UNIVERSIDAD DE LAS CIENCIAS INFORMÁTICAS

FACULTAD 3

HERRAMIENTA PARA GESTIONAR COMPONENTES EN EL MARCO DE TRABAJO SAUXE

Trabajo de Diploma para optar por el título de

Ingeniero en Ciencias Informáticas

Autor:

Julio Cesar Nápoles Puente

Tutores:

Ing. Yuniel Cedeño Mendoza

Ing. Claudia Bravo Batista

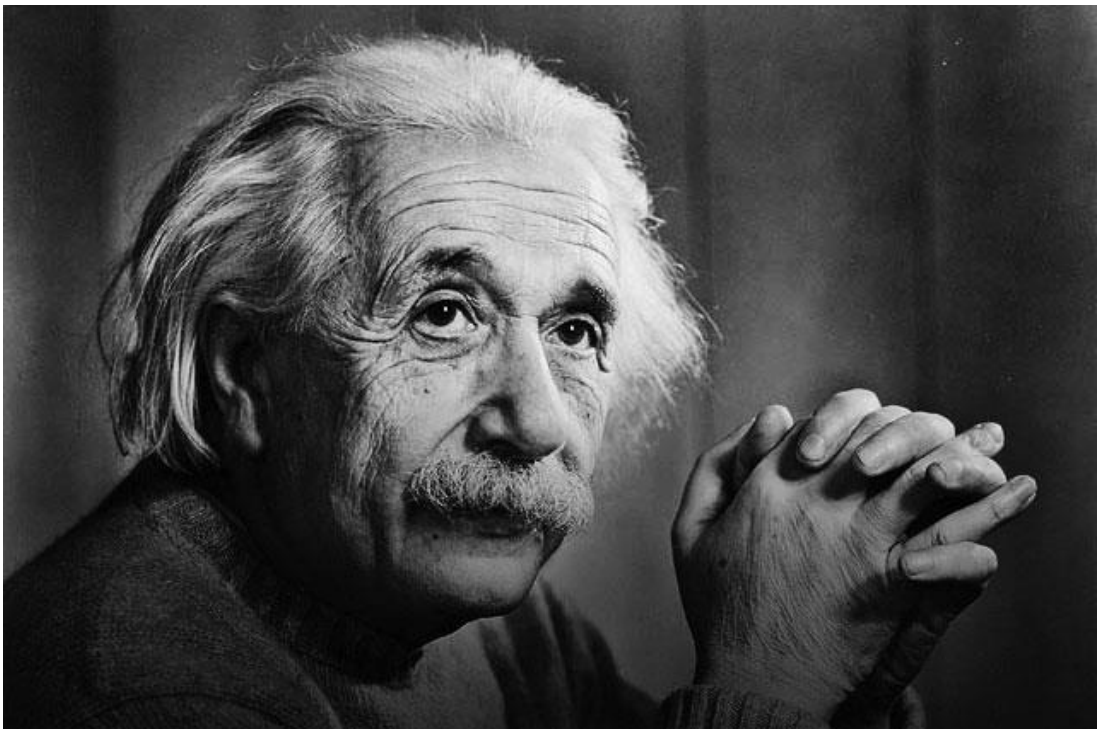
Ing. Katia Saria Preval

La Habana, Junio de 2014

"Año 56 de la Revolución"

"No entiendes realmente algo a menos que seas capaz de explicárselo a tu abuela. "

Albert Einstein



DECLARACIÓN DE AUTORÍA

Declaro que soy el único autor de este trabajo y autorizo a la Universidad de las Ciencias Informáticas a hacer uso del mismo en su beneficio.

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

Julio Cesar Nápoles Puente	Ing. Yuniel Cedeño Mendoza
Firma del Autor	Firma del Tutor
Ing. Claudia Bravo Batista	Ing. Katia Saria Preval
Firma del Tutor	Firma del Tutor

DATOS DE CONTACTO

Síntesis de los tutores:

Tutor: Ing. Yuniel Cedeño Mendoza

Graduado de Ingeniería en Ciencias Informáticas

Dirección: Universidad de las Ciencias Informáticas (UCI), Edificio: 31, Apto: 301

Teléfono Oficina: +53 – 7 – 8358292. Teléfono Apto: +53 – 7 – 835 – 8897

E-mail: ycedenom@uci.cu

Tutor: Ing. Claudia Bravo Batista

Graduado de Ingeniería en Ciencias Informáticas en Julio del 2012

Dirección: Universidad de las Ciencias Informáticas (UCI), Edificio: 27, Apto: 201

Teléfono Oficina: +53 – 7 – 8358292. Teléfono Apto: +53 – 7 – 835 – 8767

E-mail: cbravo@uci.cu

Tutor: Ing. Katia Saria Preval

Graduado de Ingeniería en Ciencias Informáticas en Julio del 2013

Dirección: Universidad de las Ciencias Informáticas (UCI), Edificio: 142, Apto: 203

Teléfono Oficina: +53 – 7 – 8358292. Teléfono Apto: +53 – 7 – 837 – 3466

E-mail: mailto:ksaria@uci.cu

AGRADECIMIENTOS

Quiero agradecer a las personas que han contribuido de una forma u otra al cumplimiento de este sueño.

Agradecer a las personas que han colaborado conmigo a lo largo de mi carrera.

Agradecer a mis amigos que han compartido los buenos y los malos momentos en la universidad.

Agradecer a mis tutores y en especial a Claudia que me ha guiado y soportado durante la realización de este trabajo.

Agradecer a mi familia completa por su apoyo incondicional.

Agradecer a mis padres que sin la guía de ellos hubiese sido imposible.

DEDICATORIA

A mi madre Inalbis Puente Rodríguez, a mi padre Cesar Enrique Nápoles Sevilla, a toda mi familia en general y a mis tutores.

RESUMEN

En el Centro de Informatización de la Gestión de Entidades de la Universidad de las Ciencias Informáticas se encuentra el Departamento de Tecnología que centra su labor en el desarrollo del marco de trabajo Sauxe, que es una base tecnológica para la construcción de aplicaciones web de gestión. En Sauxe se pueden desarrollar componentes como elementos de software que integran el sistema, pero se ha detectado que se dedica demasiado tiempo en la definición de los componentes y que no existe un control de los elementos que constituyen los componentes. La presente investigación hace una propuesta del desarrollo de un herramienta que permita “Gestionar componentes” en el marco de trabajo Sauxe, de esta forma contribuirá a disminuir el tiempo de definición de dicho componente y por consiguiente el tiempo de implementación del mismo. El desarrollo de la herramienta está basado en un modelo de componentes propuesto por el estudiante Abraham Calás, avalado por el Departamento de Tecnología. Se utilizan para el desarrollo de la investigación los métodos teóricos y empíricos: Análisis Histórico Lógico, Analítico Sintético, Modelación y el de Entrevista. Las herramientas utilizadas son el *NetBeans* como IDE de desarrollo, para el control de versiones *Subversion* y para el modelado *Visual Paradigm*. Se empleó el lenguaje PHP del lado del servidor y *Javascript del lado del cliente*. El proceso ingenieril esta guiado por el modelo de desarrollo propuesto por el centro CEIGE, donde se aplican las fases de desarrollo de software que se refieren en el modelo.

PALABRAS CLAVES: componentes, dependencias, eventos, observadores

ÍNDICE

INTRODUCCIÓN **14**

CAPÍTULO I. FUNDAMENTACIÓN TEÓRICA **18**

INTRODUCCIÓN 18

1.1 CONCEPTOS ASOCIADOS AL DOMINIO DEL PROBLEMA 18

1.2 MODELO DE COMPONENTES DEL MARCO DE TRABAJO SAUXE 19

1.3 ESTADO DEL ARTE 21

1.3.1 Gestionar componentes en los principales marcos de trabajo *21*

1.3.2 Symfony 2 ..... *22*

1.3.3 Zend Framework 2 ..... *23*

1.3.4 Spring *24*

1.3.5 OSGi *25*

1.3.6 Spring Dynamic Modules (Spring DM) *27*

1.3.7 Valoración de las herramientas existentes *27*

1.4 TENDENCIAS Y TECNOLOGÍAS ACTUALES 29

1.4.1 Metodología de desarrollo *29*

1.4.2 Herramientas CASE *29*

1.4.3 Herramientas para el desarrollo colaborativo *30*

1.4.4 Herramientas para el desarrollo *30*

1.4.5. Librerías y marcos de trabajo *31*

1.4.5 Lenguaje de Modelado *32*

1.4.6 Lenguajes de programación *33*

1.5 PATRONES 34

¿QUÉ ES UN PATRÓN? 34

1.5.1 Patrones de diseño *34*

1.5.2 Patrones Arquitectónicos *35*

1.6 CONCLUSIONES PARCIALES 36

CAPÍTULO II. PROPUESTA DE SOLUCIÓN **37**

INTRODUCCIÓN 37

2.1 MODELO CONCEPTUAL 37

2.1.1 Descripción del Modelo conceptual *37*
2.2 REQUISITOS 38
2.2.1. Requisitos funcionales *38*
2.2.3. Requisitos no funcionales *40*
2.3. MODELO DE DISEÑO 41
2.3.1. Diagrama de clases *41*
2.4. MODELO DE DATOS 43
2.5. CONCLUSIONES DEL CAPÍTULO 44
CAPÍTULO III. IMPLEMENTACIÓN Y PRUEBA 45
INTRODUCCIÓN 45
3.1 MODELO DE IMPLEMENTACIÓN 45
3.1.1 Diagrama de Componentes *45*
3.1.2 Diagrama de Despliegue *46*
3.2 ESTÁNDARES DE CODIFICACIÓN 47
3.2.1 Estándares de nomenclatura *47*
3.2.2 Estilo del código *49*
3.3 MÉTRICAS DE DISEÑO 51
3.3.1 Tamaño operacional de clase (TOC) *52*
3.3.2 Relación entre Clases (RC) *57*
3.3.3 Matriz interferencia de indicadores de calidad *61*
3.3.4 Valoración de los resultados de las métricas *62*
3.5 PRUEBAS DE SOFTWARE 63
3.5.1 Tipos de pruebas *63*
3.5.2 Pruebas de Caja Negra *67*
3.5.3 Valoración de los resultados de las pruebas de software. *68*
3.6 VALIDACIÓN DE LA INVESTIGACIÓN 69
3.6.1 Diseño de experimento *69*
3.7 CONCLUSIONES DEL CAPÍTULO 70
CONCLUSIONES GENERALES 72
RECOMENDACIONES 73
BIBLIOGRAFIA 74

ANEXOS ..... ¡ERROR! MARCADOR NO DEFINIDO.

ANEXO 1 DISEÑOS DE CASOS DE PRUEBA ..... **¡ERROR! MARCADOR NO DEFINIDO.**

ANEXO 2 ENCUESTA PARA VALIDAR LA HERRAMIENTA GESTIONAR COMPONENTE ..... **¡ERROR! MARCADOR NO DEFINIDO.**

ANEXO 3 DESCRIPCIÓN DE REQUISITOS FUNCIONALES ..... **¡ERROR! MARCADOR NO DEFINIDO.**

GLOSARIO DE TÉRMINOS ..... ¡ERROR! MARCADOR NO DEFINIDO.

ÍNDICE DE FIGURAS

Figura 1: Interfaces propuestas para un componente en Sauxe por (Abraham, 2013). 20
Figura 2: Ejemplo de configuración de un servicio llamado “Example1”. 23
Figura 3: Ejemplo de consumo a un servicio llamado “example”. 23
Figura 4: Ejemplo de módulo que brinda un servicio llamado “Auth”. 24
Figura 5: Ejemplo que consume el servicio “Auth” en un controlador. 24
Figura 6: Ejemplo de configuración de componentes con el contenedor de Spring. 25
Figura 7: Ejemplo de configuración a través del fichero MANIFEST.MF. 26
Figura 8: Ejemplo de registro de un servicio en OSGi. 26
Figura 9: Ejemplo de consumo de un servicio en OSGi. 26
Figura 10: Ejemplo de exposición de un bean de Spring como un servicio OSGi. 27
Figura 11: Fases del ciclo de vida de proyectos del CEIGE. 29
Figura 12: Modelo conceptual. 37
Figura 13: Diagrama de clases de la herramienta “Gestionar componentes”. 42
Figura 14: Modelo de Datos de la herramienta “Gestionar componentes”. 43
Figura 15 Diagrama de componentes. 46
Figura: 16 Modelo de despliegue. 47
Figura 17: Estilo del código: Sangría o indexado. 50
Figura 18: Estilo del código: Sangría o indexado. 51
Figura 19: Representación de las clases según la cantidad de operaciones. 55
Figura 20: Resultados de la evaluación de la métrica TOC para el atributo de calidad Responsabilidad. 56
Figura 21: Resultados de la evaluación de la métrica TOC para el atributo de calidad Complejidad. 56
Figura 22: Resultados de la evaluación de la métrica TOC para el atributo de calidad Reutilización. 56
Figura 23: Resultado de la aplicar la métrica TOC. 57
Figura 24 Resultados de la evaluación de la métrica RC para el atributo de calidad Acoplamiento. 60
Figura 25 Resultados de la evaluación de la métrica RC para el atributo de calidad Complejidad de Mantenimiento. 60
Figura 26 Resultados de la evaluación de la métrica RC para el atributo de calidad Reutilización. 60
Figura 27 Resultados de la evaluación de la métrica RC para el atributo de calidad Cantidad de Pruebas. 61

Figura 28: Resultado de la aplicación de la métrica RC. 61
Figura 29 Matriz de cubrimiento. 62
Figura 30 Grafo de flujo asociado a la funcionalidad cargarCarpetasAppAction() 64
Figura 31: Resultados de las pruebas funcionales. 68
Figura 32: Resultados de las encuestas. 70
Figura 33: Prototipo funcional de la interfaz registrar componente. ..... **¡Error! Marcador no definido.**
Figura 34: Prototipo funcional de la interfaz modificar componente. ..... **¡Error! Marcador no definido.**
Figura 35: Prototipo funcional de la interfaz de confirmación de eliminación. **¡Error! Marcador no definido.**
Figura 36: Prototipo funcional de la interfaz de confirmación de habilitación. **¡Error! Marcador no definido.**

ÍNDICE DE TABLAS

Tabla 1: Prefijos para tipos de datos. 48
Tabla 2: Atributos de calidad evaluados por la métrica TOC. 52
Tabla 3: Criterios de evaluación para la métrica TOC. 52
Tabla 4: Atributos de calidad evaluados por la métrica RC. 53
Tabla 5: Criterios y categorías de evaluación 53
Tabla 6: Resultado de la métrica TOC. 54
Tabla 7: Instrumento de evaluación de la métrica TOC. 54
Tabla 8: Tamaño de clases. 55
Tabla 9: Atributos de calidad que evalúa RC. 57
Tabla 10: Criterios de evaluación de la métrica RC. 58
Tabla 11 : Resultado de la métrica RC. 58
Tabla 12 : Instrumento de evaluación de la métrica RC. 58
Tabla 13 Cantidad de dependencias por clases 59
Tabla 14 Evaluación según cantidad de relaciones. 59
Tabla 15 Resultado de la matriz interferencia de indicadores de calidad. 62
Tabla 16: Escenario del caso de prueba Registrar componente. ..... **¡Error! Marcador no definido.**
Tabla 17: Descripción de variables del caso de prueba Registrar componente. **¡Error! Marcador no definido.**
Tabla 18: Juego de datos a probar del caso de prueba Registrar componente. **¡Error! Marcador no definido.**
Tabla 19: Descripción del requisito Registrar componente. ..... **¡Error! Marcador no definido.**
Tabla 20: Descripción del requisito Modificar componente. ..... **¡Error! Marcador no definido.**
Tabla 21: Descripción del requisito Deshabilitar componente. ..... **¡Error! Marcador no definido.**
Tabla 22: Descripción del requisito Habilitar componente. ..... **¡Error! Marcador no definido.**

INTRODUCCIÓN

La sociedad actual se encuentra cada vez más inmersa en el desarrollo tecnológico, siendo de interés la creación de sistemas informáticos de mayor complejidad, con una estructura que satisfaga las necesidades de miles de usuarios y que sean construidos en el menor tiempo posible. Adaptando la idea de Sommerville (Profesor de Ingeniería de Software en la universidad de Strathclyde Andrews) en el presente contexto, "*la única forma de tratar la complejidad y entregar un software en el menor tiempo posible es reutilizar componentes de software a tener que reimplementarlos"* (Sommerville, 2005), es decir, utilizar los fundamentos de la ingeniería basada en componentes.

Un componente de software es una unidad de composición con interfaces especificadas contractualmente y dependencias de contexto explícitas únicamente. Puede ser desplegado de forma independiente y está sujeto a la composición por terceras partes. (Szyperski, 2002)

La definición de la arquitectura basada en componentes cubre aspectos únicamente lógicos y es totalmente independiente de la tecnología con la cual se implementarán los mismos y sobre la cual se hará el despliegue del sistema. Esta vista lógica nos permite medir el nivel de acoplamiento del sistema y razonar sobre los efectos de modificar o reemplazar un componente. La independencia de la tecnología nos permite abstraernos de los tecnicismos de éstas, así como elegir la más apta dependiendo del sistema que se esté desarrollando. (Vignaga, Perovich, 2005)

En Cuba, la Universidad de las Ciencias Informáticas (UCI) tiene entre sus proyecciones la construcción de sistemas informáticos, algunos de los cuales utilizan una arquitectura basada en componentes, viéndose reflejado en el Centro de Informatización de la Gestión de Entidades (CEIGE), que tiene como objetivo crear sistemas que apoyen a la informatización de los procesos operativos y la toma de decisiones de las entidades. Como base tecnológica para el desarrollo de sus productos se creó el marco de trabajo Sauxe. El mismo contiene un conjunto de componentes reutilizables que provee la estructura genérica y el comportamiento para una familia de abstracciones, logrando una mayor estandarización, flexibilidad, integración y agilidad en el proceso de desarrollo.

La definición de componentes que se realiza en el marco de trabajo Sauxe comprende un periodo de tiempo de dos jornadas de trabajo, resultado arrojado después de haber realizado encuestas abiertas y talleres con especialistas del departamento. Actualmente para realizar el registro de un componente se debe insertar la información en un documento Excel, donde se incluye las características generales de los

mismos, detectando que no se prescinde de todos los elementos que contiene un componente como los servicios que consume y que brinda, las operaciones que tienen los servicios, las dependencias entre los componentes, ni los eventos generados u observados, además que se hace muy engorroso llevar el control del estado de cada uno de los componentes que se van desarrollando en Sauxe. También se debe de tener en cuenta los siguientes factores:

-Se necesita un personal que tenga los conocimientos necesarios para crear un componente.

-Se emplea demasiado tiempo para definir el componente.

- Durante la definición del componente se pueden incurrir en la introducción de errores.

