Universidad de las Ciencias Informáticas

Facultad 3

Título: “Herramienta de soporte para el proceso de verticalizaciones del sistema Cedrux.”.

Trabajo de Diploma para optar por el título de
Ingeniero en Ciencias Informáticas.

Autor(es): Odelaisy García Borges

Fabiel Torres Marrero

Tutor(es): Yusnier Matos Arias

Ciudad de La Habana
Junio del 2012
“Año 53 de la Revolución”

Pensamiento:

"Programar sin una arquitectura en mente es como explorar una gruta sólo con una linterna: no sabes dónde estás, dónde has estado y hacia dónde vas.".

Danny Thorpe

Declaración de autoría

Declaramos ser autores de la presente tesis y reconocemos a la Universidad de las Ciencias Informáticas los derechos patrimoniales de la misma, con carácter exclusivo.

Para que así conste, firmamos la presente a los ____ días del mes de _____ del año 2012.

Odelaisy García Borges

Firma del Autor

Fabiel Torres Marrero

Firma del Autor

Ing. Yusnier Matos Arias

Firma del Tutor

Datos de contacto

Tutor: Ing. Yusnier Matos Arias

Edad: 26 años.

Ciudadanía: cubano.

Institución: Universidad de las Ciencias Informáticas (UCI).

Título: Ingeniero en Ciencias Informáticas.

Categoría docente: Profesor instructor.

E-mail: ymarias@uci.cu

Ingeniero en Ciencias Informáticas, graduado en 2008 en la Universidad de las Ciencias Informáticas. Instructor. Tres años de experiencia en el desarrollo de software. Tres años de graduado.

Agradecimientos

De Odelaisy:

Agradezco a:

A mis padres, por todo lo que me enseñaron, por indicarme siempre cuál era el camino a seguir y ser los principales artífices de que yo haya llegado hasta aquí.

A mi abuela y a mi tía Eneida por ser ejemplos, por toda la ayuda moral, espiritual y profesional que me han dado.

A mis hermanos Yunier y Odeimys porque siempre me dieron aliento y me apoyaron en todo momento.

A Esther, Fabianne y Felipe, por su apoyo, cariño y preocupación.

A mi novio por estar siempre presente y darme aliento en todo momento.

A Dariel, por el apoyo que me ha dado en estos cinco años.

A mi Familia en general por brindarme en todo momento su apoyo incondicional.

De Fabiel:

Agradezco a:

A mis padres por su incondicional apoyo a lo largo de mi vida estudiantil en general.

A mi hermana por ser un ejemplo para mí, por toda la ayuda moral, y profesional que me ha dado.

A la familia de mi novia en general por siempre apoyarme y tratarme como si fuera u hijo mas.

A mi novia por estar presente en todo momento.

De los dos:

Agradecemos a:

A nuestro tutor Yusnier Matos Arias que hizo suyo este trabajo, guiándonos por el camino correcto, siempre dispuesto a ayudar ante cualquier inquietud o duda. Por toda su ayuda y esfuerzo estaremos siempre agradecidos.

A nuestros compañeros por estos años de estudio que nos han colmado de experiencias positivas y han dejado en nosotros la huella de la amistad.

A Eduardo por toda la ayuda brindada. Muchas gracias.

A todos los compañeros y profesores que nos dieron apoyo y ayuda, en especial a Ariel por ser incondicional estos cinco años.

Dedicatoria

De Odelaisy:

A mis padres, a los tres, por todo el amor y apoyo que me han dado siempre. Porque por ellos es que quiero ser mejor cada día, para que se sientan orgullosos del gran trabajo que han hecho conmigo.

A Jeny por ser la luz de mis ojos.

A mis hermanos, que han estado presentes en todo momento brindándome su apoyo y cariño.

A todos mis tíos y en especial a mi tía Eneida, por su amor cariño y preocupación.

A mi abuela querida, por estar siempre ahí para mí, por la confianza, el apoyo y el inmenso amor que me ha dado.

A mi abuelo del alma Nerey, porque siempre me dio amor y aunque ya no está yo se que se sentiría muy orgulloso de la mujer en la que me he convertido.

A toda mi familia….

De Fabiel:

A mis padres por estar siempre presentes para mí y por darme su cariño y apoyo incondicional.

A la niña (Fabianne) por ser mi guía y mi reto de ser como ella.

Resumen

Cedrux es un Sistema Integral de Gestión desarrollado por el Centro de Informatización de la Gestión de Entidades (CEIGE). Para su desarrollo se han tenido en cuenta los aspectos generales de las áreas de negocio de las entidades empresariales y presupuestadas cubanas. Sin embargo, es necesario que provea a su vez, funcionalidades particulares de dichas entidades. Para esto, en ocasiones, se hace necesario realizar cambios estructurales en su arquitectura. Estos cambios hoy se realizan con pocos elementos para un análisis previo de sus implicaciones sobre la propia estructura arquitectónica, y por tanto, sobre el sistema.

El objetivo de este trabajo es desarrollar una herramienta que provea información sobre elementos arquitectónicos que se ven afectados con los cambios en la estructura arquitectónica. Esta herramienta es desarrollada como un *plugin*[1] para el IDE Eclipse e integrada con la herramienta *AcmeStudio*, la cual es un *front-end*[2] del Lenguaje de Descripción Arquitectónica (ADL, del inglés *Architectural Description Languages*) *Acme*. Hace uso de técnicas de análisis de dependencia y recibe como entrada modelos realizados en *AcmeStudio*. Los resultados del trabajo se validan mediante la realización de pruebas a la herramienta, aplicación de la técnica Criterio de expertos y el método Delphi para la evaluación de las variables de investigación.

Palabras claves: *AcmeStudio*, análisis de dependencias, verticalización.

[1]

[2] Front-end: Parte del software que interactúa con los usuarios.

Tabla de Contenido

Introducción

En el departamento Desarrollo de Productos del Centro de Informatización de la Gestión de Entidades (CEIGE), se desarrolla el Sistema de Gestión Integral Cedrux. De este han sido liberados varios módulos por el equipo de Calisoft[3].

Cedrux es un paquete de soluciones integrales de gestión, para las entidades presupuestadas y empresariales. Se desarrolla siguiendo los principios de independencia tecnológica.

Este sistema está concebido de modo que pueda ser verticalizado teniendo en cuenta necesidades y particularidades de clientes específicos. De acuerdo con Manuel Baraza en (Baraza, 30), “verticalizar un software, es hacer de este un sistema más accesible, es decir, adaptarlo más y/o mejor a la capacidad de los usuarios, a la capacidad de entendimiento para quien consulta la aplicación, y capacidad de entendimiento y de ejecución para quien consulta y edita.”.

En el contexto de este trabajo, se define verticalizar como el proceso de adaptar funcionalidades del sistema teniendo en cuenta necesidades de clientes específicos. En determinadas ocasiones los clientes se interesan en partes específicas del sistema, es decir, por subconjuntos de funcionalidades que este ofrece. Incluso pueden estar interesados en contar con las funcionalidades que ofrece el software. Sin embargo en este hay procesos implementados que no responden del todo al modo en que se desarrolla en sus entornos o empresas. En estos casos, es necesario realizar adaptaciones que por lo general conllevan a cambios en la estructura de componentes. En otras palabras, se hace necesario agregar, modificar o eliminar componentes del sistema o interacciones entre estos.

Durante el desarrollo de las verticalizaciones de Cedrux, los arquitectos han identificado afectaciones. Dichas afectaciones están dadas fundamentalmente por los pocos elementos con que se cuenta para determinar las implicaciones que tienen sobre la arquitectura, los cambios en la estructura de componentes. En el departamento Desarrollo de Productos de CEIGE, se han propuesto acciones que ayudan a mitigar las afectaciones identificadas por los arquitectos, entre estas se pueden mencionar las siguientes:

[3] Calisoft: Centro de calidad para soluciones informáticas.

- Definición de un conjunto de artefactos para registrar las dependencias entre componentes. Estos se basan en hojas de cálculo con información de los componentes y las conexiones entre ellos. (Fernández, septiembre de 2011)
- Se llevó a cabo una investigación con el fin de seleccionar un Lenguaje de Descripción Arquitectónica (del inglés, ADL[4]) para modelar la arquitectura del sistema Cedrux y que aportara elementos para tomar decisiones relacionadas con la arquitectura. Este ADL fue *Acme*. (Valero, Julio de 2009).

Las acciones anteriores, han resuelto la problemática planteada en parte. El ADL seleccionado, por ejemplo, permite modelar la arquitectura de Cedrux teniendo en cuenta su taxonomía. Además cuenta con una herramienta gráfica, denominada *AcmeStudio* basada en el Entorno de Desarrollo Integrado (del inglés, IDE[5]) Eclipse, por lo que puede ser extendida con relativa facilidad para agregarle funcionalidades.

Sin embargo todavía se presentan dificultades como las siguientes:

- Se torna demasiado compleja la actualización y la propia revisión de los artefactos definidos para llevar la información de los componentes dada la considerable cantidad de estos y sus conexiones.
- No muestra información sobre dependencias entre componentes
- *AcmeStudio* no permite guardar la información sobre los modelos en hojas de cálculo (formato xls); formato compatible con los de los artefactos usados en CEIGE.
- *AcmeStudio* no permite guardar información relacionada con el análisis que realiza.
- El análisis que realiza *AcmeStudio* sobre los modelos realizados no brindan información sobre impacto de cambios en la estructura de componentes.
- La información que muestra sobre los modelos está muy relacionada con la Lógica de Predicados que usa, por lo que otros análisis que se complejicen con el uso de este formalismo no los realiza. Entre estos pudieran mencionarse: criticidad de un componente, complejidad de un componente, existencia de determinados ciclos en la estructura de componentes.

El hecho de mantener los problemas anteriores, ha influido en el tiempo en que se implementan las verticalizaciones de Cedrux.

[4] Architecture Descrition Language.
[5] Integrated Development Environment

A partir de estas problemáticas se genera el siguiente **problema a resolver** de la presente investigación: la herramienta *AcmeStudio* no provee funcionalidades que permitan identificar los elementos que se afectan ante cambios en la estructura de componentes durante las verticalizaciones de Cedrux, esto provoca un aumento en el tiempo durante la implementación de las mismas.

Del problema descrito anteriormente, se determinó que el **objeto de estudio** de este trabajo se enmarca en herramientas y técnicas para la realización de análisis arquitectónico.

Para dar solución al problema se formula como **objetivo general**: Desarrollar un *plugin* para Eclipse que extienda la herramienta *AcmeStudio* con funcionalidades que permitan identificar las implicaciones de las modificaciones a la estructura de componentes durante las verticalizaciones de Cedrux, de manera que contribuya con la reducción de tiempo de implementación de las mismas.

Para dar cumplimiento al objetivo general propuesto se declaran los siguientes **objetivos específicos**:

- Establecer el Marco Teórico de la investigación referente a las herramientas y técnicas para la realización de análisis arquitectónico, para sentar las bases de la propuesta de solución.
- Implementar la herramienta para realizar análisis del impacto de cambios en la estructura de los componentes.
- Aplicar la herramienta en el proceso de verticalización de Cedrux analizándose y evaluándose los resultados obtenidos con el fin de validar la investigación.

El **campo de acción** está dirigido a las técnicas y herramientas para el análisis de dependencias entre componentes de software.

Basado en lo anterior se plantea la siguiente **idea a defender**: Si se desarrolla una herramienta que permita realizar análisis de dependencias entre los componentes de una estructura de componentes dada, y se aplica en el proceso de verticalizaciones de Cedrux, entonces, se reducirán los atrasos en la implementación de dichas verticalizaciones, dados por demoras en la identificación de elementos afectados por los cambios en la estructura de componentes.

Las **tareas** de investigación planificadas para dar solución al problema y dar cumplimiento al objetivo planteado son:

- Revisión de documentación relacionada con el objeto de estudio del trabajo.
- Análisis de la información a partir de la revisión bibliográfica realizada.
- Descripción del contexto sobre el que se enfoca la solución que se propone.
- Identificación de las funcionalidades que se deben extender en la herramienta *AcmeStudio* enfocadas a los análisis que se realizan durante las verticalizaciones de Cedrux.
- Modelación de dominio.
- Levantamiento de requisitos.
- Diseño de la solución.
- Validación del diseño de la solución.
- Desarrollo de la herramienta.
- Integración de la herramienta con la biblioteca *AcmeLib* en un *plugin* para Eclipse.
- Diseño de pruebas
- Realización de pruebas de funcionalidades.
- Validación de la solución mediante su aplicación en un caso de estudio.
- Análisis de los resultados de la validación de la herramienta.

Para el desarrollo del presente trabajo se utilizaron los siguientes métodos de investigación científica:

- **Analítico sintético**: Con el uso de este método, se lograron extraer los elementos más importantes relacionados con el objeto de estudio, así como los rasgos que lo caracterizan y distinguen, a través de un análisis de teorías y documentos.
- **Histórico lógico**: Se utilizó para realizar un estudio de las técnicas de dependencias que utilizan los sistemas de análisis de arquitectura de software.

El presente trabajo se encuentra estructurado como se explica a continuación: Introducción, Capítulos 1,2 y 3, Conclusiones, Anexos y Bibliografía.

Capítulo 1. Fundamentación teórica: En este capítulo se realiza una valoración de las algunas herramientas que utilizan técnicas de análisis de dependencias. También se tratan conceptos necesarios para poder desarrollar la herramienta propuesta.

Capítulo 2. Diseño e implementación: En este capítulo se presentan las características del *plugin*, se definen la principales funcionalidades que debe tener el diseño y los artefactos generados. Además se valida el diseño haciendo uso de métricas.

Capítulo 3. Validación y pruebas: En este capítulo se efectúa la validación de la variable establecida, se presentan las pruebas de software realizadas para la validación de la propuesta de solución respectivamente.

CAPÍTULO 1: FUNDAMENTACIÓN TEÓRICA

1 Introducción

En el presente capítulo se realiza una breve descripción de algunas herramientas que realizan análisis arquitectónico y las técnicas que utilizan para ello. Además se tratan otros temas vinculados con la arquitectura de software. También se realiza un análisis con el fin de seleccionar la metodología que guiará el proceso de desarrollo del software. Por último se exponen las herramientas utilizadas para la implementación del *plugin*.

1.2 Arquitectura de software

Hoy día, no existe una única definición de arquitectura de software, cada arquitecto defiende su punto de vista en dependencia del contexto en el que se desarrolla. La cuestión es: cada escenario es distinto y por ende requiere de diferentes interpretaciones, en este epígrafe se exponen algunas de las definiciones de arquitectura de software.

Dentro de las definiciones de arquitectura de software más reconocida está la de Paul Clements (Reinoso, Marzo 2004), quien define a la AS como una vista general del sistema que incluye los componentes principales del mismo. Otro punto de vista lo establece David Garlan (Reinoso, 2004), este establece que la arquitectura de software constituye un puente entre el requerimiento y el código, ocupando el lugar que en los gráficos antiguos se reservaba para el diseño. Finalmente el estándar IEEE Std 1471-2000 define la AS como sigue:

"*La Arquitectura de Software es la organización fundamental de un sistema encarnada en sus componentes, las relaciones entre ellos y el ambiente y los principios que orientan su diseño y evolución.*".

Las definiciones vistas anteriormente, demuestran que existe una tendencia a considerar la AS, como la base organizacional de un sistema que viene determinada principalmente por los **componentes** como elementos básicos de la funcionalidad del sistema, caracterizados por una interfaz segmentada en **puertos** y **conectores** quienes describen la interacción entre los diferentes componentes para permitir un bajo acoplamiento entre ellos. En esta descripción, se emplea también una serie de conexiones que definen la composición de todos ellos formando una estructura única denominada **configuración**. Basado en los conceptos analizados, se decide adoptar la definición de AS que se

propone en el estándar IEEE Std 1471-2000 como base conceptual de cada uno de los elementos arquitectónicos que se presentan en la investigación.

Para el diseño y distribución de arquitecturas existen diversos lenguajes descriptivos, que proporcionan características para modelar un sistema de software. Estos lenguajes en su mayoría cuentan con herramientas que permiten realizar análisis sobre la arquitectura descrita, para comprobar la consistencia de la misma. En el epígrafe 1.3 se exponen diferentes Lenguajes de Descripción Arquitectónica (ADL) enfocando el estudio de estos a las herramientas y técnicas de análisis asociadas a los mismos.

Es importante también para la investigación referirse al análisis arquitectónico, dirigiendo el estudio a las diferentes clasificaciones de análisis de dependencias. El objetivo es seleccionar el tipo de dependencia que se ajuste a las necesidades de la herramienta que se pretende implementar.

1.2.1 Análisis arquitectónico

En el contexto de este trabajo el estudio referente al análisis de arquitecturas se centró principalmente en el análisis de dependencias y análisis de impacto de cambios.

El objetivo del ingeniero de software es crear un software de calidad, pero sobre todo busca la confianza del usuario. Ésta puede perderse cuando el tiempo de respuesta para su resolución de fallos excede de unos mínimos aceptables. Por ello se debe realizar un análisis de la arquitectura eligiendo técnicas de análisis donde primen la eficiencia y trazabilidad. (Simarro, 2004).

Análisis de dependencia

Durante el proceso de verticalizaciones se desarrollan modificaciones en la estructura de componentes que pueden incidir de manera negativa en el sistema. Se hace relevante el estudio del análisis de dependencias por la importancia que tiene para los arquitectos, conocer con anterioridad a los cambios, el impacto que pudieran provocar los mismos a la arquitectura del software, identificando durante el análisis, las dependencias y restricciones que existen entre los componentes.

Las dependencias pueden ser identificadas basándose en la información sintáctica disponible en una especificación formal de la arquitectura. El análisis de dependencias aplicado al código de un programa se basa en las relaciones entre sentencias y variables

en un programa. Se han adaptado éstas técnicas para identificar y explotar relaciones de dependencia en el nivel arquitectónico. (Simarro., Diciembre 2004).

Las relaciones de dependencias en el nivel arquitectónico, surgen de las conexiones entre componentes y las restricciones sobre sus interacciones. Estas relaciones involucrarían alguna forma de control de flujo de datos, pero más generalmente involucran estructura y comportamiento fuente. La estructura fuente de la arquitectura tiene relación con las dependencias del sistema tales como ¨*imports*¨, mientras que el comportamiento tiene que ver con las dependencias de interacción dinámica. (Simarro., Diciembre 2004).

