Universidad de las Ciencias Informáticas

Facultad de Ciencias y Tecnologías Computacionales

Generador visual de entidades para Doctrine 2.0

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

Autores

Ismary Izquierdo Domínguez

Yazmin del Carmen Sagardoy Hernández

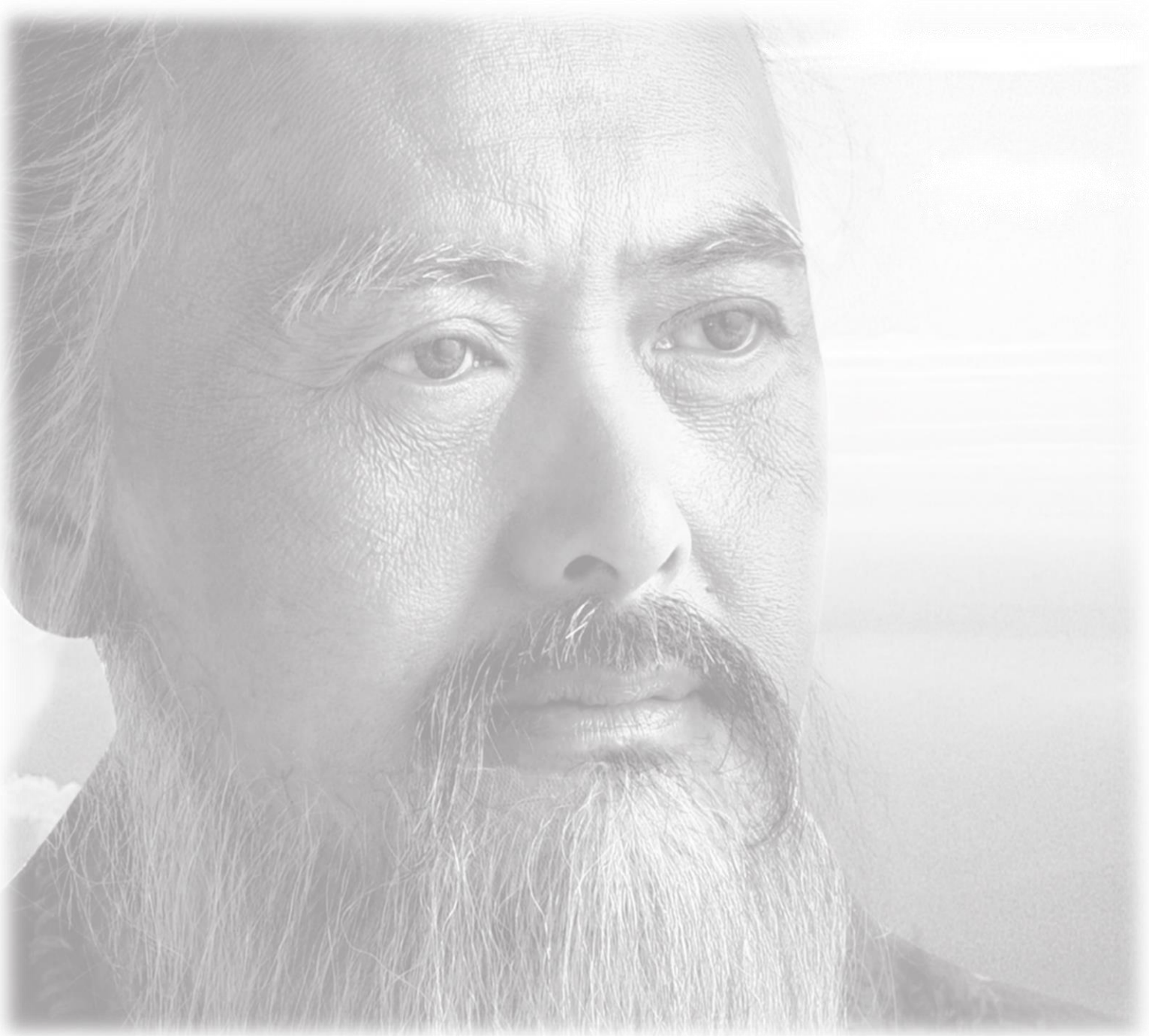
Tutores

MSc. Yadira Robles Aranda

Ing. Karel Rodríguez Carmenates

La Habana, 2 de junio

2017



"Cuando el objetivo te parezca difícil, no cambies de objetivo; busca un nuevo camino para llegar a él."