Luego de definir un componente se procede a su estructuración, estableciendo la perspectiva de empaquetamiento que considera un componente como unidad de empaquetamiento, la perspectiva de servicio que considera un componente como proveedor de servicios y la perspectiva de integridad que considera un componente como un elemento encapsulado. Lo que implica que si no se define correctamente un componente, su construcción puede acarrar errores.

A partir de la situación descrita con anterioridad, se define el siguiente **problema a resolver:** ¿Cómo disminuir el tiempo de desarrollo de los componentes en el marco de trabajo Sauxe?

Con el fin de darle respuesta al problema se identifica el siguiente **objetivo general:** desarrollar una herramienta que permita “Gestionar componentes” en el marco de trabajo Sauxe para disminuir el tiempo de desarrollo de los mismos.

Para darle cumplimiento al objetivo general se propone el proceso de gestión de componentes en aplicaciones web como **objeto estudio**, siendo el **campo de acción** el desarrollo de componentes en el marco de trabajo Sauxe.

A partir del objetivo general se desglosan los siguientes **objetivos específicos:**

1. Construir el marco teórico de la investigación sobre el proceso de gestión de componentes en sistemas de gestión empresariales.
2. Realizar el análisis y diseño de la herramienta para “Gestionar componentes” en el marco de trabajo Sauxe.
3. Realizar la implementación de la herramienta para “Gestionar componentes” en el marco de trabajo Sauxe.

4. Validar la herramienta para “Gestionar componentes” en el marco de trabajo Sauxe aplicando pruebas de caja negra y pruebas de caja blanca.
5. Validar la investigación realizando un cuasiexperimento.

Se tiene como **idea a defender** de la presente investigación**:** Si se desarrolla una herramienta para “Gestionar componentes” en el marco de trabajo Sauxe se logrará disminuir el tiempo de desarrollo de los componentes.

Los métodos científicos de investigación son la forma de abordar la realidad con el propósito de descubrir su esencia y sus relaciones. Durante el desarrollo de la investigación se utilizarán un conjunto de métodos científicos para la obtención y procesamiento de la información y la elaboración de las conclusiones. Los cuales se mencionan a continuación:

Métodos Teóricos:

1. **Método Análisis Histórico – Lógico:** se aplica con el objetivo de conocer la existencia de marcos de trabajo que realicen el proceso de gestión de componentes, para tener una mayor visión de la forma de ejecución de estos procesos, en este tipo de sistema y utilizar estos como puntos de referencia.
2. **Método Analítico – Sintético:** se aplica con el objetivo de analizar todos los documentos relacionados con la estructura de los componentes en el marco de trabajo Sauxe, la estandarización de los mismos así como sus interfaces para poder gestionarlos.
3. **Método Modelación:** se aplica para generar los artefactos que se construyen durante el proceso de Ingeniería de Software, como modelo para comprender las propiedades, funcionalidades y relaciones que contiene la herramienta “Gestionar componentes”.

Métodos Empíricos:

1. **Método Entrevista:** se aplica para obtener información sobre el desarrollo de componentes en sistemas de gestión empresariales que utiliza el CEIGE.

Con el desarrollo de la investigación se espera alcanzar la construcción de una herramienta que permita “Gestionar componentes”, así como la documentación técnica de la misma. Dicha herramienta contribuirá a disminuir el tiempo de desarrollo de los componentes en el marco de trabajo Sauxe.

El presente trabajo de diploma se encuentra estructurado en tres capítulos que estarán guiados por los siguientes temas a tratar:

CAPÍTULO I. FUNDAMENTACIÓN TEÓRICA

Está centrado en el análisis de los elementos y conceptos relacionados con la estructura de la herramienta para “Gestionar componentes”. Se realiza una presentación del estado del arte en el mundo de las aplicaciones que proponen soluciones similares a la que se quiere brindar con la herramienta para “Gestionar componentes” en el marco de trabajo Sauxe. Además se hace una descripción de las herramientas y tecnologías que se emplearán para la construcción de la solución.

CAPÍTULO II. PRESENTACIÓN DE LA SOLUCIÓN PROPUESTA

Está centrado en modelar el análisis y diseño de la herramienta para “Gestionar componentes”. Además se generan los artefactos relacionados con la implementación.

CAPÍTULO III. IMPLEMENTACIÓN Y VALIDACIÓN DE LA SOLUCIÓN

Se realiza la validación de los requisitos identificados, aplicando técnicas de validación. Se aplican métricas para la validación del diseño de la solución propuesta. Se presentan los resultados de las pruebas de caja blanca y caja negra realizadas a la solución. Se presentan los resultados del cuasiexperimento, utilizando como instrumento las encuestas.

CAPÍTULO I. FUNDAMENTACIÓN TEÓRICA

Introducción

El presente capítulo constituye el marco teórico de la investigación a realizar. En el mismo se describen los conceptos necesarios para tener una mayor comprensión del problema propuesto, profundizando en temas como la gestión de componentes de software. Además se realiza una reseña sobre el contexto en el que se encuentra el proceso de gestión de componentes en marcos de trabajo a nivel global. Se describen también las tecnologías a utilizar durante el proceso de desarrollo de software a partir de lo que propone el documento "Vista de Entorno de Desarrollo Tecnológio-Sauxe_v2.0".

1.1 Conceptos asociados al dominio del problema

Componente:

-Un elemento de software que se ajusta a un modelo de componentes y que puede ser desplegado y compuesto de forma independiente sin modificación según un estándar de composición. (Councill y Heineman, 2001)

Componente de software:

- En este trabajo se utiliza la definición de Sterling Software, que plantea que un componente de software es:

- Una parte reutilizable que encapsula elementos del modelo (por ejemplo una DLL[1], documentos, tablas, biblioteca, etc.).
- Un paquete de software el cual ofrece servicios a través de sus interfaces.
- Un paquete de software que puede ser usado para construir aplicaciones o componentes más grandes. (Software, 2000)

Elementos que describen un componente:

Servicio:

-Un servicio es un objeto PHP (acrónimo de *Hypertext Preprocessor*) que realiza alguna tarea "global", un objeto creado para un propósito definido. Cada servicio se utiliza en toda una aplicación siempre que se necesite la funcionalidad específica que este proporciona. Al explicitar un servicio en alguno de los archivos de configuración también se explicitan las operaciones que debe recibir este servicio así como la ubicación de la clase que implementa el servicio.

[1] Dynamic Library Link, biblioteca de vínculos dinámicos.

Dependencia:

-Es un servicio que necesitan los componentes para estar funcionales. Cada dependencia puede ser utilizada por varios componentes.

Eventos:

-Mecanismo de comunicación por el que se pueden propagar las situaciones que ocurren en un sistema de forma asíncrona. La comunicación entre emisores y receptores de los eventos se puede realizar tanto de forma directa como indirecta. Los eventos suelen ser emitidos por los componentes para avisar a los componentes de su entorno de cambios en su estado o de circunstancias especiales, como pueden ser las excepciones.

Tipos de Eventos

-Observados: son los eventos que observan otros componentes.

-Generados: son los eventos que generan cada componente.

Interfaces:

-Las interfaces de un componente determinan tanto las operaciones que el componente implementa como las que precisa utilizar de otros componentes durante su ejecución (Fuentes, y otros, 2001). En términos prácticos, las interfaces de los componentes serán interfaces PHP, que contendrán comentarios en PHPNotation (Notación para el Modelado de Procesos de Negocio por sus siglas en inglés *Business Process Modeling Notation*) para especificar los tipos de datos de las operaciones de cada servicio y/o el tipo de dato del retorno, en caso de que existan.

-**Interfaces Proveídas:** permite que un componente provea los servicios a otros componentes.

-**Interfaces Requeridas:** permite que un componente requiera los servicios que otros componentes proveen.

Metadatos:

-Descripciones estructuradas y opcionales que están disponibles de forma pública para ayudar a localizar objetos o datos estructurados y codificados que describen características para ayudar a identificar, descubrir, valorar y administrar las instancias descritas.

1.2 Modelo de componentes del marco de trabajo Sauxe

Un modelo de componentes es una definición de los estándares para la creación de componentes, documentación y despliegue. Estos estándares son utilizados por los desarrolladores para asegurar que estos puedan interoperar. Se han propuesto muchos modelos de componentes, pero los modelos más

importantes son el modelo de CORBA de OMG, el modelo *Enterprise Java Beans de Sun* y el modelo COM+ de Microsoft. (Somerville, 2005)

Para el desarrollo de la propuesta a solucionar se ha seleccionado el modelo de componentes para el marco de trabajo Sauxe, propuesto por Abraham Calás Torres (Abraham, 2013), en el mismo, se realiza un estudio de la infraestructura del marco de trabajo, como deben ser definidos los componentes y qué mecanismo de integración utilizar, además de que realiza un análisis y describe los modelos más importantes a nivel internacional, tomando de estos las utilidades que puedan ser adecuadas al marco de trabajo Sauxe.

En el modelo se especifica cuáles serán los tipos de interfaces que conforman el componente, mostrándose en la Figura 1.

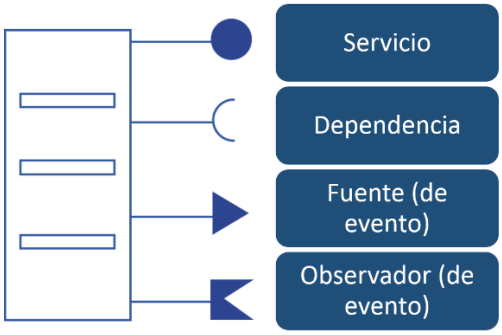


Figura 1: Interfaces propuestas para un componente en Sauxe por (Abraham, 2013).

El modelo establece cómo documentar los componentes. El proveer una documentación descriptiva de los componentes, permite a los clientes comprenderlos y utilizarlos en aplicaciones para los que no fueron concebidos, siempre y cuando estén construidas bajo el mismo modelo. Una documentación adecuada facilita la búsqueda en repositorios de componentes, evaluación, adaptación a nuevos entornos, integración con otros componentes y acceso a información de soporte (Montilva, 2003). La carencia de documentación de los componentes existentes es la causa principal de los atrasos en la realización de las tareas según los encuestados, una correcta documentación pudiera causar un efecto positivo en los cronogramas de trabajo. Se propone que la documentación contenga los siguientes datos de los componentes (Abraham, 2013):

1. Nombre del componente.
2. Requisitos funcionales y no funcionales.
3. Interfaces que brinda y necesita.
4. Eventos generados y suscritos.

5. Versión del componente.
6. Índice de peso del componente.
7. Otros datos de interés.

Mecanismo de integración

El modelo especifica las reglas para la integración e independencia de los componentes. Parte del mecanismo de integración es todo el código escrito en el marco de trabajo que se encarga de comunicar los componentes brindando seguridad y transparencia en las operaciones. Esta parte también se describe en el modelo de componentes, para que los usuarios conozcan las funcionalidades que brinda el marco de trabajo en cuanto a la gestión de componentes. En este aspecto se debe estudiar qué elementos deben ser cambiados en la actual configuración, la cual limita la modularización ya que los servicios que ofrecen los componentes están declarados en un solo fichero XML (Lenguaje de marcas extensible por sus siglas en inglés *eXtensible Markup Language*) y es muy pobre en cuanto a la información que se brinda de los mismos. (Abraham, 2013)

Convención de nombres

Se especifica en el modelo una convención de nombres únicos para que los componentes se puedan distribuir y puedan ser encontrados sin conflictos. Se recomienda el uso de nombres jerárquicos ya que son fáciles de entender. Por ejemplo; sauxe.seguridad.roles, (Abraham, 2013)

Soporte para evolución

Inevitablemente, a medida que surgen nuevos requerimientos, los componentes tendrán que ser cambiados o remplazados. El modelo de componentes incluye reglas para regular cuándo y cómo se permite el remplazo de componentes (Somerville, 2005). Un ejemplo de cuando un componente podría ser remplazado, es cuando lleve un largo período de tiempo desplegado y las interfaces que proporcionan o generan eventos no son requeridas u observadas por ningún otro componente. (Abraham, 2013)

1.3 Estado del arte

1.3.1 Gestionar componentes en los principales marcos de trabajo

En la actualidad al iniciar el desarrollo de una herramienta se hace un estudio del estado de las principales soluciones que existen, con el objetivo de solucionar la problemática existente. Los *frameworks* o marcos de trabajo integran un conjunto de componentes que resuelven muchos de los escenarios que pueden encontrarse en el transcurso de un desarrollo, estos permiten agilizar el proceso y obtener resultados en breves períodos de tiempo. Como son soluciones muy genéricas, en la mayoría de los casos no abarcan

todos los posibles contextos en el que se va a desempeñar una aplicación. Con la necesidad existente en el país de contar con un sistema ERP (Proceso de relación de empresas pos sus siglas en inglés Enterprise Relantionship Proyect) que mejore y agilice la gestión de los recursos, surge el proyecto Sauxe perteneciente al CEIGE, por la premura de la tarea se necesita dar una respuesta en un período inmediato. (Oiner, 2010) En este caso es imprescindible el estudio de soluciones homólogas de la gestión de componentes que puedan ser reutilizadas y de esta forma disminuir el tiempo de desarrollo, el esfuerzo y los costos. En el mundo existen un conjunto de marcos de trabajo que proveen una cantidad de componentes reutilizables, dentro de ellos se estudiarán los más utilizados, desarrollados sobre tecnologías libres para evaluar si es factible reutilizar algunas de sus herramientas y de esta forma dar una respuesta en un período de tiempo menor.

1.3.2 Symfony 2

Symfony es un *framework* de desarrollo web, implementado en lenguaje PHP, que cuenta en la actualidad con dos ramas estables. La versión 1.4 (la última de la primera generación) y la versión 2, lanzada oficialmente en julio de 2011. (Mata, 2012)

La base de la nueva filosofía de trabajo de Symfony 2 son los bundles: directorios que contienen todo tipo de archivos dentro una estructura jerarquizada. Los bundles de las aplicaciones suelen contener clases PHP y archivos web (JavaScript, CSS e imágenes). No obstante, no existe ninguna restricción sobre lo que puedes incluir dentro de un bundle. (Eguiluz, 2012)

En Symfony 2, con el objetivo de proporcionar ayuda al crear instancias, organizar y recuperar objetos (servicios) en una aplicación, se implementó un contenedor de servicios que permite la estandarización y centralización de la forma en que se construyen los objetos en una aplicación. Symfony 2 proporciona facilidades para el trabajo, y acentúa una arquitectura que promueve el código reutilizable y disociado. (Potencier, 2011)

La configuración de los *bundles* se puede realizar por dos vías, a través del archivo config.yml (Los documentos YML son archivos de texto asociados con Javascript YML Document o YAML) en la carpeta config/ del proyecto, o a través del archivo services.yml que se encuentra en la carpeta config/ dentro del bundle; de cualquier forma en esta configuración solo se especifican los servicios que provee un bundle, pudiéndose consumir dichos servicios desde cualquier parte de la aplicación. Esta configuración se registra en caché desde la primera petición.

Un servicio es cualquier objeto PHP que realiza alguna tarea “global”, un objeto creado para un propósito X. Cada servicio se utiliza en toda una aplicación siempre que se necesite la funcionalidad específica que este proporciona.
Al explicitar un servicio en alguno de los archivos de configuración también se explicitan las operaciones que debe recibir este servicio así como la ubicación de la clase que implementa el servicio.

```
services:
   example1:
      class:      Desymfony\ExampleBundle\Controller\Example1
```

Figura 2: Ejemplo de configuración de un servicio llamado “Example1”.

```
$var=$this->get('example'); //llamada al servicio de nombre example
```

Figura 3: Ejemplo de consumo a un servicio llamado “example”.

Dada la concepción del mecanismo para la integración entre componentes en *Symfony* es posible el consumo de los servicios que estos brindan desde cualquier *bundle* de una aplicación. Entre los componentes existe un escaso nivel de dependencia pero estas no quedan explícitas en la configuración de los componentes.
En *Symfony 2*, los componentes consumen servicios que brindan otros componentes, a diferencia de lo que en la actualidad ocurre en el marco de trabajo Sauxe, ya que directamente se consumen métodos que se encuentran en otros componentes.
Una deficiencia notable en *Symfony 2* es que sus componentes no declaran sus dependencias, siendo necesario para los desarrolladores inspeccionar el código.

1.3.3 Zend Framework 2

Zend Framework 2 (ZF2) es un *framework* de código abierto para desarrollar aplicaciones web y servicios web con PHP5.3. Es una implementación que usa código 100% orientado a objetos. (Zend Technologies, 2012)
Esta nueva versión del *framework* liberada en el 2012, introduce un enfoque al desarrollo de aplicaciones basadas en módulos, que mejora la flexibilidad y organización del código con respecto a versiones anteriores. ZF2 utiliza una nueva arquitectura basada en servicios que se vale de un contenedor de inyección de dependencias para crear los objetos que son solicitados por otros módulos.

Cualquier módulo puede actuar como proveedor de servicios. Para hacerlo se implementa el método *getServiceConfig()* en donde se definen en arreglos (*arrays*) de PHP los servicios a brindar.

```
class Module {
    public function getServiceConfig() {
        return array(
          'service_manager' => array(
             'services' => array(
               'Auth' => new SomeModule\Authentication\AuthenticationService(),
                ),
        );
    }}
```

Figura 4: Ejemplo de módulo que brinda un servicio llamado “Auth”.

Si se quiere consumir el servicio en algún controlador, se hace uso del contenedor de inyección de dependencias que se encarga de crear el objeto.

```
$serviceManager = $this->getServiceLocator();
$apiServiceResult = $serviceManager->get('Auth');
```

Figura 5: Ejemplo que consume el servicio “Auth” en un controlador.

Al configurar un módulo en ZF2 se pueden especificar los servicios que se brindan y las dependencias que tienen los servicios, no así las dependencias que tienen los módulos, por lo que se hace necesario revisar el código para encontrarlas. El uso de *arrays* PHP para la configuración de los módulos hace complejo comprobar si está correcta la estructura.

1.3.4 Spring

Spring es un *framework* creado con el objetivo de facilitar el desarrollo de aplicaciones en Java que promueve el bajo acoplamiento a través de la inversión de control. El proyecto de desarrollo de este *framework* se inició en el año 2003. (Breidenbach, y otros, 2005)

La clase BeanFactory es el contenedor de Spring, cuya responsabilidad es crear y brindar componentes. BeanFactory carga por defecto las configuraciones de todos los componentes, pero estas no serán utilizadas ni se creará ninguna instancia mientras no sea necesario. Cuando se realice una llamada al método getBean() pasándole por parámetro el nombre del componente con el que se desea trabajar, BeanFactory creará una instancia del componente configurando sus propiedades a través de la inyección de dependencias.