La dependencia estructural permite localizar especificaciones que contribuyan a la descripción de algún estado o interacción. Cuando se analiza una arquitectura y sus dependencias, es muy importante capturar y entender las relaciones entre estados o interacciones con otros estados o interacciones. (Simarro., Diciembre 2004).

La mayoría de las dependencias estructurales se pueden encontrar mediante la inspección del código fuente (es decir, el análisis estático del código fuente), las dependencias estructurales también existen en el nivel de modelos y ejecución de la aplicación. (Trosky B. Callo Arias, Marzo 2011)

Dependencias de Comportamiento: A diferencia de las dependencias estructurales, las de comportamiento o interacción a menudo implican abstracciones que no están directamente proporcionadas por los lenguajes de programación: el uso de interfaces públicas (por ejemplo, programas o dispositivos externos), la difusión de eventos, cliente-servidor, protocolos, ordenamiento temporal, etc. (Trosky B. Callo Arias, Marzo 2011)

Dependencias de trazabilidad: En un proceso iterativo, un desarrollador no puede descartar los requisitos después de que el diseño se construyó, ni desechar el diseño después de que el código fuente está programado. Por lo tanto, los desarrolladores tienen la necesidad de mantener las interrelaciones entre los diferentes artefactos. Estas interrelaciones se denominan dependencias de trazabilidad y caracterizan las dependencias entre requisitos, diseño, y el código. (Trosky B. Callo Arias, Marzo 2011)

El análisis de dependencia se realiza con frecuencia para determinar el impacto tienen los cambios sobre la arquitectura de un sistema de software. (Trosky B. Callo Arias, 2011).

Consideraciones del análisis arquitectónico

En el estudio anterior, se trataron tres tipos de análisis: el análisis de dependencia estructural, el de comportamiento y el de trazabilidad. Siendo el análisis estructural basado en la arquitectura el que se ajusta a las funcionalidades que se pretenden extender a la herramienta gráfica de modelado *AcmeStudio*. En la **Figura 1** se explican las dos (2) variantes de ejecución de este análisis. En ambos casos los objetivos (tal como se muestra en el recuadro inferior de la figura) y relaciones (tal como se muestra en el recuadro central de la figura) son comunes, la diferencia radica en que el análisis basado en el código se centra en las declaraciones, variables y procedimientos y el análisis basado en la arquitectura se centra en los componentes, puertos y conexiones del modelo.

Figura 1. Comparación de análisis de dependencia

1.3 Lenguajes de Descripción de Arquitectura

Los ADLs, proporcionan notaciones para descomponer un sistema en componentes y conectores y especificar cómo se combinan estos elementos para formar cierta configuración. (Río, 2002) Además suministran soporte de herramientas para el desarrollo de soluciones basadas en arquitectura y su posterior evolución.

Steve Vestal (Vestal., Febrero de 1993) Define que un ADL debe modelar o soportar conceptos como:

- Componentes.
- Conexiones.

- Composición jerárquica, en la que un componente puede contener una sub-arquitectura completa.
- Soporte de herramientas para modelado, análisis, evaluación y verificación.
- Composición automática de código aplicativo.

Tracz en (Wolf, 1997) define un ADL como una entidad consistente en cuatro “Cs”: componentes, conectores, configuraciones y restricciones (restricciones, del inglés *constraints*).

Componentes: Un componente es una unidad de composición de aplicaciones software, que posee un conjunto de interfaces y un conjunto de requisitos, y que ha de poder ser desarrollado, adquirido, incorporado al sistema y compuesto con otros componentes de forma independiente, en tiempo y espacio. (Szyperski, 1998)

Conectores: Representan interacciones entre componentes. Al igual que los componentes, los conectores tienen una interfaz, la cual consiste en un conjunto de roles. Cada rol define el comportamiento esperado de uno de los participantes en la relación Ejemplos típicos podrían ser tuberías (*pipes*), paso de mensajes, llamadas a procedimientos, protocolos cliente-servidor o conexiones entre una aplicación y un servidor de base de datos. Los conectores también tienen una especie de interfaz que define los roles entre los componentes participantes en la interacción. (Kicillof, 2004).

Configuraciones: Se constituyen como grafos de componentes y conectores. En los ADL más avanzados la topología del sistema se define independientemente de los componentes y conectores que lo conforman. Los sistemas también pueden ser jerárquicos: componentes y conectores pueden resumir la representación de lo que en realidad son complejos subsistemas. (Kicillof, 2004).

Restricciones: Representan condiciones de diseño que deben acatarse incluso en el caso que el sistema evolucione en el tiempo. Restricciones típicas serían restricciones en los valores posibles de propiedades o en las configuraciones topológicas admisibles. Por ejemplo, el número de usuarios que se puede conectar simultáneamente a un determinado servicio. (Kicillof, 2004).

Principales características de los ADL

- Composición: Permiten la representación del sistema como la composición de una serie de partes.

- Configuración y Abstracción: Mediante las cuales se describen los roles o papeles abstractos que juegan los componentes dentro de la arquitectura.
- Flexibilidad: Permiten la definición de nuevas formas de interacción entre componentes.
- Reutilización: Pues permiten la reutilización tanto de los componentes como de la propia arquitectura.
- Heterogeneidad: Pueden combinar descripciones heterogéneas.
- Análisis: Permiten diversas formas de análisis de la arquitectura y de los sistemas desarrollados a partir de ella. (Kicillof, 2004).

No es objeto de esta tesis un análisis exhaustivo de los ADLs, sino que el estudio de estos se centró en indicadores como:

- Soporte de herramientas: Se tiene en cuenta la disponibilidad de herramientas asociadas al ADL, para facilitar el uso del mismo.
- Disponibilidad: En este se aspecto se tiene en cuenta que sea un ADL libre y bien documentado.
- Tipos de análisis arquitectónicos con los que están asociados.

Basado en los indicadores formulados anteriormente, a continuación se hace un resumen de ADLs como: *Rapide, Wright, C2 y Acme*.

Rapide

El lenguaje se ha diseñado para dar soporte al desarrollo basado en componentes de sistemas multilenguaje de gran tamaño, utilizando las definiciones arquitectónicas como marco fundamental.

El modelo de ejecución adoptado por este ADL está basado en los *posets*[6] los cuales son identificados a través de patrones de eventos. El objetivo de estos *posets* es describir las secuencias válidas de eventos que pueden ocurrir en un determinado sistema, los cuales se corresponden con la invocación de las operaciones entre los componentes, estos eventos se dividen en acciones que modelan comunicaciones asíncronas y funciones que modelan comunicaciones síncronas; de esta forma es posible simular las especificaciones realizadas con este ADL, así como verificar si las trazas de eventos generados cumplen determinadas restricciones. (1997).

[6]Posets: Conjunto de eventos parcialmente ordenados.

En la simulación simplemente testea la arquitectura, y no proporciona un análisis exhaustivo del escenario. Nada garantiza que no pueda surgir una inconsistencia en una ejecución diferente. *Rapide* ha desarrollado un conjunto de herramientas, las mismas no han evolucionado desde 1997, y tampoco han avanzado más allá de la fase de prototipo. (Kicillof, 2004).

La semántica de este ADL se basa en los conceptos de interfaz, *posets* y patrones de conexión. Dicha semántica es apoyada por un conjunto de herramientas, las cuales son:

- ***Rpdc***: que es el compilador de código fuente *Rapide*, el cual reporta errores de sintaxis y semánticos. Genera un ejecutable que al correr produce un archivo ".l*og*", que es un registro de los eventos producidos durante la ejecución. (Francisco Losaivo, noviembre 2003)

- ***Pov*** (*Partial Order Viewer*): que permite observar en forma gráfica la traza de eventos producidos por los ejecutables. (Francisco Losaivo, noviembre 2003)

- ***Raptor***: la cual es una herramienta para la animación de arquitecturas. Tales animaciones ayudan a los especialistas a comprender la arquitectura de un sistema, pues muestran el funcionamiento de cada uno de los componentes, siempre desde la óptica de los posets. (Francisco Losaivo, noviembre 2003)

Wright

Es un ADL con énfasis en la comunicación y formalización de conexiones arquitectónicas. Tiene como propósitos principales: la integración de una metodología formal con una descripción arquitectónica y la aplicación de procesos formales, tales como: el álgebra de proceso y el refinamiento de procesos a una verificación automatizada de las propiedades de las arquitecturas de software. En *Wright* la interfaz de los componentes se representa a través de una serie de puertos y la de los conectores a través de roles, donde los puertos describen el comportamiento de los componentes y los roles el comportamiento que deben tener los conectores una vez se conecten los componentes. (Río, febrero de 2002)

Wright se basa en *Communicating Sequential Process* (CSP[7]) como modelo semántico. Realiza análisis estático para verificar la consistencia del modelo. Permite la definición de componentes, conectores y roles. En *Wright* existe la posibilidad de utilizar cualquier

[7] CSP: Communicating Sequential Process. En español: Comunicación de procesos secuencial.

herramienta de análisis o técnica que pueda usarse para comunicación de procesos. (Río, febrero de 2002)

Acme

Acme se define como un lenguaje capaz de soportar el mapeo de especificaciones arquitectónicas entre diferentes ADL, o en otras palabras, como un lenguaje de intercambio de arquitectura. Además define siete (7) elementos básicos que son: Componentes, conectores, sistemas, puertos, roles, representaciones y mapas de representación. (Valero, 2009).

Acme soporta una variedad de *front-ends*[8] de carácter gráfico, que se componen por: *AcmeStudio* que es un entorno gráfico basado en *Windows y Linux*, susceptible de ser configurado para soportar visualizaciones específicas de estilos e invocación de herramientas auxiliares. (2007).

Armani es un lenguaje de restricción que extiende Acme basándose en lógica de predicados, y que es por tanto útil para definir invariantes (limitaciones fundamentales de diseño) y heurísticas de estilos (reglas generales). (Kicillof, 2004)

La disponibilidad de la herramienta gráfica *AcmeStudio* para el modelado tiene suficiente documentación y página web. Hay documentación y herramientas disponibles libremente en Internet. La herramienta ha sido desarrollada en la plataforma Eclipse en el lenguaje de programación *Java* y puede ser extendida mediante *plugin*.

C2 (Chiron 2)

C2 es mayormente considerado como estilo arquitectónico y no como ADL, sin embargo ADLs como: C2 SADL, C2SADEL entre otros; se basan en C2. El mismo plantea que un sistema de software está compuesto por componentes ordenados en capas o niveles que se coordinan y comunican mediante peticiones de servicios y notificaciones de eventos, y por conectores los cuales tienen como única función conectar a los componentes y llevar a cabo la comunicación entre ellos, en cada nivel o capa hay un solo conector el cual actúa de delimitador para su nivel correspondiente. En el caso de los componentes solo se pueden vincular con un conector, mientras un conector se puede vincular con más de un componente y más de un conector. A la hora de la comunicación por medio de dos tipos de mensajes, los de requerimiento que sólo se pueden enviar “hacia arriba” y los de

[8] Front-Ends: Parte del software que interactúa con los usuarios.

notificación sólo “hacia abajo”. En C2 no existe limitación para los lenguajes de la implementación, aunque normalmente soporta desarrollos en C++, Ada, C# y *Java*. Entre las herramientas independientes de plataforma que dispone C2 está el entorno gráfico *ArchStudio* implementado en estilo C2. (Kicillof, marzo de 2004).

1.4 Herramientas analizadas

En el presente epígrafe se muestra un resumen del estudio referente a herramientas como Aladdin e IDM. El análisis de las mismas, va encaminado a identificar las técnicas que estas herramientas emplean para analizar las dependencias que existen en una arquitectura de software.

En el caso específico de *Acme,* el objetivo del estudio de este lenguaje está dirigido a identificar las funcionalidades que se desean extender a la herramienta gráfica de modelado *AcmeStudio*, enfocando dichas funcionalidades a las necesidades actuales de los arquitectos de Cedrux a la hora de realizar análisis de dependencias sobre la arquitectura que se modela con *Acme*.

Aladdin

Aladdin es un analizador de dependencias en arquitecturas de software. Para su funcionamiento esta herramienta toma como entrada el diseño de la arquitectura a analizar, definida tanto mediante el ADL *Acme* como *Rapide*. Ambos lenguajes aportan como entrada la especificación de componentes, conectores y puertos de la arquitectura a analizar. Estos aparecen en *Aladdin* como un conjunto de cajas y flechas en un diagrama, donde las flechas representan la comunicación entre dos componentes o conectores mediante uno o varios puertos. Además permite desplazarse hacia delante o hacia atrás desde una caja dada, atravesando flechas desde el principio hasta el final. (Judith A. Stafford, 1998).

La herramienta permite ir sobre los puertos de los componentes e ir avanzando o retrocediendo por algún camino determinado, para alcanzar todos los puertos de salida o entrada. Este análisis es esencial para encontrar posibles dependencias falsas entre componentes. Esto es posible mediante la utilización de un ADL para la descripción de las entradas, y aprovechando esta información para generar un conjunto reducido. (Judith A. Stafford, 1998).

Aladdin, es una herramienta que implementa el encadenamiento (*Chaining*), como técnica de análisis de dependencia, también usa un algoritmo resumen que opera sobre la

descripción del comportamiento de interacción de un componente para identificar las relaciones posibles entre pares de puertos de entrada y salida. La conexión resultante se denomina conexión de transición. (Judith A. Stafford, 1998).

Integrated Dependencies Model (IDM)

IDM es una herramienta capaz de procesar diagramas de dependencia escritos en UML e informar si esas dependencias efectivamente se cumplen en la realidad. IDM está formado por los siguientes módulos: (Fontdevilla, 2007).

- **IDM Vista**: Un *plugin* (módulo agregado al entorno de desarrollo) de Eclipse para la evaluación gráfica de los modelos. La funcionalidad de IDM Vista está integrada con Eclipse, de manera tal que un archivo .xmi (XML *Metadata Interchange*, que contiene un diagrama UML en formato XML) es abierto con IDM Vista por Eclipse. Así, basta abrir un archivo .xmi para ejecutar los controles y ver el resultado. (Fontdevilla, 2007).
- **IDM Consola**: Una aplicación de consola capaz de realizar los controles sobre los diagramas sin interfaz gráfica de usuario.
- **IDM Núcleo**: Un conjunto de librerías que describen el modelo de dependencias entre elementos de distintas vistas. Incluye también una plantilla para simplificar la creación de modelos de dependencia en UML. (Fontdevilla, 2007).

La herramienta está diseñada para acompañar el proceso de desarrollo, con las siguientes características:

- Define una vista mixta para documentar relaciones entre elementos de distintas vistas.
- Promueve la documentación de información específica tradicionalmente difícil de ubicar.
- Convierte los diagramas en entidades activas, con comportamiento, agregando una nueva dimensión a la información registrada.
- Agrega un atractivo más para motivar a los desarrolladores a generar la documentación.
- Permite ejecutar múltiples veces la evaluación de un diagrama, por ejemplo antes y después de realizado un cambio, ayudando a mantener la consistencia del sistema en desarrollo.
- Utiliza estándares de la industria como UML y XMI.

- Está diseñada para integrarse al proceso de desarrollo sin reemplazar a otras herramientas, si no agregando su funcionalidad.
- No requiere información muy detallada en los diagramas para funcionar, a diferencia de otras herramientas estilo CASE. (Fontdevilla, 2007)

IDM puede considerarse como una herramienta de "ejecución de diagramas" o "*running* UML", puesto que genera y ejecuta controles para todas las dependencias documentadas. A diferencia de una herramienta de ingeniería directa (*forward engineering*), no se aplica a generar el sistema en estudio sino a analizarlo a posteriori. (Fontdevilla, 2007).

AcmeStudio

AcmeStudio es un entorno de desarrollo de arquitectura, escrito como un *plugin* para el IDE Eclipse, que utiliza al ADL *Acme* como base de funcionamiento. Este ADL permite definir arquitecturas y combinar estilos arquitectónicos. La herramienta proporciona un sistema para la definición de nuevos componentes y tipos de conectores, las normas sobre su composición, e instalaciones para el empaquetado de estos en los estilos arquitectónicos. (Schmerl, 2003)

AcmeStudio admite diferentes herramientas de análisis para ser integradas, y se aplica a ciertos estilos arquitectónicos. Ejemplos de estos análisis que se han integrado son: el análisis del rendimiento basado en la teoría de colas, lo que requiere componentes y conectores que tienen propiedades que definen su tiempo de servicio, tiempos de retardo, repeticiones, las visitas, los tiempos de respuesta, etc. El análisis de planificación requiere componentes para especificar las prioridades, plazos, periodicidad, entre otros aspectos. Las herramientas de análisis tienen acceso a los modelos arquitectónicos, y puede ser notificado de cambios en el modelo para actualizar su análisis. Además, *AcmeStudio* permite el análisis para añadir puntos de vista de análisis y acciones específicas para la interfaz de usuario, a través de una combinación de Eclipse se pueden personalizar puntos de extensión para *AcmeStudio*. (Bradley Schmerl, 2003).

Este entorno de desarrollo de arquitectura está escrito como un *plugin* para Eclipse que apoya al ADL *Acme*, términos como *plugin* y Eclipse serán tratados más adelante.

Actualmente es provechoso el modelado de la arquitectura de Cedrux con la herramienta *AcmeStudio*. Esta herramienta brinda facilidades en cuanto a los elementos que maneja, como es el caso de componentes, conectores, puertos, roles, etc. Sin embargo, como se planteaba en la introducción del presente trabajo esta resuelve solamente una parte de

los problemas identificados por los arquitectos del centro durante la etapa de análisis en el proceso de las verticalizaciones, ya que la misma hace uso de la lógica de predicado como lenguaje formal mediante la definición de reglas.

El lenguaje basado en la lógica de predicados de *Acme-Armani* le permite realizar análisis de consistencia sobre los modelos realizados en *AcmeStudio*. Sin embargo hay otros tipos de análisis que no tiene incluidos de forma natural. Lo anterior no es del todo inconveniente, puesto que está basado en una arquitectura que permite la extensión en este aspecto. Es decir, permite que le sean agregadas funcionalidades para realizar análisis.

Una deficiencia de esta herramienta es que no muestra información suficiente relacionada con el análisis que efectúa, al mismo tiempo que no permite guardar reportes del resultado del mismo (análisis) para una posterior consulta.