Confucio

Declaración de autoría

Se declara que Ismary Izquierdo Domínguez y Yazmin del Carmen Sagardoy Hernández son las únicas autoras del trabajo de diploma “Generador visual de entidades para Doctrine 2.0” y se le reconoce a la Universidad de las Ciencias Informáticas los derechos patrimoniales de la misma, con carácter exclusivo.

Para que así conste se firma la presente a los ____ días del mes de junio del año 2017.

_____	_____
Firma de la autora	Firma de la autora
Ismary Izquierdo Domínguez	Yazmin del Carmen Sagardoy Hernández
_____	_____
Firma de la tutora	Firma del tutor
MSc. Yadira Robles Aranda	Ing. Karel Rodríguez Carmenates

Dedicatoria

Ismary

De todo corazón a mis padres Berkis y Luis, así como mi hermano Ismael, por todo su sacrificio y apoyo en el transcurso de la carrera. A ellos por todo el amor que me han regalado a lo largo de mi vida.

Ismary

Esta tesis quiero dedicarla en primer lugar a las personitas que me dieron la vida, y que en cada minuto de ella siempre me apoyaron y guiaron mis pasos. Mis padres, ellos que siempre de una manera u otra estuvieron conmigo y nunca me dieron la espalda y que han sabido aconsejarme para que hoy día esté aquí no solo cumpliendo mi sueño sino el de ellos también.

Por otro lado, quiero dedicarla: a mi hermanito Jonathan quién espero que el día de mañana se convierta también en un gran ingeniero y que pueda crear juego como él dice que quiere hacer. A mis abuelos los cuales sé que deben estar muy orgullosos de ver que su nietecita ya se está convirtiendo en una ingeniera, y a toda mi familia en general también se las dedico. Los quiero mucho.

Yazmin

Agradecimientos

De Ismary Izquierdo Domínguez:

Agradezco a mi familia y amigos más cercanos por ser parte del camino para lograr esta meta: ser Ingeniera en Ciencias Informáticas. Gracias por ayudarme a cumplir mi sueño.

A mi padre Luis por confiar en mí, porque desde pequeña me llamaba ingeniera y hoy, donde quiera que esté, sabe que nuestro sueño se hace realidad. A mi madre Berkis por darme siempre lo mejor que pudo, y más. Porque a pesar de los obstáculos me ha guiado por un buen camino, y es ella mi primer pensamiento cuando necesito una razón para seguir adelante. A mi hermano Ismael por ese amor incondicional que sentimos uno por el otro … con él aprendí que ser hermanos es mucho más que compartir un lazo sanguíneo y un ADN. A mi padrastro Damián por acogerme como una hija. Siempre estaré agradecida.

A mis amigas de toda la vida Elianita, Mary y Yuni, porque nos puede separar grandes distancias, largo tiempo, ... pero cuando nos reunimos nuevamente no cambian los sentimientos: estas son amistades verdaderas, son mis hermanas de corazón.

A todos los profesores que han contribuido en mi formación profesional y humanista. En especial a dos de ellos que después de cumplir su labor se convirtieron en excelentes amistades: Raciel Pérez y Dianet Utria. A Silvano Merced Len por ser un educador ejemplar, por su preocupación y ayuda, por dedicar, cada día, un tiempo para pensar en sus estudiantes. A mis tutores por desempeñar una excelente labor. A ellos mis más gratos agradecimientos.

A Nelson por los años que compartimos juntos y ayudarme tanto con los estudios.