En Spring la configuración de un componente se puede hacer a través de archivos XML, anotaciones Java o texto plano. El siguiente código XML muestra cómo se inyecta la dependencia “notifier” dentro del componente “businessService”.

```
<beans>
    <bean id="notifier" class="foo.bar.notify.JmsNotifier" />
    <bean id="businessService" class="for.bar.service.impl.BusinessServiceImpl">
        <property name="notifier" ref="notifier" />
    </bean>
</beans>
```

Figura 6: Ejemplo de configuración de componentes con el contenedor de Spring.

En Spring, los componentes no son responsables de gestionar su asociación con otros componentes, sino que se les proporcionan referencias a los componentes de los que dependen a través del contenedor. (Breidenbach, y otros, 2005)

Este *framework* al pertenecer a una plataforma distinta a la del marco de trabajo Sauxe, no es utilizable en términos prácticos. Una de las deficiencias halladas es que en la configuración de sus componentes son identificables sus dependencias pero no los servicios que proveen.

1.3.5 OSGi

OSGi es un *framework* que define un sistema de módulos dinámicos para Java, que ofrece un mejor control sobre la estructura del componente, la capacidad de gestionar de forma dinámica el ciclo de vida del código, y un enfoque de acoplamiento flexible del código. (Hall et al. 2011)

Una de las limitantes de Java es que proporciona algunos aspectos de la modularidad en la forma de orientación a objetos, pero nunca fue pensado para apoyar la programación de grandes módulos. OSGi suple la ausencia de modularidad de las aplicaciones desarrolladas en Java, proporcionando una plataforma completa y ligera para la implementación de aplicaciones basadas en componentes y orientadas a servicios dentro de una JVM (por las siglas en inglés de Máquina Virtual de Java).

Los componentes en OSGi también llamados bundles son unidades físicas de modularidad en forma de ficheros JAR que contienen códigos, recursos y metadatos (Hall, y otros, 2011). El fichero META-INF/MANIFEST.MF dentro del JAR (un archivo de java por sus siglas en inglés *Java Archive*) es quien contiene las características de modularidad del *bundle*, que pueden ser: nombre, versión, descripción, paquetes externos de los que depende, paquetes internos que exporta y otras. Con esta información el

framework puede de manera automática identificar *bundles* dentro de la aplicación y ponerlos a disposición de otros bundles.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Spring DM Hello World
Bundle-SymbolicName: com.manning.sdmia.helloworld
Bundle-Version: 1.0.0
Export-Package: com.manning.sdmia
```

Figura 7: Ejemplo de configuración a través del fichero MANIFEST.MF.

Para brindar servicios, OSGi promueve la separación de la interfaz y la implementación. Los servicios de OSGi son interfaces de Java que constituyen el contrato conceptual entre los proveedores y los clientes de servicios.

Al tener definido las interfaces e implementaciones del servicio que se quiere brindar, el próximo paso es documentarlo en el registro de servicios de OSGi utilizando el método registerService del objeto bundleContext.

```
ServiceRegistration registration =
    bundleContext.registerService(interfaces, new LSE(), metadata);
```

Figura 8: Ejemplo de registro de un servicio en OSGi.

Para consumir el servicio desde otro bundle se utiliza el método getService del objeto bundleContext. Este método necesita como parámetro una referencia que le permita buscar sobre los metadatos de los servicios registrados. El resultado devuelto es una instancia de la clase registrada en el registro de servicios.

```
StockListing listing =
    (StockListing) bundleContext.getService(reference);
```

Figura 9: Ejemplo de consumo de un servicio en OSGi.

Este *framework* al pertenecer a una plataforma distinta a la del marco de trabajo Sauxe no es utilizable en términos prácticos. Entre los aspectos positivos se encuentra que la configuración del componente está ubicada en el mismo componente y el contrato de los servicios se establece a través de interfaces de java.

1.3.6 Spring Dynamic Modules (Spring DM)

Spring DM proporciona una potente solución modular para el desarrollo de aplicaciones basadas en Spring que pueden desplegarse en un entorno de ejecución OSGi. El objetivo de la tecnología es hacer el trabajo de Spring y OSGi juntos de una manera sencilla, así como hacer frente a las limitaciones de las dos tecnologías. (Cogoluegnes, y otros, 2011)

Spring DM utiliza un nuevo tipo de *bundle* que importa o exporta paquetes de Java, pero que no debe preocuparse por la inicialización y exportación de sus servicios o por buscar sus dependencias, ya que estas tareas serán realizadas por el *framework* haciendo uso de las configuraciones en XML.

En la siguiente imagen se aprecia una configuración XML que muestra cómo exponer un *bean* de Spring como un servicio de OSGi. Al hacer uso del *bean* por algún módulo del sistema, Spring DM registra el servicio de manera automática y lo destruye al terminar su uso.