Análisis del estudio sobre las herramientas analizadas

En este acápite se hace un análisis de las herramientas estudiadas, y los aportes que brinda cada una a la investigación. El resultado del estudio se muestra en la **Tabla 1**:

Herramientas	**Aporte**		**Consideraciones**
	Tipo de análisis	**Técnica**	
Aladdin	Dependencia de análisis estructural (análisis estático).	– Encadenamiento.	Esta herramienta no es soportada desde hace varios años. Por lo que cualquier información relacionada con la misma es obsoleta, sin embargo esto no influye en la investigación puesto que la técnica se mantiene invariable y puede ser aprovechada en el presente trabajo.

IDM	Dependencia de análisis estructural (análisis estático).	– Matriz de dependencias. – Diagramas de dependencias	Esta herramienta permite documentar las dependencias del modelo que se analiza. Recibe como entrada la arquitectura modelada en UML.

Tabla 1. Análisis de las herramientas analizadas.

El estudio de la herramienta *AcmeStudio* se realiza para mostrar lo que se desea extender a esta herramienta, enfocado a las necesidades de Cedrux/Sauxe, tal como lo muestra la **Tabla 2**:

AcmeStudio	
Problemas	**Soluciones**
- No muestra información suficiente del análisis que realiza. - La información que muestra no se encuentra en un formato adecuado. - No permite guardar reportes del análisis que realiza. - Resulta complejo realizar un análisis manual de las dependencias del modelo arquitectónico.	- Identificar dependencias estructurales basadas en la arquitectura modelada. - Mostrar información completa referente a las dependencias identificadas. - Generar y guardar reportes en formato pdf. - Guardar la matriz de dependencia generada a través del modelo en una hoja de cálculo. Esto tributa al problema de la actualización de los modelos actuales de la arquitectura de Cedrux.

Tabla 2. *AcmeStudio*: problemas y soluciones asociados.

1.5 Técnicas de análisis dependencia

Chaining

El encadenamiento (*Chaining*) es una técnica de análisis de dependencia que tiene como objetivo reducir las porciones de una arquitectura que deben ser examinados por un arquitecto con un propósito, como las pruebas o la depuración. En el encadenamiento, los

enlaces representan las relaciones de dependencia que existen en una especificación de la arquitectura, produciendo una cadena de dependencias que se puede seguir durante el análisis. (Judith A. Stafford, 1997).

Los eslabones individuales de la cadena dentro de una cadena se asocian a los elementos arquitectónicos que están directamente relacionados mientras que una cadena de dependencias incluye a las asociaciones entre los elementos arquitectónicos que están indirectamente relacionados. El encadenamiento es el proceso de aplicación de los algoritmos de la dependencia a las descripciones de la arquitectura con el fin de crear estos conjuntos de elementos relacionados. (Judith A. Stafford, 1997).

El encadenamiento (*Chaining*), es una técnica de análisis de dependencia estática. (Judith A. Stafford, 1998) Esta técnica de análisis se utiliza a nivel de implementación para ayudar a la optimización de programas, comprobación de anomalías, la comprensión del programa, las pruebas y depuración.

Diagramas de dependencia

Una dependencia entre dos entidades A y B es una relación (A, B) tal que si B sufre un cambio, A puede sufrir un cambio. Esta definición evidencia el alto nivel de abstracción de la relación: Cualquier cambio en B puede provocar un cambio en A. (Fontdevila, 2007)

Este alto nivel de abstracción le imprime a la dependencia algunas características importantes: No implica una connotación estructural (la relación no requiere que un elemento contenga a otro ni que sea de la misma vista), no describe necesariamente un cambio de forma (es decir que los cambios pueden no ser aparentes o estar en la implementación), y puede representar aspectos puramente semánticos (A y B pueden seguir vinculados pero con la esencia de esa vinculación alterada). (Fontdevila, 2007)

Matriz de dependencia

En esta investigación se trata el tema de: matriz de dependencias como una técnica de análisis, donde el objetivo del arquitecto de software es documentar las relaciones entre los componentes del sistema, representando la arquitectura como un grafo de dependencias.

Con el fin de comprender el proceso de creación de una matriz de dependencias, resulta provechoso el estudio realizado por el Msc. Osmar Leyet Fernández en (Fernández, septiembre de 2011), donde define una matriz bidimensional de integración (dependencias) entre los componentes del sistema Cedrux, especificándose los puntos de

contactos entre cada uno de estos componentes y mediante qué servicios ocurre la interacción.

Consideraciones sobre las técnicas de análisis de dependencias estudiadas

- La matriz de dependencias mencionada anteriormente es buena y aplicable para documentar dependencias entre los componentes de una AS. Sin embargo muestra limitantes con respecto a la actualización de la misma, debido que resulta engorroso realizar este proceso manual.
- En la implementación se utiliza el encadenamiento para realizar análisis al modelo. Esta técnica permite desplazarse hacia adelante y hacia atrás a través de la cadena de dependencias facilitando la implementación de las funcionalidades a desarrollar en las cuales se utiliza una estructura de datos que permita hacer uso de esta técnica.

1.6 Arquitectura de Cedrux

Se hace necesario mostrar una descripción del contexto en el que se aplica la solución especificando los elementos que se tienen en cuenta dentro de este. En este caso, se exponen de manera breve algunos aspectos de la arquitectura de Cedrux con el fin de esclarecer el significado que tienen para el presente trabajo.

La arquitectura de Cedrux define siete (7) vistas arquitectónicas. En el presente trabajo para el análisis que se propone inciden las vistas de integración y la de componentes. Estas brindan elementos como:

- La vista de integración que como se expone en (Fernández, septiembre de 2011), esta se encarga de los procesos de integración a nivel de sistema. Esta integración puede ser interna (entre componentes de un mismo subsistema) y externa (entre distintos subsistemas).
- La vista de componentes, que como se expone en (Fernández, septiembre de 2011), se encarga de las definiciones de la taxonomía de componentes. Es decir se conceptualizan los tipos de componentes, se especifican sus características, además de la composición interna de cada uno de ellos.

De las clasificaciones de componentes que aparecen en *(Fernández, septiembre de 2011)*, en este trabajo se tiene en cuenta la referente a la estructura de empaquetamiento

puesto que esta clasificación es la que se asocia directamente con la estructura de componentes.

De acuerdo con esa clasificación, existen dos tipos de componentes:

1. Componente simple: abstracción de parte o la totalidad de determinadas áreas funcionales, reglas del negocio o del sistema, definiciones tecnológicas implementadas o flujos definidos como parte del proceso de configuración del sistema, es la menor unidad existente en la estructura de empaquetamiento del sistema, autónomo en su funcionamiento y capaz de implementar interfaces o contratos de comunicación. *(Fernández, septiembre de 2011)*
2. Componente contenedor: En el marco de este trabajo se amplía la definición dada por *(Fernández, septiembre de 2011)* como sigue: Elemento conformado por componentes simples y/o componentes contenedores que interactúan entre ellos, Los elementos que integran un componente contenedor pueden interactúan con otros elementos.

La interacción entre estos componentes se realiza mediante un conector. El conector que se define en Cedrux se conoce como ioc. Este se utiliza de dos maneras:

1. ioc-externo: Este se usa para la interacción entre componentes que no están incluidos en su nivel jerárquico dentro de un mismo componente.
2. ioc-interno: Este se usa para la interacción entre componentes que descienden de un mismo componente en algún nivel de la jerarquía a la que pertenecen.

Cada uno de estos componentes tiene características que permiten realizar análisis sobre ellos con el fin de tomar decisiones. Estas características además de otras son las siguientes:

- Servicios que brinda el componente.
- Restricciones sobre las interacciones.
- Componentes dependientes del mismo.

Estas características influyen en la implicación que tiene en la estructura una modificación ya sea eliminar o reutilizar un componente. A partir de ellas se pueden tomar decisiones arquitectónicas para evitar un impacto negativo en el proceso de verticalizaciones.

A partir de la conceptualización de verticalización expuesta en la introducción y la descripción de las vistas arquitectónicas en las que se incide, así como el problema

formulado y el estudio realizado en este capítulo, se realiza un análisis para exponer las pretensiones de este trabajo.

Pretensiones de la herramienta a desarrollar

- Automatizar el análisis de impacto de cambios sobre la estructura de componentes.
- Desarrollar una herramienta que realice análisis de dependencias que utilice técnicas como: encadenamiento y matriz de dependencias.
- Definición de los elementos que se modelan:
    - Puertos: se definen dos tipos de puertos (salida y entrada).
    - Conexión: se definen dos tipos de conexiones (*Binding* y *Attachment*).
- Integrar la herramienta en un *plugin* para eclipse que extienda funcionalidades de AcmeStudio.
    - Mostrar reportes que permitan tomar decisiones con respecto a las implicaciones de las modificaciones en la estructura de componentes antes mencionadas en este epígrafe.
    - Generar una matriz que muestre la interacción entre los componentes, de manera que apoye el análisis de dependencia a la estructura de componentes. Guardar la matriz para ayudar en la actualización del artefacto Matriz de integración propuesta por Msc. Osmar Leyet Fernández.

En el resto de este capítulo se define qué se va a utilizar (herramientas, técnicas, metodología) para cumplir las pretensiones anteriores.

1.7 Herramientas y lenguajes utilizados para el desarrollo del *plugin*

Ambiente de desarrollo integrado. Eclipse

Metafóricamente, Eclipse es como una tienda para herreros, donde no solamente se hacen productos, sino que además se hacen las herramientas para hacer los productos. Cuando se descarga el Eclipse SDK, se obtiene un equipo de instrumentos para desarrollo en Java o *Java Development Toolkit* (JDT) para escribir y depurar programas en Java; además se obtiene un ambiente de desarrollo de *plugin*, *Plugin Development*

Enviroment para heredar de Eclipse. Si todo lo que se quiere es un Ambiente de Desarrollo Integrado (IDE[9]) para Java, no se necesita nada además que el JDT. Esto es

para lo que la mayoría de las personas usan Eclipse. (Lilian Teresa Castro Mecías, Mayo 2008).

Aunque Eclipse es escrito en Java y su principal uso es como IDE para Java, este es un lenguaje neutral. El soporte para desarrollo en Java es proveído por un componente enchufado o *plugin*, pero además están disponibles *plugins* para otros lenguajes, como C/C++, Cobol, C#. (Lilian Teresa Castro Mecías, Mayo 2008).

Arquitectura de Eclipse:

Eclipse sirve como IDE Java y cuenta con numerosas herramientas para desarrollo de software. A la plataforma base de Eclipse se le pueden añadir extensiones (*plugins*) para incrementar su funcionalidad. (Gallardo, 2002).

El entorno y espacio de trabajo son partes indispensables de los *plugins* de la Plataforma Eclipse, que proporcionan los puntos de extensión utilizados por la mayoría de los *plugins*, como se muestra en la **Figura 2**. Un *plugin* requiere un punto de extensión para conectar a fin de funcionar. (Gallardo, 2002).

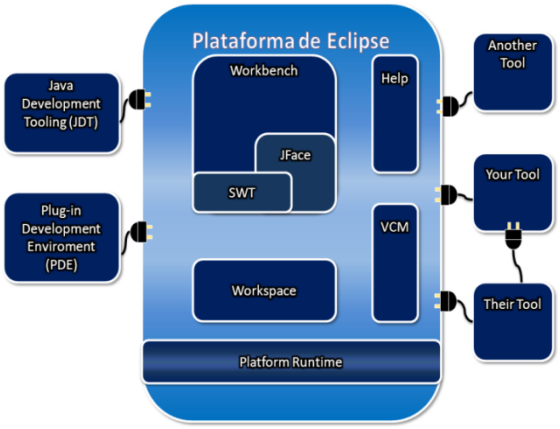


Figura 2. Plataforma de Eclipse (Hervis, 2010)

El *Workbench* es la interfaz gráfica de usuario (GUI) del Eclipse, siendo el componente de la plataforma responsable de la presentación y de la coordinación de la misma, posee

[9] IDE: Entorno de desarrollo integrado, pueden ser aplicaciones por sí solas o pueden ser parte de aplicaciones existentes es decir, consiste en un editor de código, un compilador, un depurador y un constructor de interfaz gráfica de usuario.

puntos de extensión para extender funcionalidades de la GUI para otros *plugin*. (Gallardo, 2002).

El *plugin Team* facilita el soporte del sistema de control de versiones (o administración de configuración) para manejar los recursos de usuario en un proyecto y define el flujo de trabajo necesario para salvar y recuperar del repositorio. (Gallardo, 2002).

El componente *Help* es un sistema de documentación extensible. Las herramientas provistas pueden adicionar documentación en formato HTML y, usando XML, define la estructura de navegación. (Gallardo, 2002).

Análisis del estudio de Eclipse

- Es un IDE con muchas distribuciones y soporta una gran cantidad de lenguajes.
- Es una herramienta de código abierto. Corre en una gran cantidad de sistemas operativos incluyendo Windows y Linux.
- Eclipse provee la utilidad de comenzar el programa con los *plugins* especificados, permitiendo acceder a distintas aplicaciones sin necesidad de levantar todas a la vez al momento de ejecutarlo.
- La plataforma Eclipse está construida en base a *plugins*. Este mecanismo permite desarrollar, integrar y correr nuevos *plugins*.

Plugin y puntos de extensión

Un *plugin* es un módulo de hardware o software que añade una característica o un servicio específico a un sistema más grande.

Es la unidad mínima de funcionalidad de Eclipse, que puede ser distribuida de manera separada. Herramientas pequeñas se escriben como un único *plugin*, mientras que en las complejas la funcionalidad está en varios de éstos. La plataforma Eclipse consiste aproximadamente, en 100 *plugins* trabajando juntos. A los límites entre ellos, que permiten conectar un *plugin* a otro, se le denomina: puntos de extensión. (González, 2009). Los puntos de extensión son el mecanismo por el cual un *plugin* puede adicionarle funcionalidades a otro.

Estructura de un plugin

Cada *plugin* posee un manifiesto del *plugin*, en el cual se describe su interconexión con otros *plugins*, o sea, se declara un número determinado de puntos de extensión y a su

vez, se exponen las extensiones que se han realizado de otros puntos de extensión definidos por otros *plugins*. El manifiesto del *plugin* está representado por un par de ficheros (ver **Figura 3**): el MANIFEST.MF y el *plugin*.xml. A continuación se hace una breve descripción de los mismos:

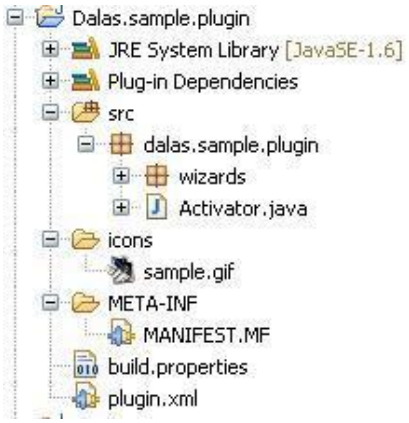


Figura 3. Arquitectura de *plugin.* (Hervis, 2010)

MANIFEST.MF: este fichero es generado dentro del directorio META-INF cuando es creado un proyecto *plugin*. Es un manifiesto del paquete OSGi[10] donde se describen las dependencias que posee el *plugin* con otros *plugins* en tiempo de ejecución y sus atributos, como son: el nombre que posee, el proveedor, la versión del mismo, entre otros.

plugin.xml: se encuentra en la misma raíz del proyecto y consiste en un archivo de formato XML que describe todas las extensiones realizadas por el *plugin* y declara, además, los puntos de extensión que el propio *plugin* ha definido. (González, 2009).

Además de estos elementos, en el directorio de un *plugin* se encuentran otros archivos y carpetas como son:

- *icons*: directorio en el cual se encuentran las imágenes o iconos que va a utilizar el *plugin*.
- *plugin.properties*: contiene cadenas de caracteres que son referenciadas en el *plugin*.xml.
- *about.html*: archivo en formato HTML que es utilizado para mostrar informaciones relacionadas con licencias.

Lenguaje de programación: Java

10 OSGi: Se refiere a Open Services Gateway Initiative (Iniciativa de enlace de servicios abiertos), es una corporación independiente, sin ánimo de lucro que trabaja para definir y promover especificaciones abiertas de software.

Java es un lenguaje multiplataforma, lo que significa que no está enlazado a un sistema operativo en concreto y los programas desarrollados con él mismo funcionan correctamente tanto en Windows como en Linux, este solo necesita de la máquina virtual. Este un lenguaje de programación orientado a objetos. El lenguaje posee una sintaxis clara y bien definida similar a la del lenguaje de programación C y C++, pero tiene un modelo de objeto más simple y elimina herramientas de bajo nivel, que suelen inducir a muchos errores, como la manipulación directa de punteros o memoria. Reúne capacidades funcionales para la creación del *plugins*, también existe mucha información, documentación e infinidad de ejemplos disponibles en Internet, es de la filosofía de Software Libre, pues no hay que pagar por licencias o patentes para obtener las actualizaciones.

Visual Paradigm

Visual Paradigm es un producto que facilita a las organizaciones diseñar visualmente y con diagramas, integrar y desplegar sus aplicaciones empresariales y sus respectivas bases de datos. La herramienta ayuda al equipo de software a destacarse en el proceso de modelado, construcción y despliegue, maximizando y acelerando las contribuciones del equipo y los individuos. Además, soporta un set de lenguajes para la Generación de Código y la Ingeniería Inversa en *Java*, C++, PHP, entre otros. ()

1.8 Metodologías de desarrollo de software

El tema de la metodología a seleccionar es uno de los más complicados en cualquier proceso de desarrollo de software, ya que son muchos los factores que se deben incluir en el análisis. En el marco de este trabajo se analizan RUP, XP y *OpenUp*, de manera que se pueda seleccionar la más adecuada, con el objetivo de guiar el análisis, implementación y documentación del proyecto a desarrollar.

Metodología RUP

*Rational Unified Process (*RUP, del inglés*)* es un proceso de desarrollo de software, y se basa en *Unified Modeling Language (*UML, del inglés*)* para la realización de los modelos. El objetivo fundamental de esta metodología es: asegurar que el software propuesto se haga con la calidad requerida y en el tiempo pactado (Pérez, Julio de 2007).

RUP se caracteriza por ser dirigido por casos de usos, centrado en la arquitectura, iterativo e incremental, lo que significa que cada fase se desarrolla en iteraciones que

involucran actividades de todos los flujos de trabajo, obteniendo un producto que irá creciendo incrementalmente en cada iteración, por lo que se considera una metodología pesada, cuya duración de fases y requerimientos de documentación son muy grandes para proyectos pequeños o con poco personal. (González, Mayo de 2009).

