para internacionalización, se define distinto a la clase que implementa la misma interfaz para los descriptores de propiedades.

4.1 Arquitectura de CledaI18N

Los criterios con los que se define la arquitectura del componente de internacionalización y descripción de propiedades son:

1. La dependencia entre los componentes que conforman la arquitectura debe ser mínima, delimitándolos por la funcionalidad de cada uno de ellos.
2. Pueda insertarse en el workflow de compilación de Maven o Eclipse.

La arquitectura para CledaI18N se observa en la figura 20.

[19] Patrón de diseño Visitor, ver Apéndice B. Pág. 106.

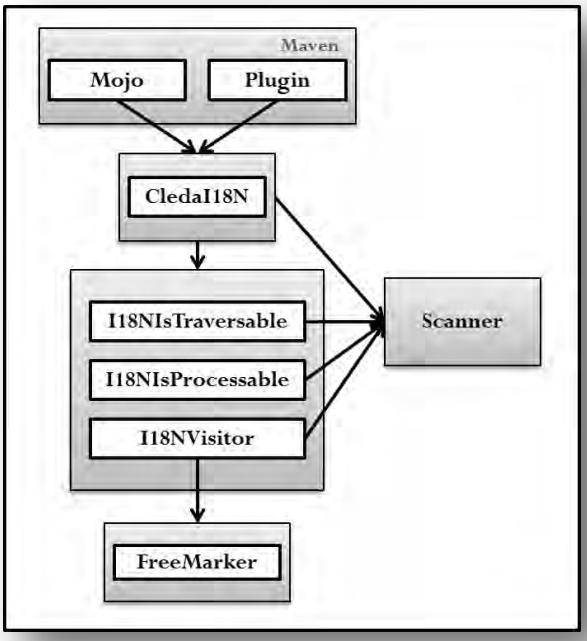

Figura 20 Arquitectura de CledaI18N

La especificación de cada uno de los componentes de CledaI18N se observan en la siguiente figura:

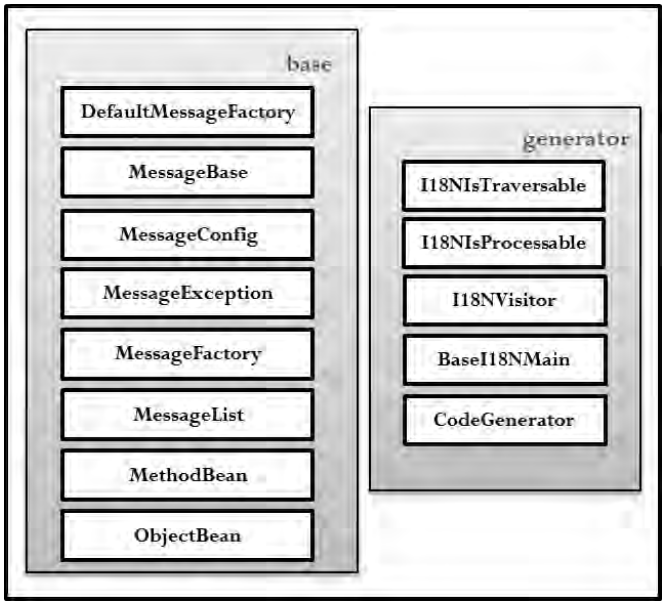

Figura 21 Componentes de CledaI18N

El paquete *generator* contiene las clases involucradas en el proceso de recorrido de directorios, revisión de archivos y generación de las clases de internacionalización. Las clases que implementan las interfaces del *Scanner*[20] son *I18NISTraversable*[21], *I18NIsProcessable*[22] e *I18NVisitor*[23] . Estas se detallan más adelante.

En la figura 22, se puede observar el diagrama de componentes para CledaI18N:

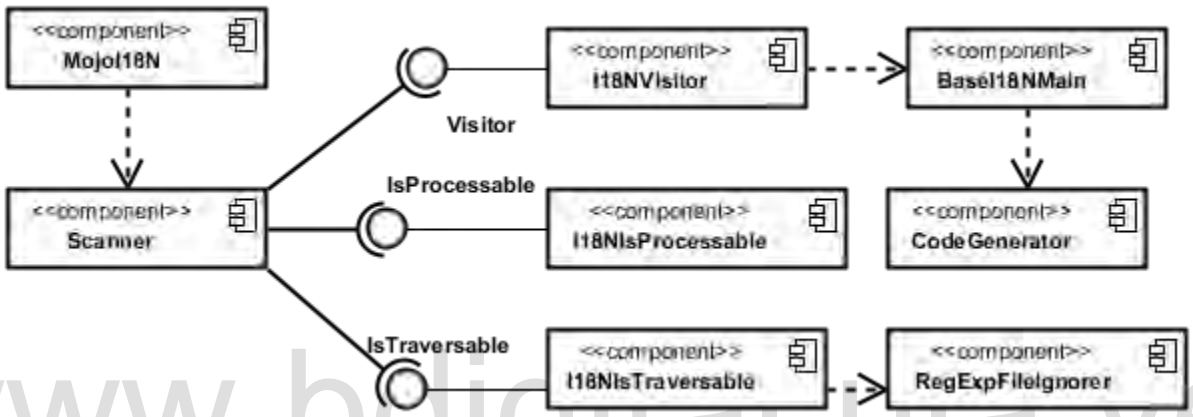

Figura 22 Diagrama de Componentes de CledaI18N

4.2 Arquitectura de CledaProp

Los criterios con los que se define la arquitectura de CledaProp son los mismos de CledaI18N (ver página anterior).

La arquitectura para CledaProp se observa en la figura 23.

[20] *Scanner*: ver Pág. 53.
[21] *I18NIsTraversable*: ver Tabla 8. Pág. 56.
[22] *I18NIsProcessable*: ver Tabla 9. Pág. 57.
[23] *I18NVisitor*: ver Tabla 10. Pág 59.

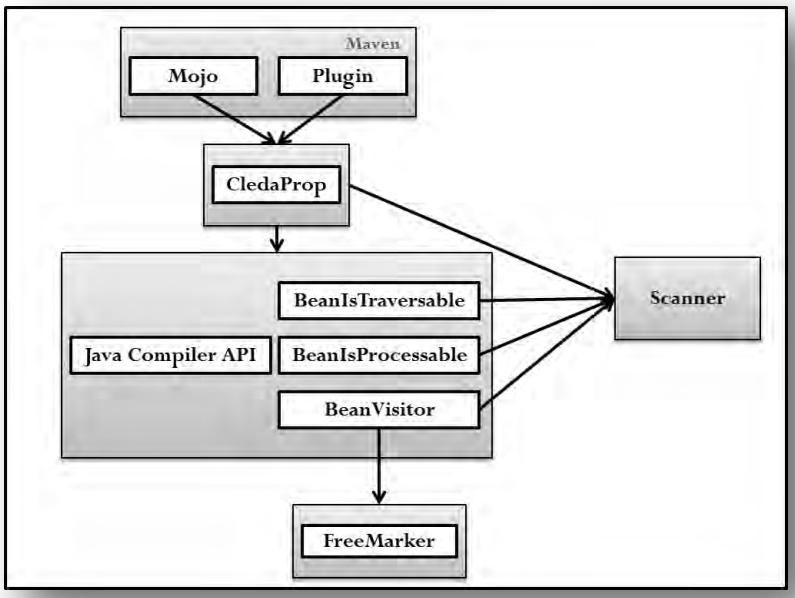

Figura 23 Arquitectura de CledaProp

La especificación de cada uno de los componentes de CledaProp se observan en la siguiente figura:

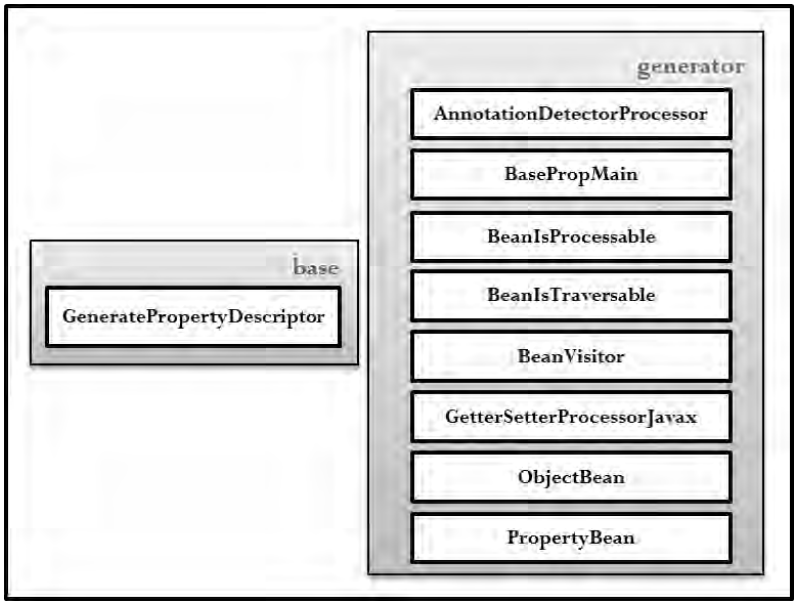

Figura 24 Componentes de CledaProp

El paquete *base* contiene la definición de la etiqueta (o anotación) *@GeneratePropertyDescriptor* que deben llevar los JavaBeans a los que se les desea generar el descriptor de propiedades.

El paquete *generator* contiene las clases involucradas en el proceso de recorrido de directorios, revisión de archivos y generación de los descriptores de propiedades. *AnnotationDetectorProcessor* es la clase que se encarga de revisar los archivos en búsqueda de la etiqueta *@GeneratePropertyDescriptor*, para determinar que dicho archivo es un JavaBean. Implementa la interfaz *AbstractProcessor* de la *Java Compiler API*. *GetterSetterProcessorJavax* es la clase que se encarga de revisar los JavaBean en búsqueda de los métodos de acceso (*getters* y *setters*) de las propiedades. Implementa la interfaz *AbstractProcessor* de la *Java Compiler API*. Las clases que implementan las interfaces del *Scanner* son *BeanIsTraversable*[24], *BeanIsProcessable*[25] y *BeanVisitor*[26] . Estas se detallan más adelante.

En la figura 25, se puede observar el diagrama de componentes para CledaProp:

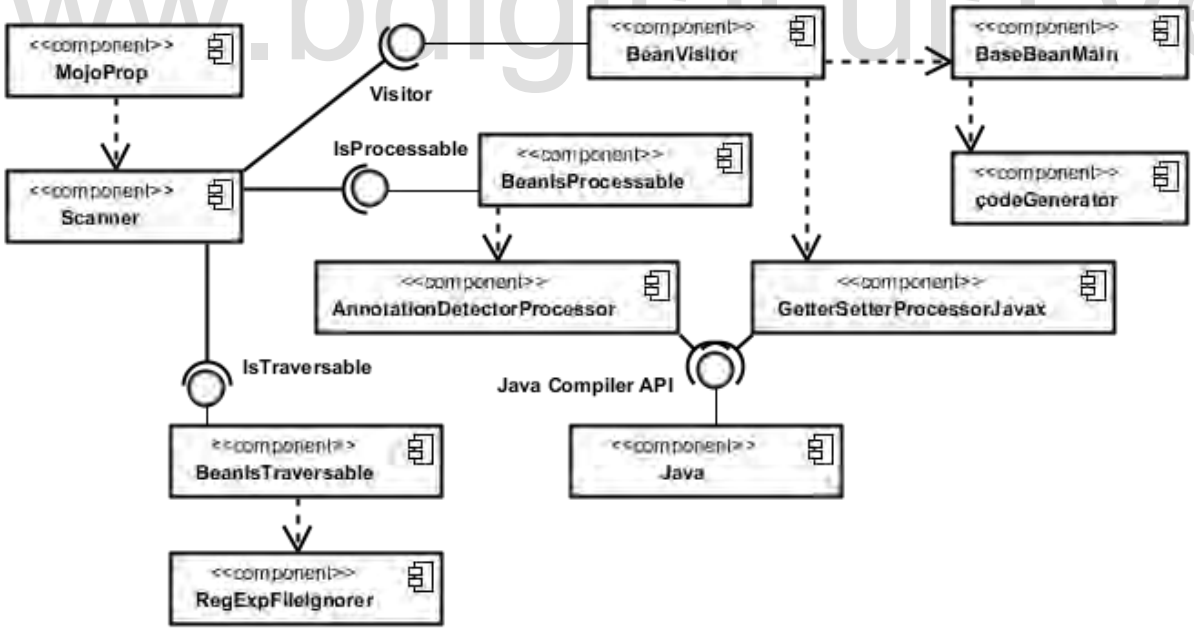

Figura 25 Diagrama de Componentes para CledaProp

[24] *BeanIsTraversable*: ver Tabla 8. Pág. 56.
[25] *BeanIsProcessable*: ver Tabla 9. Pág. 57.
[26] *BeanVisitor*: ver Tabla 10. Pág. 59

4.3 Componente Maven

Tanto CledaI18N como CledaProp, tienen un componente Maven. Éste componente está conformado por dos estructuras: el *Mojo*[27] (*Maven Old Java Object*) que se encarga de la ejecución del programa, y el p*lugin* que se encarga de la ejecución de Mojo.

Se tendrá un *Mojo* para la ejecución del generador de código de internacionalización (CledaI18N) y otro *Mojo* para la ejecución del generador de código del descriptor de propiedades (CledaProp). De esta manera se limita la funcionalidad de la herramienta para cada uno de los objetivos:

Se realiza un *plugin* que contiene ambos *Mojo*, y en el momento de la ejecución de este *plugins*, se indica que *Mojo* se desea ejecutar.