```
<beans>
    <osgi:service ref="myService"
                  interface="com.manning.spring.osgi.simple.MyService"/>
    <bean id="myService"
          class="com.manning.spring.osgi.simple.impl.MyServiceImpl"/>
</beans>
```

Figura 10: Ejemplo de exposición de un *bean* de Spring como un servicio OSGi.

Con este *framework* sucede similar a los casos anteriores, no es utilizable en términos prácticos. Sin embargo posee aspectos teóricos de alta validez que pueden ser tenidos en cuenta para el diseño de la solución. Entre estos se encuentran la sencillez en términos de configuración de los componentes, la utilización de la inversión de control y la transparencia para el cliente de los servicios, respecto al mecanismo de integración.

1.3.7 Valoración de las herramientas existentes

A partir de la descripción de las herramientas presentadas, se realizó un análisis y descripción de las principales características que permiten determinar en términos prácticos qué es un componente y cómo deben interactuar entre ellos. Para lograr esto, en esta investigación se estudió cómo funcionan los mecanismos de gestión de componentes de otros reconocidos marcos de trabajos, desarrollados sobre tecnologías libres para evaluar si es factible reutilizar algunas de sus herramientas. Los indicadores seleccionados para medir estas herramientas son:

- Si gestiona o no los servicios

- Si gestiona o no las dependencias
- Ventajas y deficiencias de las herramientas

Symfony 2: el análisis de este *framework* arrojó que existe un escaso nivel de dependencia pero estas no quedan explícitas en la configuración de los componentes. Los componentes consumen servicios que brindan otros componentes, a diferencia de lo que ocurre en la actualidad en el marco de trabajo Sauxe, ya que directamente se consumen métodos que se encuentran en otros componentes.

Una deficiencia notable en *Symfony* 2 es que sus componentes no declaran sus dependencias, siendo necesario para los desarrolladores inspeccionar el código.

ZendFramework2: al configurar un módulo en ZF2 se pueden especificar los servicios que se brindan y las dependencias que tienen los servicios, no así las dependencias que tienen los módulos, por lo que se hace necesario revisar el código para encontrarlas. El uso de *arrays* PHP para la configuración de los módulos hace complejo comprobar si está correcta la estructura.

Spring: este *framework* al pertenecer a una plataforma distinta a la del marco de trabajo Sauxe, no es utilizable en términos prácticos. Una de las deficiencias halladas es que en la configuración de sus componentes son identificables sus dependencias pero no los servicios que proveen.

OSGi: este *framework* al pertenecer a una plataforma distinta a la del marco de trabajo Sauxe no es utilizable en términos prácticos. Entre los aspectos positivos se encuentra que la configuración del componente está ubicada en el mismo componente y el contrato de los servicios se establece a través de interfaces de java.

SpringDM: con este *framework* sucede similar a los casos anteriores, no es utilizable en términos prácticos. Sin embargo posee aspectos teóricos de alta validez que pueden ser tenidos en cuenta para el diseño de la solución. Entre estos se encuentran la sencillez en términos de configuración de los componentes, la utilización de la inversión de control y la transparencia para el cliente de los servicios, respecto al mecanismo de integración.

Pese a que todos estos marcos de trabajo utilizan modernas técnicas de gestión de componentes como la inyección de dependencias, solo SpringDM ofrece verdadera independencia y modularidad en los componentes. La herramienta “Gestionar componentes” que se va a implementar en este trabajo es una solución similar a la de SpringDM, solo que en una plataforma completamente distinta.

1.4 Tendencias y tecnologías actuales

1.4.1 Metodología de desarrollo

La producción del CEIGE se concentra en el desarrollo de proyectos generalmente de gran magnitud, por lo que se hace necesario contar con un modelo estandarizado, que establezca las distintas fases por las que se debe transitar y el conjunto de artefactos a generar en cada una de ellas. Teniendo como precedente dicha necesidad y en colaboración con las distintas líneas de desarrollo donde se ejecutan cada uno de estos proyectos, se ha decidido usar la metodología descrita en el documento "CEIGE-Modelo de Desarrollo de Software v1.2". Dicho documento detalla el ciclo de vida de proyectos del centro con la incorporación de los distintos subprocesos definidos por el Nivel II de CMMI (*Capability Maturity Model Integration*), certificación obtenida por el centro en julio de 2011 y reconocida por el SEI (*Software Engineering Institute*) como aval de la calidad de su proceso de desarrollo de software.

Para el ciclo de vida de los proyectos del CEIGE se tienen en cuenta las fases y actividades por áreas de procesos que plantea el Nivel II de CMMI establecido en la UCI. En la Figura 11 que a continuación se presenta se pueden apreciar el total de fases por las que pueden transitar los proyectos del centro, pudiendo realizar iteraciones a partir de la fase de Modelado del negocio. El conjunto de fases propuestas abarca el total de acciones que se realizan en las distintas líneas de desarrollo para la elaboración del servicio o producto final; sin embargo, se debe adaptar a las características particulares del proyecto que puede que no lleve a cabo alguna de estas fases o la elaboración de algunos de sus artefactos.

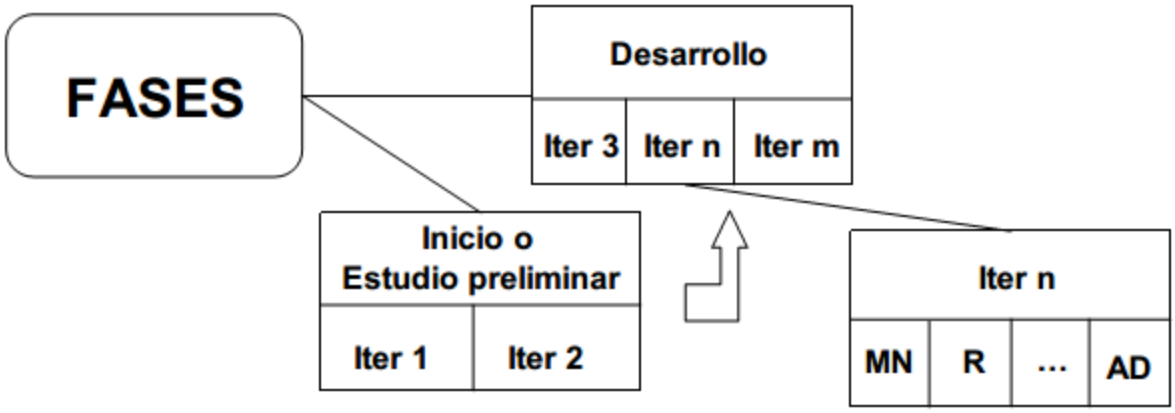


Figura 11: Fases del ciclo de vida de proyectos del CEIGE.

1.4.2 Herramientas CASE

Visual Paradigm

Visual Paradigm es una herramienta CASE (*Computer Aided Software Engineering* o Ingeniería de Software Asistida por Computadora) profesional que soporta el ciclo de vida completo del desarrollo de software: análisis y diseño orientados a objetos, construcción, pruebas y despliegue. Esta herramienta ayuda a construir aplicaciones con calidad y con un menor coste, además de disminuir el periodo de desarrollo. Permite dibujar todos los tipos de diagramas de clases, código inverso, generar código desde diagramas y documentación. La herramienta CASE también proporciona abundantes tutoriales, demostraciones interactivas y proyectos en UML (*Unified Modeling Language* o Lenguaje Unificado de Modelado). Para los ingenieros y arquitectos de software es excelente ya que permite centralizar y planificar las ideas, además de mejorar la productividad en el desarrollo y mantenimiento del software, aumentar la calidad del mismo, reducir el tiempo de desarrollo y mantenimiento, mejorar la planificación de un proyecto y realizar una gestión global en todas las fases de desarrollo de software con una misma herramienta. Esta herramienta puede ayudar en todos los aspectos del ciclo de vida del software. La mayor parte de los equipos de desarrollo del CEIGE cuentan con un dominio medio o alto de la herramienta, aspecto que le da un valor agregado a la decisión de adoptar esta solución para el modelado del sistema (K. Prada Nicot, H.y.S.G, 2009).

1.4.3 Herramientas para el desarrollo colaborativo

Subversion

Subversion es un sistema centralizado de control de versiones de código abierto. Se caracteriza por su fiabilidad como un refugio seguro para los datos valiosos, la simplicidad de su modelo, uso y capacidad para apoyar las necesidades de una amplia variedad de usuarios y proyectos, desde proyectos particulares hasta operaciones empresariales a gran escala. (Apache Subversion, 2014)

RapidSVN

RapidSVN es una plataforma cruzada GUI (Interfaz Gráfica de Usuario por sus siglas en inglés *Grafic User Interface*) con interfaz de usuario para el sistema de revisión de *Subversion*. Está escrito en C++ utilizando el marco wxWidgets (Bibliotecas multiplataforma y libres, para el desarrollo de interfaces gráficas programadas en lenguaje C++). RapidSVN está licenciado bajo la versión tres de la Licencia Pública General GNU. (RapidSVN, 2013)

1.4.4 Herramientas para el desarrollo

NetBeans

NetBeans es una herramienta de desarrollo libre y de código abierto creada por la compañía Sun MicroSystems. La misma es capaz de generar el código necesario para la creación tanto de servicios web como de clientes de servicios web para Java. Todo esto es posible hacerlo mediante asistentes que permiten la configuración visual de los *WS-Security* (Seguridad en Servicios Web), de forma tal que le ahorre tiempo al desarrollador. (*NetBeans Docs & Support*, 2013)

1.4.5. Librerías y marcos de trabajo

Sauxe

Sauxe es un marco de trabajo que contiene un conjunto de componentes reutilizables que provee la estructura genérica para el desarrollo de aplicaciones web de gestión, logrando una mayor estandarización, flexibilidad, integración y agilidad en el proceso de desarrollo, siguiendo el paradigma de independencia tecnológica por el cual apuesta el país. (Gómez Baryolo, 2010)

El marco de trabajo Sauxe da solución a un sin número de escenarios o aspectos arquitectónicos como: gestión y configuración dinámica de caché, integración de componentes de forma distribuida o no distribuida, acceso a bases de datos a través de una capa de abstracción, gestión de concurrencia de recursos, administración centralizada de transacciones, gestión dinámica de las trazas generadas por los sistemas, implementación de mecanismos de autenticación y autorización, implementación de mecanismos de mensajería y control de excepciones, gestión y configuración dinámica de precondiciones, postcondiciones y validaciones de variables, gestión y configuración de flujos de trabajo, visualización de las funcionalidades de un sistema, entre otros escenarios de alta complejidad, garantizando los atributos de calidad de los sistemas que se desarrollen con el mismo. Cuenta con una arquitectura en capas que a su vez presenta en su capa superior un MVC (Modelo-Vista-Controlador). (Gómez Baryolo, 2010)

ExtJS

Es una librería Java Script *Open-Source* (código abierto) de alto rendimiento para la creación y desarrollo de aplicaciones web dinámicas. Provee interfaces gráficas de usuario que brindan experiencias parecidas o iguales a las que se tienen con aplicaciones de escritorio. Permite la creación de aplicaciones complejas utilizando componentes predefinidos. Es extensible para la gran mayoría de los navegadores, evitando el tedioso problema de validar el código para cada uno de estos. (Frederick, Colin y Blades, 2008)

En el mercado actualmente existen múltiples librerías de JavaScript que permiten realizar todo tipo de aplicaciones complejas en el navegador web. ExtJS permite que con pocas líneas de código sea posible realizar interfaces amigables para los usuarios. Es la librería más avanzada para el desarrollo rápido de

aplicaciones con una apariencia totalmente novedosa y una arquitectura flexible. ExtJS permite construir aplicaciones complejas en Internet. (Frederick, Colin y Blades, 2008)

Esta librería incluye:

- Componentes de interfaces de usuarios de alto rendimiento y personalizables.
- Modelo de componentes extensibles.
- Un API (*Application Programming Interface* o Interfaz de programación de aplicaciones) fácil de usar.
- Licencias *Open-Source* y comerciales.

Una de las grandes ventajas de utilizar ExtJS es que permite crear aplicaciones complejas utilizando componentes predefinidos así como un manejador de *layouts* (diseños por su traducción al inglés) similar al que provee *Java Swing* (Biblioteca gráfica para Java). Gracias a esto provee una experiencia consistente sobre cualquier navegador, evitando el tedioso problema de validar que el código escrito funcione bien en cada uno (Firefox, Internet Explorer, Safari, etc.). (Frederick, Colin y Blades, 2008)

Usar una librería con interfaces gráficas como ExtJS, permite tener además estos beneficios:

- Existe un balance entre Cliente – Servidor. La carga de procesamiento se distribuye, permitiendo que el servidor, al tener menor carga, pueda manejar más clientes al mismo tiempo. (Gómez Baryolo, 2010)
- Comunicación asíncrona. En este tipo de aplicación el motor de renderizado puede comunicarse con el servidor sin necesidad de estar sujeta a un clic o una acción del usuario, dándole la libertad de cargar información sin que el cliente se dé cuenta. (Frederick, Colin y Blades, 2008)
- Eficiencia de la red. El tráfico de red puede disminuir al permitir que la aplicación elija qué información desea transmitir al servidor y viceversa; sin embargo la aplicación que haga uso de la pre-carga de datos puede que revierta este beneficio debido al incremento del tráfico. (Frederick, Colin y Blades, 2008)

1.4.5 Lenguaje de Modelado

UML

El Lenguaje Unificado de Modelado (UML por sus siglas en inglés, *Unified Modeling Language*) es un lenguaje que permite modelar, construir y documentar los elementos que forman un software orientado a objetos; es la sucesión de una serie de métodos de análisis y diseño orientados a objetos que aparecen a fines de los 80s y principios de los 90s. Se ha convertido en el estándar de la industria, ya que, a pesar de no tener un respaldo formal de una autoridad institucional o un organismo de normalización, es

ampliamente usado. Incrementa la capacidad de lo que se puede hacer con otros métodos de análisis y diseño orientados a objetos. Una de las metas principales de UML es avanzar en el estado de la integración institucional proporcionando herramientas de interoperabilidad para el modelado visual de objetos. (González Cornejo, 2013)

El lenguaje está dotado de múltiples herramientas para lograr la especificación determinante del modelo, algunas de ellas son:

- Modelamiento de Clases.
- Casos de Uso.
- Diagrama de Interacción. (González Cornejo, 2013)

1.4.6 Lenguajes de programación

PHP

PHP, es un lenguaje *Open-Source* interpretado de alto nivel, especialmente pensado para desarrollos web que puede ser incrustado en páginas HTML. La mayoría de su sintaxis es similar a C, Java y Perl. La meta de este lenguaje es permitir escribir a los creadores de páginas web, páginas dinámicas de una manera rápida y fácil, aunque se pueda hacer mucho más con PHP. (Friedhelm, Dovgal, y otros, 2012)

Lo mejor de usar PHP es que es extremadamente simple para el principiante, pero a su vez, ofrece muchas características avanzadas para los programadores profesionales. (Friedhelm, Dovgal, y otros, 2012)

JavaScript

JavaScript es un lenguaje de *scripting* basado en objetos, que se utiliza principalmente para crear páginas web dinámicas y permite el desarrollo de interfaces de usuario mejoradas. Una página web dinámica es aquella que permite la interacción entre el contenido de la misma y el usuario. JavaScript permite incorporar a dichas páginas efectos como texto que aparece y desaparece, animaciones, acciones que se activan al pulsar botones y ventanas con mensajes de aviso al usuario. Técnicamente, JavaScript es un lenguaje de programación interpretado, por lo que no es necesario compilar los programas para ejecutarlos. En otras palabras, los programas escritos con JavaScript se pueden probar directamente en cualquier navegador sin necesidad de procesos intermedios. A pesar de su nombre, no guarda relación directa con el lenguaje Java, sino que simplemente la compañía dueña del mismo lo adoptó por una cuestión de mercado. (Frederick, Colin y Blades, 2008)

1.5 Patrones

¿Qué es un patrón?

Según Christopher Alexander cada patrón describe un problema que ocurre una y otra vez en un entorno, además describe el núcleo de la solución de ese problema de forma que se puede utilizar esta solución un millón de veces, sin hacerlo nunca de la misma manera. (Martínez, J.S., 1999)

1.5.1 Patrones de diseño

Los patrones de diseño son soluciones a problemas repetidos en la construcción de software, y en ocasiones pueden incluir sugerencias para aplicar estas soluciones en diversos entornos. (Kaisler, 2005)
En el diseño que propone este trabajo, se utilizaron algunos patrones de diseño GoF (Gang of Four o Grupo de Cuatro) y GRASP (*General Responsibility Assignation Software Pattern* o Patrones Generales de Software de Asignación de Responsabilidades), para solucionar y/o evitar diferentes problemas que pudieran aparecer durante la implementación de la solución.

GRASP (Patrones Generales de Asignación de Responsabilidad)

Experto: propone que la clase que contenga toda la información necesaria, será la responsable de la creación de un objeto o la implementación de un método. El comportamiento se distribuye entre las que contienen la información requerida, siendo más fáciles de entender, mantener y ampliar, aumentando sus posibilidades de reutilización. (Pérez Mariñán, 2007)
Creador: la creación de instancias es una de las actividades más comunes en un sistema orientado a objetos. Este patrón proporciona asistencia al identificar quién debe ser el responsable de la creación (o instanciación) de nuevos objetos o clases. Mediante el uso de este patrón se logra un bajo acoplamiento, facilitando la posibilidad de mantenimiento y reutilización. (Pérez Mariñán, 2007)
Controlador: el patrón controlador funciona como intermediario entre una determinada interfaz y el algoritmo que la implementa, de tal forma que es la interfaz quien recibe los datos del usuario y los envía a las distintas clases según el método invocado. (Pérez Mariñán, 2007)
Alta cohesión: propone que la información que almacena una clase debe ser coherente y estar, en la medida de lo posible, relacionada con la clase. Al realizar un cambio en una clase con alta cohesión, todos los métodos que pueden verse afectados, estarán a la vista, en el mismo archivo. Incrementa la claridad, la reutilización y la facilidad de comprensión del diseño. (Pérez Mariñán, 2007)

Bajo acoplamiento: este patrón expresa que entre las clases deberán existir pocas ataduras, es decir, estarán lo menos relacionadas posibles, de forma tal que en caso de producirse una modificación en alguna de ellas, se tenga la mínima repercusión posible en el resto de las clases, incrementando la reutilización, y disminuyendo la dependencia entre ellas. (Pérez Mariñán, 2007)

GoF (Gang of Four)

Proxy: cuando no se desea el acceso directo a un objeto sobre el que se va a aplicar determinada acción, este patrón propone la adición de un nivel que permita solamente el acceso al objeto a través de un objeto sustituto, que será el responsable de controlar o mejorar el acceso al objeto real. (Pérez Mariñán, 2007)

Facade (Fachada): propone la definición de un único punto de conexión con un componente. Este objeto fachada presenta una única interfaz unificada y es responsable de colaborar con los componentes del subsistema. (Pérez Mariñán, 2007)

Singleton: garantiza que una clase sólo tiene una única instancia, proporcionando un punto de acceso global a la misma. Se usa cuando debe haber únicamente una instancia de una clase y debe ser claro su acceso para los clientes y cuando la "Instancia Única" debe ser especializable mediante herencia y los clientes deben poder usar la instancia extendida sin modificar su código (el de los clientes). (Pérez Mariñán, 2007)

1.5.2 Patrones Arquitectónicos

En la solución propuesta se utilizó el patrón arquitectónico Modelo-Vista-Controlador (MVC), el cual se emplea en la capa de funcionamiento y es el encargado de separar los datos de la aplicación, la interfaz de usuario y la lógica de control en tres componentes distintos (Steve Burbeck, 2013):

- Modelo: está compuesto por datos, reglas de negocio y las funcionalidades correspondientes para la comunicación con el marco de persistencia de datos Doctrine.
- Controlador: gestiona las entradas del usuario.
- Vista: muestra la información del modelo usuario.

1.6 Conclusiones parciales