Metodología XP

XP (*Extreme Programming*) es una metodología ágil, orientada directamente al objetivo y basado en las relaciones interpersonales y en la velocidad de reacción (Molpeceres, Diciembre de 2002), es principalmente para proyectos cortos y mediano alcance, integrados por equipos pequeños.

XP se basa en retroalimentación continua entre el cliente y el equipo de desarrollo, comunicación fluida entre todos los participantes, simplicidad en las soluciones implementadas y coraje para enfrentar los cambios. Esta metodología se define como especialmente adecuada para proyectos con requisitos imprecisos y muy cambiantes, y donde existe un alto riesgo técnico. (Maite Rodríguez Corbea, julio 2007).

Metodología OpenUp

OpenUP es un Proceso Unificado ligero, antes llamado BUP (Iglesias., Junio de 2007) y es una versión de RUP optimizada por IBM (Iglesias., Junio de 2007). *OpenUp* aplica una aproximación iterativa incremental dentro de un ciclo de vida estructurado. Posee una filosofía ágil, se enfoca en la naturaleza colaborativa del desarrollo de software. Está concebido para proyectos pequeños, se caracteriza por ser una metodología basada en casos de usos, centrada en la arquitectura, iterativo e incremental. Contiene fundamentalmente un conjunto de simplificaciones de roles, actividades, artefactos y guías. (Iglesias., Junio de 2007).

El esfuerzo personal en un proyecto *OpenUP* está organizado en **micro-incrementos**. Esto representa pequeñas unidades de trabajo que producen un paso de progreso del proyecto cuantificable y constante. (Eclipse, 2012).

OpenUP divide el proyecto en iteraciones: planeadas en intervalos de tiempos fijos usualmente medido en semanas. Las iteraciones concentran al equipo en entregas incrementales de valor para los involucrados en una manera predecible. Esto se realiza definiendo finas y bien engranadas tareas de una lista de elementos de trabajo. (Eclipse, 2012).

OpenUP estructura el ciclo de vida del proyecto en cuatro fases: Inicio, Elaboración, Construcción y Transición. El ciclo de vida del proyecto provee a los involucrados y a los miembros del equipo, visibilidad y puntos de decisión a lo largo de todo el proyecto. Esto permite la vigilancia eficaz, y permite tomar decisiones de qué hacer o no, en un tiempo apropiado. (Eclipse, 2012).

Importancia de la metodología OpenUp para la investigación

- Permite detectar errores tempranos a través de un ciclo iterativo.
- Es apropiado para proyectos pequeños y de bajos recursos.
- Permite disminuir las probabilidades de fracaso en los proyectos pequeños e incrementar las probabilidades de éxito.
- Evita la elaboración de documentación, diagramas e iteraciones innecesarios requeridos en la metodología RUP.
- Por ser una metodología ágil tiene un enfoque centrado al cliente y con iteraciones cortas.
- Capacidad de respuesta a cambios de requisitos a lo largo del desarrollo.
- Entrega continua y en plazos breves de software funcional.
- Mejora continua de los procesos y el equipo de desarrollo.
- Trabajo conjunto entre el cliente y el equipo de desarrollo.

Valoraciones del estudio de las metodologías

En la actualidad existen muchas metodologías para el desarrollo del software, se tienen las tradicionales como RUP y otras ajustadas especialmente al control de procesos, estableciendo las actividades involucradas, los artefactos que se deben producir y las herramientas y notaciones que se usarán, por otra parte están las conocidas metodologías ágiles como *Extreme Programming* (XP), *OpenUp* entre otras que se centran especialmente en el cumplimiento de los objetivos y un desarrollo incremental del software con iteraciones muy cortas.

En el marco de este trabajo se toma una dirección hacia el uso de una metodología ágil por características como proyecto pequeño y de bajos recursos, elaboración de documentación e iteraciones cortas que permitan detectar errores tempranos. Teniendo en cuenta estas características, los autores deciden ajustarse a la metodología *OpenUP. OpenUp* es un proceso de desarrollo iterativo del software que es mínimo, completo, y extensible. El proceso es mínimo donde solamente el contenido fundamental es incluido;

es completo y puede ser manifestado como todo proceso para construir un sistema; extensible y puede ser utilizado como fundamento sobre el cual el contenido de proceso se pueda agregar o adaptar según lo necesitado (Carmina Lizeth Torres Flores, 2008). Esta metodología está diseñada para el desarrollo de proyectos pequeños, pero basada en las mejores prácticas del RUP, de esta manera se podrá desarrollar artefactos ligeros apropiados utilizando el lenguaje UML, incrementando así las probabilidades de éxito en función de costo, tiempo y alcance.

No se decidió utilizar la metodología XP ya que ésta le da poco significado al diseño inicial y a la documentación ocasionando que se tengan que hacer muchos cambios durante el desarrollo del proyecto de software.

Conclusiones parciales

En este capítulo se trataron algunas definiciones de AS, asumiendo finalmente como base para la investigación, la definición oficial que se encuentra en el estándar IEEE Std 1471-2000. También se estudiaron herramientas que realizan análisis arquitectónico, como es el caso de: Aladdin e IDM, de las cuales se tomaron como aporte las técnicas que estas utilizan para realizar análisis de dependencia. Se realizó un análisis crítico de algunas metodologías de desarrollo (XP, RUP y *OpenUp*). Como resultado del estudio los autores proponen *OpenUp* como metodología para guiar el proceso de desarrollo de la herramienta. Además, se estudió la herramienta *AcmeStudio*, con el fin de identificar las funcionalidades que necesitan ser extendidas a dicha herramienta, para solventar los problemas actuales que tiene el equipo de arquitectos durante las verticalizaciones de Cedrux.

Con este capítulo se da cumplimiento al primer objetivo específico planteado: Establecer el Marco Teórico de la investigación referente a las herramientas y técnicas para la realización de análisis arquitectónico, para sentar las bases de la propuesta de solución.

CAPÍTULO 2: Diseño e implementación.

2 Introducción

En el presenta capítulo se realiza el diseño e implementación del *Plugin* para Eclipse, se exponen los artefactos necesarios que fueron generados con la guía de la metodología *OpenUp* durante el desarrollo de la herramienta. Además se analizan las características del *plugin* que serán automatizadas, así como la descripción de las funcionalidades que brinda la herramienta. Finalmente se validará con la ayuda de métricas el diseño propuesto.

2.1 Propuesta del sistema

La herramienta propuesta pretende realizar análisis sobre la arquitectura de Cedrux, teniendo en cuenta la taxonomía de componentes del sistema. Dentro del análisis se manejan elementos como componentes, conexiones, restricciones, conectores, puertos y roles.

Debe además permitir integrarse con la herramienta gráfica de modelado *AcmeStudio*, mediante un *plugin* para el IDE Eclipse que se integre con *AcmeLib*. Finalmente la herramienta debe permitir que se genere la matriz de dependencias asociada al modelo de la arquitectura que se analiza. Además de mostrar reportes con los componentes que se ven afectados al ser modificada la estructura de componentes. Para la manipulación de los elementos del modelo, el sistema tiene como requisito, importar modelos para convertir los elementos en objetos.

2.2 Artefactos generados

La metodología *OpenUP* durante el ciclo de vida del desarrollo de un software genera una serie de artefactos. En este epígrafe se exponen los artefactos generados durante el desarrollo de la herramienta.

Documento Visión del plugin

Este artefacto proporciona una base de alto nivel para los requisitos principales previstos para el sistema. Define la solución técnica a desarrollar, la cual se expresa con la descripción de las necesidades de las partes interesadas y las limitaciones que dan una

visión general del razonamiento, de fondo, y el contexto de los requisitos detallados. (Eclipse, 2012). Ver en el **Anexo 1**, el **Link 2**.

Plan de Proyecto

El Plan de Proyecto describe un acuerdo inicial sobre la forma en que el proyecto logra sus metas. Este plan puede ser actualizado a medida que avanza el proyecto basado en la retroalimentación y los cambios en el medio ambiente. (Eclipse, 2012).

La metodología especifica que en cualquier tipo de proyecto se debe disponer de este artefacto inclusive para aquellos proyectos de tamaños pequeños, o aquellos cuya complejidad es reducida o limitada. Ver en el **Anexo 1**, el **Link 2**.

2.3 Requisitos del software

Las técnicas utilizadas para el levantamiento de requisitos de la herramienta de análisis propuesta fueron las entrevistas realizadas al cliente, además se realizaron talleres, espacio que fue oportuno para desencadenar tormentas de ideas. De las mismas surgieron los requisitos que se muestran y detallan a continuación.

Requisitos funcionales del Software

La herramienta propuesta cuenta con cuatro (4) requisitos funcionales y un (1) requisito funcional implícito del sistema:

- Importar modelo: Es un requisito implícito del sistema, interno y transparente para los usuarios.
- Generar matriz de integración.
    - Mostrar componente origen de la relación.
    - Mostrar componente destino de la relación.
    - Mostrar el servicio a través del cual se establece la relación.
- Guardar matriz.
- Mostrar reportes.
    - Listado de los componentes libres de dependencias.
    - Listado de los componentes dependientes directos de un componente.
    - Listado de los componentes dependientes indirectos de un componente.
    - Listado de los componentes condicionantes directos.
    - Listado de los componentes condicionantes indirectos.
    - Listado de los servicios que brinda un componente.

- Listado de los componentes afectados al eliminar un componente.
- Listado con los componentes que se necesitan si se reutiliza un componente.

➢ Guardar reportes.

La descripción de las funcionalidades se explica en el epígrafe 2.2.7.

Requisitos no funcionales del Software

Los requisitos no funcionales definen propiedades y cualidades que el sistema debe tener. Es decir, las características que harán del producto algo atractivo, usable, rápido o confiable.

En el marco de este trabajo, se considera necesario que la aplicación cumpla con los siguientes requisitos no funcionales:

Escalabilidad

➢ Se integrará con la herramienta AcmeStudio a través de un *plugin*.

➢ El sistema debe ser construido sobre la base de un desarrollo evolutivo e incremental, de manera tal que nuevas funcionalidades y requisitos relacionados con el análisis puedan ser incorporados afectando el código existente de la menor manera posible.

Usabilidad

➢ El sistema debe poder satisfacer las peticiones del cliente. Podrá ser usado por cualquier persona que tenga conocimientos básicos de arquitectura de software, especialmente los arquitectos del centro. Con la aplicación de la herramienta se agiliza el trabajo de los arquitectos durante el proceso de verticalización.

Software.

➢ JDK 1.5 o superior.

➢ Eclipse IDE 3.5.

2.4 Diseño de la solución

2.4.1 Especificaciones de Casos de Uso y Modelo de Casos de Uso

Con estos artefactos se puede capturar, visualizar, establecer y explicar el comportamiento del sistema y su entorno para producir resultados de valor para los actores que interactúan con él.

Si no se cuenta con estos artefactos, la funcionalidad que el sistema debe soportar podría no ser clara, a menos que se utilicen formas alternativas de especificación de requisitos.

El modelo de casos de uso correspondiente a la solución del *plugin* propuesto se muestra en la **Figura 4**, a partir del este, se generó un artefacto con la especificación del mismo utilizando como guía, la plantilla definida por la metodología *OpenUp*. Para ver la especificación del requisito puede consultar el **Anexo 2.**

Diagrama de Caso de uso

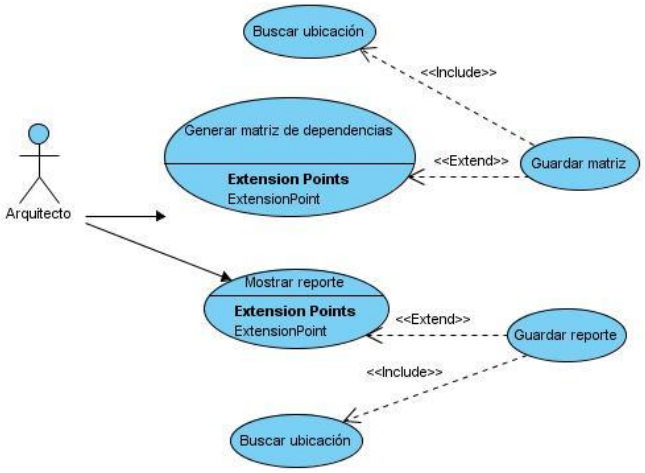


Figura 4. Modelo de Casos de uso.

Para un mayor entendimiento de los requisitos identificados se desglosó el modelo y se especificaron cada uno de estos generando, por cada requisito una especificación detallada del mismo.

1. **Generar matriz de integración:** En el Caso de uso se describe como generar una matriz partiendo de las dependencias entre los componentes representados en el modelo. Para ver la especificación del requisito puede consultar el **Anexo 3.**

2. **Guardar matriz de integración:** En el Caso de uso se describen los pasos a seguir por el actor para guardar la matriz de dependencias generada. Para ver la especificación del requisito puede consultar el **Anexo 4.**
3. **Mostrar reportes:** En el Caso de uso se describe cómo mostrar los reportes de las implicaciones de las dependencias entre componentes de la arquitectura. Para ver la especificación del requisito consultar el **Anexo 5**.
4. **Guardar reporte:** En el Caso de uso se describe el procedimiento para guardar los reportes en una dirección física, en formato pdf. Para ver la especificación del requisito consultar el **Anexo 6.**

2.4.2 Estructura del diseño

En este epígrafe se describe la realización de las funcionalidades requeridas del sistema y sirve como una abstracción del código fuente.

El diseño que se propone se fundamenta en las características expuestas anteriormente en este capítulo.

En la **Figura 5** se muestra la estructura básica que posee el *plugin*. El mismo posee cuatro (4) componentes: *Architecture, Views, Core y Model*. Estos componentes ofrecen las funcionalidades necesarias para realizar análisis de dependencias a la arquitectura que se describe con *AcmeStudio*.

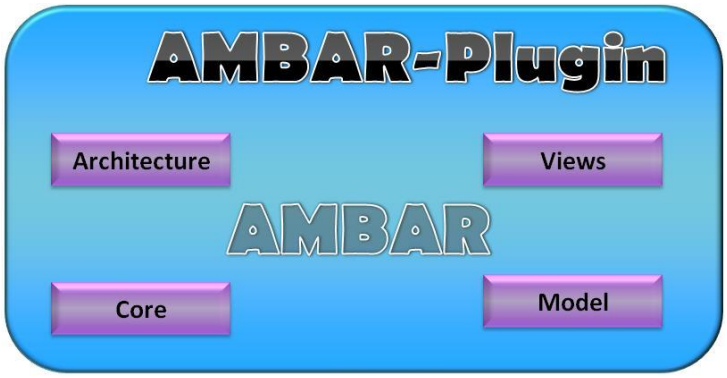


Figura 5. Estructura básica del *plugin*.

- *Views*: Es el paquete contenedor de las vistas del *plugin*. Permite al usuario visualizar las funcionalidades que agrega el *plugin* a la herramienta (**Generar matriz**, **Guardar matriz**, **Mostrar reportes** y **Guardar reporte**).
- *Core*: Es el paquete contenedor de las clases que proporcionan las estructuras de datos.

- *Architecture*: Es el paquete contenedor de las funcionalidades que provee el *plugin*.
- *Model*: Es un paquete contenedor de las entidades propias del dominio del modelo que se genera en *AcmeStudio* (**Componentes**, **Conectores**, **Puertos** y **Conexiones**). Además, contiene un paquete *Mapping*, que se encarga, entre otras tareas, de importar el modelo desde *Acme* hacia un entorno controlado por la herramienta de este trabajo.

Diagrama de paquetes

Los diagramas de Paquetes se usan para reflejar la organización de paquetes y sus elementos. Los usos más comunes de estos son; para organizar diagramas de casos de uso y diagramas de clases, además de ser grandes contenedores de clases.

Los elementos contenidos en un paquete comparten el mismo espacio de nombres, esto significa que los elementos contenidos en un mismo espacio de nombres específico deben tener nombres únicos.

Como otra característica de estos diagramas, es que cada paquete se debe identificar con un nombre único y opcionalmente mostrar todos los elementos dentro del mismo. Los paquetes están normalmente organizados para maximizar la coherencia interna dentro de cada paquete y minimizar el acoplamiento externo entre los paquetes.

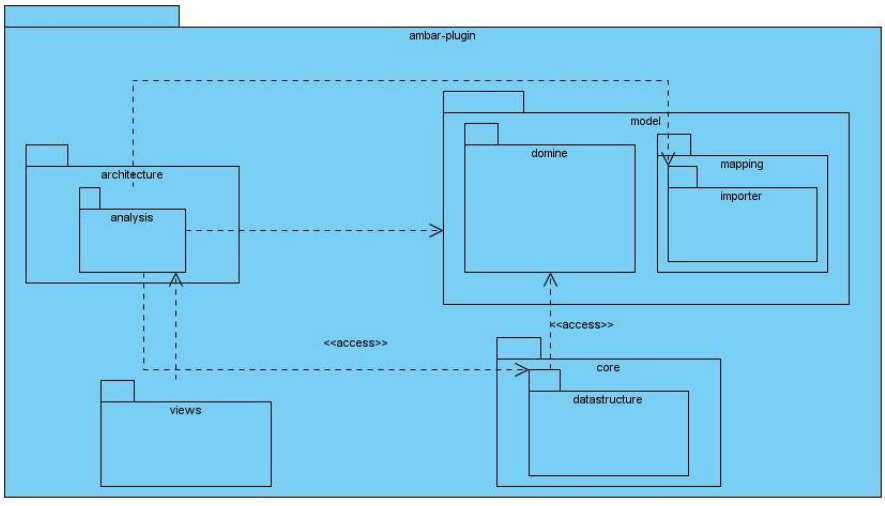


Figura 6. Diagrama de paquetes.

El diagrama de paquetes que se muestra en la **Figura 6** sirve para dar una vista de la organización interna de la estructura dentro del *plugin*, donde se denotan cuatro (4) paquetes fundamentales estos son: *Architecture, Model, Core y Views*.