4.3.1 Mojo

El *Mojo* no es más que una clase Java, que contiene ciertas anotaciones, las cuales especifican los metadatos del objetivo. En la siguiente figura se puede ver la implementación del *Mojo* que usa CledaI18N. (Figura 26)


```
@Mojo(name = "i18n", defaultPhase = LifecyclePhase.GENERATE_SOURCES)
public class MojoI18N extends AbstractMojo {

  protected static final Logger log = LoggerFactory.getLogger( //
    MojoI18N.class.getName());

// ----------------------------------------------------------------------

  @Parameter (property = "source", required = true)
  private File source;

  @Parameter (property = "target", required = true)
  private File target;

// ----------------------------------------------------------------------

  private I18NIsTraversable traversableI18N;
  private I18NIsProcessable processableI18N;

  private I18NVisitor visitorI18N;
```

[27] Mojo: *Maven plain Old Java Object*, es un objetivo ejecutable en Maven, ver Apéndice B. Pág. 101.

```
  private Scanner i18n;

// ----------------------------------------------------------------------

  public void execute() throws MojoExecutionException {
    StaticLoggerBinder.getSingleton().setLog(getLog());

    log.info(MessageFormat.format( //
        "source directory is {0}", source.getAbsolutePath()));
    log.info(MessageFormat.format( //
        "target directory is {0}", target.getAbsolutePath()));

    target.mkdirs();

    File[] directoryTree = target.listFiles();

    if (directoryTree.length > 0) {
        log.info(MessageFormat.format( //
            "deleting target directory {0}", target.getAbsolutePath()));

        CledaFileUtils.deleteDirectory(target);
        target.mkdir();
    }

    traversableI18N = new I18NIsTraversable();
    processableI18N = new I18NIsProcessable();
    visitorI18N = new I18NVisitor(processableI18N, target);

    i18n = new Scanner(traversableI18N, processableI18N, visitorI18N);
    i18n.run(source);
  }
}
```

Figura 26 Implementación de *MojoI18N.java* donde se indica la ejecución del scanner para la generación de las clases de internacionalización, formatos de fecha y números.

La sentencia:

```
@Mojo(name = "i18n", defaultPhase = LifecyclePhase.GENERATE_SOURCES)
```

Indica que el *goal* u objetivo del *Mojo* se llama "i18n", y que este se une a la ejecución del *plugin* en la fase "*GENERATE_SOURCES* del ciclo de vida del proyecto. Maneja dos parámetros:

```
@Parameter(property = "source", required = true)
private File source;

@Parameter(property = "target", required = true)
private File target
```

Estos parámetros son la ruta de entrada "*source*", donde se encuentran los archivos fuentes a procesar, y la ruta de salida "*target*", donde se encontrarán los archivos generados.

El método:

```
public void execute() throws MojoExecutionException { /* …  */}
```

Es el que indica que procesos ejecuta el *Mojo*. En este caso en particular, verifica que el directorio de salida "*target*" esté vacío. Si no es vacío, borra lo que contiene, para luego invocar al Scanner para la generación de las clases de internacionalización de cadenas, formatos de hora y números.

El *Mojo* que describe el proceso de generación de los descriptores de propiedades de los JavaBeans, es similar al de Internacionalización.

```
@Mojo(name = "prop", defaultPhase = LifecyclePhase.GENERATE_SOURCES)
public class MojoProp extends AbstractMojo {

  protected static final Logger log = LoggerFactory.getLogger( //
    MojoProp.class.getName());

// ----------------------------------------------------------------------
  @Parameter(property = "source", required = true)
  private File source;

  @Parameter(property = "target", required = true)
  private File target;
 // ----------------------------------------------------------------------
  private BeanIsTraversable traversableBean;
  private BeanIsProcessable processableBean;

  private BeanVisitor visitorBean;

  private Scanner bean;
 // ----------------------------------------------------------------------
  public void execute() throws MojoExecutionException {
    StaticLoggerBinder.getSingleton().setLog(getLog());

    log.info(MessageFormat.format( //
        "source directory is {0}", source.getAbsolutePath()));
    log.info(MessageFormat.format( //
        "target directory is {0}", target.getAbsolutePath()));

    target.mkdirs();

    File[] directoryTree = target.listFiles();

    if (directoryTree.length > 0) {
        log.info(MessageFormat.format( //
            "deleting target directory {0}", target.getAbsolutePath()));
```

```
            CledaFileUtils.deleteDirectory(target);
            target.mkdir();
        }

        traversableBean = new BeanIsTraversable();
        processableBean = new BeanIsProcessable();
        visitorBean = new BeanVisitor(target);

        bean = new Scanner(traversableBean, processableBean, visitorBean);

        bean.run(source);
    }
}
```

Figura 27 Implementación de *MojoProp.java* donde se indica la ejecución del scanner para la generación de los descriptores de propiedades de los JavaBeans.

4.3.2 POM del Mojo

El archivo *pom.xml* (*POM – Project Object Model*) es la unidad fundamental para trabajar en Maven. El contiene información acerca del proyecto y los detalles de las configuraciones usadas por Maven para la construcción del proyecto. Es una representación en XML.


```
<?xml version="1.0"?>
<project
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
       http://maven.apache.org/xsd/maven-4.0.0.xsd"
    xmlns="http://maven.apache.org/POM/4.0.0"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" >
 <modelVersion>4.0.0</modelVersion>

 <parent>
   <artifactId>Minotauro</artifactId>
   <groupId>com.minotauro</groupId>
   <version>0.0.1-SNAPSHOT</version>
   <relativePath>..</relativePath>
 </parent>
 <artifactId>MinotauroI18NMaven</artifactId>
 <packaging>maven-plugin</packaging>

 <build>
   <plugins>
      <plugin>
         <groupId>org.apache.maven.plugins</groupId>
         <artifactId>maven-plugin-plugin</artifactId>
         <version>3.2</version>

         <executions>
            <execution>
              <id>default-descriptor</id>
              <goals>
                 <goal>descriptor</goal>
```

```
            </goals>
            <phase>process-classes</phase>
          </execution>
          <execution>
            <id>help-descriptor</id>
            <goals>
               <goal>helpmojo</goal>
            </goals>
            <phase>process-classes</phase>
          </execution>
        </executions>

      </plugin>
  </plugins>
</build>

<dependencies>

  <!-- MAVEN -->

  <dependency>
    <groupId>com.googlecode.slf4j-maven-plugin-log</groupId>
    <artifactId>slf4j-maven-plugin-log</artifactId>

  </dependency>
  <dependency>
    <groupId>org.apache.maven</groupId>
    <artifactId>maven-plugin-api</artifactId>
  </dependency>

  <dependency>
    <groupId>org.apache.maven.plugin-tools</groupId>
    <artifactId>maven-plugin-annotations</artifactId>
  </dependency>

  <!-- MINOTAURO -->

  <dependency>
    <groupId>com.minotauro</groupId>
    <artifactId>MinotauroI18N</artifactId>
  </dependency>

</dependencies>
</project>
```

Figura 28 Archivo pom.xml del MojoI18N.java

En la figura 28 se puede observar el archivo de configuración *pom.xml* del *Mojo* que usa CledaI18N.

Entre la etiqueta <*dependencies*></*dependencies*> se indican las dependencias que tiene *MojoI18N.java*

El elemento <*build*></*build*> maneja cosas como la estructura de directorios del proyecto o la gestión de *plugins*.

El *pom.xml* del *MojoProp.java* es similar al de *MojoI18N.java* (Ver figura 27).

```
<?xml version="1.0"?>
<project
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
       http://maven.apache.org/xsd/maven-4.0.0.xsd"
    xmlns="http://maven.apache.org/POM/4.0.0"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" >
 <modelVersion>4.0.0</modelVersion>

 <parent>
   <artifactId>Minotauro</artifactId>
   <groupId>com.minotauro</groupId>
   <version>0.0.1-SNAPSHOT</version>
   <relativePath>..</relativePath>
 </parent>
 <artifactId>MinotauroPropMaven</artifactId>
 <packaging>maven-plugin</packaging>

 <build>
   <plugins>
      <plugin>
         <groupId>org.apache.maven.plugins</groupId>
         <artifactId>maven-plugin-plugin</artifactId>
         <version>3.2</version>

         <executions>
            <execution>
              <id>default-descriptor</id>
              <goals>
                 <goal>descriptor</goal>
              </goals>
              <phase>process-classes</phase>
            </execution>
            <execution>
              <id>help-descriptor</id>
              <goals>
                 <goal>helpmojo</goal>
              </goals>
              <phase>process-classes</phase>
            </execution>
         </executions>
      </plugin>
   </plugins>
 </build>

 <dependencies>
   <!-- MAVEN -->
   <dependency>
     <groupId>com.googlecode.slf4j-maven-plugin-log</groupId>
```

```
      <artifactId>slf4j-maven-plugin-log</artifactId>

    </dependency>
    <dependency>
      <groupId>org.apache.maven</groupId>
      <artifactId>maven-plugin-api</artifactId>
    </dependency>

    <dependency>
      <groupId>org.apache.maven.plugin-tools</groupId>
      <artifactId>maven-plugin-annotations</artifactId>
    </dependency>

    <!-- MINOTAURO -->
    <dependency>
      <groupId>com.minotauro</groupId>
      <artifactId>MinotauroProp</artifactId>
    </dependency>

    <dependency>
      <groupId>com.minotauro</groupId>
      <artifactId>MinotauroUtils</artifactId>
    </dependency>
  </dependencies>
</project>
```

Figura 29 Archivo pom.xml del MojoProp.java

4.3.3 Plugin en Maven

Los *Mojos* de Maven se ejecutan desde un *plugin*. Un *plugin* consta de solo un archivo de configuración *pom.xml* donde se indica los *Mojos* que componen a este, y la configuración de cada uno de ellos. En la figura 30 se visualiza el archivo de configuración *pom.xml* del *plugin* que ejecuta los *Mojos* de CledaI18N y CledaProp.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
      http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.minotauro</groupId>
  <artifactId>Plugin</artifactId>
  <version>0.0.1-SNAPSHOT</version>

  <build>
    <plugins>
       <plugin>
          <groupId>com.minotauro</groupId>
          <artifactId>MinotauroI18NMaven</artifactId>
          <version>0.0.1-SNAPSHOT</version>
          <configuration>
```

```
            <source>src/main/resources</source>
            <target>src/main/i18n</target>
         </configuration>
      </plugin>

      <plugin>
         <groupId>com.minotauro</groupId>
         <artifactId>MinotauroPropMaven</artifactId>
         <version>0.0.1-SNAPSHOT</version>
         <configuration>
            <source>src/main/resources</source>
            <target>src/main/propDesc</target>
         </configuration>
      </plugin>

   </plugins>
 </build>
</project>
```

Figura 30 Archivo pom.xml del *plugin* que ejecuta ambos Mojos

Las etiquetas <*plugin*></*plugin*> indican ambos *Mojos* que se desean ejecutar. Especifica los descriptores <*groupId*></*groupId*>, <*artifactId*></*artifactId*> y <*version*></*version*> de cada uno de los *Mojos*, y dentro de la etiqueta *properties* se describen los parámetros que estos *Mojos* reciben.

4.4 Scanner

La manera en que CledaI18N realizaba la generación de código anteriormente, era indicando manualmente cada uno de los archivos a procesar. Debido a que uno de los requerimientos que tendría la nueva herramienta era la automatización de este procedimiento, se realizó el Scanner.

Este *Scanner* está basado en el patrón de diseño Visitor. Hace uso de unas interfaces para el recorrido de directorios, revisión de archivos y procesamiento de estos.

Se utiliza este patrón de diseño, para no limitar el *Scanner* a solo la generación de las clases de internacionalización y los descriptores de propiedades, de modo que si se necesita automatizar el proceso de generación de código para otra herramienta de CLEDA u otro proyecto, se tiene implementado el mecanismo de revisión de directorios y archivos.

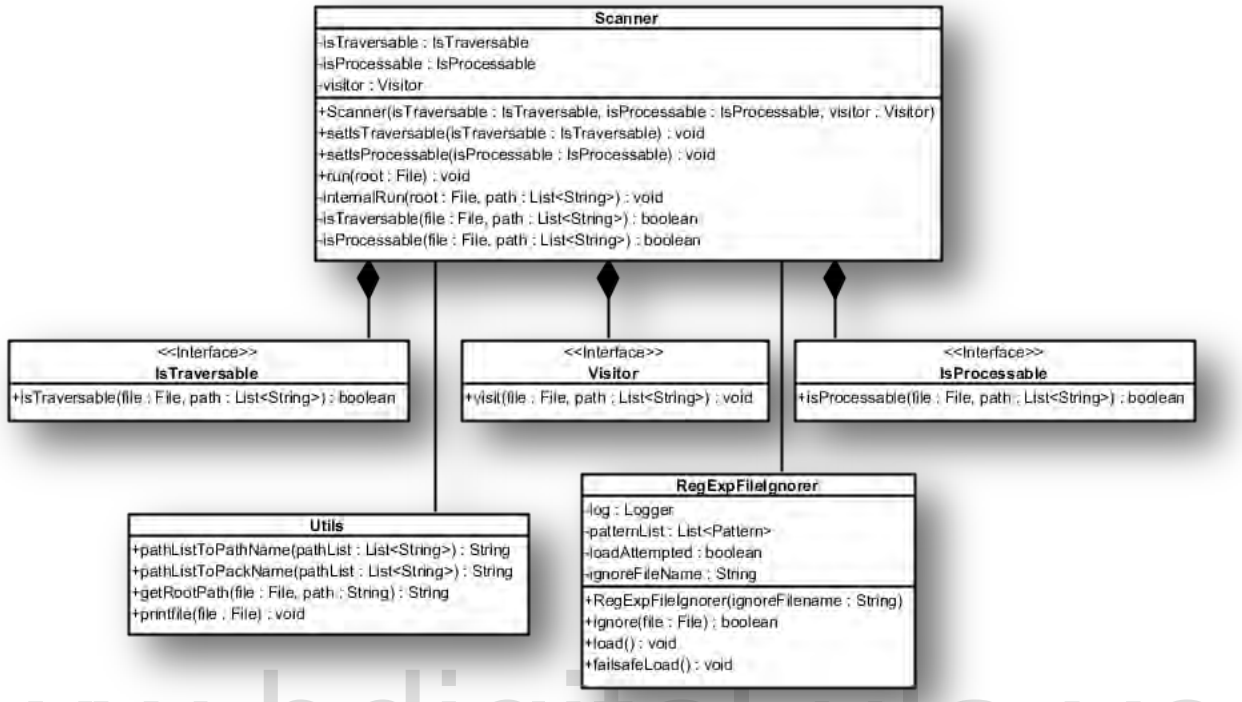

Figura 31 Diagrama de clases del *Scanner*

Tabla 7 Métodos de la figura 32

Método	Descripción
Scanner	
run(root : File)	Ejecuta el método privado *internalRun()*.
internalRun(root:File, path:List<String>)	Se encarga de la llamada a los métodos *isTraversable()* e *isProcessable()* para la verificación de los directorios y los archivos fuentes. También invoca el método *visit()* que se encargará de generar código dependiendo del *visitor* que se haya declarado.
IsTraversable	
isTraversable(file:File, path:List<String>)	Es solo la declaración del método. Las clases que implementen esta interfaz, deben sobrescribir este método.
IsProcessable	

isProcessable(file:File, path:List<String>)	Es solo la declaración del método. Las clases que implementen esta interfaz, deben sobrescribir este método.
Visitor	
visit(file:File, path:List<String>)	Es solo la declaración del método. Las clases que implementen esta interfaz, deben sobrescribir este método.
RegExpFileIgnorer	
ignore(file : File)	Verifica si un archivo se debe o no ignorar, dependiendo de si cumple o no el patrón de nombres.
failsafeLoad()	Método que maneja la carga de un archivo. Valida que no sea un archivo que se deba ignorar.
Utils	
pathListToPathName(pathList : List<String>)	Convierte la lista que contiene la ruta de un archivo en una cadena de texto que será el nombre de esta ruta. Función de utilidad para calcular la ruta que tendrán los archivos de salida.
pathListToPackName(pathList : List<String>)	Convierte la lista que contiene la ruta de un archivo en una cadena de texto que será el nombre del paquete del archivo de salida generado.
getRootPath(file : File, path : String)	Retorna la ruta donde está un archivo.

La función privada *InternalRun*() de la clase *Scanner*, es la que realiza las instrucciones necesarias para la el recorrido de directorios, revisión y validación de archivos y llamada a los generadores de código. A su vez, ella es llamada por la función pública *run*(). La figura 32 muestra la porción de código donde se define esta función.