El análisis y estudio de los conceptos asociados al dominio del problema permitieron identificar las características de los elementos de los componentes. La selección y presentación de los diferentes marcos existentes a nivel internacional permitió identificar las características relevantes y las ventajas que podrían ser utilizadas durante el desarrollo de la solución. A partir de la problemática planteada se identificó un problema a resolver. Se definió el objetivo general del cual se desglosaron los objetivos específicos. Durante el desarrollo de la investigación se utilizarán un conjunto de métodos científicos para la obtención y procesamiento de la información. Se plantea la idea a defender de la investigación. Se plantearon las herramientas y tecnologías actuales. Se explicaron los patrones utilizados.

CAPÍTULO II. PROPUESTA DE SOLUCIÓN

Introducción

En el presente capítulo, se realiza una propuesta de solución que va a estar conformada por la identificación y descripción de los requisitos funcionales y no funcionales que debe cumplir el sistema a implementar. Además de los artefactos generados durante la fase de Análisis y Diseño de la herramienta.

2.1 Modelo conceptual

El modelo conceptual es la representación visual de los conceptos u objetos del mundo real que son significativos para el problema o área de interés que se analiza, representando las clases conceptuales, no los componentes de software. El mismo, captura los tipos más importantes de objetos que existen o los eventos que suceden en el entorno donde estará el sistema, además de incluir las responsabilidades de las personas que ejecutan las actividades. (Universidad de Buenos Aires, 2009)

2.1.1 Descripción del Modelo conceptual

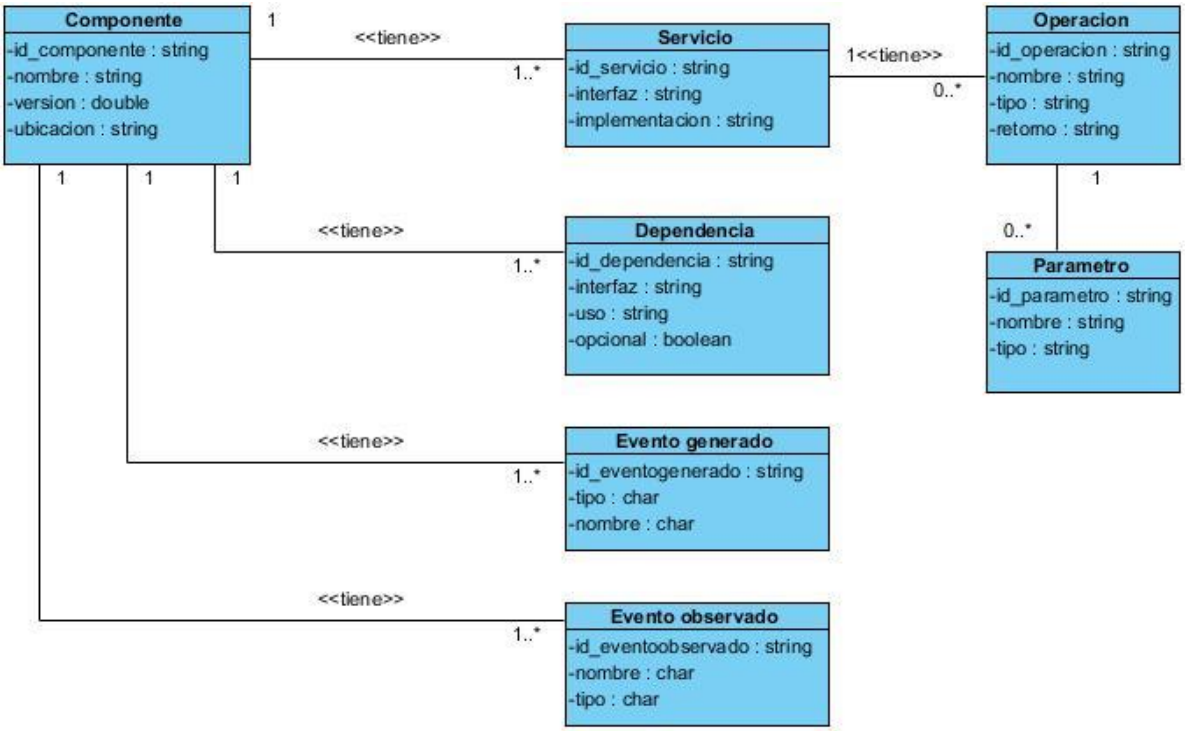


Figura 12: Modelo conceptual.

El Modelo conceptual de la solución propuesta (ver Figura 12) se encuentra representado a través del concepto componente que contiene un nombre, una versión y una ubicación, que tiene asociado una lista de servicios, dependencias, eventos generados y observados. A su vez cada servicio tiene un identificador, un nombre, el nombre de la interfaz y la interfaz que implementa, así como posee varias operaciones, dichas operaciones tendrán un nombre, un tipo de dato, un retorno que es el tipo de dato que retorna y una lista de parámetros, cada parámetro contiene también un tipo de dato. Cada dependencia tendrá un identificador, el nombre de la interfaz, el uso que es el servicio que utiliza la dependencia y opcional que es el atributo que brinda el nivel de dependencia para que sea funcional, es decir verdadero si es funcional el componente que la contiene y falso si no es funcional. Los eventos van estar en dependencia de las acciones que haga el componente.

2.2 Requisitos

El análisis de las soluciones existentes en el capítulo anterior, la representación del modelo conceptual para la herramienta “Gestionar componentes” y las entrevistas realizadas a los especialistas del proyecto permitió efectuar una correcta captura de requisitos, quedando reflejado en los epígrafes que se relacionan a continuación. También se efectuó la validación de requisitos capturados para comprobar que satisface las necesidades del cliente.

2.2.1. Requisitos funcionales

Los requisitos funcionales son las capacidades o condiciones que el sistema debe cumplir, para que el usuario resuelva un problema o cumpla sus objetivos. Deben estar claros y libres de ambigüedades, asegurando que los involucrados comprendan claramente el significado de cada uno (Sommerville, 2006). Durante el proceso de gestión de requisitos se realizó la captura de requisitos a través de la técnica entrevistas abiertas realizadas a especialistas del proyecto, al estudiante Abraham Calás que desarrollo el modelo de componentes y usando la técnica tormentas de ideas en los talleres que se realizaron en la fase inicial del desarrollo de la propuesta de solución, identificando un total de 29 requisitos funcionales. Además se realizó el análisis de los mismos agrupándolos a partir de las técnica Agrupación funcional, en la que se agrupa requisitos que tienen una relación funcional directa, por ejemplo según el tipo de datos que van a manejar o según los procesos de negocio que van a cubrir, logrando agrupar los 29 requisitos funcionales en 7 agrupaciones. A continuación se refieren las siguientes agrupaciones realizadas y en el **Anexo 3** se realiza la descripción de los requisitos agrupados en **Gestionar componente**, los restantes requisitos se encuentran en el expediente de proyecto del marco de trabajo Sauxe.

Agrupación1. Gestionar componente

RF1 Registrar componente

RF2 Modificar componente

RF3 Deshabilitar componente

RF4 Habilitar componente

RF5 Mostrar componente

Agrupación2. Gestionar servicio

RF6 Adicionar servicio

RF7 Modificar servicio

RF8 Eliminar servicio

RF9 Listar servicio

Agrupación3. Gestionar operación

RF10 Adicionar operación asociada a un servicio

RF11 Modificar operación asociada a un servicio

RF12 Eliminar operación asociada a un servicio

RF13 Listar operación asociada a un servicio

Agrupación4. Gestionar parámetro

RF14 Adicionar parámetro asociada a una operación

RF15 Modificar parámetro asociada a una operación

RF16 Eliminar parámetro asociada a una operación

RF17 Listar parámetro asociada a una operación

Agrupación5. Gestionar dependencia

RF18 Adicionar dependencia

RF19 Modificar dependencia

RF20 Eliminar dependencia

RF21 Listar dependencia

Agrupación6. Gestionar eventos generados

RF22 Adicionar eventos generados

RF23 Modificar eventos generados

RF24 Eliminar eventos generados

RF25 Listar eventos generados

Agrupación7. Gestionar eventos observados

RF26 Adicionar eventos observados

RF27 Modificar eventos observados

RF28 Eliminar eventos observados

RF29 Listar eventos observados

2.2.3. Requisitos no funcionales

Los requisitos no funcionales son las propiedades o cualidades que el producto debe tener, especificando criterios que pueden usarse para juzgar la operación de un sistema (Sommerville, 2006). El sistema a desarrollar se va a integrar al marco de trabajo Sauxe; los requisitos no funcionales fueron definidos teniendo en cuenta este elemento, agrupados por diferentes categorías los cuales se mencionan a continuación:

Categoría 1. Software

Se requiere para las PC clientes:

RNF1 Navegador Mozilla Firefox 3.6 o superior.

RNF2 Sistemas operativos GNU/Linux, Windows 98 o superior.

Se requiere para las PC servidoras:

RNF3 Sistema operativo Linux en cualquiera de sus distribuciones.

RNF4 Servidor web Apache en su versión 2.0 o superior, con los módulos de PHP 5.0 disponibles.

RNF5 PostgreSQL en su versión 8.3 o superior, como Sistema Gestor de Base de Datos.

Categoría 2. Hardware

En el caso de las PC clientes donde se ejecutará la aplicación, se requiere de las siguientes condiciones:

RNF6 Procesador Pentium IV 2x2 caché, con velocidad de procesamiento a 1 GHz como mínimo y 256Mb de RAM.

En dependencia de la funcionalidad concebida para cada uno de los servidores, son las condiciones que se requiere que deban cumplir:

RNF7 **Servidor web:** requiere un procesador Pentium IV 2X2 caché, velocidad del procesador de 1GHz como mínimo y memoria de 1GB de RAM.

Categoría 3. Usabilidad

RNF8 Los usuarios que utilicen la aplicación deben tener conocimientos de la informática.

RNF9 Los nombres de las interfaces sean sugerentes.

RNF10 Los pasos necesarios para realizar una operación de la herramienta son sencillos.

RNF11 Facilita las consultas de los componentes del marco de trabajo.

2.3. Modelo de diseño

El modelo de diseño es una abstracción del Modelo de Implementación y su código fuente, el cual fundamentalmente se emplea para representar y documentar su diseño. Es utilizado como entrada esencial en las actividades relacionadas con la implementación. El modelo de diseño puede contener: diagramas, clases, paquetes, subsistemas, interfaces, relaciones y los atributos. (Pressman, 2002)

2.3.1. Diagrama de clases

El diagrama de clases muestra un conjunto de clases, interfaces y colaboraciones y las relaciones entre éstos. Los diagramas de clases muestran el diseño de un sistema desde un punto de vista estático; un diagrama estático es el que muestra una colección de elementos (estáticos) declarativos (Grady Booch). En la figura 13 se muestra el diagrama de clases del diseño de la herramienta “Gestionar componentes”, representado por la clase servidora **ComponenteController,** que se encarga de construir o generar el resultado HTML en componente.phtml que conforma el código cliente **<<build>>**, el formulario componente envía los datos al código servidor para ser procesados los pedidos **<<submit>>**, y además forman parte del código cliente o resultado HTML, el código servidor solicita y envía la información a través de la clase Manager que es la encargada de crear los *bundles*, donde se registra el contenido de los componentes del sistema. En la construcción de los formularios se utiliza la librería ExtJs en su versión 4.2, a través de la arquitectura Modelo Vista Controlador (MVC) que siguen una estructura de directorio unificado que es el mismo para cada aplicación, todas las clases se colocan en la carpeta componente, que contiene a su vez sub-carpetas para generar el espacio de nombres de los modelos, vistas, controladores y almacenes (*stores*), cada aplicación se inicia con una instancia de la clase *Application*. La instancia de *Application* contiene la configuración global de su aplicación (por ejemplo, el nombre de la aplicación), así como mantiene referencias a todos los modelos, vistas y controladores utilizados por la aplicación. Una aplicación también contiene una función de lanzamiento, que se ejecuta automáticamente cuando todo está cargado.

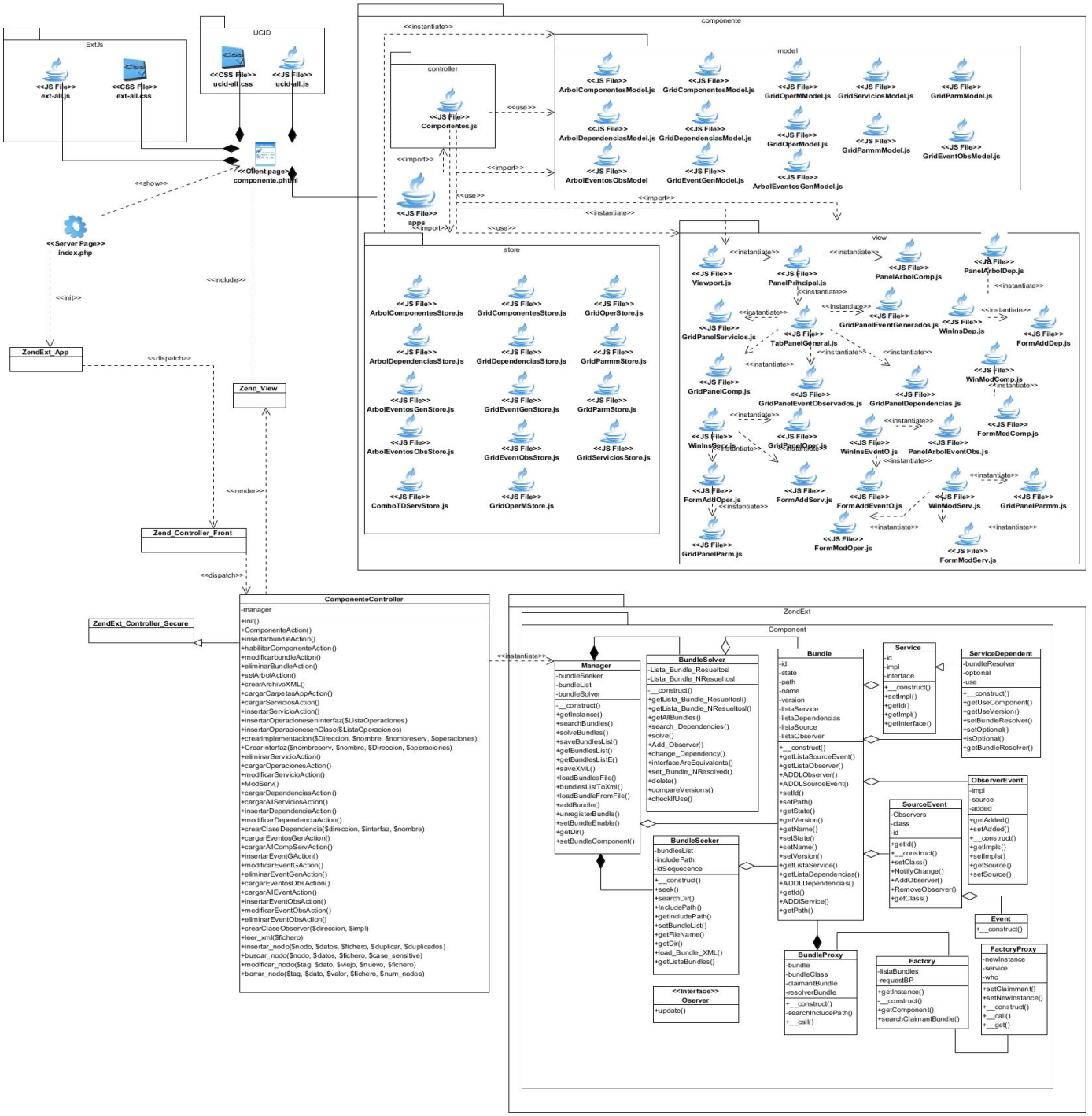


Figura 13: Diagrama de clases de la herramienta “Gestionar componentes”.

2.4. Modelo de Datos

El Modelo de Datos es usado para describir la representación lógica y física de la información persistente administrada por el sistema. Puede ser inicialmente creado a través de ingeniería inversa de un almacenamiento de datos persistentes que existan en la base de datos o puede ser inicialmente creado a partir de un conjunto de clases del diseño persistente en el modelo de diseño. (Massachusetts Institute, 2002)

El Modelo de Datos definido en la solución propuesta es orientado a objetos contiene 10 objetos persistentes. A continuación se muestra el diagrama que lo describe.

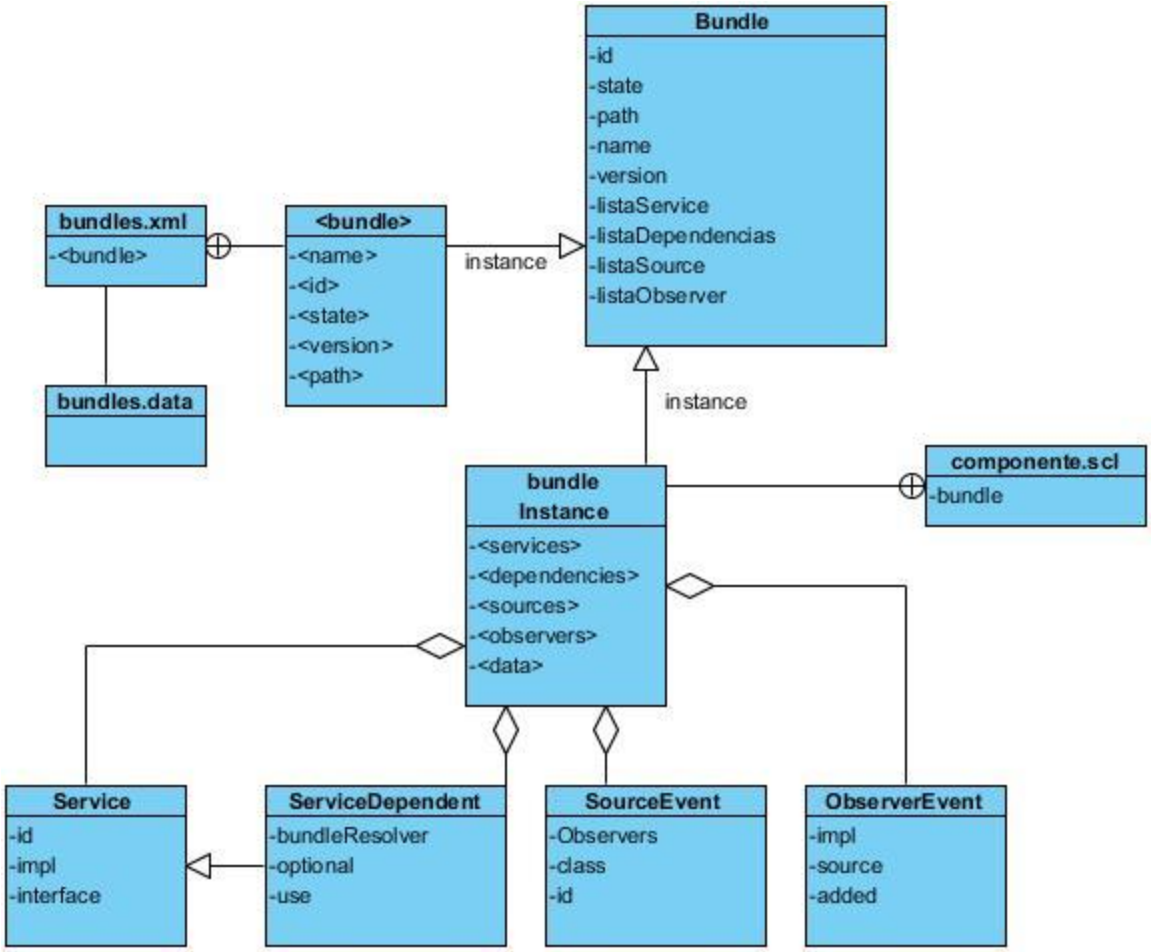


Figura 14: Modelo de Datos de la herramienta "Gestionar componentes".

En el Modelo de Datos representado anteriormente se puede observar el fichero bundle.xml que contiene una lista de *bundles* que son todos los componentes registrados y habilitados en Sauxe conjuntamente

con bundles.data con los atributos nombre, identificador, estado, versión y la dirección relativa. Toda la información de cada *bundle* es almacenada en el fichero ".scl" del mismo nombre del componente los cuales tienen como etiquetas <data> que contiene el estado y la versión, la etiqueta <<services>> tiene una lista de *Service* que son los servicios que brinda teniendo como atributos el identificador, la implementación y la interfaz. También contiene un listado de *ServiceDependent* que son las dependencias de cada *bundle* que tienen como propiedades el componente que resuelve dicha dependencia (bundleResolver), el *booleano* de opcionalidad (optional) y el *bundle* que lo usa contenidos en la etiqueta <<dependencies>>. Cada fichero ".scl" tienen además en las etiquetas <<sources>> y <<observers>> listados de *SourceEvent* y *ObserverEven*t respectivamente, lo cual el primero tiene una lista de *Observers* que son los observadores que atienden dicho evento generado, el objeto de la clase en caso de tenerlo y el identificador. Por último *ObserverEvent* que tiene como atributo el evento que observa, si esta añadido el observador y la implementación de dicho observador.

2.5. Conclusiones del capítulo

Se elaboró el modelo conceptual de la solución, donde se analizó la relación que existe entre cada uno de los conceptos identificados. Se realizó la descripción de los requisitos funcionales y no funcionales identificados durante el proceso de Ingeniería de Requisitos. Como parte del modelo de diseño se identificaron los patrones de diseño a utilizar en el desarrollo de la solución. Por último, se realizó el análisis y diseño de la herramienta para "Gestionar componentes" que servirá de base para llevar a cabo el proceso de implementación de la solución propuesta.

CAPÍTULO III. IMPLEMENTACIÓN Y PRUEBA

Introducción

En el presente capítulo se describirán los estándares de codificación a utilizar para garantizar un mejor entendimiento y legibilidad del código, así como los componentes necesarios para realizar la implementación y despliegue de la solución. Además se mostrarán los resultados de la evaluación mediante métricas de diseño para validar el diseño, pruebas de caja negra y caja blanca, técnicas escogidas para encontrar los errores de la aplicación. La realización del cuasiexpermiento permitió validar la investigación, utilizando como instrumento las encuestas.

3.1 Modelo de Implementación

El Modelo de Implementación describe cómo los elementos del Modelo de Diseño, se implementan en términos de componentes. Describe también cómo se organizan los componentes de acuerdo con los mecanismos de estructuración y modulación disponibles en el entorno de implementación y en el/los lenguajes de programación utilizados y cómo dependen los componentes unos de otros, además de los recursos necesarios para poder ejecutar el sistema desarrollado (Brito Acuña, 2013).

3.1.1 Diagrama de Componentes

Un diagrama de componentes muestra la organización y las dependencias entre un conjunto de componentes. Gráficamente, un diagrama de componentes es una colección de nodos y arcos. Los diagramas de componentes se utilizan para modelar la vista de implementación estática de un sistema. Esto implica modelar las cosas físicas que residen en un nodo, tales como ejecutables, bibliotecas, tablas, archivos y documentos. Los diagramas de componentes contienen: Componentes, Interfaces y Relaciones de dependencia, generalización, asociación y realización.

Los diagramas de componentes son fundamentalmente diagramas de clases que se centran en los componentes de un sistema (Grady Booch).

A continuación en la figura 15, se muestra el diagrama de componentes de la herramienta “Gestionar componentes”, que se encuentra representado por un componente que identifica a todos los componentes que crea la herramienta, el mismo contiene una estructura de carpetas definida por la carpeta *controller* donde se guarda la clase controladora del sistema, por la carpeta *model* donde se guarda la información de acceso a datos, la carpeta *views* donde se registra los **.js** que contruyen los formularios, la carpeta *validators* donde se registra las validaciones del propio componente. La carpeta services que tiene la

información de los servicios y las dependencias que implementan las interfaces del componente, de los eventos y los observadores que se generan. El fichero bundles.xml permite registrar todos los **.scl** que existen en el marco de trabajo, éste **.scl** contiene la información del componente, a través de las etiquetas <<service>>, <<dependency>>, <<source>> y <<observer>>.

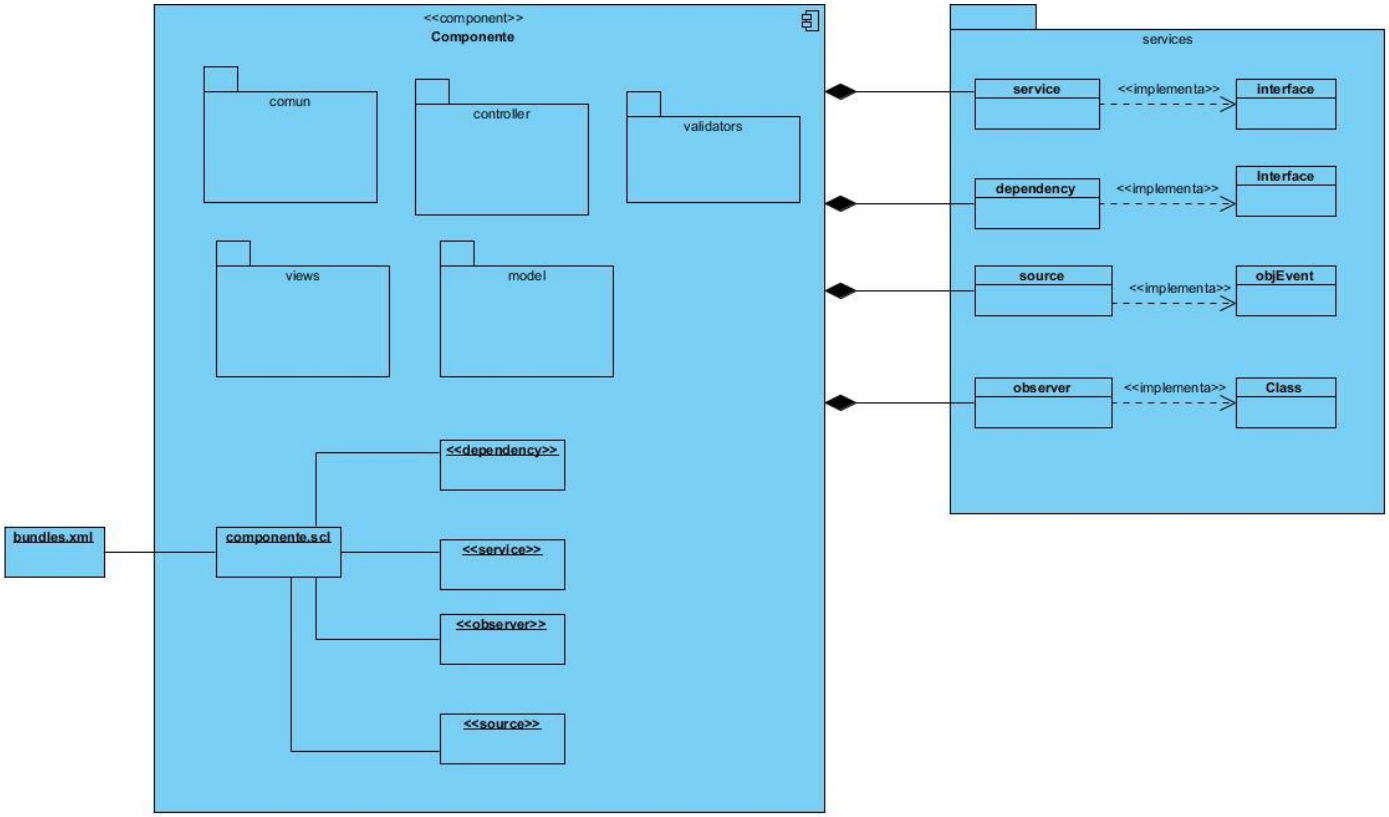


Figura 15 Diagrama de componentes.

3.1.2 Diagrama de Despliegue

Un diagrama de despliegue es un diagrama que muestra la configuración de los nodos que participan en la ejecución y de los componentes que residen en ellos. Gráficamente, un diagrama de despliegue es una colección de nodos y arcos. Lo que distingue a un diagrama de despliegue de los otros tipos de diagramas es su contenido particular, es decir, estos contienen nodos y relaciones de dependencia y asociación. (Grady Booch). A continuación se muestra el diagrama de despliegue, en el cual, se tiene una **PC_Cliente** que refleja la estación de trabajo donde el usuario podrá acceder al sistema, estableciendo una comunicación con el **Servidor_Web**, a través del protocolo HTTP, protocolo de transferencia de hipertexto

(*Hyper Text Transfer Protocol*) es un protocolo del nivel de aplicación usado para la transferencia de información entre sistemas, de forma clara y rápida. En este caso el cliente envía una petición en forma de método, una URL, y una versión de protocolo seguida de los modificadores de la petición de forma parecida a un mensaje *Multipurpose Internet Mail Extensions* (MIME), información sobre el cliente y al final un posible contenido. El servidor contesta con una línea de estado que incluye la versión del protocolo y un código que indica éxito o error, seguido de la información del servidor en forma de mensaje MIME y un posible contenido. El servidor web almacena y obtiene la información en los ficheros bundles.xml, bundles.data y componente.scl, que se representa mediante un artefacto como archivo fuente.

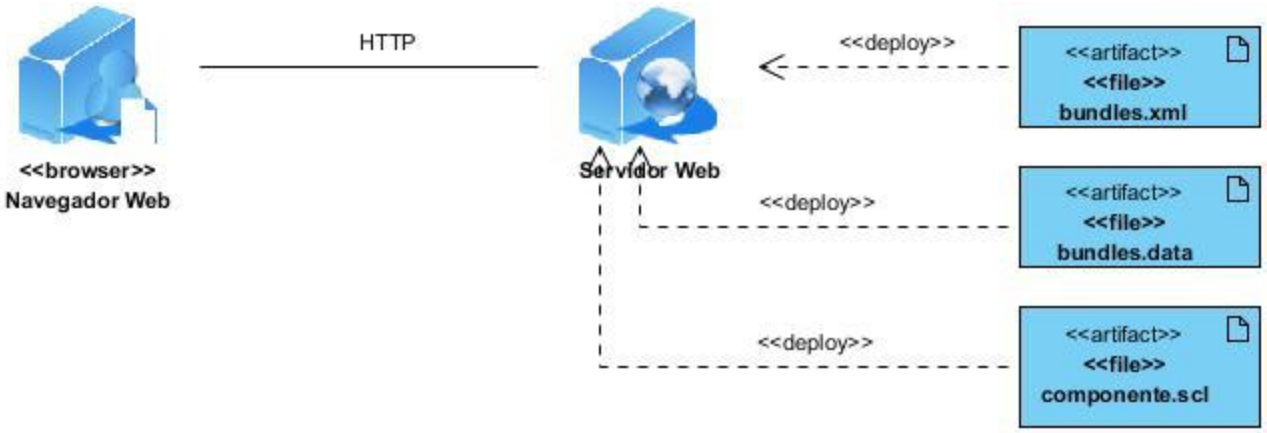


Figura: 16 Modelo de despliegue.

3.2 Estándares de codificación

Los estándares de codificación son pautas de programación que no están enfocadas a la lógica del programa, sino a su estructura y apariencia física para facilitar la lectura, comprensión y mantenimiento del código. Este documento constituye una guía para el desarrollo en las líneas de producción, desde el punto de vista arquitectónico, con el propósito de lograr una estandarización del código. (CEIGE, 2008)

Para el desarrollo de la herramienta Gestionar componente se utilizarán algunos de los estándares de codificación y normas propuestos como parte de la Línea de arquitectura determinada para el desarrollo del ERP Cuba. (CEIGE, 2008)

3.2.1 Estándares de nomenclatura

Nomenclatura de las clases

Los nombres de las clases comienzan con la primera letra en mayúscula y el resto en minúscula, en caso de que sea un nombre compuesto se empleará notación PascalCasing*. Con sólo leerlo se reconoce el propósito de la misma.

Ejemplo: GestionarComponente.

Clases controladoras

Las clases controladoras después del nombre llevan la palabra:"Controller".

Ejemplo: GestionarComponenteController

Nomenclatura de las funciones

El nombre a emplear para las funciones se escribe con la primera palabra en minúscula, en caso de que sea un nombre compuesto se empleará notación CamelCasing*, y con sólo leerlo se reconoce el propósito de la misma.

Ejemplo: habilitarComponente

• En caso de ser una acción de la clase controladora se le pone el nombre y seguida la palabra:"Action"

Ejemplo: habilitarComponenteAction

Nomenclatura de las variables

El nombre a emplear para las variables se escribe con la primera palabra en minúscula, en caso de que sea un nombre compuesto se empleará notación CamelCasing*, y comenzando con un prefijo según el tipo de datos.

Ejemplo: manager

Prefijos para los tipos de datos

Los prefijos a utilizar en la creación de variables serán los siguientes:

Tabla 1: Prefijos para tipos de datos.

Tipos de Datos	Prefijos
Arreglos	Arr
Objetos	Obj
Enteros	Int
Cadena	Str
float	Flt
Boolean	Boo

Nomenclatura de las constantes

El nombre a emplear para las constantes se escribe con todas las letras en mayúscula.

Ejemplo: MONEDA.

Nomenclatura de los atributos

El nombre a emplear para los atributos se escribe con la primera palabra en minúscula, en caso de que sea un nombre compuesto se empleará notación *CamelCasing**. Además en caso de ser un objeto se comienza con:"_" y después se escribe el nombre.

Ejemplo: intMoneda=dinero

objMoneda=_dinero

Normas de comentariado

Los comentarios deben ser lo bastante claro y preciso de forma tal que se entienda el propósito de lo que se está desarrollando.

- **En las clases**

Antes de la declaración de una clase se escribe una breve descripción donde se explique el propósito de la misma. Que se escribe de la siguiente forma:

```
/**
* Nombre de la clase *
* Descripción *
* @author *
* @package *(módulo)
* @subpackage *(sub módulo)
* @copyright *
* @version (versión - parche)
*/
```

- **En las funciones**

Antes de la declaración de la función se escribe una breve descripción donde se explique el propósito de la misma. Se escribe de la siguiente forma:

```
/**
* Nombre de la función *
* Descripción *
* @author * (en caso de que no sea el autor de la clase)
* @param *(los parámetros que se le pasan a la función con su descripción)
* @throws *(en caso de que dispare una excepción)
*/
```

3.2.2 Estilo del código

En la implementación cuando se valla a escribir una sentencia en PHP la forma de utilizar los *tab* del mismo.

Sangría o indexado

La política de sangría a utilizar en la implementación es por *Tab*.

Declaraciones dentro del cuerpo de la clase

Las clases se comienzan a declarar pegado al margen izquierdo, después de poner el nombre de la clase se pone un espacio y se abre llave en la misma línea.

```php
<?php
/**
 * Indentation
 */
class Example {
    var $theInt = 1;
    function foo($a, $b) {
    switch ( $a) {
    case 0 :
    $Other->doFoo ();
    break;
    default :
    $Other->doBaz ();
    }
    }
    function bar($v) {
    for($i = 0; $i < 10; $i ++) {
    $v->add ( $i );
    }
    }
}
?>
```

Figura 17: Estilo del código: Sangría o indexado.

En este ejemplo se muestra como debe quedar el código después de aplicarle las declaraciones de sangría.