- El paquete *Architecture::Analysis* es el contenedor de las funcionalidades que permite realizar los diferentes tipos de análisis, incluido el análisis de dependencia, entre los componentes modelados en la herramienta *AcmeStudio*.
- El paquete *Model::Domain* es el contenedor de las clases que describen los elementos del dominio.
- El paquete *Model::Domain::Mapping::Importer* es el contenedor de las clases encargadas de importar el modelo cargado desde *AcmeStudio*.
- El paquete *Core::DataStructure* es el paquete contenedor de las clases que proveen las estructuras de datos que se manipulan en la implementación del *plugin*, estas estructuras son: Estructura de grafo.
- El paquete *Views* es el contenedor de las vistas que muestra la aplicación, es decir, la herramienta muestra una vista para los diferentes reportes generados durante el análisis de dependencia, además de visualizar una matriz, representando los componentes del modelo y la relación entre los mismos.

Diagrama de clases

Un diagrama de Clases representa las clases que serán utilizadas dentro del sistema y las relaciones que existen entre ellas. Permitiendo visualizar las relaciones entre las clases que involucran el sistema, las cuales pueden ser asociativas, de herencia, de uso, etc. Un diagrama de clases está compuesto por los siguientes elementos: Clase: atributos, métodos y visibilidad. Relaciones: Herencia, Composición, Agregación, Asociación y Uso.

Los diagramas de clases que se muestran en la **Figura 7**, **Figura** 8, **Figura 9, Figura 10** y **Figura 11**, en la representación de las clases se aprecian dos colores para resaltar las clases del paquete que se está modelando de las clases que pertenecen a otros paquetes. Las clases de color azul representan las clases que son propias del paquete y las de color verde son las clases que pertenecen a otros paquetes.

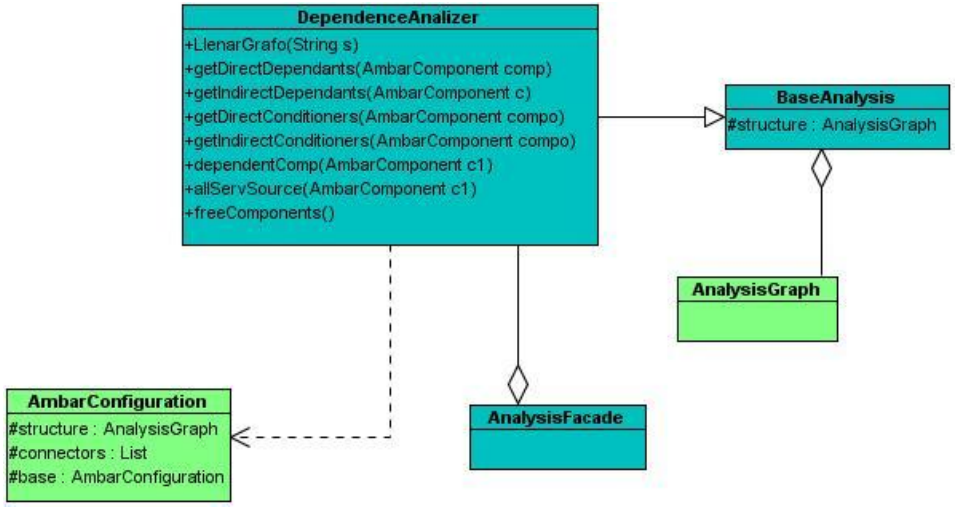


Figura 7. Diagrama de clases asociado al paquete *Architecture::Analysis* y su relación con clases de otros paquetes.

Elementos representados	Descripción
DependenceAnalizer	Clase que incluye todas las funcionalidades referentes al análisis de dependencias entre componentes.
BaseAnalysis	Esta es la clase que contiene el grafo.
AnalysisFacade	Esta clase se define como fachada de la clase *DependenceAnalizer*.

Tabla 3. Descripción del diseño de clases asociadas al paquete *Architecture::Analysis Analysis.*

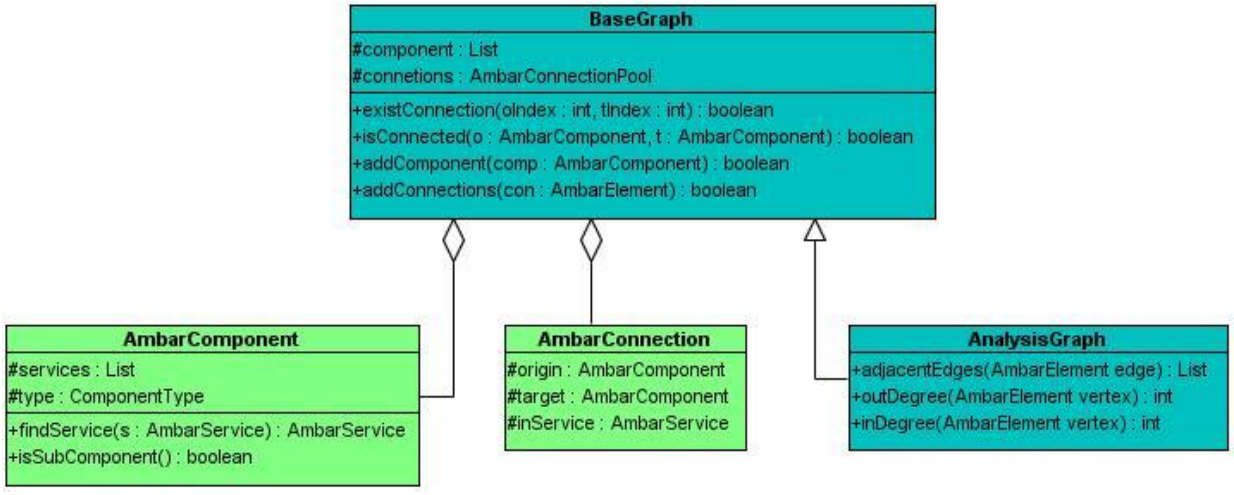


Figura 8. Diagrama de clases asociado al paquete *Core::DataStructure Analysis* y su relación con clases de otros paquetes.

Elementos representados	Descripción
BaseGraph	Clase encargada estructurar el grafo.
AnalysisGraph	Esta clase contiene métodos que permiten realizar análisis sobre el grafo.

Tabla 4. Descripción del diseño de clases asociadas al paquete *Core::DataStructure Analysis.*

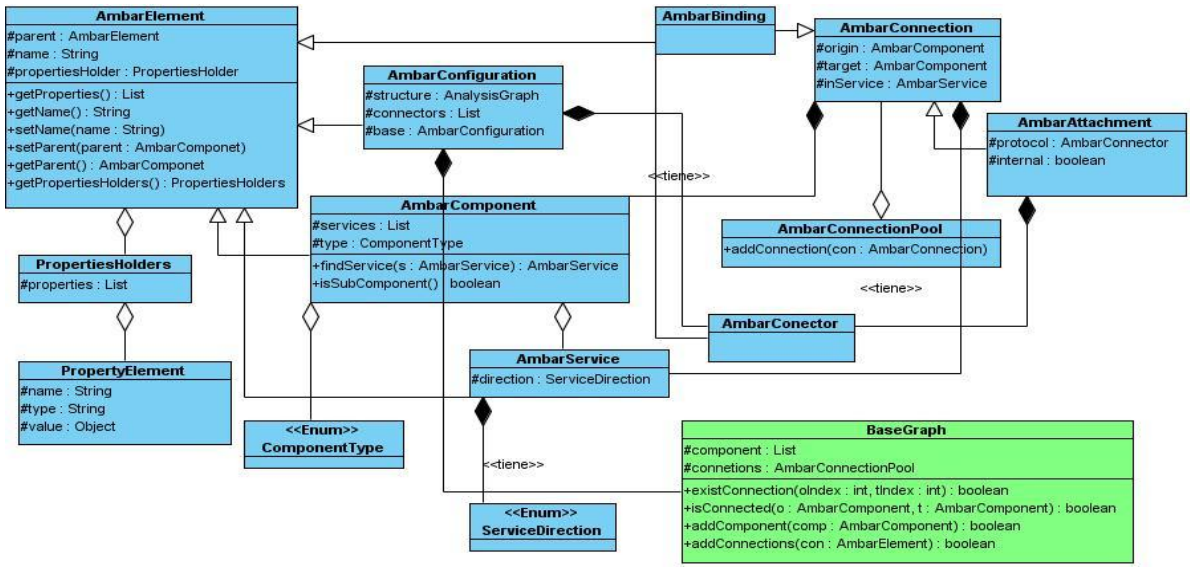


Figura 9. Diagrama de clases asociado al paquete *Model::Domain Analysis* y su relación con clases de otros paquetes.

Elementos representados	Descripción
AmbarConfiguration	Esta clase provee una configuración a los elementos propios del modelo que se crea en *Acme* y que se importa al Eclipse.
AmbarComponent	Clase que contiene los atributos que describen el objeto componente.
ComponentType	Esta clase es un *enum* que contiene los tipos de componentes.
AmbarService	Clase que contiene los atributos que describen el objeto servicio.

AmbarConnector	Clase que contiene los atributos que describen el objeto *IoCConnector*. El cual representa el tipo de conector que contiene el modelo.
AmbarConnection	Clase que contiene los atributos que describen el objeto *Connetion*.
AmbarElement	Esta clase es una generalización de los elementos provenientes del modelo que se importa desde *Acme*.
AmbarAttachment	Clase que representa un tipo de conexión.
AmbarBinding	Clase que representa un tipo de conexión.
AmbarConnectionPool	Clase que contiene los atributos que describen el objeto *ConnectionPool.*
ServiceDirection	Esta clase es un *enum* que contiene los tipos de direcciones que tiene un servicio.
PropertiesHolders	Esta clase contiene la lista de propiedades de los elementos del dominio.
PropertyElement	Esta clase define las propiedades de los elementos propios del dominio.

Tabla 5. Descripción del diseño de clases asociadas al paquete *Model::Domain*.

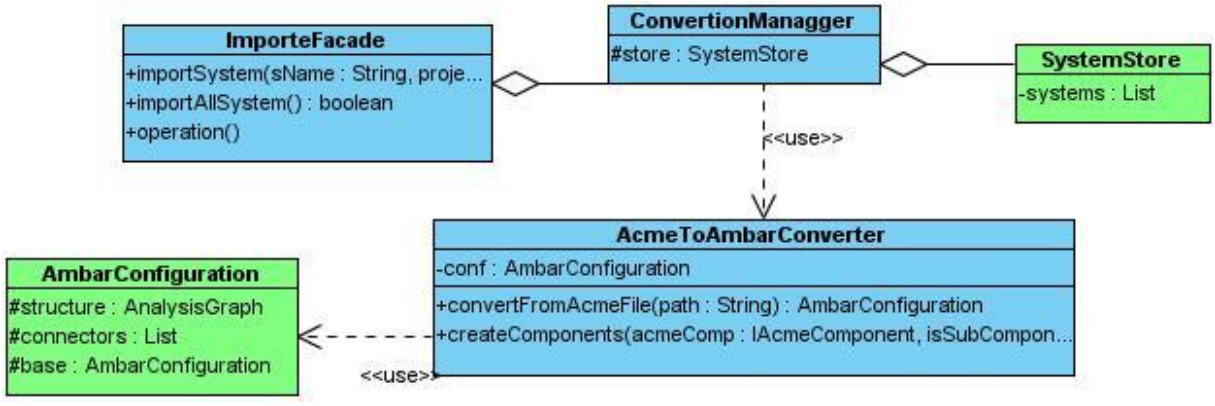


Figura 10. Diagrama de clases asociado al paquete *Model::Domain::Mapping::Importer Analysis* y su relación con clases de otros paquetes.

Elementos representados	Descripción
ImporterFacade	Esta clase es la fachada de la clase *ConvertionManagger*.
AcmeToAmbarConverter	Esta clase es la encargada de importar el modelo creado en *Acme* hacia el Eclipse.
ConvertionMannager	Es la clase que manipula los sistemas que contiene el modelo de *Acme*.

Tabla 6. Descripción del diseño de clases asociadas al paquete *Model::Domain::Mapping::Importer*.

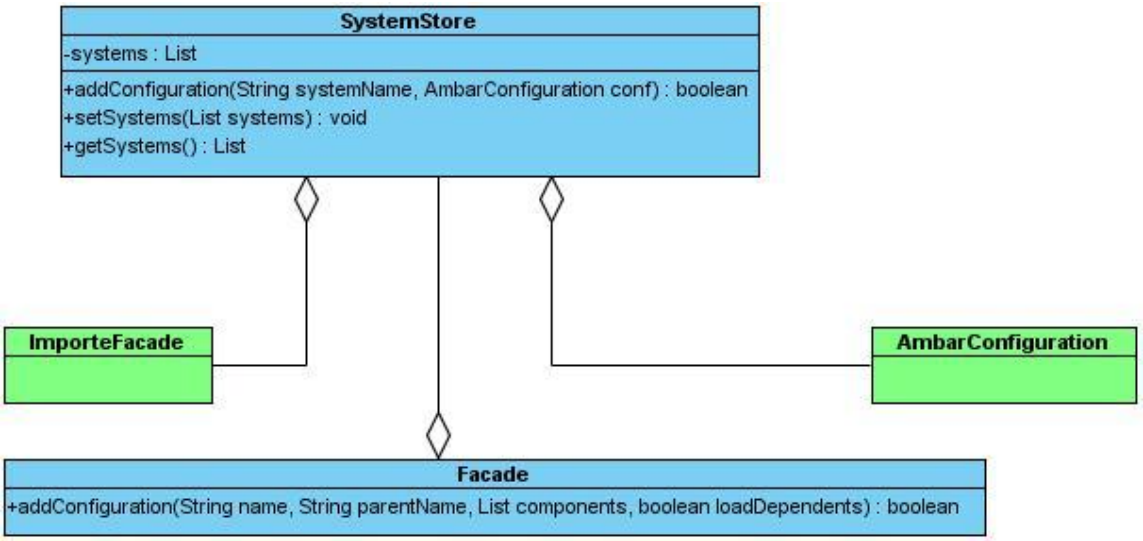


Figura 11. Diagrama de clases asociado al paquete *Architecture::DataSource Analysis* y su relación con clases de otros paquetes.

Elementos representados	Descripción
SystemStore	Esta clase es la encargada de estructurar un sistema.
Facade	Esta clase es la facha de la clase *SystemStore*.

Tabla 7. Descripción del diseño de clases asociadas al paquete *Architecture::DataSource*.

2.5 Patrones de diseño empleados

Patrones de asignación de responsabilidades

GRASP: (*General Responsability Assignment Software Patterns*) se encargan de asignar responsabilidades a los objetos en sentido general, las principales responsabilidades son conocer (atributos, relaciones con otros objetos) y hacer (tareas que debe cumplir cada objeto). (Prieto, 2009)

Entre los principales patrones GRASP utilizados se encuentran:

Experto: Se puso en práctica en el paquete con las clases del paquete *domain*, con el uso de clases que poseen responsabilidades específicas a cumplir de acuerdo con la información que manejan, el paquete cuenta con clases que poseen funciones concretas de acuerdo con la información que gestiona.

Se conserva el encapsulamiento, ya que los objetos se valen de su propia información para hacer lo que se les pide. El comportamiento se distribuye entre las clases que cuentan con la información requerida alentando con ello definiciones de clases sencillas y más cohesivas que son fáciles de comprender y mantener.

Patrón Creador: Permite asignar quien debería ser el responsable de la creación de una nueva instancia de alguna clase. Guía la asignación de responsabilidades relacionadas con la creación de objetos. El propósito general de este patrón es encontrar un creador que debemos conectar con el objeto producido en cualquier evento. Se puso en práctica en la clase *ConvertionManagger* la cual se encarga de crear un objeto *SystemStore* para posteriormente manipularlo.

Brinda soporte de bajo acoplamiento, lo cual admite menos dependencias con respecto al mantenimiento y mejores oportunidades de reutilización. Es probable que el acoplamiento no aumente, pues la clase creada tiende a ser visible a la clase creador, debido a las asociaciones actuales que llevaron a elegirla como el parámetro adecuado.

Alta cohesión: Existe afinidad entre cada clase y los métodos que implementan, estas poseen responsabilidades vinculadas acordes a la información que controlan. Su utilización mejora la claridad y facilidad con que se entiende el diseño, simplifica el mantenimiento y las mejoras de funcionalidad, generan bajo acoplamiento, soporta mayor capacidad de reutilización. Este patrón se pone de manifiesto en la mayoría de las clases porque cada una es capaz de realizar sus responsabilidades sin la utilización de las demás.

GoF: (*Gang of Four*) llamados así por los cuatro autores del libro Patrones de Diseño que recoge alrededor de veintitrés (23) patrones de los más utilizados. Están clasificados según su propósito:

- ✓ Creación: Tratan la creación de instancias.
- ✓ Estructura: Tratan la relación entre clases, la combinación de clases y la formación de estructuras de mayor complejidad.
- ✓ Comportamiento: Tratan la interacción y cooperación entre clases. (Prieto, 2009)

Fachada: Muestra cómo gestionar un sistema a través de un objeto representativo. Se puso en práctica en el componente *mapping::importer* como se muestra en la **Figura 12**:

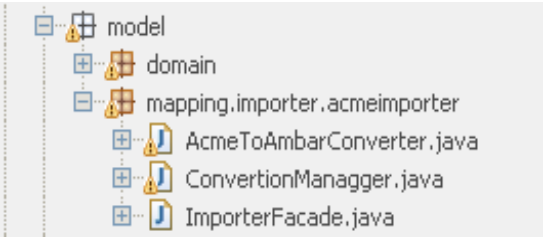


Figura 12. Representación del patrón fachada.

Bajo acoplamiento: es la idea de tener las clases lo menos ligadas entre sí que se pueda. De tal forma que en caso de producirse una modificación en alguna de ellas, tenga la mínima repercusión posible en el resto de clases, potenciando la reutilización, y disminuyendo la dependencia entre las clases.

Durante todo el trabajo con las clases del diseño se manifestó este patrón, pues no existen relaciones en 9 de las 32 clases y 9 con una sola relación de dependencia lo que significa que existe poca dependencia y cada clase realiza sus funciones sin necesitar de otra.

2.6 Validación del diseño.

En el presente epígrafe se exponen los procedimientos empleados para la validación del diseño propuesto. Se presentan las métricas Tamaño operacional de clase (TOC) y Relaciones entre clases (RC), para evaluar la calidad del diseño propuesto.

Las métricas TOC y RC incluyen medidas de los siguientes atributos de calidad:

> **Responsabilidad:** Responsabilidad que posee una clase en un marco conceptual correspondiente al modelado de la solución propuesta.