```
public void run(File root){
    internalRun(root, new ArrayList<String>());
}

//-----------------------------------------------------------------------

private void internalRun(File root, List<String> path) {
  if(!root.exist() || !root.isDirectory()) {
        return;
```

```
  }
  File[] files = root.listFiles();
  for (int i = 0; i < files.length; i++) {
    if ( files[i].isDirectory() ) {
      if ( isTraversable( files[i], path ) ) {
        path.add( files[i].getName() );
        internalRun( files[i], path );
        path.remove( path.size() - 1 );
      }
    } else {
      if ( isProcessable( files[i], path ) ) {
        visitor.visit( files[i], path );
      }
    }
  }
}
```

Figura 32 Funciones que se encargan de invocar los recorridos de los directorios, revisión y verificación de archivos, e invocación de los métodos para la generación de código.

4.4.1 Interfaz IsTraversable

Se definieron dos clases para implementar la interfaz *IsTraversable*. Una clase para verificar que los directorios sean compatibles con los directorios fuentes de internacionalización, y la otra clase es para verificar que los directorios sean compatibles con los directorios fuentes de descriptores de propiedades. Ver figura 33.

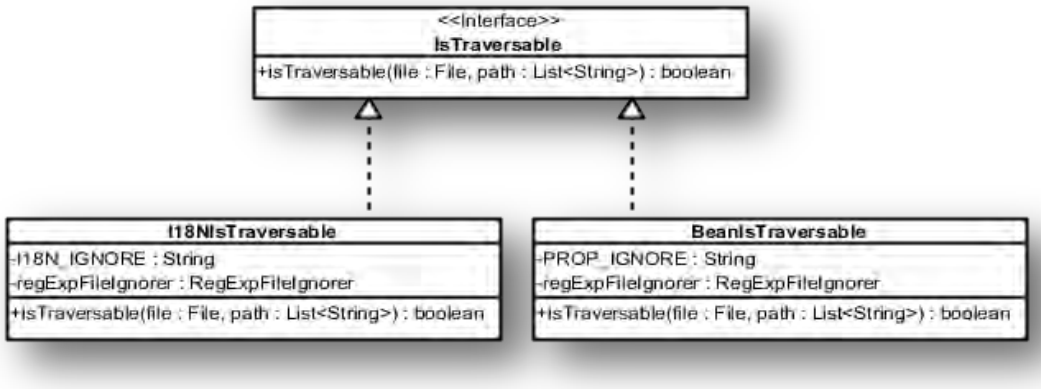

Figura 33 Diagrama de clases que implementan la interfaz *IsTraversable*

Tabla 8 Métodos de la figura 33

Método	Descripción
I18NIsTraversable	
isTraversable(file:File, path:List<String>)	Verifica que un directorio se pueda recorrer para la generación de componentes de internacionalización. Si retorna verdadero, es un directorio que se debe recorrer, de lo contrario, no se recorre.
BeanIsTraversable	
isTraversable (file:File, path:List<String>)	Verifica que un directorio se pueda recorrer para la generación de descriptores de propiedades de JavaBeans. Si retorna verdadero, es un directorio que se debe recorrer, de lo contrario, no se recorre.

4.4.2 Interfaz IsProcessable

Se definieron dos clases para implementar la interfaz *IsProcessable*. Una clase para verificar que los archivos sean de claves para la internacionalización (*.properties*), y la otra clase es para verificar que los archivos sean JavaBeans, para poder generarles los descriptores de propiedades. Ver figura 34.

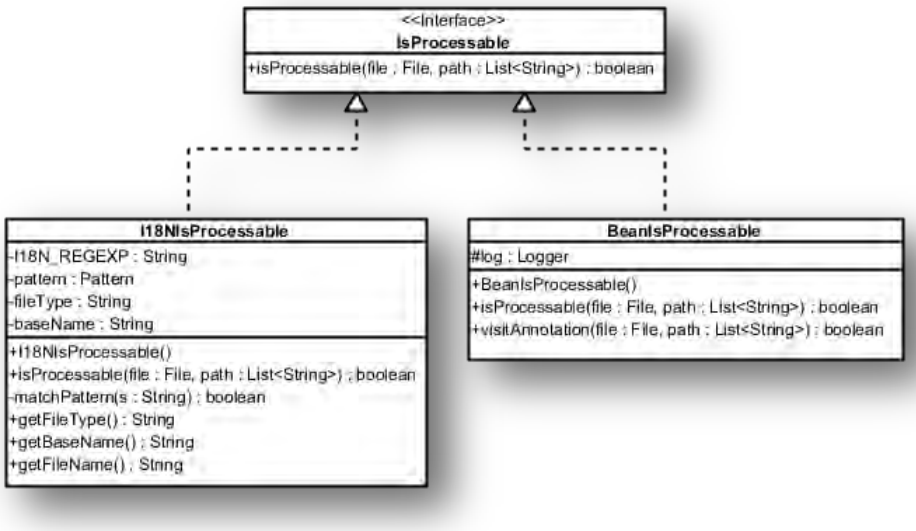

Figura 34 Diagrama de clases que implementan la interfaz *IsProcessable*

Tabla 9 Métodos de la figura 34

Método	Descripción
I18NIsProcessable	
isProcessable(file : File, path : List<String>)	Verifica que un archivo sea correcto. Si cumple con el patrón de nombres establecido para archivos de claves (*.properties*), retorna verdadero; de lo contrario retorna falso.
matchPatterns(s : String)	Compara una cadena "s" con un patrón establecido por medio de una expresión regular. Si la cadena armoniza con el patrón retorna verdadero, de lo contrario retorna falso.
getFileType()	Devuelve el tipo de archivo que se está revisando.
getBaseName()	Devuelve el nombre base del archivo que se está revisando.
getFileName()	Devuelve el nombre del archivo que se está revisando.
BeanIsProcessable	
isProcessable(file : File, path : List<String>)	Verifica que el archivo sea correcto. Esto es, primero se asegura que la extensión del archivo sea "*.java*" (en vista de que queremos generar descriptores de propiedades de JavaBeans). Luego, revisa que el archivo contenga la anotación establecida para indicar que es un JavaBean de CLEDA, esto lo hace llamando al método *visitAnnotation()*.
visitAnnotation(file : File, path : List<String>)	Instancia un *AnnotationDetectorProcessor*, y lo usa para revisar el archivo y detectar la etiqueta que indica que es un JavaBean de CLEDA (*@GeneratePropertyDescriptor*). Si consigue la etiqueta, devuelve verdadero; de lo contrario devuelve falso.

4.4.3 Interfaz Visitor

Se definieron dos clases para implementar la interfaz *Visitor*. Una para la generación de las clases de internacionalización de cadenas, formato de fechas y números, y la otra para la generación de los descriptores de propiedades de los JavaBeans. Ver figura 35.

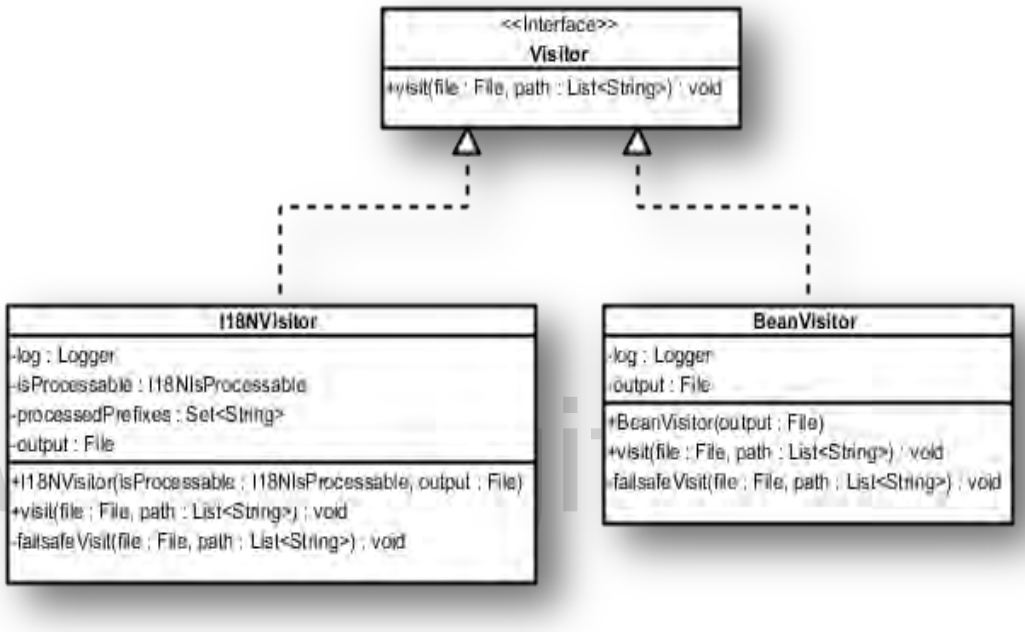

Figura 35 Diagrama de clases que implementan la interfaz *Visitor*

Tabla 10 Métodos de la figura 35

Método	Descripción
I18NVisitor	
visit(file : File, path : List<String>)	Este método invoca al método privado *failsafeVisit()*
failsafeVisit(file : File, path : List<String>)	Este método realiza varias instrucciones. Construye los nombres de los paquetes donde se almacenaran las clases de i18n generadas. Verifica que los archivos de claves (*.properties*) generen clases de i18n una sola vez, validando que los prefijos de los nombres de los archivos no se repitan. Y finalmente llama a los métodos del generador de

	código para cada uno de los tipos (internacionalización de cadenas, formatos de fecha o formatos de números).
BeanVisitor	
Visit(file : File, path : List<String>)	Este método invoca al método privado *failsafeVisit()*
failsafeVisit(file : File, path : List<String>)	Éste método instancia un compilador, y procesa un archivo que es una clase Java, en busca de los métodos *set* y *get* de cada propiedad de esa clase. Lo realiza instanciando un *GetterSetterProcessorJavax*, que no es más que una clase que extiende de la clase *AbstractProcessor* de la *Java Compiler API*. Esta clase tiene métodos que permiten la verificación y detección de las propiedades a las cuales se le generarán descriptores.

4.5 Eliminación del código generado previamente

La primera actividad que debe realizar tanto CledaI18N como CledaProp, es la eliminación del código generado previamente. Se realiza esto para evitar arrastrar código inútil o erróneo.

El procedimiento es simple: se realiza la verificación de los directorios de salida, si estos no son vacíos, se borra el directorio y se vuelve a crear. Ver figura 36

```
…4.
if(directoryTree.length > 0) {
    log.info(MessageFormat.format( //
        "deleting target directory {0}", source.getAbsolutePath()));
    CledaFileUtils.deleteDirectory(target);
    target.mkdir();
}
```

Figura 36 Porción de código donde se elimina el código generado anteriormente

Esto se realiza tanto en la generación de las clases de internacionalización como en la generación de los descriptores de propiedades de los JavaBeans.

4.6 Verificación de los directorios

El Scanner realiza los procesos de verificación de directorios, verificación de archivos, e invocación de los generadores de código.

Los directorios son verificados para evitar ingresar en directorios que no forman parte del proceso tanto de internacionalización como de descripción de propiedades de JavaBeans. Por ejemplo, no se debe revisar los directorios ocultos, o de control de versiones. Esto se realiza usando una expresión regular, que permite la validación del nombre del directorio, mediante el emparejamiento de este con el patrón que describe la expresión regular.

La clase *RegExpFileIgnorer.java* contiene las funciones y los procedimientos necesarios para esta validación. Estas funciones se pueden visualizar en la siguiente figura:

```
...
public boolean ignore (File file) {
  if (!loadAttempted) {
    load();
  }
  if(patternList == null) {
    return false;
  }

  for (Pattern pattern : patternList) {
    Matcher matcher = pattern.matcher(file.getName());
    if(matcher.matches()) {
      log.debug(MessageFormat.format( "Ignoring {0} with pattern {1}",
        file.getAbsoluteFile(), pattern.toString()));
      return true;
    }
  }
  return false;
}
//---------------------------------------------------------------------------
private void load() {
  try {
    failsafeLoad();
  } cath (IOexception) {
    throw new RuntimeException;
  }
}
//---------------------------------------------------------------------------
```

```
private void failsafeLoad() throws IOException{
  loadAttempted = true;
  File ignoreFile = new File(ignoreFileName);

  if(!ignoreFile.exists()){
    log.warn(MessageFormat.format("I18N ignore file {0} not found",
      ignoreFileName));
    return;
  }


  patternList = new ArrayList<Pattern> ();
  BufferedReader rd = new BufferedReader(new FileReader(ignoreFile));
  String line = null;

  while((line = rd.readLine()) != null){
    log.debug(MessageFormat.format("Loading pattern {0}", line));
    patternList.add(Pattern.compile(line));
  }
}
```

Figura 37 Funciones de la clase *RegExpFileIgnorer.java* que validan por medio de expresiones regulares si se ignora o no un directorio

4.7 Verificación de los archivos de claves para la internacionalización

Se verifica que los nombres de los archivos cumplan con un patrón establecido por medio de expresiones regulares. Esto es para indicar que un archivo contiene las claves de internacionalización de cadenas, formatos de fecha y número.

La expresión regular usada para validar el nombre de los archivos, se define de la siguiente manera:

```
(I18N|Date|Numb)([a-zA-Z0-9]+)(?:_([a-z]{2})(?:_([A-Z]{2}))?)?.properties
```

De modo que se validen tanto los archivos de cadenas de texto (*i18n*) como los de formatos de fechas (*date*) y formatos de números (*numb*), que puedan o no contener el identificador del idioma (en caso de que no tenga identificador de idioma, se considera el caso por defecto –*default*-) y que puedan o no contener el país (por ejemplo, está el caso de las variaciones entre el francés de Francia y el francés de Canadá).

La clase *IsI18NProcessable.java* contiene las funciones y los procedimientos necesarios para esta validación. Ver figura 38.