A mi dúo de tesis Yazmin, por toda su paciencia e incondicionalidad en mis últimos años de la carrera. Estoy agradecida por todo su apoyo, por cada detalle y ocasión que me sorprendió con su actitud. A Alejandro de León por ser mi mayor confidente, mi amigo incondicional. Por decir ¡presente! en los momentos buenos y malos. A Leo por su amor leal, su confianza y compartir conmigo parte de su vida. A su familia por todo su afecto y acogerme como una hija.

A mis compañeros de año, que fuimos todos como un pequeño grupo de clases. Porque con ellos pasé los mejores momentos de la universidad, esos que con los años no se olvidan. Agradecimientos especiales para mis amigos Marlón, Patry y Yandy, gracias por hacer un lugar para mí en sus corazones.

Finalmente, a todas las personas que me ayudaron durante mi carrera para culminar mis estudios satisfactoriamente. ¡Gracias!

De Yazmin del Carmen Sagardoy:

Quiero agradecer a mi mamá y a mi papá por todo el esfuerzo y sacrificio que han hecho para que yo pudiera culminar mis estudios y que hoy me encuentre aquí exponiendo finalmente la tesis. Quiero decirles que son los padres más bellos que pudiera tener y si me hubieran dado la oportunidad de poder escoger a los padres que quisiera sin dudarlo los escogería a ellos mismos, gracias por su amor, su paciencia, por el ejemplo que me han dado y por dedicarme cada minuto de sus vidas desde que nací.

Agradecer de forma general a toda mi familia por el apoyo que siempre recibí por parte de ellos: Por ejemplo, a mi hermanito Joni que es una de las personas más importantes en mi vida y aunque siempre estamos fajados lo quiero mucho, a mis abuelos a quienes considero mis segundos padres pues nunca me han dado la espalda y al contrario siempre me han consentido como la niñita chiquitica que soy para ellos y que, aunque tenga la edad que tenga seguiré siendo pequeñita ante sus ojos, a mi tía Yoleidys a la que no me canso de molestar para que me arregle el pelo, me ponga las uñas, y que también me sirvió como ejemplo para llegar a donde estoy hoy. A mi madrina Leono y a mi padrino Froily a los cuales quiero mucho, les agradezco que siempre se preocupen por mí, y a ti Leono también te doy las gracias por los flanes esos tan rico que haces a cada rato y por nunca decirme que no cuando te pido que me pintes el pelo.

Le doy las gracias a mi novio por siempre apoyarme y cuidarme desde que estamos juntos. Te agradezco por aguantar mi carácter que ha sido muy cambiante en estos últimos tiempos por los nervios de la tesis, de veras que te mereces una medalla. Gracias por estar a mi lado en los momentos que pensaba que no podía y demostrarme que yo sí puedo hacer todo lo que me proponga. Eres muy especial para mí.

Quiero mencionar a una personita que conocí aquí en la universidad y que desde el primer día supe que formaría parte de mi vida para siempre: Marlon, ese amigo que más que amigo es hermano, esa persona que nunca me dio de lado y que siempre me brindó su cariño y amistad, además de arroz con leche y dulce leche que a cada rato hacía en su antiguo apartamento. Juntos hemos vivido muchas cosas y espero que solo sean el principio de muchas más. Te doy las gracias por siempre estar ahí conmigo.

Agradecer a Ismary que además de ser mi dúo de tesis por encima de todo es mi amiga, decirle que sin ella esto no hubiese sido igual. Te doy las gracias por no solo ayudarme en la elaboración de este trabajo, sino que también me has sabido escuchar y apoyar en momentos que me hacía falta una mano amiga.

Además, agradecer a mis tutores Yadira y Karel que han tenido la paciencia y delicadeza de ayudarnos con la tesis sin poner peros ni contras. Decirles que sin ellos no hubiésemos logrado estar aquí hoy. Gracias por su disposición y apoyo.

Darle las gracias a los miembros del proyecto de elecciones quienes nos abrieron las puertas del laboratorio y nos recibieron como si fuéramos miembros de su equipo y nos brindaron sus meriendas en cada visita nuestra.