Complejidad del mantenimiento: Nivel de esfuerzo necesario para sustentar, mejorar o corregir el diseño de software propuesto. Puede influir significativamente en los costes y la planificación del proyecto.

Complejidad de implementación: Grado de dificultad que tiene implementar un diseño de clases determinado.

Reutilización: Significa cuán reutilizada es una clase o estructura de clase dentro de un diseño de software.

Acoplamiento: Dependencia o interconexión de una clase o estructura de clase respecto a otras.

Cantidad de pruebas: Número o grado de esfuerzo necesario para realizar las pruebas de calidad al producto (componente) diseñado. (Fernández González, 2010)

Métrica TOC: Tamaño operacional de clase.

Se refiere al número de procedimientos existentes en una clase. Determina una relación directa entre los atributos Responsabilidad y Complejidad de implementación, sin embargo establece una relación inversa entre estos últimos y el atributo Reutilización. Para ver el instrumento de medición de la métrica Tamaño operacional de clase, y la evaluación de los atributos consultar en el **Anexo 7**, las **Tablas 10** y **11** respectivamente**.**

Representación en % de la incidencia de los resultados obtenidos en el atributo **Responsabilidad**:

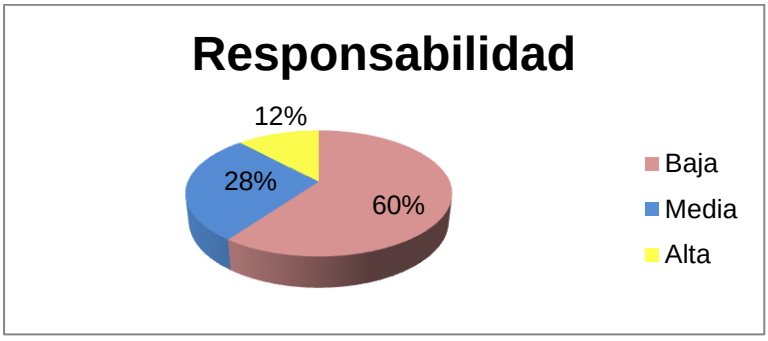


Figura 13. Representación del valor en % del atributo de calidad: Responsabilidad.

Representación en % de la incidencia de los resultados obtenidos en el atributo **Complejidad de Implementación**:

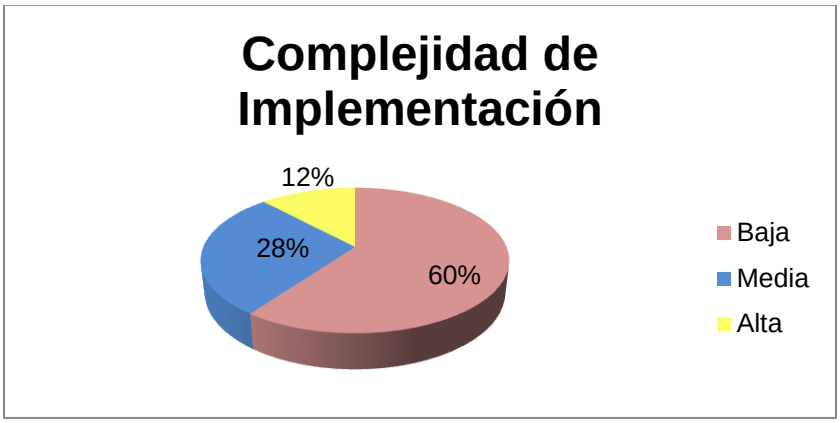


Figura 14. Representación del valor en % del atributo de calidad: Complejidad de Implementación.

Representación en % de la incidencia de los resultados obtenidos en el atributo **Reutilización**:

Figura 15. Representación del valor en % del atributo de calidad: Reutilización.

Análisis de los resultados obtenidos en la evaluación de la métrica TOC

Al aplicar la métrica TOC se determinó que la aplicación cuenta con 32 clases y un total de 200 procedimientos, con un promedio de 6 procedimientos por clase. La **Figura 13**, **Figura 14** y **Figura 15**, muestran los resultados de los atributos de calidad de Responsabilidad, Complejidad de implementación y Reutilización respectivamente.

El resultado demuestra que la mayoría de las clases presentan un bajo nivel de responsabilidad, complejidad y una alta reutilización siendo este diseño lo más simple posible, permitiendo una sencilla implementación y una amplia realización de pruebas con mayor facilidad.

Métrica RC: Relaciones entre clases.

Se refiere al número de relaciones de uso de una clase. Determinada por los atributos: Acoplamiento, Complejidad de mantenimiento, Reutilización y Cantidad de pruebas, existiendo una relación directa con los tres primeros e inversa con el último antes mencionado. Para ver instrumento de medición de la métrica Tamaño operacional de clase y la evaluación de los atributos, consultar en el **Anexo 7,** las **Tablas 12** y **13**.

Como resultado de la evaluación de la métrica RC se obtuvo lo siguiente:

Representación en % de la incidencia de los resultados obtenidos en el atributo **Acoplamiento**:

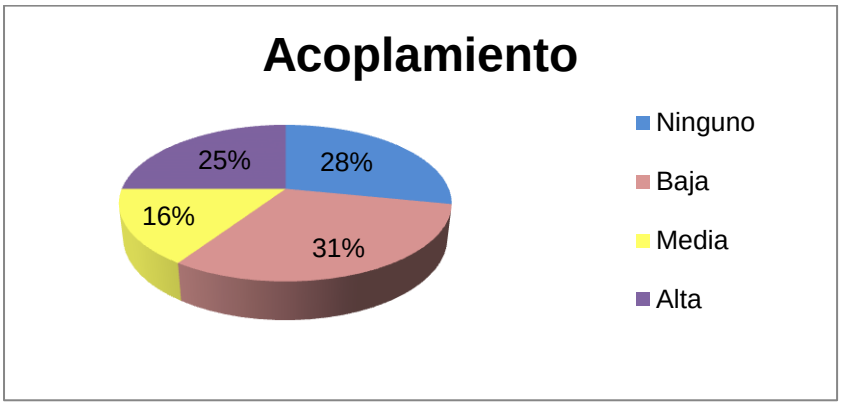


Figura 16. Representación del valor en % del atributo Acoplamiento.

Representación en % de la incidencia de los resultados obtenidos en el atributo **Complejidad de Mantenimiento**.

Figura 17. Representación del valor en % del atributo Complejidad de mantenimiento.

Representación en % de la incidencia de los resultados obtenidos en el atributo **Cantidad de pruebas**.

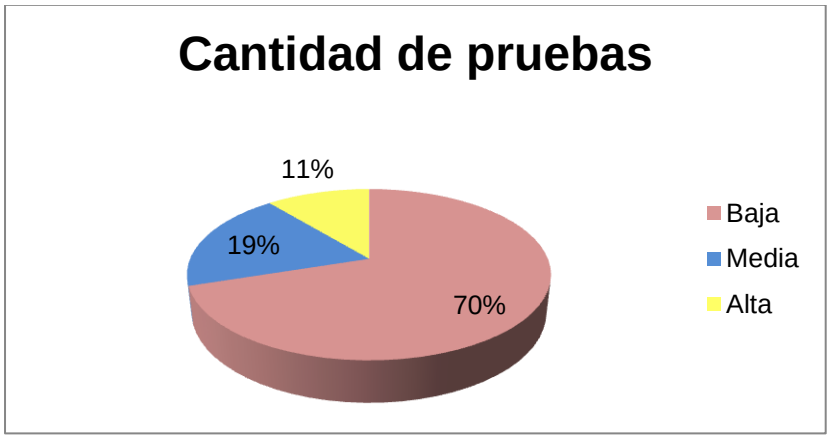


Figura 18. Representación del valor en % del atributo Cantidad de pruebas.

Representación en % de la incidencia de los resultados obtenidos en el atributo **Reutilización**.

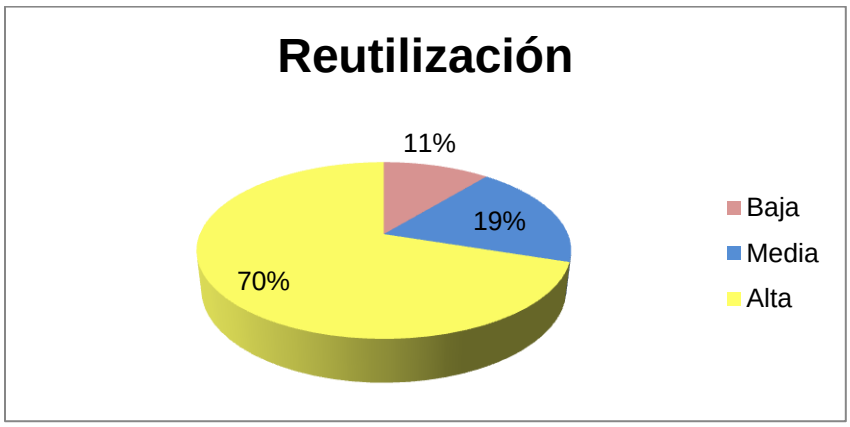


Figura 19. Representación del valor en % del atributo Reutilización.

Análisis de los resultados obtenidos en la evaluación de la métrica RC

Para aplicar la métrica **RC** se determinó que la aplicación cuenta con 32 clases y un total de 75 dependencias entre ellas, con un promedio de 2 dependencias por clase. La **Figura 16, Figura 17**, **Figura 18** y **Figura 19** muestran los resultados de los atributos de calidad: Acoplamiento, Complejidad de Mantenimiento, Reutilización y Cantidad de Pruebas respectivamente.

Luego de aplicarse dicha métrica de diseño y obtenidos los resultados de la evaluación del instrumento de medición de la misma, se puede concluir que el diseño propuesto tiene una calidad aceptable teniendo en cuenta que el 59% de las clases empleadas poseen menos de dos (2) dependencias de otras clases, lo que lleva a evaluaciones positivas del atributo de calidad: acoplamiento. Los atributos de calidad (complejidad de mantenimiento

y cantidad de pruebas) arrojan valores positivos, con un 70% de baja complejidad de mantenimiento y cantidad de pruebas. Para el atributo de calidad reutilización se concluye que de manera general el diseño cuenta con un 70% de alta reutilización. Estos resultados favorecen la reutilización de las clases así como la modificación e implantación del diseño.

Al aplicar las métricas a las clases del diseño definidas se cumple que las métricas de Tamaño de la clase y Relaciones entre clases cumplen las restricciones para la aceptación de las clases.

2.7 Implementación

En el presente epígrafe se expondrán elementos que se tuvieron en cuenta durante el proceso de implementación, como es el caso de las estructuras de datos utilizados y la descripción de las funcionalidades implementadas.

2.7.1 Estructuras de datos utilizadas

Durante el proceso de implementación se utilizaron estructuras de datos: grafos. Esta estructura se emplea para organizar objetos elementales como: los componentes, y las conexiones del modelo con el que se trabaja.

Un grafo, que consiste en un conjunto de nodos (también llamados vértices) y un conjunto de aristas (conexiones) que establecen relaciones entre los nodos. En el caso particular de la aplicación se hizo necesaria esta estructura para entender y manipular mejor cada componente y las relaciones entre ellos en el modelo que se analiza en *AcmeStudio*.

Este modelo consiste principalmente en componentes y conexiones entre estos. El grafo utilizado es dirigido debido a las especificaciones en el modelo en las que los componentes brindan y consumen servicios a través de las conexiones entre estos. Además la estructura de datos utilizada debe cumplir con la restricciones del modelo referente a que no hay aristas (dígase aristas a las conexiones) de un vértice (dígase vértices a los componentes) a sí mismo, esta propiedad es garantizada por la especificación algebraica del tipo de dato.

La representación del grafo se realizó mediante una matriz de adyacencias cuya mayor ventaja es la facilidad de manipulación y de acceso. Se utilizó este tipo de estructura de grafo también conocida como secuencial para representar las dependencias directas del modelo basado en componentes. En la matriz, cada componente es representado por una

columna y una fila. Si un componente de Ci es dependiente de otro componente de Cj, entonces en la celda [i, j] de la matriz aparece una dependencia o arco.

2.7.2 Descripción de la implementación por funcionalidades.

En el presente epígrafe se mostrará una breve descripción de las funcionalidades implementadas y de las diferentes vistas de la aplicación.

La herramienta cuenta con dos (2) subsistemas: Matriz y Reportes. En el subsistema Matriz se encuentra la funcionalidad Generar Matriz (ver en el **Anexo 8**, la **Figura 22**). El subsistema Reporte dará la opción efectuar reportes (ver en el **Anexo 8**, la **Figura 23**) y guardar los mismos luego de analizar un modelo. Además en esta vista se incluyó el logo propuesto para la herramienta.

Generar matriz

Esta funcionalidad es la encargada de obtener la información necesaria referente a los componentes del sistema y la interacción entre los mismos, para generar una Matriz de dependencias. La matriz generada muestra los componentes que brindan, y los que consumen un determinado servicio, además es parte de la funcionalidad, poder guardar dicha matriz.

La representación de la matriz de dependencia para las arquitecturas es lo suficientemente general como para apoyar las capacidades de análisis más allá de lo que se describe en este documento. La determinación de las dependencias se muestra a través de los propios componentes, considerándolos como las etiquetas de las filas y columnas de la matriz.

En la matriz de dependencia se representan por ejemplo, en el caso de que el componente A dependa del componente B, la columna B y la fila A detallan esa relación. El punto de intercepción de estas etiquetas, refleja la existencia de una conexión directa.

Reporte de dependencias

Esta funcionalidad es la encargada de mostrar los análisis de dependencias que se le realizan a los componentes del modelo. La vista de dicha funcionalidad, inicialmente muestra un listado de los componentes existentes. Una vez que el usuario selecciona el componente a analizar se habilita el campo de selección: Análisis de dependencias, el

cual le da la posibilidad al arquitecto de seleccionar la pregunta de análisis que desee realizar. Las preguntas para realizar el análisis son:

- Qué componentes no están siendo usado por ningún otro del sistema.
- Dado un componente conocer:
- Qué componentes dependen directamente de este.
- Qué componentes dependen indirectamente de este.
- De qué componentes depende directamente este.
- De qué componentes depende indirectamente este.
- Si se elimina el componente qué componentes se ven afectados.
- Si se desea reutilizar el componente qué otros componentes se necesitan.

Estas preguntas comparten el tema común de identificar los componentes de un sistema que afectan o son afectados por un componente particular de alguna manera. La idea de recoger juntos las partes relacionadas de un sistema, en un esfuerzo por reducir la cantidad de información que debe ser examinado por algún propósito. Estas dependencias son analizadas a través de las relaciones (**conexiones**) entre **componentes**, además se tienen en cuenta **restricciones** sobre los puertos y roles que aparecen en la relación, estos puertos pueden ser de entrada o salida. Las conexiones entre componentes solo se pueden realizar a través de un puerto de entrada y otro de salida o *attachment*. Si la relación es entre un componente simple y su contenedor, los puertos que se utilizan son del mismo tipo o *binding.* Las restricciones y propiedades que contienen los elementos del dominio forman parte de la **configuración** del modelo arquitectónico la cual se utiliza para manipular los datos y realizar los análisis.
En el caso de guardar reporte se guardan las respuestas a las preguntas:

1. Si se elimina el componente qué componentes se ven afectados.

   Cuando se elimina un componente C del modelo, se afectaran los componentes que dependen de C, es decir se afectan los componentes que consumen algún servicio de C.
2. Si se desea reutilizar el componente qué otros componentes se necesitan.

   Este análisis está vinculado a recoger el grupo de componentes que le brindan algún servicio al componente que se desea reutilizar, por lo que si se quiere usar un componente C en otro sistema se necesitarían todos los componentes de los que C depende.

Importar modelo

Para la importación del modelo se hizo necesaria la creación de un paquete denominado *Importer*. Este paquete es el contenedor de las clases que permiten convertir en objetos los elementos del dominio del modelo que se realiza en AcmeStudio. La conversión de estos elementos permite la manipulación de los mismos durante la implementación de las funcionalidades a automatizar.

Primeramente se captura a través de la clase *IFile* que provee la librería *Acmelib* la dirección donde aparece el proyecto con el modelo arquitectónico. Luego en la clase *AcmeToAmbarConverter* utilizando esta dirección se crea un objeto configuración con la responsabilidad de guardar los elementos que se interpreten del modelo. Esta configuración guarda los elementos (Componentes y Conexiones) en una estructura de datos grafo para análisis posteriores. La clase *ConvertionManagger* es la encargada de manipular los *Systems* (sistemas) que estos son los que contienen los modelos que se realizan en *AcmeStudio*. Esta clase contiene una lista de configuraciones en la que como se mencionaba anteriormente contiene los elementos del modelo para convertirlo a una estructura de dato adecuada (grafo) para la manipulación.

Desarrollo del plugin

Para el desarrollo del *plugin* se utilizó Eclipse que provee un asistente para que crear el esqueleto de lo que va a ser el *plugin* (ver **Figura 3**). Este IDE provee un plugin asociado denominado PDE (*Plugin Development Enviroment*) que no es más que el ambiente de desarrollo de *plugin* que propone la Fundación de Eclipse y uno de los principales subproyectos de la misma. Para la creación de los editores y las vistas se utilizó la librería *JFace* para la generación de las interfaces visuales del *plugin* y SWT (*Standart Widget Toolkit*) el cual es un *Toolkit* para la creación de GUIs (*Grafical User Interface*), también se utilizó un de *plugin* de Eclipse que permiten una programación más gráfica: *Visual Editor* este permitió insertar todo tipo de elementos, darles forma, añadir eventos entre otras. Además se utilizaron otras dependencias en la implementación de las funcionalidades del *plugin*, como las librerías org.acmestudio de *AcmeStudio* para la manipulación de los datos que provee el modelo que se realiza en la herramienta *AcmeStudio*, org.apache.poi para la generación del Excel. Este muestra la matriz de integración, dicha matriz es una de las funcionalidades más significativas de la herramienta. Para la generación de reportes en formato pdf se utilizó la librería org.lowagie.

Conclusiones parciales

En este capítulo se abordaron las principales características y funcionalidades de la herramienta, se realizó la especificación de los requisitos asociados al *plugin*. Se realizó el diseño de clases y la validación de la calidad del diseño propuesto, especificándose el uso de los patrones de diseño empleados. Con el análisis de los resultados arrojados por las métricas de diseño se concluye que este cuenta con una calidad aceptable. Se expusieron elementos de la implementación, tales como, descripción de las funcionalidades, y las estructuras de datos utilizadas.