```
…
@Override
public boolean isProcessable(File file, List<String> path) {
  fileType = null;
  baseName = null;

  if(matchPattern(file.getName())){
    return true;
  }
  return false;
}
//--------------------------------------------------------------------------


private void matchPatter(String s){
  Matcher matcher = pattern.matcher(s);
  matcher.find();
  if(!matcher.matches()){
    return false;
  }
  fileType = matcher.group(1);
  baseName = matcher.group(2);
  return true;
}
```

Figura 38 Funciones de la clase *IsI18NProcessable.java* que validan por medio de expresiones regulares si se ignora o no un archivo

4.8 Identificación de los JavaBeans

Los JavaBeans presentes en CLEDA, deben llevar una anotación que los identifique. Esta anotación es colocada por el desarrollador involucrado en el proyecto y permite generar el descriptor de propiedades respectivo automáticamente. Esta anotación está definida en CledaProp: *@GeneratePropertyDescriptor*.

Si una clase Java contiene la anotación *@GeneratePropertyDescriptor*, indica que esta clase es un JavaBean al que se le debe generar un descriptor de propiedades.

La búsqueda e identificación de esta anotación se realiza haciendo uso de la *Java Compiler API*.

Primero se verifica que el archivo evaluado sea de extensión Java, y luego busca la etiqueta *@GeneratePropertyDescriptor* en este archivo. Si la etiqueta existe, se procede a revisar los métodos del JavaBean, para asegurar que contenga los métodos *set* y *get* de cada propiedad, y así poder generar los descriptores. En la clase *BeanIsProcessable.java* está la función *isProcessable*, la cual revisa que el archivo sea de extensión *.java*, si esto es correcto, se invoca el método *visitAnnotationFile*. Este método *visitAnnotationFile* instancia un *AnnotationDetectorProcessor*, si será éste el que se encargue de detectar si el archivo contiene la anotación requerida. Ver figura 39.

```
…
private boolean visitAnnotation(File file, List<String> path) //
      throws Exception {

  JavaCompiler javaCompiler = ToolProvider.getSystemJavaCompiler();

  StandardJavaFileManager standardJavaFileManager = //
    javaCompiler.getStandardFileManager(null, null, null);

  List<File> fileList = new ArrayList<File>();

  fileList.add(file);

  Iterable<? extends JavaFileObject> compilationUnits = //
    standardJavaFileManager.getJavaFileObjectsFromFiles(fileList);

  CompilationTask task = javaCompiler.getTask( //
    null, //
. standardJavaFileManager, //
    new DiagnosticCollector<>(), //
    null, null, compilationUnits);

  List<AbstractProcessor> processorList = //
    new ArrayList <AbstractProcessor>();

  AnnotationDetectorProcessor annotationDetectorProcessor = //
    new AnnotationDetectorProcessor();
  processorList.add(annotationDetectorProcessor);

  task.setProcessors(processorList);

  task.call();
  if (annotationDetectorProcessor.isProcessable()) {
    log.info(MessageFormat.format( //
```

```
      "{0} is processable", file.getAbsolutePath()));
  }
  return annotationDetectorProcessor.isProcessable();
}
```

Figura 39 Función *visitAnnotation* de la clase *BeanIsProcessable.java*, que revisa que una clase java contenga la anotación predefinida para los JavaBeans *@GeneratePropertyDescriptor*

La clase *AnnotationDetectorProcessor* implementa la interfaz *AbstractProcessor* de la *Java Compiler API*. Esta clase sobrescribe la función *process* para que procese un conjunto de anotaciones y las compare con la anotación objetivo *(@GeneratePropertyDescriptor)*. Si encuentra la anotación, el valor de retorno es verdadero, de lo contrario, es falso. Esto se visualiza en la figura 40.

```
…
@Override
public boolean process( //
    Set<? extends TypeElement> annotations, RoundEnvironment roundEnv) {

  for (Element rootElement : roundEnv.getRootElements()) {
    if (rootElement.getKind() != ElementKind.CLASS) {
      continue;
    }
    for (AnnotationMirror aMirror : rootElement.getAnnotationMirrors()) {
      log.debug(MessageFormat.format("checking annotation {0}",
          aMirror.getAnnotationType().toString()));
      if (TARGET_ANNOTATION.equals( //
          aMirror.getAnnotationType().toString())) {
        log.debug(MessageFormat.format( //
            "found annotation {0}", TARGET_ANNOTATION));
        processable = true;
        break;
      }
    }
  }
  return false;
}
```

Figura 40 Función *process*, que pertenece a la clase *AbstractProcessor* de la *Java Compiler API*, en el cual se procesan las anotaciones

4.9 Localización de los métodos *get* y *set* de las propiedades de los JavaBeans

Java Compiler API provee toda la funcionalidad para identificar los métodos (*set* y *get*) y asociarlos a las propiedades correspondientes de un JavaBean. La clase *BeanVisitor.java* implementa el método *failsafeVisit*, que instancia un compilador de la *Java Compiler API* para la revisión del archivo en búsqueda de los métodos *get* y *set* de cada una de las propiedades. También instancia un objeto de la clase *GetterSetterProcessorJavax.java* para el procesamiento de los métodos que se encuentren.

La clase *GetterSetterProcessorJavax.java* define las funciones para el procesamiento de los métodos: identificar si son de modificación o consulta, si el tipo de retorno del método de consulta es igual al tipo de parámetro del método modificador y a su vez igual al tipo de dato de la propiedad a que corresponden los métodos, que el sufijo de los nombres de los métodos sea igual al nombre de la propiedad.

```
…
private void failsafeVisit(File file, List<String> path) //
    throws Exception {

  JavaCompiler javaCompiler = ToolProvider.getSystemJavaCompiler();
  StandardJavaFileManager standardJavaFileManager = //
  javaCompiler.getStandardFileManager(null, null, null);

  List<File> fileList = new ArrayList<File>();
  fileList.add(file);


  Iterable<? extends JavaFileObject> compilationUnits = //
    standardJavaFileManager.getJavaFileObjectsFromFiles(fileList);


  CompilationTask task = javaCompiler.getTask( //
      null, //
      standardJavaFileManager, //
      new DiagnosticCollector<JavaFileObject>(), //
      null, null, compilationUnits);

  List<AbstractProcessor> pList = new ArrayList<AbstractProcessor>();

  GetterSetterProcessorJavax getterSetterProcessorJavax = //
  new GetterSetterProcessorJavax();
  pList.add(getterSetterProcessorJavax);


  task.setProcessors(pList);
  task.call();
```

```
  String outputDirectory = output.getPath() + SystemUtils.FILE_SEPARATOR;
  String packg = Utils.pathListToPackName(path);


  BasePropMain app = new BasePropMain();
  app.prop( //
      getterSetterProcessorJavax.getPropertyBeanList(), //
      file.getName().replace(".java", ""), //
      outputDirectory, //
      packg);
}
```

Figura 41 Método *failsafeVisit* de la clase *BeanVisitor.java*

4.10 Generador de código

Se define dos generadores de código. Uno es para la generación de las clases de internacionalización de cadenas, formatos de fecha y números, y el otro es para la generación de los descriptores de propiedades de los JavaBeans.

4.10.1 Generador de código de internacionalización – CledaI18N

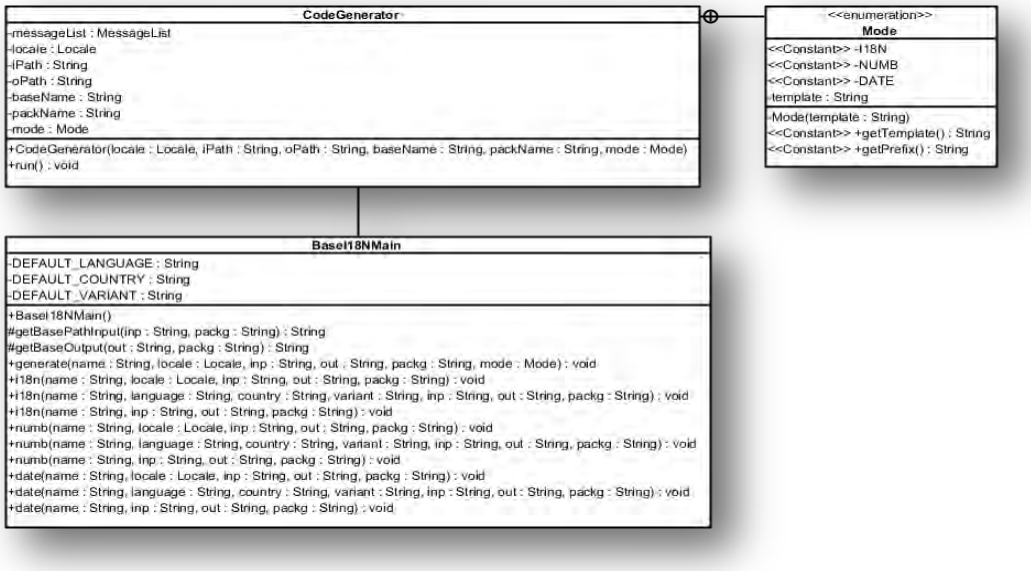

Figura 42 Diagrama de clases que intervienen en la generación de código de i18n

Tabla 11Métodos de la figura 42

Método	Descripción
Mode	
Mode(template : String)	Asigna al atributo *template* de la interfaz *Mode* una plantilla. Estas plantillas están predefinidas, y son tres. Una para cada tipo de archivo (cadenas internacionalizadas, formatos de fechas y formatos de números).
getTemplate()	Retorna la plantilla que fue asignada al generador de código.
getPrefix()	Obtiene el prefijo que ayuda a determinar cuál plantilla se debe usar. (*I18N* en caso de cadenas de texto, *Numb* en caso de formatos de números y *Date* en caso de formatos de fechas).
CodeGenerator	
CodeGenerator(locale : Locale, iPath : String, oPath : String, baseName : String, packName : String, mode : Mode)	Inicializa un generador de código con los parámetros que recibe. Es el constructor paramétrico del *CodeGenerator*.
run()	Este método es el que ejecuta las instrucciones más relevantes al momento de la generación de las clases de internacionalización. Carga todos los archivos *.properties*, comprueba que las claves sean consistentes, crea el modelo de la plantilla y finalmente la escribe.
BaseI18NMain	
getBasePathInput(inp : String, packg : String)	Construye a partir de la ruta de entrada y los nombres de los paquetes donde se encuentran los archivos de claves (*.properties*), la ruta base de estos archivos. Lo retorna en una cadena.
getBasePathOutput(out : String, packg : String)	Construye a partir de la ruta de salida y los nombres de los paquetes donde se encuentran los archivos de claves (*.properties*), la ruta base de los archivos de salida. Lo

	retorna en una cadena.
generate(name : String, locale : Locale, inp : String, out : String, packg : String, mode : Mode)	Instancia un generador de código (*CodeGenerator*) y llama al método *run()* del *CodeGenerator*.
i18n(name : String, locale : Locale, inp : String, out : String, packg : String)	Invoca al método *generate*() pasándole los mismo parámetros que recibe, y adicionalmente el *Mode.I18N*
i18n(name : String, language : String, country : String, variant : String, inp : String, out : String, packg : String)	Método que recibe el nombre del archivo, idioma, país y variante, ruta de entrada y salida, y el nombre de paquete; e invoca al método *i18n* que recibe como parámetro la varible *locale*. Construye esta variable *locale* a partir de los parámetros idioma, país y variante, por medio del método *createLocale* de la clase *CledaLocaleUtils*.
i18n(name : String, inp : String, out : String, packg : String)	Método que recibe el nombre del archivo, la ruta de entrada y de salida, y el nombre de paquete; e invoca al método *i18n* que adicionalmente recibe los parámetros de idioma, país y variante.
numb(name : String, locale : Locale, inp : String, out : String, packg : String)	Invoca al método *generate*() pasándole los mismo parámetros que recibe, y adicionalmente el *Mode.NUMB*
numb(name : String, language : String, country : String, variant : String, inp : String, out : String, packg : String)	Método que recibe el nombre del archivo, idioma, país y variante, ruta de entrada y salida, y el nombre de paquete; e invoca al método *numb* que recibe como parámetro la varible *locale*. Construye esta variable *locale* a partir de los parámetros idioma, país y variante, por medio del método *createLocale* de la clase *CledaLocaleUtils*.
numb(name : String, inp : String, out : String, packg : String)	Método que recibe el nombre del archivo, la ruta de entrada y de salida, y el nombre de paquete; e invoca al método *numb* que adicionalmente recibe los parámetros de idioma, país y variante.
date(name : String, locale : Locale, inp : String, out : String, packg : String)	Invoca al método *generate*() pasándole los mismo parámetros que recibe, y adicionalmente el *Mode.DATE*
date(name : String, language : String, country	Método que recibe el nombre del archivo, idioma, país y

: String, variant : String, inp : String, out : String, packg : String)	variante, ruta de entrada y salida, y el nombre de paquete; e invoca al método *date* que recibe como parámetro la varible *locale*. Construye esta variable *locale* a partir de los parámetros idioma, país y variante, por medio del método *createLocale* de la clase *CledaLocaleUtils*.
date(name : String, inp : String, out : String, packg : String)	Método que recibe el nombre del archivo, la ruta de entrada y de salida, y el nombre de paquete; e invoca al método *date* que adicionalmente recibe los parámetros de idioma, país y variante.

El generador de código de CledaI18N, según el modo de internacionalización (ya sea cadenas de texto, formatos de fecha o formatos de número) indica el modelo de plantilla que se utiliza. Hay tres plantillas: *class-i18n.ftl*, *class-numb.ftl* y *class-date.ftl* que corresponden a cadenas de texto, formatos de números y formatos de fecha respectivamente. El generador carga los archivos de propiedades (*.properties*) donde están contenidas las claves, rectifica que las claves sean consistentes, luego crea el modelo de la plantilla que contendrá los datos de la clase de internacionalización (nombre de la clase, nombre del paquete, lista de los métodos de acceso a las cadenas internacionalizadas, prefijo del archivo de propiedades, idioma, país y variante en caso de que lo tenga), y finalmente escribe esta plantilla en un archivo java, que será finalmente la clase de internacionalización para un idioma en particular.