A mi actual grupo FC502 agradecer de forma general ya que de una manera u otra me han brindado muchos momentos de alegrías que no voy a olvidar. Algunas de esas personitas son: la Patry (que, aunque no está aquí hoy sé que desde Las Vegas está muy contenta por nosotras), a Yane, Suinny, Claudiña, Lici, Mida, Milagrillo, Lenia, y a los chicos como el Padri y Yandy que han sido grandes amigos y consejeros, a Asiel, Jorgito, Roly que en ocasiones me animó con sus dibujos, Karel, Idel, Cano, Rene, Ariel, Raimel, Rolando, Luis Miguel Fonseca, Osmin y bueno espero que no me haya faltado nadie por mencionar. ¡¡¡A todos muchas gracias!!!

Resumen

La técnica de programación orientada a objetos y la base de datos de tipo relacional son dos paradigmas diferentes, pero generalmente coexisten durante el proceso de desarrollo de software. Convertir los objetos del lenguaje de programación a registros de la base de datos y el proceso inverso, constituyen tareas comunes de los programadores. Para facilitar esta tarea, los desarrolladores se apoyan en herramientas de mapeo relacional de objetos. La inexistencia de este tipo de herramienta en el proyecto Sistema de Gestión e Información del Proceso Electoral (SIGEL) dificulta el trabajo de los especialistas. Por lo que se decidió desarrollar una herramienta que permita la generación de entidades para Doctrine 2.0 a partir del diseño del modelo de datos. Durante el desarrollo de la herramienta fueron utilizadas tecnologías libres como el Entorno de Desarrollo Integrado NetBeans IDE, el lenguaje de programación Java, además de una variante de la guía ofrecida por la metodología de desarrollo AUP, empleada en los proyectos productivos de la Universidad. Una vez realizada la implementación se le hicieron pruebas a la solución con el objetivo de comprobar el funcionamiento del código y la interfaz, obteniendo finalmente la aplicación deseada. Esta facilita el trabajo de los desarrolladores cuando se requiere mapear grandes volúmenes de clases del marco de trabajo Doctrine 2.0, así como cuando se desea realizar alguna modificación al diseño del modelo de datos correspondiente.

Palabras claves: base de datos, Doctrine 2.0, entidades, marco de trabajo.

Abstract

The object-oriented programming technique and the relational-type database are two different paradigms, but they generally coexist during the software development process. The conversion of programming language objects into database records and the reverse process, are common tasks of programmers. To facilitate this task, developers rely on relational object mapping tools. The lack of this type of tool in the System of Management and Information of the Electoral Process (SMIEP) makes difficult the work of the specialists. Therefore, it was decided to develop a tool that allows the generation of Doctrine 2.0 entities from the design of the data model. During the development of the tool, free technologies such as the integrated development environment NetBeans IDE, the Java programming language, were used as well as a variant of the guide offered by the AUP development methodology used in the University's productive projects. After the implementation, the solution was tested with the objective of checking the operation of the code and the interface, finally obtaining the desired application. This facilitates the work of developers when it is required to map large class volumes of the Doctrine 2.0 framework, as well as when it is desired to make some modification in the design of the corresponding data model.

Keywords: *database, Doctrine 2.0, entities, framework.*

Índice de contenido

Índice de Figuras

Índice de Tablas

Introducción

El desarrollo de las Tecnologías de la Información y las Comunicaciones (TIC) acelera su ritmo a medida que el mundo se hace más próspero. Dicho desarrollo ha provocado un crecimiento a nivel mundial de la informatización, así como el aumento en la generación y almacenamiento de la información. Para atender estas necesidades surgen las bases de datos, que no son más que una serie de datos organizados y relacionados entre sí, los cuales son recolectados y explotados por los sistemas de información de una empresa o negocio en particular (Mato García, 1999). Dichas bases de datos son administradas por los Sistemas Gestores de Bases de Datos (SGBD) que permiten la administración de los datos para realizar tomas de decisiones.