```php
<?php
/**
 * Indentation
 */
class Example {
    var $theInt = 1;
    function foo($a, $b) {
        switch ( $a) {
            case 0 :
                $Other->doFoo ();
            break;
            default :
                $Other->doBaz ();
        }
    }
    function bar($v) {
        for($i = 0; $i < 10; $i ++) {
            $v->add ( $i );
        }
    }
}
?>
```

Figura 18: Estilo del código: Sangría o indexado.

3.3 Métricas de diseño

Las métricas de software son una medida cuantitativa que permite a los desarrolladores tener una visión profunda de la eficacia del proceso del software y de los proyectos que dirigen utilizando el proceso como un marco de trabajo. Se reúnen los datos básicos de calidad y productividad. Estos datos son entonces analizados, comparados con promedios anteriores, y evaluados para determinar las mejoras en la calidad y productividad. Las métricas son también utilizadas para señalar áreas con problemas de manera que se puedan desarrollar los remedios y mejorar el proceso del software (Pressman, 2009).

Las métricas empleadas están diseñadas para evaluar los siguientes atributos de calidad:

Responsabilidad. Consiste en la responsabilidad asignada a una clase en un marco de modelado de un dominio o concepto, de la problemática propuesta (Pressman, 2009).

Complejidad de implementación. Consiste en el grado de dificultad que tiene implementar un diseño de clases determinado (Pressman, 2009).

Reutilización. Consiste en el grado de reutilización presente en una clase o estructura de clase, dentro de un diseño de software (Pressman, 2009).

Acoplamiento. Consiste en el grado de dependencia o interconexión de una clase o estructura de clase, con otras, está muy ligada a la característica de Reutilización (Pressman, 2009).

Complejidad del mantenimiento. Consiste en el grado de esfuerzo necesario a realizar para desarrollar un arreglo, una mejora o una rectificación de algún error de un diseño de software. Puede influir indirecta, pero fuertemente en los costes y la planificación del proyecto (Pressman, 2009).

Cantidad de pruebas. Consiste en el número o el grado de esfuerzo para realizar las pruebas de calidad (Unidad) del producto (componente, modulo, clase, conjunto de clases, etc.) diseñado (Pressman, 2009).

Las métricas identificadas como instrumento para evaluar la calidad del diseño de la herramienta para "**Gestionar componentes"** en el marco de trabajo Sauxe descrito en el capítulo anterior y su relación con los atributos de calidad son las siguientes:

3.3.1 Tamaño operacional de clase (TOC)

Está dado por el número de métodos asignados a una clase y evalúa los siguientes atributos de calidad (lngSoftwarell-CUFM, 2013):

Tabla 2: Atributos de calidad evaluados por la métrica TOC.

Atributo de calidad	Modo en que lo afecta
Responsabilidad	Un aumento del TOC implica un aumento de la responsabilidad asignada a la clase.
Complejidad de implementación	Un aumento del TOC implica un aumento de la complejidad de implementación de la clase.
Reutilización	Un aumento del TOC implica una disminución del grado de reutilización de la clase.

Para los cuales están definidos los siguientes criterios y categorías de evaluación:

Tabla 3: Criterios de evaluación para la métrica TOC.

Atributo	Categoría	Criterio
Responsabilidad	Baja	<=Promedio
	Media	Entre Promedio y 2*Promedio
	Alta	>2*Promedio
Complejidad de implementación	Baja	<=Promedio
	Media	Entre Promedio y 2*Promedio
	Alta	>2*Promedio
Reutilización	Baja	>2*Promedio

	Media	Entre Promedio y 2*Promedio
	Alta	<=Promedio

Relaciones entre clases (RC): Está dado por el número de relaciones de uso de una clase con otra y evalúa los siguientes atributos de calidad (IngSoftwarell-CUFM, 2013):

Tabla 4: Atributos de calidad evaluados por la métrica RC.

Atributo de calidad	**Modo en que lo afecta**
Acoplamiento	Un aumento del RC implica un aumento del Acoplamiento de la clase.
Complejidad de mantenimiento	Un aumento del RC implica un aumento de la complejidad del mantenimiento de la clase.
Reutilización	Un aumento del RC implica una disminución en el grado de reutilización de la clase.
Cantidad de pruebas	Un aumento del RC implica un aumento de la Cantidad de pruebas de unidad necesarias para probar una clase.

Para los cuales están definidos los siguientes criterios y categorías de evaluación:

Tabla 5: Criterios y categorías de evaluación

Atributo	**Categoría**	**Criterio**
Acoplamiento	Ninguno	0
	Bajo	1
	Medio	2
	Alto	>2
Complejidad de mantenimiento	Baja	<=Promedio
	Media	Entre Promedio y 2*Promedio
	Alta	>2*Promedio
Reutilización	Baja	>2*Promedio
	Media	Entre Promedio y 2*Promedio
	Alta	<=Promedio
Cantidad de pruebas	Baja	<=Promedio

	Media	Entre Promedio y 2*Promedio
	Alta	>2*Promedio

Resultados obtenidos:

Evaluando la cantidad de operaciones de cada una de las clases con que cuenta la herramienta "Gestionar componentes", según los criterios establecidos en la Tabla 10, se obtuvo el siguiente resultado reflejado en la tabla 9.

Tabla 6: Resultado de la métrica TOC.

Clasificación	Responsabilidad	Complejidad de implementación	Reutilización
Baja	70%	70%	13%
Media	17%	17%	17%
Alta	13%	13%	70%

Tabla 7: Instrumento de evaluación de la métrica TOC.

No	Clase	Cantidad de Procedimientos	Responsabilidad	Complejidad	Reutilización
1	**ComponenteController**	**40**	Alta	Alta	Baja
2	Bundle	19	Alta	Alta	Baja
3	BundleProxy	3	Baja	Baja	Alta
4	BundleSeeker	10	Media	Media	Media
5	BundleSolver	13	Media	Media	Media
6	Event	1	Baja	Baja	Alta
7	Factory	**4**	Baja	Baja	Alta
8	FactoryProxy	5	Baja	Baja	Alta
9	Manager	16	Media	Media	Media
10	Observer	1	Baja	Baja	Alta
11	ObserverEvent	7	Baja	Baja	Alta
12	Service	**5**	Baja	Baja	Alta
13	ServiceDependent	**7**	Baja	Baja	Alta
14	SourceEvent	7	Baja	Baja	Alta
15	SourceObserver	**7**	Baja	Baja	Alta
16	TransactionalService	2	Baja	Baja	Alta

Cantidad de clases por intervalos de procedimientos

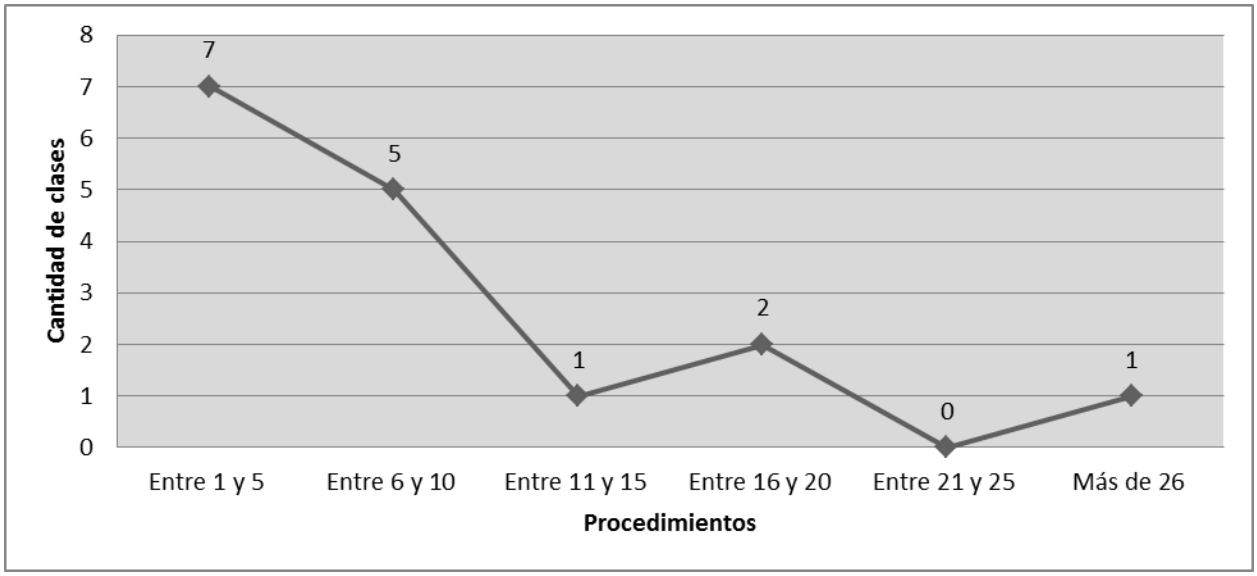


Figura 19: Representación de las clases según la cantidad de operaciones.

Teniendo en cuenta el umbral de cantidad de operaciones con que cuenta cada clase, está establecido que un tamaño de clase pequeño es aquel que tiene un valor menor o igual que 3, un tamaño medio para aquellas clases que tengan entre 4 y 10 operaciones y un tamaño de clase grande es aquel que es mayor o igual que 16. Se concluye que la herramienta cuenta con 15 clases fundamentales cuyo promedio de cantidad de operaciones equivale a 18. Los valores de tamaño quedan distribuidos de la siguiente manera:

Tabla 8: Tamaño de clases.

Umbral	Tamaño	Cantidad de Clases
Pequeño	<=3	4
Medio	>=4 y <=10	8
Grande	>=16	3

Como se puede observar en la Tabla 12 el 26.7% de las clases analizadas están clasificadas de umbral Pequeño, Medio 53% y Grande 20%, lo cual se considera un resultado positivo según los parámetros de calidad Responsabilidad, Complejidad de Implementación y Reutilización propuestos para esta métrica.
A continuación se grafican los resultados obtenidos de la aplicación de la métrica TOC.

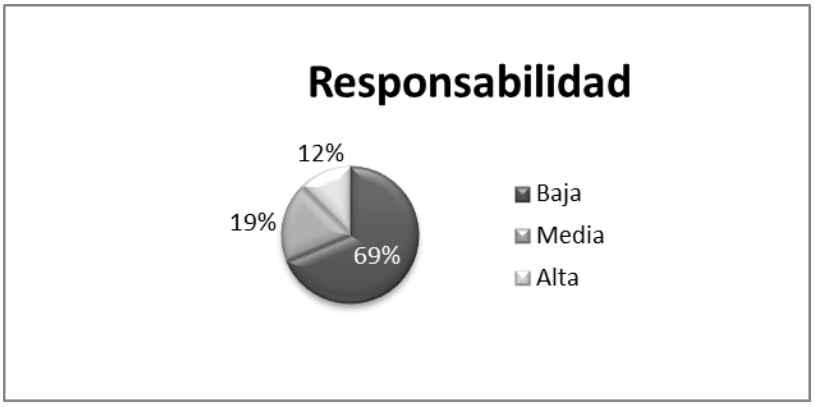


Figura 20: Resultados de la evaluación de la métrica TOC para el atributo de calidad Responsabilidad.

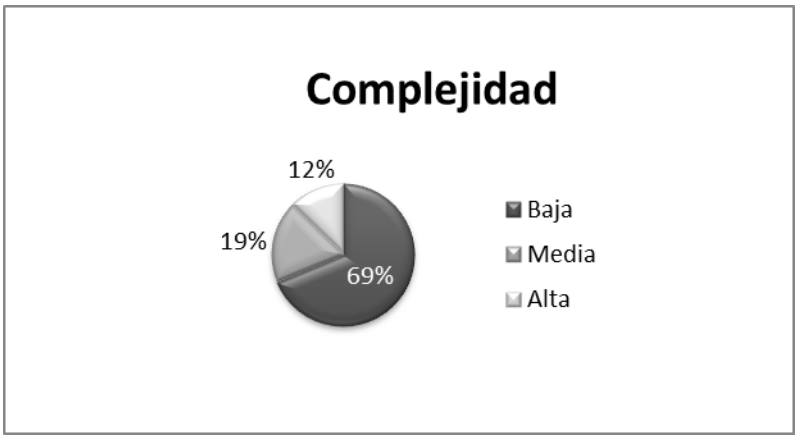


Figura 21: Resultados de la evaluación de la métrica TOC para el atributo de calidad Complejidad.

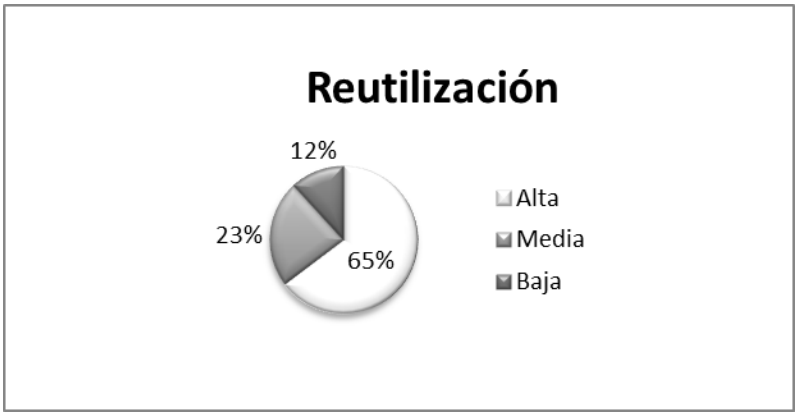


Figura 22: Resultados de la evaluación de la métrica TOC para el atributo de calidad Reutilización.

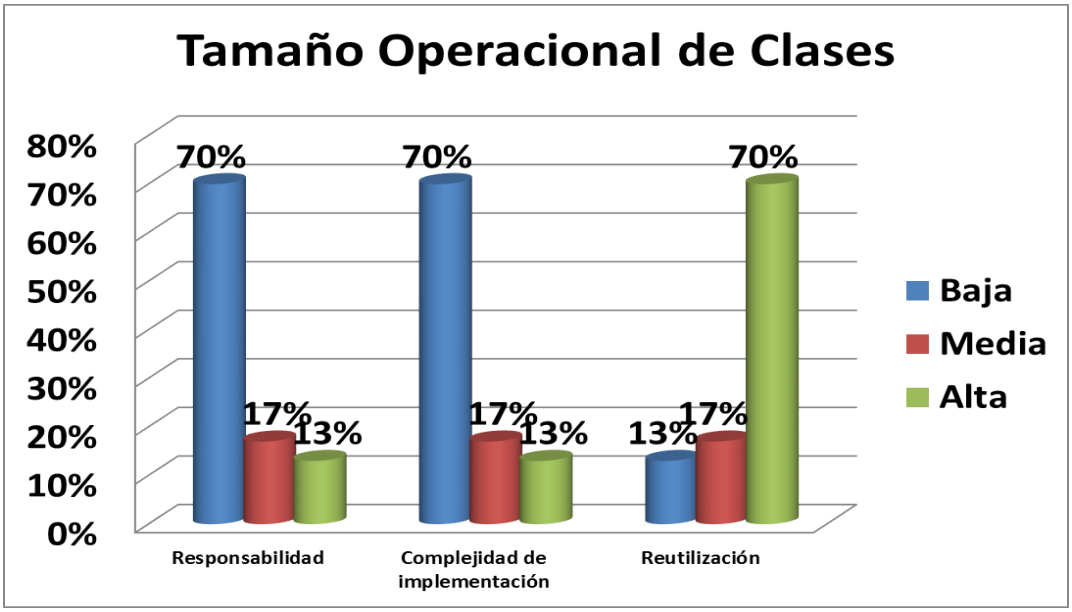


Figura 23: Resultado de la aplicar la métrica TOC.

Luego de aplicar la métrica se han obtenido resultados positivos, esto evidencia que la solución está correctamente diseñada garantizando un diseño robusto con una alta reutilización.

3.3.2 Relación entre Clases (RC)

Está dado por el número de relaciones de uso de una clase con otra y evalúa los siguientes atributos de calidad:

Tabla 9: Atributos de calidad que evalúa RC.

Atributo de calidad	Modo en que lo afecta
Acoplamiento	Aumento del RC provoca aumento del Acoplamiento de la clase.
Complejidad de mantenimiento	Aumento del RC provoca aumento de la complejidad del mantenimiento de la clase.
Reutilización	Aumento del RC provoca disminución en el grado de reutilización de la clase.
Cantidad de pruebas	Aumento del RC provoca aumento de la cantidad de pruebas de unidad necesarias para probar una clase.

Para la evaluación de dichos atributos de calidad, se definieron los siguientes criterios y categorías de evaluación:

Tabla 10: Criterios de evaluación de la métrica RC.

	Categoría	Criterio
Acoplamiento	Ninguno	0
	Baja	1
	Medio	2
	Alto	>2
	Categoría	Criterio
Complejidad Mantenimiento	Baja	<= Prom.
	Media	Entre Prom. y 2*Prom.
	Alta	> 2*Prom.
	Categoría	Criterio
Reutilización	Baja	>2* Prom.
	Media	Entre Prom. y 2*Prom.
	Alta	<= Prom.
	Categoría	Criterio
Cantidad de Pruebas	Baja	<= Prom.
	Media	Entre Prom. y 2*Prom.
	Alta	> 2*Prom.

Resultados obtenidos: Evaluando la cantidad de relaciones entre clases con que cuenta la herramienta "Gestionar componentes", según los criterios establecidos en la Tabla 14, se obtuvo el siguiente resultado reflejado en la tabla 15.

Tabla 11 : Resultado de la métrica RC.

	Acoplamiento	Complejidad de mantenimiento	Reutilización	Cantidad de pruebas
Ninguno	27%			
Baja	45%	83%	13%	83%
Media	11%	4%	4%	4%
Alta	17%	13%	83%	13%

Tabla 12 : Instrumento de evaluación de la métrica RC.

No	Clase	Cantidad de Relaciones de Uso	Acoplamiento	Complejidad Mantenimiento	Reutilización	Cantidad de Pruebas
1	ComponenteController	1	Baja	Media	Media	Media
2	Bundle	8	Alta	Alta	Baja	Alta
3	BundleProxy	2	Media	Alta	Baja	Alta
4	BundleSeeker	2	Media	Alta	Baja	Alta
5	BundleSolver	2	Media	Alta	Baja	Alta
6	Event	1	Baja	Media	Media	Media

7	Factory	3	Alta	Alta	Baja	Alta
8	FactoryProxy	1	Baja	Media	Media	Media
9	Manager	3	Alta	Alta	Baja	Alta
10	Observer	0	Ninguno	Baja	Alta	Baja
11	ObserverEvent	1	Baja	Media	Media	Media
12	Service	2	Media	Alta	Baja	Alta
13	ServiceDependent	2	Media	Alta	Baja	Alta
14	SourceEvent	2	Media	Alta	Baja	Alta
15	SourceObserver	0	Ninguno	Baja	Alta	Baja
16	TransactionalService	0	Ninguno	Baja	Alta	Baja

Teniendo en cuenta la cantidad de dependencias entre las clases mostradas, está establecido que una cantidad de relaciones pequeña será igual a 1, una cantidad media será entre 1 y 3 relaciones y grande será más de 3 relaciones.

Tabla 13 Cantidad de dependencias por clases

Criterio	Categoría	Cantidad de clases	Promedio
0 dependencias	Muy Bueno	3	5.660377358
1 dependencias	Bueno	3	5.660377358
2 dependencias	Regular	6	11.32075472
3 dependencias	Malo	2	3.773584906
> 3 dependencias	Muy Malo	1	1.886792453
Total		15	28.30188679

Como se puede observar en la tabla 18 el 48 % de las clases están clasificadas de Pocas y un 50% de Medias y 2% de Muchas según sus relaciones, lo cual representa un resultado positivo según los atributos de Acoplamiento, Complejidad del Mantenimiento, Reutilización y Cantidad de Pruebas.

Tabla 14 Evaluación según cantidad de relaciones.

Umbral	**Relaciones**	**Cantidad de Clases**
Pocas	<=1	7
Medias	>1 y <= 3	8
Muchas	>3	1

A continuación se grafican los resultados obtenidos al aplicar la métrica RC.

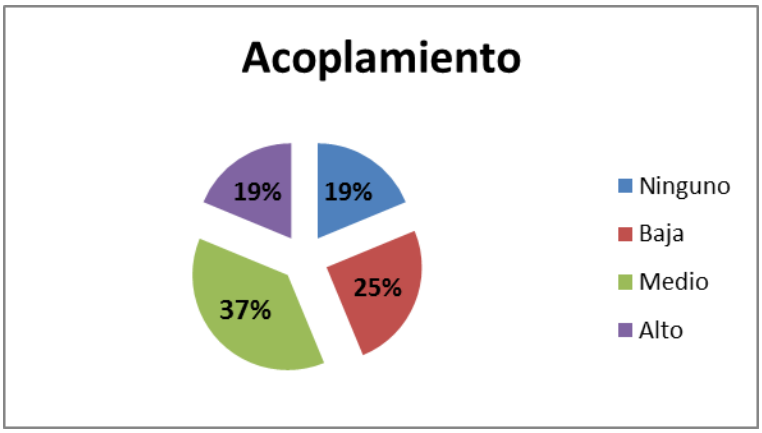


Figura 24 Resultados de la evaluación de la métrica RC para el atributo de calidad Acoplamiento.

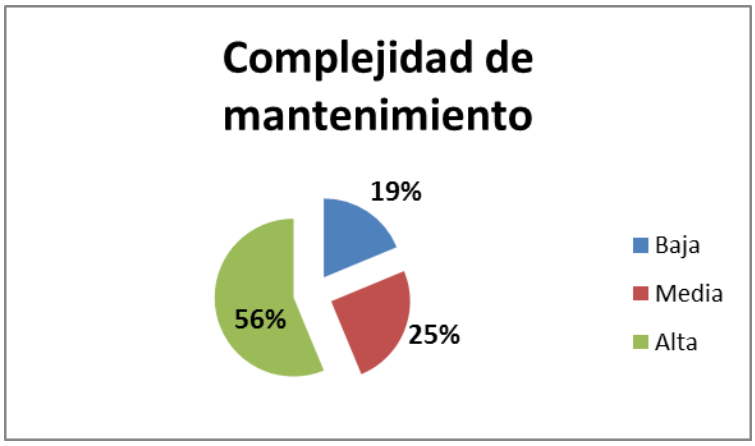


Figura 25 Resultados de la evaluación de la métrica RC para el atributo de calidad Complejidad de Mantenimiento.

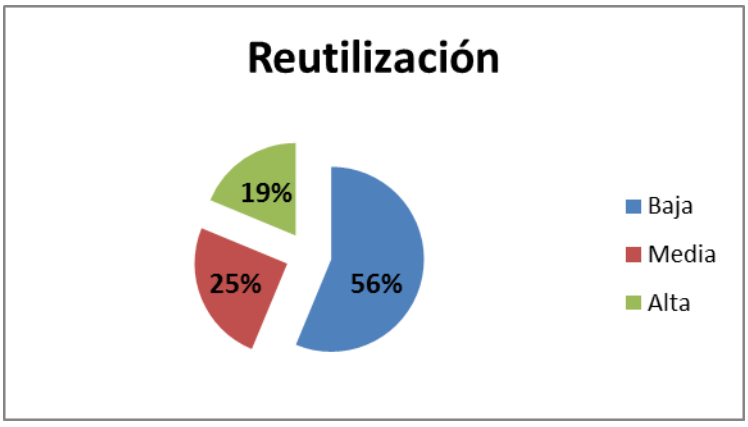


Figura 26 Resultados de la evaluación de la métrica RC para el atributo de calidad Reutilización.

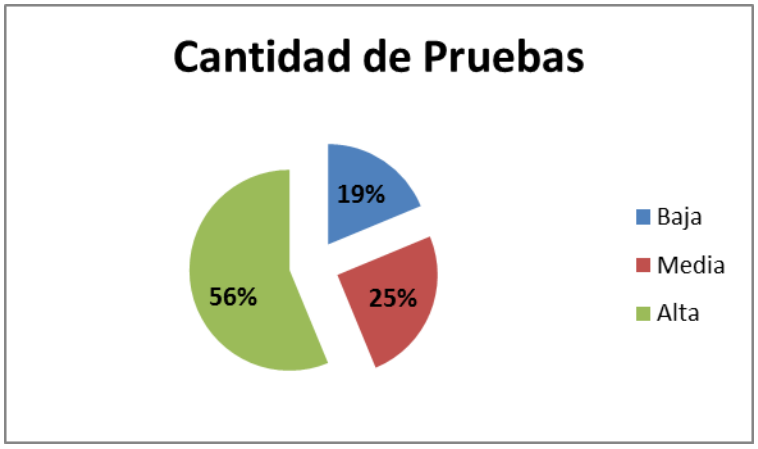


Figura 27 Resultados de la evaluación de la métrica RC para el atributo de calidad Cantidad de Pruebas.

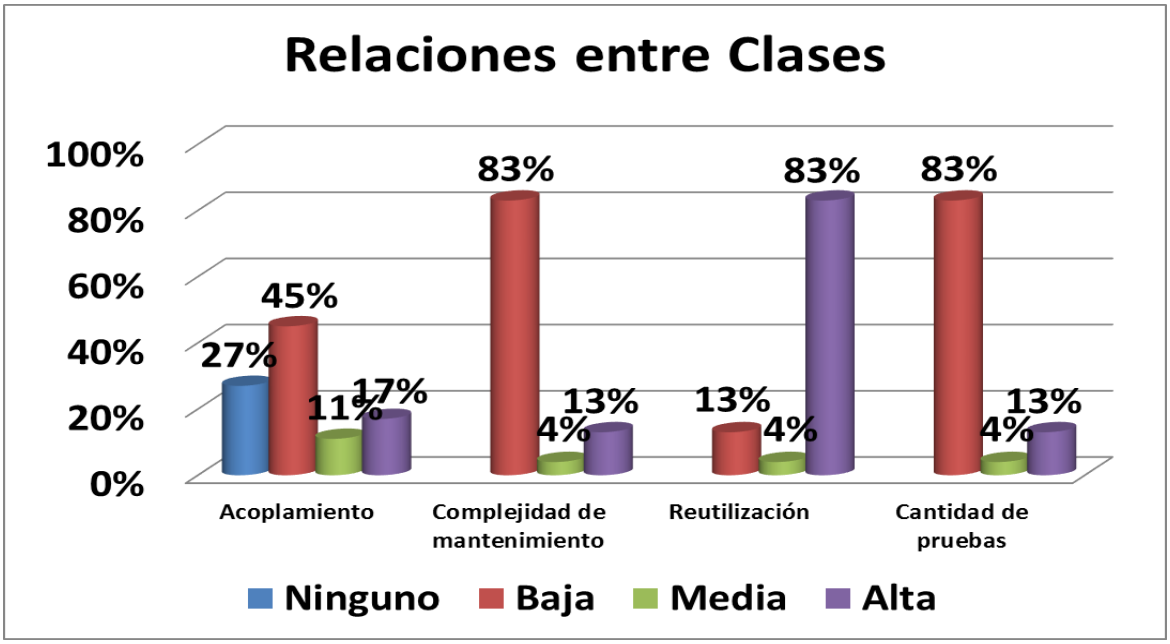


Figura 28: Resultado de la aplicación de la métrica RC.

De igual manera se garantiza que la solución será fácil de mantener y/o modificar en caso de ser necesario. También se garantiza la cantidad de pruebas necesarias para probar las clases de la solución.

3.3.3 Matriz interferencia de indicadores de calidad

La matriz de interferencia de indicadores de calidad es una representación de los atributos de calidad y las métricas utilizadas para evaluar la calidad del diseño propuesto para el componente. Con ella se puede conocer si los resultados obtenidos de las relaciones atributo/métrica son positivos o no, llevando estos

resultados a una escalabilidad numérica donde, si los resultados son positivos se le asigna el valor uno, si son negativos toma valor cero y si no existe relación es considerada como nula y es representada con un guión simple (-). Luego se puede obtener un resultado general para cada atributo promediando todas sus relaciones no nulas (La Llave Iglesias, 2012). En la tabla se muestra el resultado final reflejado en la matriz interferencia de indicadores de calidad.

Tabla 15 Resultado de la matriz interferencia de indicadores de calidad.

	Malo	Regular	Bueno
Responsabilidad	0		1
Complejidad de implementación	0	0	1
Reutilización	0	0	1
Acoplamiento	0	0	1
Complejidad de mantenimiento	0	0	1
Cantidad de pruebas	0	0	1

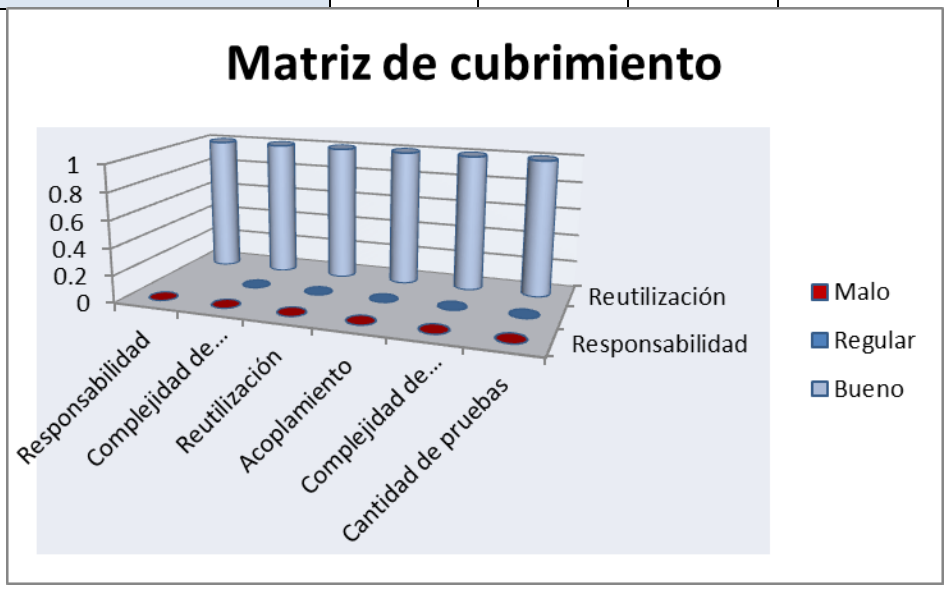


Figura 29 Matriz de cubrimiento.

3.3.4 Valoración de los resultados de las métricas

Luego de aplicar de las métricas TOC y RC se han obtenido resultados positivos, esto evidencia que la solución está correctamente diseñada garantizando un diseño robusto con una alta reutilización. De igual manera se garantiza que la solución será fácil de mantener y/o modificar en caso de ser necesario. También se garantiza la cantidad de pruebas necesarias para probar las clases de la solución.

3.5 Pruebas de Software

Las pruebas constituyen una actividad en la cual un sistema o componente es ejecutado bajo condiciones o requisitos especificados, los resultados son observados y registrados, y una evaluación es hecha de algún aspecto del sistema o componente (Grady Booch).

3.5.1 Tipos de pruebas

Hay dos formas de probar cualquier producto construido (y casi cualquier cosa): 1) Si se conoce la función específica para la que se diseñó el producto, se aplican pruebas, que demuestren que cada función es plenamente operacional, mientras se buscan los errores de cada función; 2) Si se conoce el funcionamiento interno del producto, se aplican pruebas para asegurarse de que todas las piezas encajan; es decir, que las operaciones internas se realizan de acuerdo con las especificaciones y que se han probado todos los componentes internos de manera adecuada. El primer término se refiere a las pruebas de caja negra y el segundo a las de caja blanca (Pressman, 2005).

3.5.1.1 Pruebas de Caja Blanca

Se comprueban los caminos lógicos del software proponiendo casos de prueba que ejerciten conjuntos específicos de condiciones y/o bucles. Se puede examinar el estado del programa en varios puntos para determinar si el estado real coincide con el esperado o mencionado. Requieren del conocimiento de la estructura interna del programa y son derivadas a partir de las especificaciones internas de diseño o el código. De los métodos de prueba basados en caja blanca estudiados se propone: **La prueba del camino básico** que permite al diseñador de casos de prueba obtener una medida de la complejidad lógica de un diseño procedimental y usar esa medida como guía para la definición de un conjunto básico de caminos de ejecución. Los casos de prueba obtenidos del conjunto básico garantizan que durante la prueba se ejecuta por lo menos una vez cada sentencia del programa. (Pressman, 2002)

A continuación se procede a mostrar el desarrollo de la prueba del camino básico aplicado a la funcionalidad **cargarCarpetasAppAction()** que permite cargar todos los componentes registrados en el sistema. Primeramente se procede a enumerar las sentencias del código y a partir del mismo se construye el grafo de flujo asociado (ver figura 30).

Código de la función **cargarCarpetasAppAction()**

```
function cargarCarpetasAppAction()  {
if ($this->_request->getPost('node')  == 'root')  // 1
$src = '../../../apps'; //2
```