```
…
public void run() throws Exception {
  File root = new File(iPath);

  String fileName = mode.getPrefix() + baseName;

  log.info("Properties base name: " + baseName);
  log.info("Properties file name: " + fileName);

  String[] list = (String[]) ArrayUtils.addAll( //
      root.list(new WildcardFileFilter(fileName + /**/"_*.properties")), /
      root.list(new WildcardFileFilter(fileName + /*  */".properties")));

  if (list == null) {
    log.error("Nothing found for: " + baseName);
    return;
  }
```

```
for (int i = 0; i < list.length; i++) {
  String name = iPath + SystemUtils.FILE_SEPARATOR + list[i];

  log.info("Loading: " + list[i] + " - " + name);

  messageList.load(new FileInputStream(name));
}

if (!messageList.test()) {
  throw new Exception("!messageList.test()");
}

ObjectBean objectBean = new ObjectBean();

objectBean.setLanguage(locale.getLanguage());
objectBean.setCountry(locale.getCountry());
objectBean.setVariant(locale.getVariant());

objectBean.setResName(/*  */fileName);
objectBean.setClsName("_" + fileName);
objectBean.setPckName(packName);
objectBean.setMethodBeanList(messageList.getKeyList());

String aux = oPath.replace("/", SystemUtils.FILE_SEPARATOR);
File folder = new File(aux);

folder.mkdirs();

FileWriter fWriter = new FileWriter(oPath + SystemUtils.FILE_SEPARATOR +
    objectBean.getClsName() + ".java");

  FreeMarkerUtil.getInstance().process(mode.getTemplate(),
    objectBean, fWriter);
  fWriter.close();
}
```

Figura 43 Método *run* del generador de código de i18n, donde se ejecutan las sentencias involucradas en la generación de las clases de internacionalización

4.10.2 Generador de código de los descriptores de propiedades de los JavaBeans - CledaProp

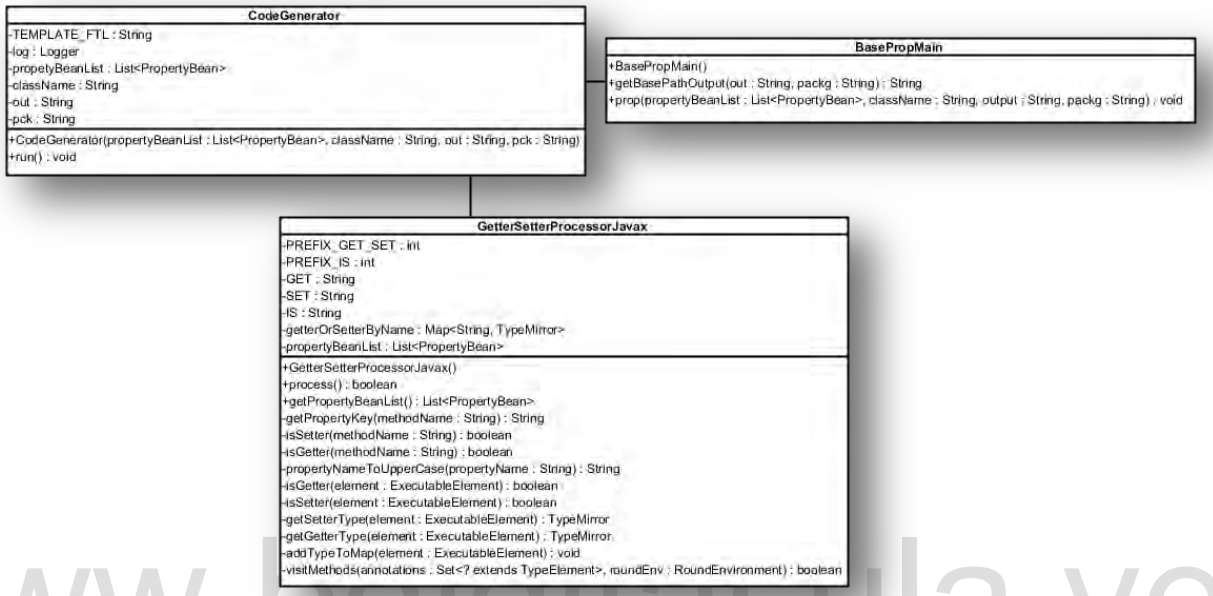

Figura 44 Diagrama de clases que intervienen en la generación de los descriptores de propiedades de JavaBeans

Tabla 12 Métodos de la figura 44

Método	Descripción
GetterSetterProcessorJavax	
GetterSetterProcessorJavax()	Constructor vacío por defecto de la clase.
process()	Método que pertenece a la interfaz *AbstractProcessor* y que debe sobrescribirse. Llama a la función *visitMethods()*
getPropertyBeanList()	Retorna la lista de propiedades del JavaBean, a los que posteriormente se les generará el descriptor.
getPropertyKey(methodName : String)	Retorna el nombre de la propiedad a la que corresponde el nombre de método que recibe por parámetro.

isSetter(methodName : String)	Si el método que recibe por parámetro corresponde a un *Setter*, devuelve verdadero, del contrario, devuelve falso.
isGetter(methodName : String)	Si el método que recibe por parámetro corresponde a un *Getter*, devuelve verdadero, del contrario, devuelve falso.
propertyNameToUpperCase(propertyName : String)	Convierte el nombre de la propiedad a mayúsculas.
visitMethods(annotations : Set<? Extends TypeElement>, roundEnv : RoundEnvironment)	Revisa los métodos que tiene el JavaBean. Por cada nombre de propiedad, si contiene un *Setter* y un *Getter*, se agrega a la lista de propiedades del JavaBean que se les generará el descriptor.
CodeGenerator	
CodeGenerator(propertyBeanLis : List<PropertyBean>, className : String, out : String, pck : String)	Inicializa un generador de código con los parámetros que recibe. Es el constructor paramétrico del *CodeGenerator*.
run()	Este método es el que ejecuta las instrucciones más relevantes al momento de la generación de los descriptores de propiedades. Construye un modelo de plantilla y la escribe con las propiedades que están en la lista *propertyBeanList*.
BasePropMain	
BasePropMain()	Constructor por defecto vacío de la clase.
getBasePathOutput(out : String, packg : String)	Construye a partir de la ruta de salida y el nombre de paquete donde se encuentran el JavaBean, la ruta base de los archivos de salida. Lo retorna en una cadena.
prop()	Instancia un generador de código (*CodeGenerator*) y llama al método *run()* del *CodeGenerator*.

El generador de código del descriptor de propiedades de JavaBeans, solo indica un tipo de plantilla para los descriptores de propiedades: *class-prop.ftl.* Este crea el modelo de la plantilla que

contendrá los datos del descriptor de propiedades (la lista de las propiedades del JavaBean, el nombre del JavaBean, el nombre del paquete), luego escribe esta plantilla en un archivo java, que será finalmente el descriptor de propiedades de un JavaBean en particular.

```
…
public void run() throws Exception {
  log.info(MessageFormat.format( //
      "Generating property descriptor for {0}", className));

  ObjectBean objectBean = new ObjectBean();

  objectBean.setClsName("_Prop" + className);
  objectBean.setPckName(pck);
  objectBean.setPropertyBeanList(propertyBeanList);

  String aux = out.replace("/", SystemUtils.FILE_SEPARATOR);
  File folder = new File(aux);

  folder.mkdirs();

  FileWriter fWriter = new FileWriter( out + SystemUtils.FILE_SEPARATOR +
     objectBean.getClsName() + ".java");

  FreeMarkerUtil.getInstance().process(TEMPLATE_FTL, objectBean, fWriter);
  fWriter.close();
}
```

Figura 45 Método *run* del generador de código de los descriptores de propiedades de JavaBeans

4.11 Ejecución del *plugin* de Maven

Para la ejecución de los plugins de Maven, se deben conocer los descriptores del proyecto a ejecutar[28]. Se debe especificar en la consola el siguiente comando:

```
mvn groupId:artifactId:version:goal
```

donde *groupId*, *artifactId* y *version* son los descriptores especificados en el archivo de configuración *pom.xml*, y *goal* es el objetivo de la ejecución, que indica que *Mojo* se va a ejecutar.

[28] Ver Apéndice B. Tabla 20. Pág. 105

Los descriptores de CledaI18N y CledaProp son los siguientes:

Descriptor	CledaI18N	CledaProp
groupId	com.minotauro	com.minotauro
artifactId	MinotauroI18NMaven	MinotauroBeanMaven
version	0.0.1-SNAPSHOT	0.0.1-SNAPSHOT

El *goal* para CledaI18N es: i18n, y el *goal* para CledaProp es: prop, de manera que los comandos para ejecutar CledaI18N y CledaProp son:

```
mvn com.minotauro:MinotauroI18NMaven:0.0.1-SNAPSHOT:i18n
mvn com.minotauro:MinotauroBeanNMaven:0.0.1-SNAPSHOT:prop
```

respectivamente.

Capítulo 5

Pruebas

5.1 Pruebas unitarias

Las pruebas unitarias permiten comprobar el buen funcionamiento de un módulo de código. Esto permite asegurar que cada componente que conforma un determinado software, funcione correctamente por separado.

Las pruebas consisten en ejecutar los métodos con una entrada conocida para obtener una salida esperada. Se hacen pruebas con entradas correctas e incorrectas, de modo que con las primeras, el método debe pasar, y con las segundas, debe fallar

Se hicieron pruebas para los módulos que conforman el Scanner, para asegurar que el recorrido de directorios y validación de archivos se hace de manera correcta. Se muestran los resultados en la siguiente tabla:

Tabla 13Resultados de las pruebas unitarias de CledaI18N y CledaProp

CledaI18N			
Método a probar	**Entrada**	**Salida esperada**	**Salida obtenida**
Recorrido de un directorio	demo	true	true
	.hg	false	false
Revisión de un archivo	I18NApp_en.properties	true	true
	I18NApp_en.java	false	false
	foo.properties	false	false

	foo_en.properties	false	false
	I18NApp_en.txt	false	false
CledaProp			
Método a probar	**Entrada**	**Salida esperada**	**Salida obtenida**
Recorrido de un directorio	demo	true	true
	.hg	false	false
Revisión de un archivo	FooBean.java	true	true
	FooBean.txt	false	false
	FooBean.cpp	false	false
	FooBean.properties	false	false
	TestBean.java	true	true

5.2 Pruebas cajas negras

Las pruebas cajas negras, sirven para detectar fallos en la entrada o salida del sistema. Estas se hacen variando sus parámetros y analizando sus salidas.

Las pruebas funcionales se realizaron bajos los siguientes entornos de software:

- Sistema Operativo Linux – Distribución Ubuntu 12.10
- JDK (Java Development Kit) 1.7.0_17
- JRE (Java Runtime Environment) 1.7.0_17
- Apache Maven 3.0.5

5.2.1 Generación de las clases de internacionalización

Se crearon varios archivos de propiedades con claves en distintos idiomas, para la generación de clases de internacionalización. Los formatos de fechas utilizados se encuentran en las siguientes tablas:

Tabla 14 Formatos de fecha y hora en Ingles para Estados Unidos y el Reino Unido (Gran Bretaña)

Formato	_en_US	_en_GB
dateOnly	MM/dd/yyyy	dd/MM/yyyy
dateLong	EEEE, MMMM d, yyyy	d MMMM yyyy
dateTime	MM/dd/yyyy, HH.mm.ss	dd/MM/yyyy, HH:mm:ss
timeOnly	HH.mm.ss	HH:mm:ss
monthYear	MM/yyyy	MM/yyyy
monthOnly	MMMM	MMMM

Tabla 15 Formatos de fecha y hora en Español, Alemán y Frances para Francia.

Formato	_es	_de	_fr_FR
dateOnly	dd-MM-yyyy	dd.MM.yyyy	dd/MM/yyyy
dateLong	EEEE, d, MMMM, yyyy	EEEE, d. MMMM yyyy	EEEE d MMMM yyyy
dateTime	dd-MM-yyyy, HH:mm:ss	dd.MM.yyyy, HH:mm:ss	dd/MM/yyyy, HH:mm:ss
timeOnly	HH:mm:ss	HH:mm:ss	HH:mm:ss
monthYear	MM-yyyy	MM.yyyy	MM/yyyy
monthOnly	MMMM	MMMM	MMMM

Donde los parámetros son:

MM: meses en dos dígitos (01 al 12).

dd: días en dos dígitos (01 al 31).

yyyy: años en cuatro dígitos (2014, 2015).

EEEE: nombre de los días en calendario gregoriano (Lunes, Monday, lundi, Montag).

HH: horas en dos dígitos (00 a 23, ó 01 a 12).

mm: minutos en dos dígitos (00 a 59).

ss: segundos en dos dígitos (00 a 59).

MMMM: nombre de los meses en calendario gregoriano (Enero, January, janvier,Januar).

Los formatos numéricos utilizados se encuentran en las siguientes tablas:

Tabla 16 Formatos numéricos en inglés, para Estados Unidos y el Reino Unido (Gran Bretaña)

Formato	_en_US	_en_GB
defaultNumberFormat	#.#	#.#
default2NumberFormat	###,###.00	###,###.00
default4NumberFormat	###,###.0000	###,###.0000

Tabla 17 Formatos numéricos en español, alemán y francés para Francia

Formato	_es	_de	_fr_FR
defaultNumberFormat	#,#	#,#	#,#
default2NumberFormat	###.###,00	###.###,00	###.###,00
default4NumberFormat	###.###,0000	###.###,0000	###.###,0000

Las claves de cadenas de texto utilizadas, se encuentran en la siguiente tabla:

Tabla 18 Claves en inglés (para US y GB), español, alemán y francés (FR).

Clave	_en_US	_en_GB	_es	_de	_fr_FR
title	title	title	titulo	titel	titre
date	date	date	fecha	datum	date
technology	technology	technology	tecnología	technologie	technologie
collection	collection	collection	colección	sammlung	collection
tools	tools	tools	herramientas	tools	outils
machinery	machinery	machinery	maquinaria	maschinen	machinerie
modifications	modifications	modifications	modificaciones	modifikationen	modifications
arrangements	arrangements	arrangements	arreglos	vereinbarungen	arrangements
procedures	procedures	procedures	procedimientos	verfahren	procédures
uses	uses	uses	usos	anwendungen	utilisations
humans	humans	humans	humanos	menschen	humains
engineering	engineering	engineering	ingeniería	maschinenbau	ingénierie
discipline	discipline	discipline	disciplina	disziplin	discipline
seek	seek	seek	buscar	suchen	chercher

study	study	study	estudio	studie	étude
new	new	new	nuevo	neu	nouveau
animal	animal	animal	animal	tier	animal
specie	specie	specie	especie	hartgeld	espèces
ability	ability	ability	habilidad	fähigkeit	capacité
control	control	control	control	steuerung	contrôle
adapt	adapt	adapt	adaptar	anpassen	adapter
natural	natural	natural	natural	natürliche	natural
environments	environments	environments	ambientes	umgebungen	environnements
term	term	term	plazo	begriff	terme
areas	áreas	areas	áreas	bereiche	domaines
examples	examples	examples	ejemplos	beispiele	exemples
include	include	include	incluir	enthalten	inclure
construction	construction	construction	construcción	bau	construction
information	information	information	información	informationen	information
comunication	comunication	comunication	comunicación	komunikation	comunication

Las clases resultantes del proceso de internacionalización son:

```
// ---------------------------------------
// Generated code, do not edit
// ---------------------------------------
package i18n.tesis;

import java.util.Locale;

import com.minotauro.i18n.base.MessageBase;
import com.minotauro.i18n.base.MessageException;

public class _I18NI18NFormat {

 public static final Locale locale = new Locale("en", "", "");

 public static final String RES_NAME = "i18n.tesis.I18NFormat";
```

```
 private _I18NI18NFormat() {
    // Empty
 }

 public static String ability() throws MessageException {
  return MessageBase.getInstance().locateValue(locale, RES_NAME, "ability", new
     Object[]{});
 }

 public static String adapt() throws MessageException {
  return MessageBase.getInstance().locateValue(locale, RES_NAME, "adapt", new
     Object[]{});
 }
…
}
```

Figura 46 Clase de i18n generada para la internacionalización de cadenas de texto. Solo se incluyeron dos métodos para obtener las cadenas traducidas. *_I18NI18NFormat.java*


```
// ----------------------------------------
// Generated code, do not edit
// ----------------------------------------
package i18n.tesis;

import java.text.DecimalFormat;
import java.text.ParseException;
import java.util.Locale;

import com.minotauro.i18n.base.MessageBase;
import com.minotauro.i18n.base.MessageException;

public class _NumbNumbFormat {

  public static final Locale locale = new Locale("en", "", "");

  public static final String RES_NAME = "i18n.tesis.NumbFormat";

  private _NumbNumbFormat() {
    // Empty
  }
```

```
public static DecimalFormat getDefaultNumberFormatFormatter() {
    return MessageBase.getInstance().
        getNumberFormatter(locale, RES_NAME, "defaultNumberFormat");
  }

  public static String formatDefaultNumberFormat(double val) throws
    MessageException {
       return MessageBase.getInstance().
           formatNumberValue(locale, RES_NAME, "defaultNumberFormat", val);
  }

  public static double parseDefaultNumberFormat(String val) throws
    MessageException, ParseException {
      return MessageBase.getInstance().
          parseNumberValue(locale, RES_NAME, "defaultNumberFormat", val);
  }
}
```

Figura 47 Clase de i18n generada para la internacionalización de los formatos de números. Solo se incluyeron tres métodos para obtener los formatos localizados. *_NumbNumbFormat.java*


```
// ----------------------------------------
// Generated code, do not edit
// ----------------------------------------
package i18n.tesis;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.Locale;

import com.minotauro.i18n.base.MessageBase;
import com.minotauro.i18n.base.MessageException;

public class _DateDateFormat {

  public static final Locale locale = new Locale("en", "", "");

  public static final String RES_NAME = "i18n.tesis.DateFormat";

  private _DateDateFormat() {
    // Empty
```

```
  }

  public static SimpleDateFormat getDateLongFormatter() {
    return MessageBase.getInstance().getDateFormatter(locale, RES_NAME,
          "dateLong");
  }

  public static String formatDateLong(Calendar val) throws MessageException {
    return MessageBase.getInstance().formatCalendarValue(locale, RES_NAME,
          "dateLong", val);
  }

  public static Calendar parseDateLong(String val) throws MessageException,
     ParseException {
           return MessageBase.getInstance().parseCalendarValue(locale, RES_NAME,
                 "dateLong", val);
     }
}
```

Figura 48 Clase de i18n generada para la internacionalización de los formatos de fechas. Solo se incluyeron tres métodos para obtener las formatos localizados. *_DateDateFormat.java*

Para la generación de las clases de internacionalización de cadenas, formatos de fecha y formatos de número, la salida fue la deseada.

Las clases para la internacionalización de cadenas contienen los métodos estáticos para acceder a cada una de las claves de los archivos .properties.

Las clases para la internacionalización de los formatos de fecha contienen los métodos estáticos para convertir los distintos formatos de fecha.

Las clases para la internacionalización de los formatos numéricos contienen los métodos estáticos para convertir los distintos formatos numéricos.

5.2.2 Generación de los descriptores de propiedades de los JavaBeans

A continuación se presenta un segmento de código de uno de los JavaBeans descritos en CLEDA.

```
…
@MappedSuperclass
@GeneratePropertyDescriptor
public class MBase implements Lifecycle {

  protected int id;
  protected boolean systEntry;
  protected boolean busyEntry;

  public MBase() {
    // Empty
  }


  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  public int getId() {
    return id;
  }


  public void setId(int id) {
    this.id = id;
  }


  public boolean getSystEntry() {
    return systEntry;
  }


  public void setSystEntry(boolean systEntry) {
    this.systEntry = systEntry;
  }
…
}
```

Figura 49 Segmento de código del JavaBean *MBase.java* de CLEDA. Se aprecian las propiedades y los métodos de acceso a estas.

```
/*
 * Generated code, do not edit
 */
package com.minotauro.cleda.model;

public class _PropMBase {

  private _PropMBase() {
```

```
    // Empty
  }

  public static final String ID = "id";
  public static final String SYST_ENTRY = "systEntry";
}
```

Figura 50 Descriptor de propiedades del JavaBean *MBase.java - _PropMBase.java*

Para la generación de los descriptores de propiedades de JavaBeans, la salida fue la deseada.

El descriptor de propiedades del JavaBean contiene las constantes estáticas que describen cada propiedad presente en el JavaBean.

5.3 Ejemplo de la implementación de un formulario internacionalizado con el componente CledaI18N

A continuación se muestra un ejemplo donde un formulario básico muestra el contenido internacionalizado. Hace uso de las clases de internacionalización de cadenas, formatos de fecha y de números generados con CledaI18N.

La variable que determina el objeto *Locale* del usuario, se asigna en el panel de selección de lenguaje.

Figura 51 Panel de selección de idioma

Se carga una ventana con los componentes internacionalizados. Los títulos de las etiquetas se muestran dependiendo del *Locale*, haciendo uso de los métodos estáticos de las clases de internacionalización generadas.

```
…
lblName = new JLabel(_I18NBankingform.name());

lblAddress = new JLabel(_I18NBankingform.address());

lblTelephone = new JLabel(_I18NBankingform.telephone());

lblEmail = new JLabel(_I18NBankingform.email());

lblCardNumber = new JLabel(_I18NBankingform.cardNumber());

lblPin = new JLabel(_I18NBankingform.pin());

lblExpiryDate = new JLabel(_I18NBankingform.expiry());

lblAccountBalance = new JLabel(_I18NBankingform.accountBalance());

lblCardBalance = new JLabel(_I18NBankingform.cardBalance());

lblDate = new JLabel(_DateBankingform.formatDateLong(calendar));

tfExpiryDate = new JTextField(_DateBankingform.formatDateOnly(calExpiry));

tfAccountBalance =
    new JTextField(_NumbBankingform.formatDefault2NumberFormat(1000000.00));
tfCardBalance =
    new JTextField(_NumbBankingform.formatDefault2NumberFormat(2400.00));
…
```

Figura 52 Uso de los métodos estáticos que acceden a las cadenas internacionalizadas

Las salidas para los distintos idiomas implementados, son:

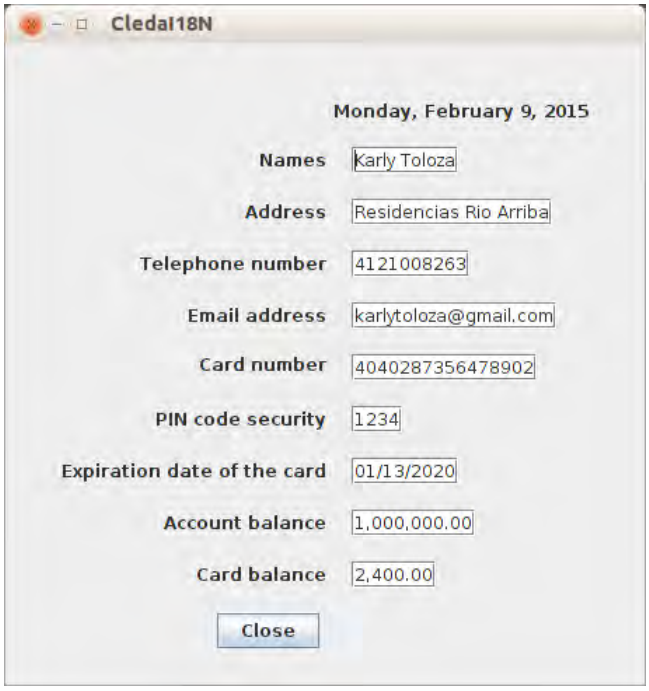

Figura 53 Formulario con los componentes localizados en lenguaje Inglés

Figura 54 Formulario con los componentes localizados en lenguaje Alemán

Figura 55 Formulario con los componentes localizados en lenguaje Español

Figura 56 Formulario con los componenten localizados en lenguaje Francés

Capítulo 6

Conclusiones y Recomendaciones

6.1 Conclusiones

Se consiguió mejorar la arquitectura de CledaI18N, separando los componentes de internacionalización y descripción de propiedades en dos módulos, diferentes uno del otro (CledaI18N y CledaProp). Se automatizó el proceso de generación de código, partiendo desde la automatización de la revisión de directorios y archivos hasta el proceso de generación de las clases, tanto de las clases de internacionalización como de los descriptores de propiedades de JavaBeans.

Se solidificó la idea de hacer internacionalización eliminando por completo la presencia de cadenas de texto (claves) en el código fuente, reemplazando las claves por métodos estáticos que manipulan las cadenas localizadas. De igual manera, se estableció un precedente para usar esta técnica de internacionalización en otros lenguajes de programación, específicamente en lenguajes estáticamente tipados, por ejemplo C y C++.

Dentro del proceso de generación de descriptores de propiedades de JavaBeans, se consiguió construir estos descriptores haciendo uso de un procesador de anotaciones y de métodos de clases, implementados con la interfaz *Java Compiler API*.

Se creó un *plugin* en Maven que permite usar las herramientas de internacionalización y descripción de propiedades CledaI18N y CledaProp.

Uno de los objetivos de este proyecto era el desarrollar un *plugin* para Eclipse que permitiera usar las herramientas de internacionalización y descripción de propiedades de JavaBeans, sin embargo el tiempo estipulado para el desarrollo del proyecto no fue suficiente para la culminación de este objetivo.

Finalmente, como se observó que el Componente de Internacionalización y Descripción de Propiedades del *Framework* CLEDA, puede brindar soporte a aplicaciones externas a CLEDA, se decidió distribuir éste componente como una herramienta independiente llamado MINOTAURO, de forma que el Framework CLEDA hace uso de MINOATURO.

6.2 Recomendaciones

Como no se alcanzó el objetivo de crear un *plugin* para Eclipse, se recomienda la realización de este, de modo que se pueda incorporar la generación de código al *workflow* de compilación de Eclipse.

El método que usa CledaI18N para la internacionalización puede ser aplicado en otros lenguajes de programación estáticamente tipados. Se recomienda la estandarización de éste, de modo que se pueda usar para internacionalización en otros lenguajes como C++.

Bibliografía

Alexander, C. Ishikawa, S., & Silverstein, M.(1997). A Patter Language. (1ra ed.). Oxford University Press.

Apache (2014a). Apache Maven Project. Consultado el 4 de Abril, 2014, desde: http://maven.apache.org/what-is-maven.html

Apache (2014b). Introduction to Maven 2.0 Plugin Development. Consultado el 4 de Abril, 2014, desde: http://maven.apache.org/guides/introduction/introduction-to-plugins.html

Arnoldus, J., Van den Brand, M., Serebrenik, A., & Brunekreef, J. (2012). Code Generation with Templates (1ra ed.). París, Francia: Atlantis Press.

Darwin, I. (2001). Java Cookbook (2da ed.). Sebastopol, CA: O'Reilly Media.

Fowler, M., (2004). UML Distilled: A Brief Guide to the Standard Object Modeling Language (3rd ed). Boston, MA: Pearson Education, Inc.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J.(1995). Design Patterns: Elements of Reusable Object-Oriented Software (1ra ed.). Indianapolis, IN: Addison-Wesley

GNU Operating System (2014, Diciembre 25). Gettext. Consultado el 12 de Enero, 2015, desde: https://www.gnu.org/software/gettext/

Gutiérrez, G., Salas, A., y Perez, A. (2007). Los cledas: Una nueva arquitectura para optimizar el desarrollo de aplicaciones web. 6ta. Conferencia Iberoamericana en Sistemas, Cibernética e Informática.

Java.net: The Source of Java Technology Collaboration (2008). Source Code Analysis Using Java 6 APIs. Consultado el 4 de Abril, 2014 desde: https://today.java.net/pub/a/today/2008/04/10/source-code-analysis-using-java-6-compiler-apis.html

Oliver, A., Moré, J. y Climent, S. (2011). Traducción y Tecnologías. Editor UOC.

ORACLE (2014a). Java SE Documentation: Internationalization Overview. Consultado el 4 de Abril, 2014, desde: http://docs.oracle.com/javase/7/docs/technotes/guides/intl/overview.html

ORACLE (2014b). Java SE Documentation: The Reflection API. Consultado el 11 de Diciembre, 2014, desde: http://docs.oracle.com/javase/tutorial/reflect/

PHP (2015). PHP Documentation. Consultado el 10 de Enero, 2015, desde: http://php.net/manual/en/function.gettext.php

Ramos, I. y Lozano, M. (2000). Ingeniería del Software y Bases de Datos: tendencias actuales. España: Ediciones de la Universidad de Castilla – La Mancha.

Sommerville, I. (2005). Ingeniería del Software (7ma ed.). Reino Unido: Pearson Addison Wesley.

Sun Microsystems (1997, Agosto 8). JavaBeansTM. Mountain View, CA: Graham Hamilton.

Viviona I. (2011). Java Users. Buenos Aires, AR: Fox Andiga.

Wikipedia. (2014a, Enero 17). Internationalization Tag Set. Consultado el 4 de Abril, 2014, desde Wikipedia, the free encyclopedia: http://en.wikipedia.org/wiki/Internationalization_Tag_Set

Wikipedia (2014b, Julio 31). JavaBean. Consultado el 10 de Agosto, 2014, desde Wikipedia, la enciclopedia libre: http://es.wikipedia.org/wiki/JavaBean

Wikipedia (2014c, Septiembre 24). Eclipse (software). Consultado el 3 de Noviembre, 2014, desde Wikipedia, la enciclopedia libre: http://es.wikipedia.org/wiki/Eclipse_(software)

Wikipedia (2014d, Octubre 24). Maven. Consultado el 12 de Septiembre, 2014, desde Wikipedia, la enciclopedia libre: http://en.wikipedia.org/wiki/Maven

Wikipedia (2015a, Enero 27). Gettext. Consultado el 27 de Enero, 2015, desde Wikipedia, la enciclopedia libre: http://es.wikipedia.org/wiki/Gettext

Wikipedia (2015b, Febrero 16). Framework. Consultado el 04 de Marzo, 2015, desde Wikipedia, la enciclopedia libre: http://es.wikipedia.org/wiki/Framework

World Wide Web Consortium – W3C (2007, Abril 27). Internationalization Tag Set (ITS) Version 1.0. Consultado el 3 de Noviembre, 2014, desde: http://www.w3.org/TR/its/

Zukowski, J., (2006). JavaTM 6 Plataform Revealed. Berkeley, CA: Apress, Inc.

Apéndice A

Plantillas usadas para la generación de código

Las plantillas que utilizan CledaI18N y CledaProp están implementadas de la siguiente manera:


```
[#ftl]
/*
 * Generated code, do not edit
 */
package ${pckName};

public class ${clsName} {

  private ${clsName}() {
    // Empty
  }

  [#list propertyBeanList as propertyBean]
  public static final String ${propertyBean.propertyName} =
      "${propertyBean.propertyValue}";
  [/#list]
}
```

Figura 57 Plantilla base para los descriptores de propiedades *class-prop.ftl*

```
[#ftl]
// ----------------------------------------
// Generated code, do not edit
// ----------------------------------------
package ${pckName};

import java.util.Locale;
import com.minotauro.i18n.base.MessageBase;
import com.minotauro.i18n.base.MessageException;

public class ${clsName} {

  public static final Locale locale = new Locale("${language}",
      "${country}", "${variant}");

  public static final String RES_NAME = "${pckName}.${resName}";

  private ${clsName}() {
    // Empty
  }
  [#list methodBeanList as methodBean]
  public static String ${methodBean.name}([@compress single_line=true]
    [#if methodBean.args > -1]
      [#list 0..methodBean.args as indx]
        Object arg${indx}[#if indx != methodBean.args], [/#if][/#list][/#if])
           throws MessageException {
    [/@compress]

    return MessageBase.getInstance().locateValue(locale, RES_NAME,
          "${methodBean.key}", new Object[]{[@compress single_line=true]
    [#if methodBean.args > -1]
       [#list 0..methodBean.args as indx]
           arg${indx}[#if indx != methodBean.args],
               [/#if][/#list][/#if]});[/@compress]
  }

  [/#list]
}
```

Figura 58 Plantilla base para las clases de internacionalización de cadenas de texto *class-i18n.ftl*

```
[#ftl]
// ----------------------------------------
// Generated code, do not edit
// ----------------------------------------
package ${pckName};

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.Locale;
import com.minotauro.i18n.base.MessageBase;
import com.minotauro.i18n.base.MessageException;

public class ${clsName} {

  public static final Locale locale = new Locale("${language}", "${country}",
"${variant}");

  public static final String RES_NAME = "${pckName}.${resName}";

  private ${clsName}() {
    // Empty
  }

  [#list methodBeanList as methodBean]
  public static SimpleDateFormat get${methodBean.capitalizedName}Formatter() {
    return MessageBase.getInstance().getDateFormatter(locale, RES_NAME,
          "${methodBean.key}");
  }
  public static String format${methodBean.capitalizedName}(Calendar val) throws
    MessageException {
        return MessageBase.getInstance().formatCalendarValue(locale, RES_NAME,
            "${methodBean.key}", val);
  }
  public static Calendar parse${methodBean.capitalizedName}(String val) throws
    MessageException, ParseException {
         return MessageBase.getInstance().parseCalendarValue(locale, RES_NAME,
             "${methodBean.key}", val);
  }
  [/#list]
}
```

Figura 59 Plantilla base para las clases de internacionalización de formatos de fecha *class-date.ftl*

```
[#ftl]
// ----------------------------------------
// Generated code, do not edit
// ----------------------------------------
package ${pckName};

import java.text.DecimalFormat;
import java.text.ParseException;
import java.util.Locale;

import com.minotauro.i18n.base.MessageBase;
import com.minotauro.i18n.base.MessageException;

public class ${clsName} {

  public static final Locale locale = new Locale("${language}", "${country}",
    "${variant}");

  public static final String RES_NAME = "${pckName}.${resName}";

  private ${clsName}() {
    // Empty
  }
  [#list methodBeanList as methodBean]
  public static DecimalFormat get${methodBean.capitalizedName}Formatter() {
    return MessageBase.getInstance().getNumberFormatter(locale, RES_NAME,
      "${methodBean.key}");
  }
  public static String format${methodBean.capitalizedName}(double val) throws
     MessageException {
         return MessageBase.getInstance().formatNumberValue(locale, RES_NAME,
            "${methodBean.key}", val);
  }
  public static double parse${methodBean.capitalizedName}(String val) throws
     MessageException, ParseException {
         return MessageBase.getInstance().parseNumberValue(locale, RES_NAME,
            "${methodBean.key}", val);
  }
  [/#list]
}
```

Figura 60 Plantilla base para las clases de internacionalización de formatos de números *class-numb.ftl*

Expresión Regular usada para la validación de los nombres de archivos de los *.properties*

```
(I18N|Date|Numb)([a-zA-Z0-9]+)(?:_([a-z]{2})(?:_([A-Z]{2}))?)?.properties
```

Anotación usada para indicar que una clase es un JavaBean de CLEDA al que se le desea generar un descriptor de propiedades


```
package com.minotauro.prop.base;

import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.SOURCE)
public @interface GeneratePropertyDescriptor {
  // Empty
}
```

Apéndice B

Componente teórico adicional

Entorno de Desarrollo Integrado Eclipse

Muchos programadores encuentran que usar un conjunto de herramientas por separado (un editor de texto, un compilador, un ejecutor, por no mencionar un depurador) es demasiado. Un entorno de desarrollo integrado (IDE – *Integrated Development Environment*) integra todas estas en un solo conjunto de herramientas con una interfaz gráfica de usuario.

Eclipse es un IDE de código abierto y multiplataforma. Dispone de un editor de texto. La compilación es en tiempo real. Tiene pruebas unitarias con *JUnit*, control de versiones con CVS, integración con *Ant*, asistentes para creación de proyectos, clases, pruebas, etc., y refactorización.

A través de *plugins* disponibles en repositorios libres, es posible añadir control de versiones con *Subversion*, *Git* o *Mercurial*. También permite la integración con Hibernate y un sinfín de herramientas adicionales disponibles en el *Marketplace* de Eclipse. (Wikipedia, 2014c).

Herramienta de desarrollo de software Maven

Maven, una palabra en Yiddish que significa "acumulador de conocimiento". Es una herramienta que puede ser usada para construir y manejar cualquier proyecto basado en lenguaje de programación Java.

Lo hace de la siguiente manera:

- Haciendo el proceso de construcción fácil.
- Proporcionando un sistema de construcción uniforme.
- Proporcionando información de calidad sobre los proyectos.
- Proporcionando directrices para las mejores prácticas de desarrollo.
- Permitiendo la migración transparente hacia nuevas funciones.

Maven tiene un modelo de configuración de construcción basado en un formato XML. Utiliza un *Project Object Model* (*POM*) para describir el proyecto de software a construir, sus dependencias, sus módulos y componentes externos, y el orden de construcción de los elementos.

Su arquitectura está basada en *plugins*. Usa cualquier aplicación controlable a través de la entrada estándar, permitiendo que cualquier desarrollador escriba *plugins* para su interfaz con herramientas como compiladores, herramientas de pruebas unitarias, entre otros. La filosofía de Maven es la estandarización de las construcciones generadas, a fin de utilizar modelos ya existentes en la producción de software. (Apache, 2014a).

Ciclo de vida de un proyecto Maven

Maven 2.0 se basa en el concepto central de un ciclo de vida de construcción. Lo que esto significa es que el proceso para la construcción y la distribución de un artefacto en particular (proyecto) está claramente definido.
La construcción de un proyecto, significa que sólo es necesario aprender un pequeño conjunto de comandos para construir cualquier proyecto Maven, y el *POM* se asegurará de obtener los resultados deseados.

Hay tres ciclos de vida de construcción: *default*, *clean* y *site*. El ciclo de vida *default* maneja la implementación del proyecto, el ciclo de vida de *clean* maneja la limpieza del proyecto, mientras que el ciclo de vida *site* se encarga de la creación de la documentación del sitio del proyecto.

En la tabla 19 se puede observar las partes del ciclo de vida principal de un proyecto Maven.

Tabla 19 Ciclo de vida de un proyecto Maven

compile	Compila el código fuente del proyecto, y así genera los archivos .class
test	Ejecuta los test utilizando alguna unidad adecuada de un *framework* de pruebas (p.e JUnit). Estas pruebas no requieren que el código esté empaquetado o desplegado.
package	Toma el código compilado y lo empaqueta en algún formato distribuible, tal como JAR.
install	Instala el paquete en el repositorio local, para usarlo como dependencia en otros proyectos a nivel local.
deploy	Se realiza en un entorno de integración o liberación, copia el paquete final al repositorio remoto para compartir con otros desarrolladores y otros proyectos.

Mojo (Maven plain Old Java Object)

Un *Mojo* es un objetivo ejecutable en Maven, y un *plugin* es una distribución de uno o más *Mojos* relacionados. Pueden ser definidos como clases de Java con anotaciones, o secuencia de comandos *Beanshell*. Un *Mojo* especifica metadatos sobre un objetivo: nombre del objetivo, en qué fase del ciclo de vida se encaja, y que parámetros espera. (Apache, 2014b).

En la figura 59 se puede visualizar un ejemplo de un *Mojo* simple, no recibe parámetros y solo genera de salida una impresión:

```
package sample.plugin;

import org.apache.maven.plugin.AbstractMojo;
import org.apache.maven.plugin.MojoExecutionException;
import org.apache.maven.plugins.annotations.Mojo;