Los SGBD proporcionan a los usuarios una visión abstracta de los datos. O sea, el usuario utiliza esos datos mediante una vista que el usuario es capaz de comprender, pero no tiene idea de cómo están almacenados físicamente. Los modelos de datos son el instrumento principal para ofrecer esta abstracción. Siendo las bases de datos relacionales un ejemplo representativo del modelo lógico, estas son las más comunes y extendidas, ya que permiten modelar problemas reales mediante las relaciones entre tablas o entidades, lo cual es su función fundamental (Mato García, 1999).

Los modelos de datos se pueden clasificar, dependiendo de los tipos de datos que ofrecen para describir la estructura de la base de datos, en conceptuales, físicos o lógicos. Estos últimos ocultan algunos detalles de cómo se almacenan los datos, pero pueden implementarse de manera directa en un SGBD. Ejemplos de modelos lógicos son las bases de datos relacionales cuya función fundamental es modelar problemas reales mediante las relaciones entre tablas o entidades (Marqués, 2011).

Una base de datos relacional es un conjunto de tablas y las relaciones entre estas (Silberschatz, y otros, 2002). O sea, las tablas van a estar formadas por columnas que representan los atributos pertenecientes a un concepto y filas que representan ejemplos de dichos conceptos. Las relaciones pueden establecerse entre una o más tablas. Las tablas están relacionadas cuando tienen atributos en común.

Para facilitar el trabajo con las bases de datos en el desarrollo de aplicaciones informáticas se emplean herramientas como los *frameworks* o marcos de trabajo. Estos emplean la técnica de programación denominada Mapeo Objeto-Relacional (ORM, por sus siglas en inglés *Object- Relational Mapping*), la cual consiste en la persistencia automatizada y transparente de las tablas en una base de datos relacional,

usando metadatos[1] que definen el mapeo entre los objetos y la base de datos (Bauer, et al., 2004). Symfony constituye un ejemplo de marco de trabajo específicamente para crear aplicaciones y sitios web en el lenguaje de programación PHP. Este incluye la librería Doctrine 2.0 que proporciona herramientas para simplificar el acceso y manejo de la información de la base de datos.

Doctrine es uno de los ORM más usados en el ámbito de las aplicaciones web. Algunas de las razones de su constante utilización es que posee una comunidad activa y un bajo nivel de configuración para iniciar un proyecto. Mediante la singular notación que provee, se pueden llegar a describir con precisión las entidades, atributos y las relaciones que se utilizan para almacenar y manipular la información. Como consecuencia se ha evidenciado en los últimos años un crecimiento de su documentación.

Varios centros de desarrollo de la Universidad de las Ciencias Informáticas (UCI) hacen uso del ORM Doctrine en el desarrollo de sus soluciones. Un ejemplo de ello es el Centro de Tecnologías y Gestión de Datos (DATEC), el cual se encarga de crear bienes y servicios informáticos relacionados con la gestión de datos. Esta área del conocimiento agrupa tanto a los sistemas de información, como a los denominados sistemas de inteligencia empresarial o de negocio. Uno de los proyectos productivos que se desarrollan en DATEC es el Sistema de Gestión e Información del Proceso Electoral (SIGEL), el cual comprende todos los procesos eleccionarios (Parciales y Generales) así como los procesos intermedios.

Recientemente se comenzó a desarrollar una nueva versión de la aplicación usando nuevas tecnologías. Como marco de trabajo se usa Symfony 2.7 y como ORM para la persistencia de los datos Doctrine 2.0. Una de las tareas del especialista es diseñar un modelo de datos y generar las entidades correspondientes en Doctrine 2.0. Actualmente este proceso se puede realizar de tres formas: utilizando herramientas como el ORM Designer y el Visual Paradigm para UML, mediante la consola de Symfony o de forma manual donde se debe escribir los códigos necesarios para cada entidad.

En el caso de la primera variante, las herramientas que se disponen no responden a las características del proyecto. Por ejemplo, el Visual Paradigm para UML no genera entidades en Doctrine 2.0 como se requiere mientras que el ORM Designer es una herramienta privativa, la cual necesita conexión permanente a internet y su costo es elevado. El resto de las variantes constituyen una tarea compleja cuando se trata de

[1] Metadatos: datos que describen otros datos.

sistemas que utilicen un gran número de entidades; y aún más cuando es necesaria la actualización de entidades y relaciones en el diseño del modelo de datos.