```
else
$src = $this->_request->getPost('carpeta');  //3
$dir = opendir($src);  // 4
$data = array(); //4
while (false !== ( $file = readdir($dir)))  { //5
if (( $file != '.' ) && ( $file != '..' )) { //6
if (is_dir($src  . DIRECTORY_SEPARATOR  . $file) && $file != '.svn')  { //7
if (file_exists($src  . DIRECTORY_SEPARATOR  . $file . DIRECTORY_SEPARATOR  . $file  . '.scl')) { //8
$data[] = array('text' => $file, 'dir' => $src . DIRECTORY_SEPARATOR  . $file, 'expandable'  => false, 'icon' =>
'/images/icons/componente.png',  'leaf' => true); //9
} else { $data[] = array('text'  => $file, //10
'dir'  => $src . DIRECTORY_SEPARATOR  . $file, 'expandable'  => true,
'leaf'  => false); } //11
} else {
//echo "archivo ".$src . DIRECTORY_SEPARATOR  . $file."<br>"; //12 }
} //13
} //14
closedir($dir);  //15
$result = array('success' => true, 'data' => $data); //15
echo json_encode($result);  //15
return; //15
```

Grafo de flujo asociado a la funcionalidad cargarCarpetasAppAction()

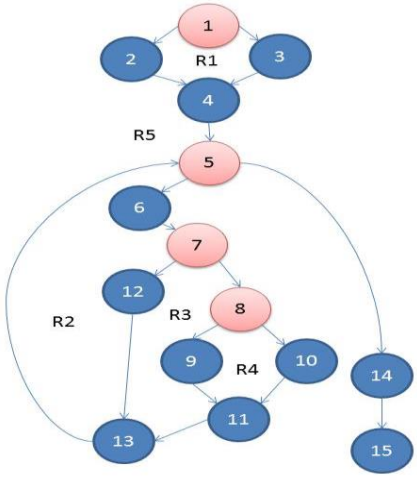


Figura 30 Grafo de flujo asociado a la funcionalidad cargarCarpetasAppAction()

Luego de haber construido el grafo se realiza el cálculo de la complejidad ciclomática mediante las tres fórmulas descritas a continuación, las cuales deben arrojar el mismo resultado para asegurar que el cálculo de la complejidad es correcto.

1. V (G) = (A – N) + 2

Siendo “A” la cantidad total de aristas y “N” la cantidad total de nodos.

V (G) = (18- 15) + 2

V (G) = 5

2. V (G) = P + 1

Siendo “P” la cantidad total de nodos predicados (son los nodos de los cuales parten dos o más aristas).

V (G) = 4 + 1

V (G) = 5

3. V (G) = R

Siendo “R” la cantidad total de regiones, para cada fórmula “V (G)” representa el valor del cálculo.

V (G) = 5

El cálculo efectuado mediante las tres fórmulas ha dado el mismo valor, dando como resultado 5, lo que indica que existen 5 posibles caminos por donde el flujo puede circular, y determina el número de pruebas que se deben realizar para asegurar que se ejecute cada sentencia al menos una vez. Seguidamente es necesario representar los caminos básicos por los que puede recorrer el flujo:

Camino básico #1: 1-2-4-5-14-15

Camino básico #2: 1-3-4-5-14-15

Camino básico #3: 1-2-4-5-6-7-12-13-5-14-15

Camino básico #4: 1-2-4-5-6-7-8-9-11-13-5-14-15

Camino básico #5: 1-2-4-5-6-7-8-10-11-13-5-14-15

Para cada camino se realiza un caso de prueba. En este caso se pone un ejemplo para el **Camino básico#1.** Ver **Anexo 4** los casos de prueba para los caminos básicos #2, #3, #4, #5.

Caso de prueba para el Camino básico #1:

Descripción: Los datos de entrada cumplirán con los siguientes requisitos:

La variable **$this->_request->getPost('node')** = 'root'

La variable **$file = readdir($dir))** toma valor **false**

Condición de ejecución: Se cumple la primera condición entra a la condicional ***if*** y la variable toma valor $**src** = '../../../apps' , no se cumple la condición del **while** porque no existe la dirección, es decir no existe ningún componente registrado ni carpetas en la dirección.

Resultados esperados: No se muestra los componentes en el sistema que están en ese nivel ni las carpetas de la dirección.

Caso de prueba para el Camino básico #2:

Descripción: Los datos de entrada cumplirán con los siguientes requisitos:

La variable **$this->_request->getPost('carpeta')** = 'herramientas'

La variable **$file = readdir($dir))** toma valor **false**

Condición de ejecución: No se cumple la primera condición entra a la condicional ***else*** y la variable toma valor $**src** = '../../../apps' , no se cumple la condición del **while** porque no existe la dirección, es decir no existe ningún componente registrado.

Resultados esperados: No se muestran las carpetas ni los componentes que contiene la carpeta "herramientas".

Caso de prueba para el Camino básico #3:

Descripción: Los datos de entrada cumplirán con los siguientes requisitos:

La variable **$this->_request->getPost('node')** = 'root'

La variable **$file = readdir($dir))** toma valor **true**

Condición de ejecución: Se cumple la primera condición entra a la condicional ***if*** y la variable toma valor $**src** = '../../../apps', se cumple la condición del **while** porque existe la dirección, se cumple la condición **if** porque tiene nombre la carpeta, no se cumple el tercer **if** porque no existe el fichero o es .svn y entra al condicional **else** correspondiente, luego vuelve a entrar al while y no se cumple la condición por lo que sale del mismo y cierra el archivo.

Resultados esperados: Se muestra los componentes en el sistema que están en ese nivel y las carpetas de la dirección.

Caso de prueba para el Camino básico #4:

Descripción: Los datos de entrada cumplirán con los siguientes requisitos:

La variable **$this->_request->getPost('node')** = 'root'

La variable **$file = readdir($dir))** toma valor **true**

Condición de ejecución: Se cumple la primera condición entra a la condicional ***if*** y la variable toma valor $**src** = '../../../apps', se cumple la condición del **while** porque existe la dirección, se cumple la condición **if**

porque tiene nombre la carpeta, se cumple el **if** tercero porque existe el fichero y no es .svn, no se cumple el cuarto **if** porque no existe el fichero.scl y el arreglo $data [] toma los datos de la carpeta seleccionada mostrando la carpeta seleccionada y sus características.

Resultados esperados: Se muestra los componentes en el sistema que están en un primer nivel y las carpetas de la dirección seleccionada.

Caso de prueba para el Camino básico #5:

Descripción: Los datos de entrada cumplirán con los siguientes requisitos:

La variable **$this->_request->getPost('node')** = 'root'

La variable **$file = readdir($dir))** toma valor **true**

Condición de ejecución: Se cumple la primera condición entra a la condicional ***if*** y la variable toma valor $**src** = '../../../apps', se cumple la condición del **while** porque existe la dirección, se cumple la condición **if** porque tiene nombre la carpeta, se cumple el **if** tercero porque existe el fichero y no es .svn, se cumple el cuarto **if** porque existe el fichero.scl y el arreglo $data [] toma los datos del fichero.scl mostrando el componente que contiene el fichero.scl

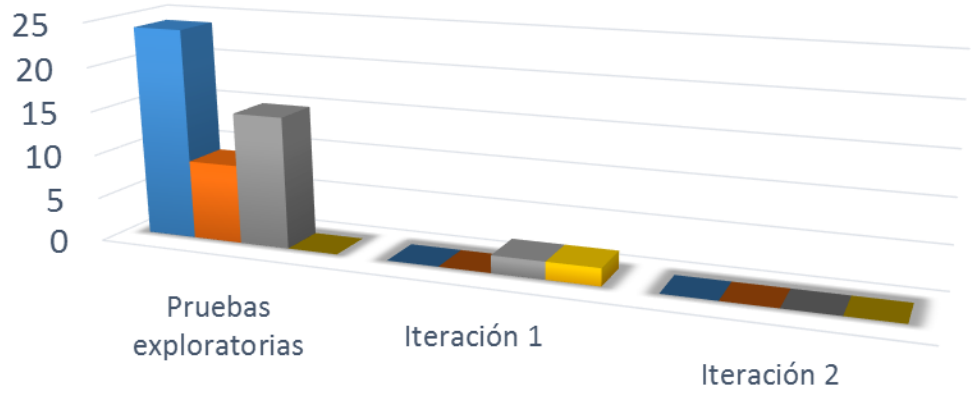
Resultados esperados: Se muestra los componentes en el sistema que están en ese nivel y las carpetas de la dirección.

3.5.1.2 Valoración de las Pruebas de Caja Blanca

Luego de realizado la prueba de caja blanca para el método cargarCarpetasAppAction() que permite cargar todos los componentes registrados en el sistema. Se obtuvo un total de cinco caminos básicos aplicando la complejidad ciclomática, quedando definido cinco diseños de casos de pruebas. Una vez aplicados todos los casos de prueba, se estará seguro de que todas las sentencias del programa se han ejecutado por lo menos una vez.

3.5.2 Pruebas de Caja Negra

Pruebas de caja negra: se refiere a las pruebas que se llevan a cabo sobre la interfaz del software. O sea, los casos de prueba pretenden demostrar que las funciones del software son operativas, que la entrada se acepta de forma adecuada y que se produce un resultado correcto, así como que la integridad de la información externa se mantiene. De las técnicas de prueba basados en caja negra se selecciona **Técnica de Partición de Equivalencia**: esta técnica divide el campo de entrada en clases de datos que tienden a ejercitar determinadas funciones del software (Pressman, 2005). Ver **Anexo 1**.



	Pruebas exploratorias	Iteración 1	Iteración 2
No conformidades	24	0	0
Aplicación	9	0	0
Documentación	15	2	0
No procede	0	2	0

Figura 31: Resultados de las pruebas funcionales.

Se detectaron 9 no conformidades de la aplicación y 15 no conformidades de la documentación para un total de 24 no conformidades en la prueba exploratoria. Para una primera iteración fueron corregidas 22 no conformidades y de ellas 2 no proceden porque no corresponde a la implementación de la herramienta y se incluyen 2 más, para una segunda y última iteración no se detectaron ninguna no conformidad. Lo que permitió detectar el mínimo de errores. Ver figura 31.

3.5.3 Valoración de los resultados de las pruebas de software.

Luego de realizar las pruebas de software de caja blanca y caja negra se obtuvo como resultado que la solución implementada cumple con los requerimientos definidos y que la ocurrencia de errores en los escenarios identificados es nula, garantizando así un software correctamente funcional.

3.6 Validación de la Investigación

3.6.1 Diseño de experimento

Para validar la investigación se utilizó un "cuasi-experimento" se refiere a diseños de investigación experimentales en los cuales los sujetos o grupos de sujetos de estudio no están asignados aleatoriamente. Los diseños cuasi-experimentales más usados siguen la misma lógica e involucran la comparación de los grupos de tratamiento y control como en las pruebas aleatorias. En otros diseños, el grupo de tratamiento sirve como su propio control (se compara el "antes" con el "después") y se utilizan métodos de series de tiempo para medir el impacto neto del programa (Rossi y Freeman, 1993). Quedando validada la variable dependiente que es el tiempo.

Análisis estadístico

La validez y aplicabilidad de la propuesta se podrá comprobar a través de su aplicación en un estudio de caso.

Instrumentos

Para medir las variables operacionales se utilizarán las encuestas. Ver **Anexo 2**.

Como resultado de los cuestionarios realizados se obtuvo que para un total de 4 encuestados que han creado componentes sobre el marco de trabajo Sauxe, coinciden que se demoran aproximadamente más de una jornada de trabajo en definir un componente, sin embargo con la utilización de la herramienta el resultado es menor de 10 minutos para los 4 casos ver Figura 32.

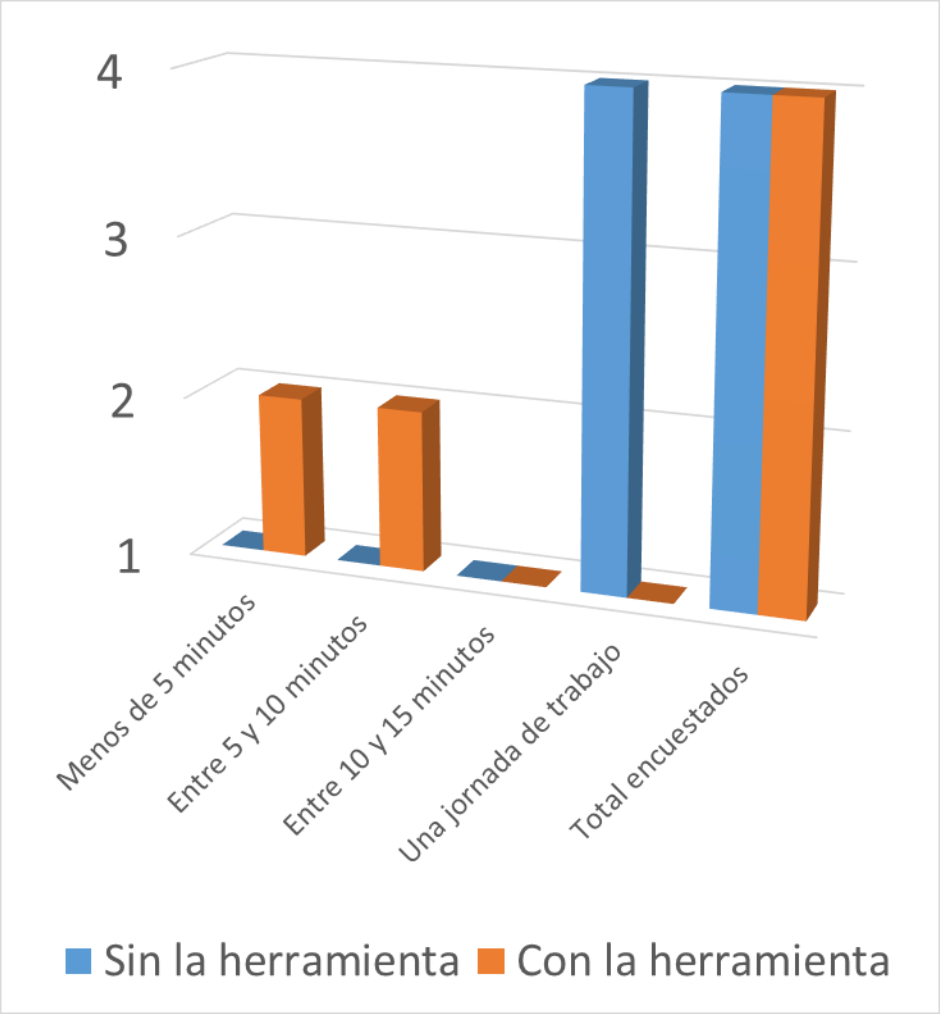


Figura 32: Resultados de las encuestas.

Los resultados del cuestionario revelan además que de forma manual, para gestionar un componente tiene una duración aproximada de más 8 horas (una jornada laboral), para una sola persona. Mientras que haciendo uso de la herramienta, solo una duración de 10 minutos, para una reducción del 95,85% del tiempo para gestionar componentes que se dedicaba antes de aplicada la propuesta.

3.7 Conclusiones del capítulo

Se implementó la Herramienta para “Gestionar componentes” en el marco de trabajo Sauxe logrando reducir el tiempo empleado en la definición de estos componentes de forma manual así como la ocurrencia de errores en este proceso y pudiendo llevar un control de los componentes. Se validó la

solución desarrollada empleando técnicas para la validación de los requisitos, métricas para la validación del diseño y pruebas de caja blanca y caja negra para la validación de la aplicación. Todo esto permitió garantizar que se ha obtenido un software de alta calidad.

Conclusiones Generales

Al concluir la investigación para el desarrollo de la Herramienta “Gestionar componentes” en el marco de trabajo Sauxe, se pudo arribar a las siguientes conclusiones:

- Se ha realizado el análisis y estudio de los conceptos asociados al dominio del problema que permitieron identificar las características de los elementos de los componentes.

- La selección y presentación de los diferentes marcos existentes a nivel internacional permitió identificar las características relevantes y las ventajas que podrían ser utilizadas durante el desarrollo de la solución.

- A partir de la problemática planteada se identificó un problema a resolver. Se definió el objetivo general del cual se desglosaron los objetivos específicos.
- Se elaboró el modelo conceptual de la solución, donde se analizó la relación que existe entre cada uno de los conceptos identificados.
- Se realizó el análisis y diseño de la herramienta para la gestión de los componentes que sirvió como base para llevar a cabo el proceso de implementación de la solución.
- Se realizó la implementación de una herramienta que gestionará los componentes que se desarrollan con el marco de trabajo Sauxe que fueron definidos a través de un modelo de componentes propuesto por el Departamento de Tecnología del CEIGE y aplicado al *framework* Sauxe, elaborado por el estudiante Abraham Calás, avalada por el doctor Oiner Gómez Baryolo.
- La herramienta brindará la posibilidad de reducir el tiempo en que son definidos los componentes en el marco de trabajo, así como el tiempo de desarrollo que conlleva la implementación de este elemento.
- Además se definieron las pautas ingenieriles siguiendo el modelo de desarrollo propuesto por el CEIGE, permitiendo comprender todo el ciclo de desarrollo de la herramienta.

RECOMENDACIONES

Se recomienda que se integre la herramienta “Gestionar componentes” a la herramienta “Generador de módulo”, permitiendo crear la estructura de las carpetas y luego definir los componentes en el marco de trabajo Sauxe.

BIBLIOGRAFIA

Acuña, Karenny Brito. *RUP Diseño e implementación.*

Aires, Universidad de Buenos. 2009. *Modelo Conceptual.* 2009.

Apache Subversion. [En línea] [Citado el: 27 de enero de 2014.] http://subversion.apache.org.

Baryolo, Dr. Oiner Gómez. 2010. *Solución informática de autorización en entornos multientidad y multisistema.* s.l. : Universidad de las Ciencias Informáticas, 2010.

1976 . *Camino básico.* 1976 .

CEIGE, Departamento de Tecnología. 2008. *Normas y estándares de Codificación del ERP.* La Habana : s.n., 2008.

Cornejo, José Enrique González. «¿Qué es UML? [En línea] http://www.docirs.cl/uml.htm.

Esser, S. 2009. *Secure Programming with the Zend-Framework.* 2009.

Grady Booch, James Rumbaugh, Ivar Jacobson. *El lenguaje Unificado de Modelado.*

http://www.rapidsvn.org/. 2013. RapidSVN. [En línea] 28 de noviembre de 2013.

IngSoftwareII-CUFM. [En línea] [Citado el: 27 de mayo de 2013.] http://cufmingsoftware.wordpress.com/estandares-de-diseno.

Kaisler. 2005. *Software Paradigms.* 2005.

M. A. Friedhelm Betz, N. L. Antony Dovgal, G. R. Hannes Magnusson, J. V. Damien Seguy, y otros. 2012. *Manual de PHP.* 2012.

Martínez, J.S. 1999. *Una Arquitectura para una Herramienta de Patrones de Diseño.* 1999.

Pérez Mariñán, P. 2007. *Patrones de Diseño.* 2007.

PostgreSQL, Manual del usuario de de PostgreSQL. 2008. *PostgreSQL.* 2008.

Pressman, Roger S. 2002. *Ingenieria de Software, Un enfoque práctico.* s.l. : McGraw-Hill Companies, 2002. 8448132149.

Pressman, Roger. 2009. *Software Engineering. A practitioner´s Approach.* 2009.

R. Frederick, S. «Cutter» Colin y Blades, y Shay. 2008. *Learning ExtJS.* 2008.

Sommerville, Ian. 2006. *Software Engineering.* s.l. : 9th ed. Pearson Education, Inc., , 2006.

Y. Aquino, A. 2009. *Implementación del módulo de Contabilidad General del Sistema Integral de Gestión Cedrux.* 2009.