CAPÍTULO 3: Validación y pruebas.

3 Introducción

En el capítulo se realiza el análisis de los resultados de la puesta en práctica de la herramienta mediante un caso de estudio. Los resultados obtenidos se brindarán en función de validar el cumplimiento de la variable establecida, demostrando la disminución del tiempo en que se analizan las dependencias arquitectónicas. Se mostrarán los resultados de la ejecución del criterio de expertos haciendo uso del método *Delphi*. También se desarrolla la etapa de pruebas para garantizar la calidad del producto desarrollado.

3.1 Criterio de Expertos

La herramienta propuesta en el presente trabajo será validada además a través de criterios de expertos mediante el uso de técnicas propuestas por el método *Delphi*.

Delphi es un método creado por la *Rand Corporation* con el objetivo de elaborar pronósticos referentes a posibles acontecimientos en varias ramas de la ciencia, la técnica y la política. (2006)

"… el *Delphi es la utilización sistemática del juicio intuitivo de un grupo de expertos para obtener un consenso de opiniones informadas.*". (2006)

Para la aplicación práctica del método es necesario tener en cuenta dos cuestiones fundamentales:

- La elaboración del cuestionario.
- La selección de los expertos.

A continuación se explica cómo fueron aplicadas estas cuestiones en el presente trabajo.

Selección del grupo de expertos

Para la selección del grupo de expertos se realizaron las siguientes actividades:

Determinación de las áreas del conocimiento que deben dominar los expertos

Partiendo del problema planteado en la introducción del trabajo, se determinó que los expertos a consultar debían dominar las siguientes áreas del conocimiento: arquitectura

de software, modelado de arquitectura, lenguajes y herramientas para el modelado arquitectónico.

Elaboración del listado de expertos candidatos

Luego de determinar las áreas del conocimiento se elaboró un listado de expertos candidatos teniendo en cuenta su experiencia productiva o docente en las áreas identificadas. El listado inicial estaba conformado por nueve (9) expertos.

Obtención del consentimiento de los expertos para participar

El siguiente paso fue la obtención del consentimiento de los expertos para participar en la validación. En este caso de los nueve (9) expertos seleccionados, siete (7) estuvieron de acuerdo en participar.

Determinación del coeficiente de conocimiento de los expertos

Las características de los expertos influyen decisivamente en la confiabilidad de los resultados obtenidos. Estas características son: calificación técnica, capacidad de emitir una decisión al respecto, conocimientos específicos sobre el tema a evaluar, disposición a participar, entre otros. (2006)

Para la selección de los expertos es útil emplear la valoración por competencias. Este método consiste en calcular el coeficiente de competencia (k) del experto a partir de la autovaloración del experto sobre su conocimiento o información sobre el tema (kc) y el coeficiente de argumentación o valoración (ka) mediante la siguiente ecuación (2006)

k = (kc + ka)/2

El código de interpretación de los coeficientes de competencias es como sigue:

Si 0,67 <=k < =1,0 coeficiente de competencia alto.

Si 0,33 <= k < =0,66 coeficiente de competencia medio

Si k <= 0,32 coeficiente de competencia bajo

Recomendándose incluir en el grupo a los expertos con coeficiente de competencia alto y medio.

El **Anexo 9** muestra la encuesta aplicada a los expertos para determinar su coeficiente de competencias, los resultados se reflejan en la siguiente tabla:

Experto N°	kc	ka	k	Nivel
1	0.59	0.79	0.69	Alto
2	0.64	0.57	0.61	Medio
3	0.59	0.64	0.61	Medio
4	0.59	0.57	0.58	Medio
5	0.55	0.71	0.63	Medio
6	0.77	0.71	0.74	Alto
7	0.77	0.78	0.78	Alto

Tabla 8. Resultado del coeficiente de competencia de expertos.

De los siete (7) expertos encuestados tres (3) poseen un coeficiente de competencia alto y los restantes cuatro (4) un coeficiente de competencia medio, por lo que se decidió que todos pueden ser incluidos en la evaluación de la propuesta.

- Selección de los expertos.

Para la selección de los expertos es necesario determinar el número de expertos que debe tener el grupo. No existe una norma generalizada para determinar el número óptimo de expertos, sin embargo según **Eneko Astigarraga** en (Astigarraga) si se tiene en cuenta el criterio de hasta 7 expertos el error disminuye exponencialmente, después de 30, aunque el error disminuye lo hace de manera poco significativa y no compensa el incremento de costos y esfuerzo, por lo que se sugiere utilizar un número de expertos en el intervalo de 7 a 30.

Dado que siete (7) es un número de expertos que estuvieron dispuestos a participar en la validación, alcanzando niveles de competencia altos y medios, se decidió que los siete (7) formaran parte del Comité de Expertos.

Validación de la propuesta

Para validar la herramienta desarrollada se aplicó la encuesta mostrada en el **Anexo 10**, la cual se encuentra asociada a pruebas realizadas sobre la aplicación.

Indicadores de medición

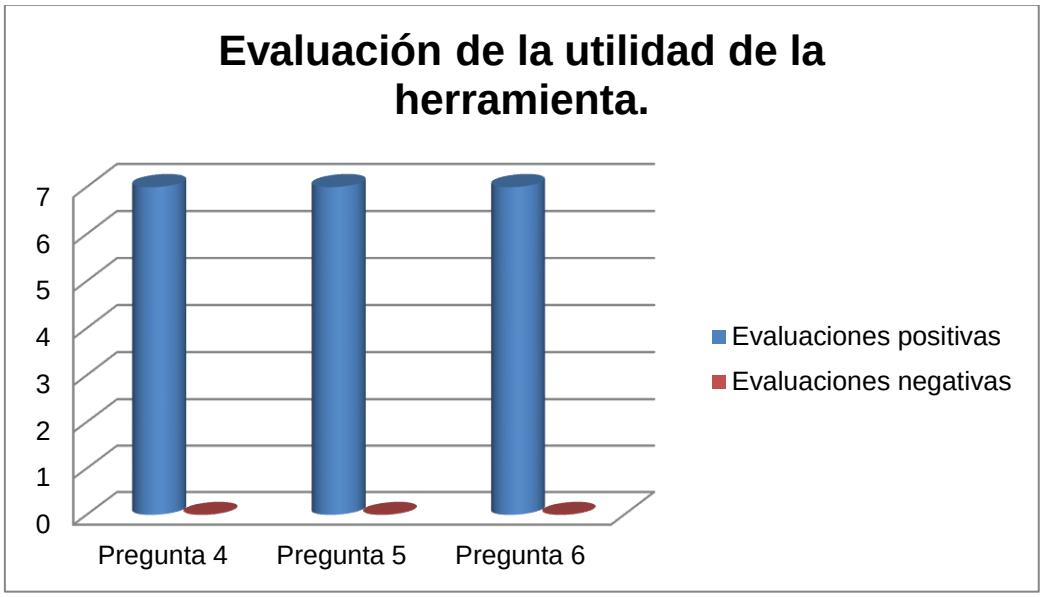
Para el análisis de los resultados de la puesta en práctica de la presente investigación, se toma como variable demostrativa la disminución del tiempo en que se realizan análisis de dependencias sobre el caso de estudio elaborado.

El objetivo de dicha encuesta es:

- Determinar la utilidad de la herramienta propuesta para resolver el problema planteado en la Introducción del presente trabajo. Para esto se formularon los incisos a y b de la pregunta 3 y las preguntas 4, 5 y 6.
- Demostrar que se da cumplimiento a la variable establecida.
- La pregunta 7 está dedicada a elaborar una valoración general de la utilidad de la herramienta con respecto al cumplimiento de los problemas planteados en la introducción del trabajo.

Utilidad del procedimiento.

Para valorar este aspecto se tuvieron en cuenta las respuestas de los expertos en la encuesta mostrada en el **Anexo 10**, enmarcada en demostrar la necesidad de disminuir el tiempo en que se realizan los análisis de dependencias arquitectónicas a la hora de tomar decisiones de este tipo durante las verticalizaciones de Cedrux, permitiendo ajustarse a las características de cada entidad del país, el objetivo principal de aplicar la encuesta es el de demostrar la efectividad de la herramienta desarrollada. Las respuestas a los puntos 4, 5 y 6 podían ser positivas o negativas y el resultado se muestra a continuación.



Gráfica 1. Evaluación de la utilidad de la herramienta.

La **Gráfica 1** muestra la evaluación emitida por los expertos a la pregunta 5, donde se puede concluir que de los siete (7) expertos encuestados los siete (siete), dieron una evaluación positiva.

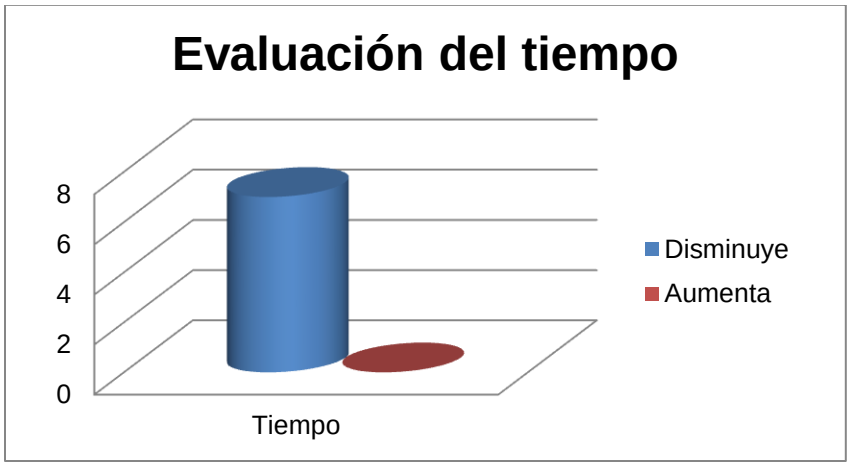
Evaluación del tiempo

La tabla que se presenta a continuación, muestra el resultado de las pruebas realizadas por los expertos, para esto se elaboró en la herramienta AcmeStudio, un modelo arquitectónico para realizar análisis sobre el mismo. En la prueba participaron 7 expertos, a los cuales se les pidió que realizaran análisis manual sobre el modelo y posteriormente un análisis haciendo uso del plugin, el objetivo de esta prueba era demostrar que el análisis realizado sobre la aplicación consumía menos tiempo que el análisis manual al mismo tiempo validar la precisión de los datos.

	Tiempo de prueba sobre el caso de estudio		
N°	**Manual**	**Sobre la aplicación**	**Valoración**
1	10 min	1 min	El experto concluyó que: El plugin permite de una manera rápida determinar relaciones de dependencias entre componentes y determinar cómo se afectan estos ante posibles cambios. Creo que es una funcionalidad de gran

			importancia para el equipo de arquitectura.
2	7 min	1 min	El experto concluyó que: La aplicación es de gran utilidad para la visualización de las dependencias entre componentes, los filtros que brinda para mostrar las implicaciones de eliminar o reutilizar un componente se convierten en una herramienta de análisis muy potente para los arquitectos. Además tiene la virtud de ser fácil de utilizar y permite disminuir considerablemente el tiempo de análisis.
3	10 min	1 min	El experto concluyó que: Con la utilización del plugin realizado se disminuye considerablemente el tiempo de búsqueda de dependencias entre componentes beneficiando el equipo de arquitectura.
4	15 min	1 min	El experto concluyó que: El plugin desarrollado es muy útil para el equipo de arquitectura puesto que permite disminuir el tiempo de análisis de dependencias entre componentes.
5	30 min	1 min	El experto concluyó que: La aplicación posibilita disminuir considerablemente el tiempo empleado en determinar las dependencias existentes entre componentes y subsistemas, lo cual ayuda y facilita a los arquitectos a la hora de empaquetar y modularizar la aplicación.
6	1h y 15 min	2 min	El experto concluyó que: La herramienta facilita el análisis ante posibles cambios, garantizando agilidad en el proceso y obtener resultados confiables, influyendo positivamente en la calidad de la toma de decisiones.
7	5 min	1 min	El experto concluyó que: La herramienta permite ahorrar el tiempo que se dedica al análisis de dependencias entre componentes.

Tabla 9. Evaluación del tiempo.



Gráfica 2. Evaluación del tiempo.

Teniendo en cuenta los resultados obtenidos, se puede apreciar en la **Tabla 9** y en la **Gráfica 2** que el uso del plugin agiliza en un tiempo considerable el análisis de dependencia sobre el modelo arquitectónico presentado. En este caso los siete (7) expertos dieron una valoración positiva del cumplimiento de la variable: tiempo.

3.2 QUASAR en la evaluación de la arquitectura

El método de evaluación de la calidad de la arquitectura QUASAR, se realiza mediante casos de calidad, los cuales son presentados por el arquitecto o el grupo de arquitectura para la evaluación y se dividirán en tres aspectos:

- Demandas: Son definidas como afirmaciones que hace el equipo de desarrollo acerca de si el software cumple adecuadamente con las metas de calidad o si tiene en cuenta un conjunto de requerimientos asociados y relacionados con la calidad.
- Argumentos: Son las razones por las cuales el software debe cumplir con las demandas que se hacen acerca de determinados requerimientos relacionados con la calidad.
- Evidencias: Son las evidencias que se presentan para respaldar los argumentos que se realizan.

El *plugin* que se presenta debe cumplir con los requisitos no funcionales que se exponen en el epígrafe 2.3, tributando a los factores y subfactores de calidad que se muestran en la tabla:

3.2.1 Casos de calidad

Escalabilidad

1. Demandas:

La herramienta es flexible: El sistema debe permitir el desarrollo de nuevas funcionalidades, modificar o eliminar funcionalidades después de su construcción y puesta en marcha inicial.

La herramienta debe permitir integrarse con *AcmeStudio* mediante un *plugin* para Eclipse.

2. Argumentos:

Para garantizar la escalabilidad de la herramienta, se estructuró la aplicación con un alto nivel de desacoplamiento entre sus clases y objetos, permitiendo la individualidad de los elementos. Esto contribuye a minimizar la implicación de las modificaciones que tienen lugar sobre determinadas funcionalidades.

3. Evidencias:

Utilización de patrones GRASP: Alta Cohesión, Bajo Acoplamiento, Experto.

Utilización de patrones GoF: Fachada.

La utilización de patrones de diseño que garanticen el bajo acoplamiento y la alta cohesión en el sistema, estos últimos mencionados en el epígrafe 2.5 donde se explica la aplicación de estos.

Usabilidad

1. Demandas:

La herramienta es operable: La herramienta deberá ser usada y controlada por el cliente. En este caso la aplicación será manipulada por los arquitectos de Cedrux.

2. Argumentos:

Para garantizar la usabilidad de la aplicación, esta será utilizada y controlada por los arquitectos del Centro. La usabilidad puede verse además a través de un diseño sencillo de las interfaces con las que interactúa el cliente.

3. Evidencias:

Como evidencia de los argumentos de este caso de calidad se presentan las valoraciones de los expertos encuestados en las pruebas que se realizaron sobre la aplicación. Durante las pruebas los expertos demostraron un dominio y control total de la aplicación.

Software

1. Demandas:

- La herramienta deberá desarrollarse en el IDE Eclipse en su versión 3.5.
- La herramienta deberá utilizar JDK en su versión 1.5 o superior.
- El plugin debe ser compatible con la última versión de la herramienta AcmeStudio.

2. Argumentos:

Para el modelado de la arquitectura de Cedrux se utiliza AcmeStudio en su versión 3.5

3. Evidencias:

En el sitio oficial de *AcmeStudio* se define que para desarrollar extensiones a esta herramienta para la versión 3.5, se utilice el IDE Eclipse 3.5 y JDK 1.5 o superior. Esta evidencia se puede consultar en el siguiente enlace: http://acme.able.cs.cmu.edu/.

3.3 Pruebas de software

Una vez concluido el proceso de implementación, el software debe ser probado, para detectar y corregir errores, antes de entregar el producto. Las pruebas aplicadas a la herramienta son: las pruebas de caja blanca y caja negra.

3.3.1 Pruebas de caja blanca

La prueba de caja blanca, es un método de diseño de casos de prueba que utiliza la estructura de control del diseño procedimental para obtener los casos de prueba. Para la solución desarrollada, la prueba de Caja Blanca aplicada fue: la prueba del camino básico. Esta técnica permite obtener una medida de la complejidad lógica de un diseño procedimental (Pressman, 2005). La medida alcanzada sirve como guía para la definición de un conjunto básico de caminos de ejecución. Los casos de prueba obtenidos del conjunto básico, garantizan que durante el proceso de prueba, se ejecuten al menos, una vez, cada sentencia del programa.

Para utilizar esta técnica se hizo necesario calcular la complejidad ciclomática del método a analizar, posteriormente se enumeraron las sentencias de código (**Figura 20**), a partir de dicha enumeracion se elaboró el grafo de flujo de la funcionalidad seleccionada (**Figura 21**).

```
private List<AmbarConnection> findConnections(AmbarComponent t,
        AmbarService out) {
    // TODO Auto-generated method stub
    int index = structure.getComponentIndex(t);1
    int count = structure.getSize();1
    List<AmbarConnection> list = new LinkedList<AmbarConnection>();1

    for (int i = 0; i < count; i++) {2

        AmbarConnectionPool pool = structure.getConnectionPool(i, index);3
        List<AmbarConnection> conns = pool.getConnections();4

        for (AmbarConnection con : conns) {5
            if (con.getOutService().equals(out))6
                list.add(con);7
        }8
    }9
    return list;10
}
```

Figura 20. Código fuente de la funcionalidad findConnections(AmbarComponent t, AmbarService out).