Por la problemática antes descrita se define como **problema de investigación**: las insuficiencias presentes en el proceso de generación de entidades para Doctrine 2.0 a partir del diseño de un modelo de datos, dificultan el trabajo a los especialistas de SIGEL.

Se identifica como **objeto de estudio** el proceso de generar entidades a partir del diseño del modelo de datos; enmarcado en el **campo de acción**: el proceso de generar entidades Doctrine 2.0 a partir del diseño del modelo de datos.

Para resolver el problema planteado se propone como **objetivo general**: desarrollar una herramienta que permita la generación de entidades para Doctrine 2.0 a partir del diseño del modelo de datos que contribuya al trabajo de los especialistas de SIGEL.

Objetivos específicos:

1. Elaborar el marco teórico de la investigación para el estudio de tendencias relacionado con el proceso de generación de entidades para Doctrine 2.0.
2. Definir los requisitos de software para determinar las principales funcionalidades del proceso de generación de entidades para Doctrine 2.0.
3. Diseñar la solución a partir de los requisitos para facilitar la implementación del proceso de generación de entidades para Doctrine 2.0.
4. Implementar el proceso de generación de entidades para Doctrine 2.0 para dar cumplimiento al objetivo general de la investigación.
5. Verificar el Generador visual de entidades para Doctrine 2.0 mediante las pruebas correspondientes para garantizar su funcionamiento.

Para dar cumplimiento al objetivo general se definen las siguientes **tareas de investigación**:

1. Análisis de los principales conceptos, términos y herramientas necesarias para abordar adecuadamente sobre el proceso de desarrollo del Generador visual de entidades para Doctrine 2.0.
2. Selección de las herramientas y la metodología para guiar el proceso de desarrollo del Generador visual de entidades para Doctrine 2.0.

3. Especificación de los requisitos funcionales y no funcionales a tener en cuenta para el desarrollo del Generador visual de entidades para Doctrine 2.0.
4. Elaboración del modelo de Casos de Uso del sistema a partir de la definición de requisitos para reflejar la relación del especialista con las funcionalidades del Generador visual de entidades para Doctrine 2.0.
5. Definición de la arquitectura a utilizar en el Generador visual de entidades para Doctrine 2.0 para modelar la estructura del sistema.
6. Realización del modelo de diseño para describir gráficamente las clases con sus atributos y relaciones pertenecientes al Generador visual de entidades para Doctrine 2.0.
7. Realización del diagrama de componente para ver cómo se divide internamente en componentes el Generador visual de entidades para Doctrine 2.0.
8. Implementación del Generador visual de entidades para Doctrine 2.0 a partir de las herramientas seleccionadas y los requisitos identificados para dar respuesta al objetivo general.
9. Aplicación de pruebas al software que permitan verificar el Generador visual de entidades para Doctrine 2.0.
10. Resolución de las no conformidades detectadas durante la fase de prueba para garantizar el cumplimiento del objetivo general de la investigación.

Para guiar el estudio se plantean las siguientes **preguntas de investigación**:

- ¿En qué se fundamenta teóricamente el proceso de generación de entidades para Doctrine 2.0 a partir de un modelo de datos?
- ¿Qué tecnologías, metodología y herramientas utilizar para el desarrollo del Generador visual de entidades para Doctrine 2.0?
- ¿Qué características y capacidades se deben tener en cuenta para lograr el correcto funcionamiento del Generador visual de entidades para Doctrine 2.0?
- ¿Cómo estructurar el proceso de implementación del Generador visual de entidades para Doctrine 2.0 que permita una mejor representación de los componentes a desarrollar?

- ¿Qué pruebas aplicar para comprobar el correcto funcionamiento Generador visual de entidades para Doctrine 2.0?