@Mojo(name = "sayhi")
public class GreetingMojo extends AbstractMojo
{
    public void execute() throws MojoExecutionException
    {
        getLog.info("Hello, world.");
    }
}
```

Figura 61 Ejemplo de Mojo simple. (Apache, 2014).

En la forma más simple, un *Mojo* consiste de solo una clase. Al procesar el árbol del código fuente para encontrar *Mojos*, `plugin-tools` busca clases con la anotación `@Mojo` (de Java 5) o la anotación de javadoc "`goal`". Cualquier clase con esta anotación son incluidas en el archivo de configuración del *plugin*.

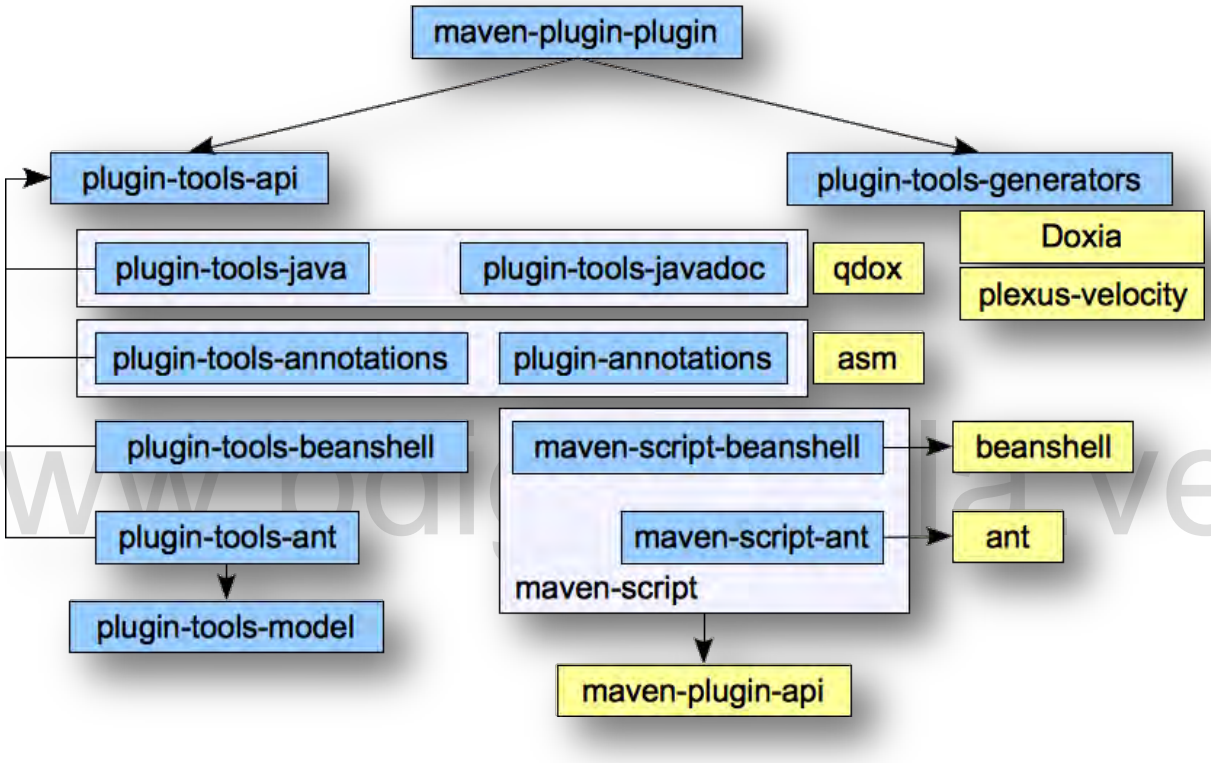

Figura 62 Dependencias dela API Maven Plugin Tools. (Tomado de Apache, 2014b).

`plugin-tools-api` es la interfaz que permite extraer los descriptores para todos los *plugins* compatibles con Maven. Contienen las herramientas necesarias para producir *plugins* de Maven, y de generar contenido como descriptores, ayuda y documentación.

Definición de Plugin de Maven

Una vez que los *Mojos* han sido escritos, es tiempo de construir el *plugin* que ejecutará esos *Mojos*. Para hacerlo de una manera correcta, los descriptores del proyecto necesitan tener ciertas configuraciones:

Tabla 20 Descriptores para la definición de un *plugin* en Maven

`groupId`	Este es el ID del grupo para el plugin, y debe coincidir con el prefijo común a los paquetes utilizados por los *Mojos*.
`artifactId`	Este es el nombre del plugin
`version`	Esta es la versión del plugin
`packaging`	Esto se debe establece en "maven-plugin"
`dependencies`	Una dependencia se debe declarar a la API Maven Plugin Tools, para resolver "AbstractMojo" y clases relacionadas.

En la figura 61 se muestra un ejemplo del archivo *POM* (*pom.xml*) que define los parámetros mostrados en la tabla 20, para el *Mojo* del ejemplo anterior (Figura 59):

```
<project>
   <modelVersion>4.0.0</modelVersion>
    <groupId>sample.plugin</groupId>
   <artifactId>hello-maven-plugin</artifactId>
   <version>1.0-SNAPSHOT</version>
   <packaging>maven-plugin</packaging>

   <name>Sample Parameter-less Maven Plugin</name>
   <dependencies>
      <dependency>
          <groupId>org.apache.maven</groupId>
          <artifactId>maven-plugin-api</artifactId>
          <version>2.0</version>
      </dependency>
   </dependencies>
</project>
```

Figura 63 Configuración del pom.xml que describe un *plugin* de Maven. (Apache, 2014).

Generación de Código

Según Arnoldus, J., Van den Brand, M., Serebrenik, A., y Brunekreef, J. (2012), un generador de código es un programa capaz de generar código basado en la especificación de una entrada. Este programa está escrito para automatizar el trabajo repetitivo, pero también cuando un sistema necesita instanciar representaciones textuales de un modelo, como páginas de HTML en aplicaciones web. Los generadores de código son subclases de metaprogramas.

La generación de código es una proyección de datos de entrada a una salida de código. Estos datos de entrada perteneces a un lenguaje con su propia sintaxis y semántica, definida independientemente del generador de código. Un generador de código traduce esta entrada en otra representación. El código de salida puede ser cualquier cosa, desde código máquina en caso de compiladores, hasta código de un lenguaje de programación.

JavaBeans

Los componentes JavaBeans son programas de software reutilizables que se pueden desarrollar y ensamblar fácilmente para crear aplicaciones de software sofisticadas. Se usan para encapsular varios objetos en un único objeto, para simplificar el uso de varios objetos a uno solo. (Sun Microsystems, 1997).

Para que una clase en Java funciones como una JavaBean, debe obedecer ciertas convenciones sobre nomenclatura. Estas convenciones permiten tener herramientas que puedan utilizar, reutilizar, sustituir y conectar JavaBeans:

- Debe tener un constructor sin argumentos.
- Sus propiedades (atributos) deben ser accesible mediante métodos *get* y *set* que siguen la convención de nomenclatura estándar de Java.
- Debe ser serializable.

La figura 62 muestra un ejemplo sencillo de un JavaBean tomado de Wikipedia (2014b). Se observa que *PersonaBean* tiene dos propiedades, con sus métodos modificadores y de consulta que siguen el estándar Java, el constructor sin argumentos, y que implementa la interfaz Serializable de Java.

```
public class PersonaBean implements java.io.Serializable {

    private String nombre;
    private int edad;

    public PersonaBean(){
    // Empty
    }
    public void setNombre(String n){
        this.nombre = n;
    }
    public void setEdad(int e){
        this.edad = e;
    }
    public String getNombre(){
        return this.nombre;
    }
    public int getEdad() {
        return this.edad;
    }
}
```

Figura 64 Ejemplo de una clase en Java que corresponde a un JavaBean

Java Compiler API

Como indica Zukowski, J., (2006), el *framework Java Compiler* (*javax.tools*) es una API (*Application Program Interface* – Interfaz de programación de aplicaciones) para ejecutar compiladores (y otras herramientas) como un SPI (*Service Interface* – Interfaz de servicio) por el cual ciertos aspectos de un compilador pueden ser personalizados.

Algunos usos típicos de esta API son:

- Ayudar a los servidores de aplicaciones a minimizar el tiempo que toma desarrollar aplicaciones, por ejemplo, evitando la sobrecarga de utilizar un compilador externo para compilar los fuentes de los Servlets generados de páginas JSP.
- Herramientas de desarrollo como IDEs y analizadores de código que puedan invocar el compilador desde el editor o construir herramientas que reduzcan significativamente el tiempo de compilación.

Patrón de diseño Visitor

Los patrones de diseño describen problemas que ocurren repetitivamente en un entorno, y propone el núcleo para la solución de ese problema, de tal manera que esa solución pueda ser utilizada una y otra vez. (Christopher, A. et al, 1977).

Particularmente, el patrón de diseño *Visitor* representa una operación a realizar sobre los elementos de un objeto. El *Visitor* permite definir una nueva operación sin cambiar las clases o los elementos sobre los que opera. (Gamma, E. et al 1995).

El patrón *Visitor* se usa cuando:

- un objeto contiene muchas clases diferentes de objetos con diferentes interfaces, y se desea realizar operaciones sobre estos objetos que dependen de sus clases concretas.
- muchas operaciones distintas y no relacionadas deben realizarse en los objetos en una estructura de objeto, y que se quiera evitar "contaminar" sus clases con estas operaciones. El *Visitor* le permite mantener operaciones relacionadas entre sí definiéndolos en una clase. Cuando la estructura del objeto es compartida por muchas aplicaciones, utilice *Visitor* para poner las operaciones en sólo aquellas aplicaciones que los necesitan.
- las clases que definen la estructura del objeto rara vez cambian, pero que a menudo se quiere definir nuevas operaciones sobre la estructura. Cambiar la estructura del objeto requiere la redefinición de la interfaz para todos los visitantes, que es potencialmente costoso. Si la estructura del objeto cambia con frecuencia, entonces es probablemente mejor definir las operaciones en esas clases.

La estructura del patrón de diseño Visitor se puede visualizar en la figura 63:

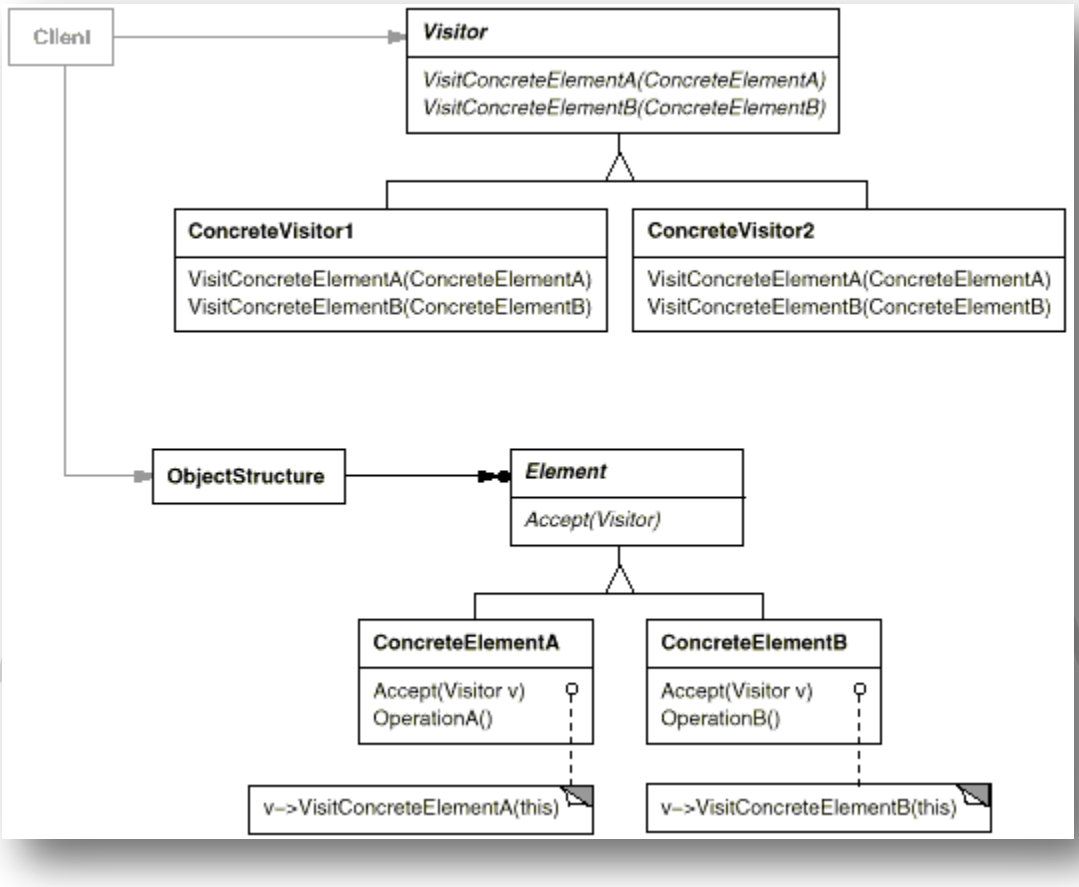

Figura 65 Estructura del Patrón de Diseño Visitor (Tomado de Gamma, E. et al , 1995).

Los participantes de este patrón de diseño son:

- *Visitor* (Visitante): declara una operación de visita para cada elemento concreto en la estructura de objetos, que incluye el propio objeto visitado.
- *ConcreteVisitor* (Visitante Concreto): implementa las operaciones del *Visitor* y acumula resultados como estado local.
- *Element* (Elemento): define una operación "*Accept*" que toma un *Visitor* como argumento.
- *ConcreteElement* (Elemento Concreto): implementa la operación "*Accept*".

La figura 62 muestra un diagrama de interacción e ilustra las colaboraciones entre una estructura de objeto, un *visitor* y dos elementos:

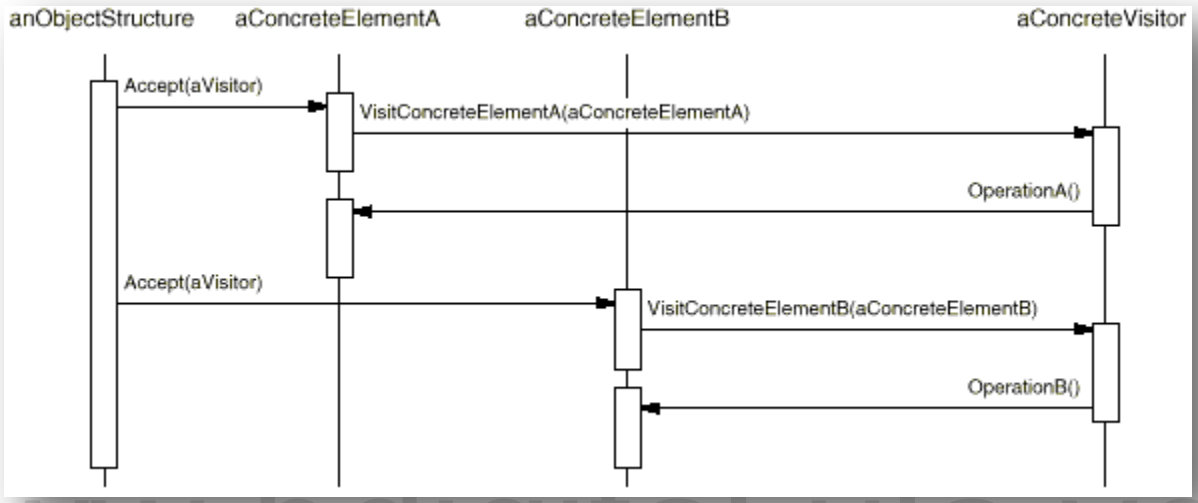

Figura 66 Diagrama de Colaboración del Patrón de Diseño Visitor (Tomado de Gamma, E. et al, 1995).

El cliente que usa el patrón Visitor debe crear un objeto de *ConcreteVisitor* y luego recorrer la estructura de objetos, visitando a cada elemento con el *Visitor*. Cuando un elemento es visitado, llama a la operación del *Visitor* que corresponde a la clase. El elemento se suministra a sí mismo como un argumento a esta operación para dejarle al *Visitor* acceso a su estado, de ser necesario.