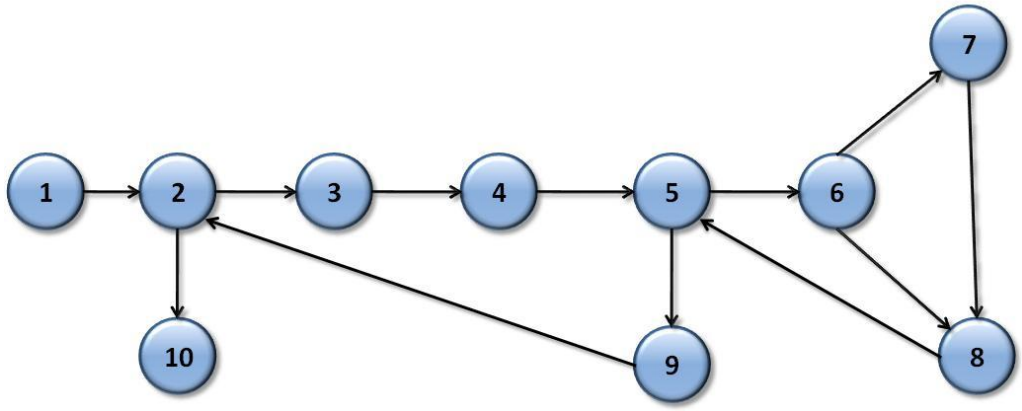


Figura 21. Grafo asociado al algoritmo findConnections(AmbarComponent t, AmbarService out).

Cálculo de la complejidad ciclomática a partir de un segmento de código.

La complejidad ciclomática, es una métrica del software que proporciona una medición cuantitativa de la complejidad lógica de un programa. Cuando se usa en el contexto del método de prueba del camino básico, el valor calculado como complejidad ciclomática define el número de caminos independientes del conjunto básico de un programa, proporcionando un límite superior para el número de pruebas que se deben realizar para asegurar que cada sentencia es ejecutada al menos una vez . (Pressman, 2005)

La complejidad ciclomática de la funcionalidad: freeComponent(), se obtubo utilizando las tres (3) vías posibles, con el objetivo de verificar los resultados arrojados en cada una.

Las fórmulas para realizar dicho cáculo son:

1. **V (G) = (A - N) + 2**

Siendo A la cantidad total de aristas del grafo y N la cantidad de nodos.

V (G) = (12 -10) +2

V (G) = 4

2. **V (G) = P +1**

Siendo P la cantidad de nodos predicado (son aquellos de los cuales parten dos o más aristas).

V (G) = 3+1

V (G) = 4

3. **V (G) = R**

Siendo R la cantidad de regiones que posee el grafo.

V (G) = 4

Análisis de los resultados.

Según los resultados obtenidos en cada uno de estos cálculos se puede concluir que la complejidad ciclomática del código analizado es cuatro (4), determinándose a su vez que existen cuatro (4) caminos posibles por donde puede circular el flujo y que esta misma cantidad representa el límite superior de casos de prueba que se le pueden aplicar a dicho código.

Caminos básicos

A continuación se muestran los caminos básicos por donde puede circular el flujo.

Camino básico # 1: 1-2-10.

Camino básico # 2: 1-2-3-4-5-9-2-10.

Camino básico # 3: 1-2-3-4-5-6-8-5-9-2-10.

Camino básico # 4: 1-2-3-4-5-6-7-8-5-9-2-10.

Para cada uno de los caminos obtenidos se realiza un caso de prueba. Los casos de prueba realizados son los siguientes:

Casos de prueba.

Caso de prueba para el camino básico #1.

Camino básico # 1: 1-2-10.

Descripción: El dato de entrada cumplirá con el siguiente requisito:

Los parámetros de entrada son nulos ya que la lista de componentes de la estructura se encuentra vacía, por lo que no se devuelve ninguna conexión.

Entrada: AmbarComponent t= null, AmbarService out= null.

Resultados esperados: Se espera que el método devuelva una lista de conexiones vacía.

Camino básico # 2: 1-2-3-4-5-9-2-10.

Descripción: el dato de entrada cumple con el siguiente requisito:

Los parámetros de entrada AmbarComponent t AmbarService out no son nulos, la estructura contiene varios componentes que a su vez no tienen conexiones.

Entrada: AmbarComponent t, AmbarService out.

Resultados esperados: Se muestra una lista de conexiones vacía.

Camino básico # 3: 1-2-3-4-5-6-8-5-9-2-10.

Descripción: el dato de entrada cumple con el siguiente requisito:

Los parámetros de entrada AmbarComponent t AmbarService out no son nulos, la estructura contiene varios componentes que a su vez tienen conexiones, aunque ninguna de ellas coincide con el servicio pasado por parámetro.

Entrada: AmbarComponent t, AmbarService out.

Resultados esperados: Se muestra una lista de conexiones vacía.

Camino básico # 3: 1-2-3-4-5-6-7-8-5-9-2-10.

Descripción: el dato de entrada cumple con el siguiente requisito:

Los parámetros de entrada AmbarComponent t AmbarService out no son nulos, la estructura contiene varios componentes que a su vez tienen conexiones, y algunas de ellas coincide con el servicio pasado por parámetro.

Entrada: AmbarComponent t, AmbarService out.

Resultados esperados: Se muestra una lista de conexiones.

3.3.2 Pruebas de caja negra

Las pruebas de caja negra, también denominadas prueba de comportamiento, se centran en los requisitos funcionales del software (Pressman, 2005). La misión de estas pruebas es detectar errores como:

- Funciones incorrectas o ausentes.
- Errores de interfaz.
- Errores en estructuras de datos o en accesos a bases de datos externas.
- Errores de rendimiento.
- Errores de inicialización y de terminación.

Las pruebas se llevan a cabo sobre la interfaz del software, y es completamente indiferente el comportamiento interno y la estructura del programa. Los casos de prueba de caja negra pretenden demostrar que:

- Las funciones del software son operativas.
- La entrada se acepta de forma adecuada.
- Se produce una salida correcta.
- La integridad de la información externa se mantiene.

Casos de prueba

La descripción de los diseños de casos de pruebas de cada una de las funcionalidades se puede consultar en los **Anexos 11, 12, 13 y 14**.

Resultado de las pruebas

La aplicación desarrollada fue revisada y probada por las analistas de CEIGE a partir de los diseños de casos de prueba realizados a cada una de las funcionalidades implementadas, mostrando buena calidad en las mismas.

Las analistas que fungieron como probadoras fueron dos (2), las cuales realizaron las tareas que se mencionan en el párrafo que sigue.

Se realizaron dos (2) iteraciones, en la primera se detectaron cuatro (4) no conformidades (NC), tres (3) de ellas en la descripción de los diseños de casos de prueba y una (1) en la aplicación. En la segunda iteración fueron resultas las NC detectadas en la iteración anterior y no se encontraron nuevas NC. Finalizando con la aprobación de la aplicación.

Conclusiones parciales

Durante el presente capítulo se exponen las métricas utilizadas para la validación del diseño de la propuesta de solución, las misma arrojaron resultados positivos, llegándose a la conclusión que se cuenta con un diseño simple y con una calidad aceptable. Además se describieron y aplicaron las pruebas de caja blanca y caja negra, para evaluar el sistema desarrollado, las cuales mostraron que el sistema consta con un adecuado funcionamiento, demostrando así el cumplimiento de las necesidades del cliente y la calidad requerida en el mismo.

Conclusiones

Una vez finalizado este trabajo de diploma se puede arribar a las siguientes conclusiones:

- Se realizó un estudio calificador de algunos ADLs, y herramientas que realizan análisis de dependencias, así como, las técnicas que estas emplean para ello. Se trataron definiciones claves para la investigación.
- Se elaboró el diseño de la herramienta a implementar, validando el mismo teniendo en cuenta: las métricas TOC y RC, las cuales mostraron resultados satisfactorios en su aplicación.
- Se realizó la implementación de la herramienta, así como la aplicación de pruebas de software para su validación, dándole solución a las necesidades de los arquitectos del centro CEIGE durante el proceso de verticalizaciones.
- Se logró realizar análisis de dependencias entre los componentes de un modelo arquitectónico.
- Se contribuyó a mejorar el tiempo de implementación de las verticalizaciones.

Recomendaciones

1. Agregar soporte para otros tipos de análisis de dependencias así como otros tipos de dependencias.
    a. Dependencias de comportamiento, causales, de trazabilidad.
2. Agregar soporte para otros tipos de análisis arquitectónico.
    a. Validación del diseño enfocado al cumplimiento con determinados indicadores de calidad.
3. Agregar soporte para técnicas de Inteligencia Artificial con el fin de proveer la herramienta con capacidades para sugerir mejoras en el diseño a partir de una base de conocimiento.
4. Perfeccionar las interfaces de usuario así como la información que se muestra a este, de manera que sea más intuitivo el trabajo con la herramienta provea información más útil.
5. Aplicar la herramienta en otras fases del desarrollo, para mitigar errores durante la evolución del sistema de manera general.

Bibliografía

2007. Acme web. [En línea] 21 de 3 de 2007. [Citado el: 11 de 12 de 2011.] http://www.cs.cmu.edu/~acme/docs/language_overview.html.

Astigarraga, Eneko. *Método Delphi.* San Sevastián. : Universidad de Deusto.

Baraza, Manuel. 30. http://www.channelbiz.es/. [En línea] 2009 de abril de 30. [Citado el: 20 de enero de 2012.] Manuel Baraza, subdirector de la unidad de soluciones empresariales de Ibermática, explica los entresijos de desarrollar software de gestión a medida.. http://www.channelbiz.es/2009/04/30/hacia-la-verticalizacion-conceptual/.

Bradley Schmerl, David Garlan. 2003. *AcmeStudio: Supporting Style-Centered Architecture Development.* s.l. : School of Computer Science Carnegie Mellon University, 2003. PDF.

Carmina Lizeth Torres Flores, Germán Harvey Alférez Salinas. 2008. *Establecimiento de una Metodología de Desarrollo de Software para la Universidad de Navojoa Usando OpenUP.* Ciudad de Navojoa : s.n., 2008. pág. 2.

2006. *CRITERIO DE EXPERTOS: METODO DELPHY.* 2006. pág. 21.

1997. *DRAFT. Guide to the Rapide 1.0.* s.l. : Universidad de Stanford, 1997.

Eclipse. 2012. epf.eclipse.org. Introduction to OpenUp. *epf.eclipse.org. Introduction to OpenUp.* [En línea] 24 de 1 de 2012. [Citado el: 13 de 2 de 2012.] http://epf.eclipse.org/wikis/openup/index.htm.

epf.eclipse.org. Introduction to OpenUp. [En línea] [Citado el: 13 de 2 de 2012.] http://epf.eclipse.org/wikis/openup/index.htm.

Fernández González, Mairelys y Zorrilla Rivera, Osley. 2010. *Diseño e implementación del componente Ajuste al Costo del Subsistema Costos y Procesos del Sistema Integral de Gestión de Entidades CEDRUX.* Habana : UCI, 2010.

Fernández, Osmar Leyet. septiembre de 2011. *Propuesta metodológica para la obtención de los componentes de software en los proyectos del sistema Cedrux.* UCI. Ciudad de La Habana : s.n., septiembre de 2011.

Fontdevila, Diego. 2007. *Tesis de Ingeniería: Un modelo integrado de dependencias.* Facultad de Ingeniería Universidad de Buenos Aires. Buenos Aires : s.n., 2007. pág. 23, PDF.

Fontdevilla, Ing. Diego. 2007. *Tesis de Ingeniería:Un modelo integrado de dependencias.* Buenos Aires : Facultad de Ingeniería Universidad de Buenos Aires, 2007.

Francisco Losaivo, Christian Guillén D. noviembre 2003. *Diseño Arquitectónico de Sistemas Distribuidos en RAPIDE.* s.l. : Revista Sociedad Chilena de Ciencia de la Computación, noviembre 2003. pág. 3, pdf.

Gallardo, David. 2002. IBM. [En línea] 1 de 12 de 2002. [Citado el: 10 de 12 de 2011.] http://www.ibm.com/developerworks/library/os-ecplug/.

González, Héctor Luis López. 2009. *Plug-in para Eclipse para la generación de elementos arquitectónicos personalizados.* Ciudad de la Habana : Universidad de Ciencias Informáticas., 2009. pág. 8, PDF.

—. Mayo de 2009. *Plug-in para Eclipse para la generación de elementos arquitectónicos personalizados.* UCI. Ciudad de la Habana : s.n., Mayo de 2009. pág. 17.

Hervis, Reynaldo Mariño Echemendia y Yandy González. 2010. *"Diseño e implementación de un plugin de Eclipse que agilice el desarrollo de única Dalas".* Facultad 15, Universidad de Ciencias Informáticas. La Habana : s.n., 2010. Trabajo de Diploma para optar por el título de Ingeniero en Ciencias Informáticas.

Iglesias., Elizabeth Martínez. Junio de 2007. *Adaptación de la metodología RUP a los nuevos módulos del proyecto Registro y Notarias.* UCI. Ciudad de La Habana : s.n., Junio de 2007. pág. 14.

—. Junio de 2007. *Adaptación de la metodología RUP a los nuevos módulos del Proyecto Registros y Notarías.* UCI. Ciudad de La Habana. : s.n., Junio de 2007. pág. 15.

Judith A. Stafford, Debra J. Richardson, and Alexander L. Wolf. 1998. *Aladdin: A Tool for Architecture-Level Dependence Analysis.* Colorado : University of Colorado, 1998.

—. 1997. *Chaining: A Software Architecture Dependence Analisys Technique.* Colorado : University of Colorado, 1997.

Kicillof, Carlos Reinoso y Nicolás. 2004. *Lenguajes de descripción de arquitectura.* Buenos Aires : Universidad de Buenos Aires, 2004.

Kicillof, Carlos Reynoso – Nicolás. marzo de 2004. *Lenguajes de Descripción de Arquitectura.* Universidad de Buenos Aires. Buenos Aires : s.n., marzo de 2004. pág. 18. version 1.0.

Larman, Craig. *UML y Patrones, 2da Edición.* Vancouver,Canadá : s.n. pág. 162, pdf.

Lilian Teresa Castro Mecías, Rolando Bermúdez Peña. Mayo 2008. *Desarrollo de una Librería de Componentes JavaScript basado en Dojo Toolkit.* Ciudad Habana : Universidad de las Ciencias Informáticas, Mayo 2008. pág. 12.

Mairelys Fernández González, Osley Zorrilla Rivera. 2010. Biblioteca de la Universidad de las Ciencias Informática. [En línea] 2010. [Citado el: 2 de mayo de 2012.] http://biblioteca2.uci.cu/.

Maite Rodríguez Corbea, Meylin Ordóñez Pérez. julio 2007. *LA METODOLOGÍA XP APLICABLE AL DESARROLLO DEL SOFTWARE EDUCATIVO EN CUBA.* Universidad

de las Ciencias Informáticas. Ciudad Habana : Universidad de las Ciencias Informáticas., julio 2007. pág. 36, PDF.

Molpeceres, Alberto. Diciembre de 2002. *Proceso de desarrollo: RUP, XP, FDD.* Diciembre de 2002. pág. 3.

OpenUp. [En línea] [Citado el: 28 de 2 de 2012.] http://epf.eclipse.org/wikis/openup/index.htm.

Pérez, Lilianne Cantillo Romero y Karelys Mesa. Julio de 2007. *Ingeniería de requisitos: elicitación, análisis y negociación, y especificación.* UCI. Ciudad de La Habana : s.n., Julio de 2007. pág. 25.

Pressman, Roger S. 2005. *Ingeniería del Software: Un enfoque práctico.* 2005.

Prieto, Félix. 2009. *Patrones de diseño.* 2009.

Reinoso, Carlos Billy. Marzo 2004. *Introduccion a la Arquitectura de Software.* Buenos Aires : Universidad de Buenos Aires, Marzo 2004. pág. 11.

—. 2004. *Introduccion a la Arquitectura de Software.* Buenos Aires : Universidad de Buenos Aires, 2004. pág. 11.

Río, Agustín Cernuda del. febrero de 2002. *Sistema de verificación de componentes software.* s.l. : Universidad de Oviedo, febrero de 2002.

—. 2002. *Sistema de verificación de componentes software.* UNIVERSIDAD DE OVIEDO. Oviedo : s.n., 2002. pág. 30.

Rodríguez, Juan José Rosales. 2011. *Desarrollo de la versión 2.0 del importador de datos del producto auditoría CEDRUX.* La Habana, Cuba. : s.n., 2011. pág. 16.

Schmerl, Bradley. 2003. *AcmeStudio user's Manual.* s.l. : School of Computer Science, Carnegie Mellon University, 2003.

Simarro, D. Juan Matías. 2004. *APLICACIÓN DEL ANÁLISIS DE DEPENDENCIAS A LA ARQUITECTURA SOFTWARE.* s.l. : UNIVERSIDAD DE CASTILLA-LA MANCHA ESCUELA POLITÉCNICA SUPERIOR, Departamento de informática, 2004. pág. 16.

Simarro., D. Juan Matías. Diciembre 2004. *APLICACIÓN DEL ANÁLISIS DE DEPENDENCIAS A LA ARQUITECTURA DE SOFTWARE.* s.l. : UNIVERSIDAD DE CASTILLA-LA MANCHA ESCUELA POLITÉCNICA SUPERIOR, Departamento de informática, Diciembre 2004. pág. 19.

1997*Succeedings of the Second International Software Architecture Workshop (ISAW-2)*ACM SIGSOFT Software Engineering1997Notes, pp. 42-56.

Szyperski, Clements. 1998. *"Component Software. Beyond Object-Oriented Programming.".* 1998.

Trosky B. Callo Arias, Pieter van der Spek, Paris Avgeriou. Marzo 2011. *A practice-driven systematic review of dependency analysis solutions.* s.l. : Empirical Software Engineering, Marzo 2011. pág. 33.

Trosky B. Callo Arias, Pieter van der Spek, Paris Avgeriou. 2011. *A Systematic Review of Dependency Analysis Solutions.* s.l. : Empirical Software Engineerin, 2011. pág. 37. Vol. Chapter 2.

Valero, Yagnieris Montero Morales y Elier Carmenate. 2009. *" Selección de un Lenguaje de Descripción Arquitectónica y modelado de las decisiones arquitectónicas del proyecto ERP Cuba".* Habana : Universidad de las Ciencias Informaticas, 2009. pág. 14.

—. Julio de 2009. *" Selección de un Lenguaje de Descripción Arquitectónica y modelado de las decisiones arquitectónicas del proyecto ERP Cuba".* UCI. Ciudad de La Habana : s.n., Julio de 2009. págs. 23-29.

Vestal., Steve. Febrero de 1993. *"A cursory overview and comparison of four Architecture Description Languages". .* s.l. : Technical Report, Honeywell Technology Center, , Febrero de 1993.

www.visual-paradigm.com. UML CASE Tools - Free for Learning UML, Cost-Effective for Business Solutions. [En línea] [Citado el: 12 de 02 de 2012.] http://www.visual-paradigm.com/product/vpuml/..