Para la realización de las tareas de investigación se utilizaron los **métodos científicos** que se explican a continuación:

Métodos teóricos:

- **Histórico – lógico**: se utiliza para conocer y comprender el estado del arte del proceso de generar entidades Doctrine 2.0 a partir de un diseño del modelo de datos, así como las características de las herramientas informáticas que existen en la actualidad que realizan este proceso.
- **Analítico – sintético:** se utiliza con el fin de analizar sistemas homólogos y bibliografías referentes al proceso de generar entidades Doctrine 2.0 a partir de un diseño del modelo de datos, para sintetizar los elementos importantes que se relacionan con esta investigación. Este método se utiliza también para tomar decisiones en cuanto a los resultados arrojados por las entrevistas realizadas, y para asentar las bases teóricas para el desarrollo del trabajo de diploma.
- **Inductivo – deductivo:** se utiliza con el fin de concluir nuevos conocimientos y predicciones referentes al proceso de generación de entidades para Doctrine 2.0 a partir de un diseño del modelo de datos, que posteriormente son sometidos a verificaciones empíricas.
- **Modelación:** se utiliza con el objetivo de crear modelos y diagramas que son abstracciones del producto final, permitiendo tener un dominio inicial de la información que se va a modelar, representar de forma estática los requisitos y obtener una versión del Generador visual de entidades para Doctrine 2.0 para validar los requisitos con el cliente.

Métodos empíricos:

- **Entrevista:** permite establecer una relación directa con el cliente; se utiliza para conocer sus necesidades y los principales problemas durante el proceso de generación de entidades para Doctrine 2.0 a partir de un diseño del modelo de datos y así recopilar información para el desarrollo de la propuesta de solución. Las entrevistas pueden ser estructuradas o no estructuradas. Para la construcción del Generador visual de entidades para Doctrine 2.0 se utiliza la no estructurada, en la que se trabaja con preguntas abiertas, sin un orden preestablecido, adquiriendo características de

conversación. Esta técnica consiste en realizar preguntas de acuerdo a las respuestas que vayan surgiendo durante la entrevista.

El presente documento contiene tres capítulos, los cuales se describen a continuación:

Capítulo 1. Fundamentación teórica del Generador visual de entidades para Doctrine 2.0: Se estudian conceptos para lograr un mejor entendimiento del proceso para generar entidades Doctrine 2.0 a partir del diseño de un modelo de datos. Se realiza un análisis de las herramientas de mapeo relacional de objetos en aras de conocer las tendencias que existen en la actualidad. Además, se explica la selección de las herramientas, metodología y tecnologías empleadas para el desarrollo de la solución.

Capítulo 2. Análisis y diseño del Generador visual de entidades para Doctrine 2.0: Está basado en las tareas realizadas por las disciplinas de Análisis y Diseño definidas por la variante de la metodología AUP utilizada en la universidad. Se incluye el modelo de dominio el cual refleja los conceptos relacionados en el dominio del problema. Son identificados los requisitos funcionales y no funcionales que se deben tener en cuenta para la implementación de la solución. Se define la arquitectura para modelar su estructura, se realizan diagramas de casos de uso del sistema y clases del diseño aplicando los patrones correspondientes.

Capítulo 3. Implementación y pruebas del Generador visual de entidades para Doctrine 2.0: Se abordarán los principales elementos para el desarrollo de las disciplinas de implementación y pruebas. Se muestra el diagrama de componentes y son descritos los estándares de codificación empleados en la implementación. Se evidencian las pruebas aplicadas a la solución con el objetivo de comprobar las funcionalidades del Generador visual de entidades para Doctrine 2.0.

Capítulo 1: Fundamentación teórica del Generador visual de entidades para Doctrine 2.0

Se estudian conceptos para lograr un mejor entendimiento del proceso para generar entidades Doctrine 2.0 a partir del diseño de un modelo de datos. Se realiza un análisis de las herramientas de mapeo relacional de objetos en aras de conocer las tendencias que existen en la actualidad. Además, se explica la selección de las herramientas, metodología y tecnologías empleadas para el desarrollo de la solución.

1.1. Modelo de datos

Un modelo de datos es un conjunto de conceptos que sirven para describir la estructura de una base de datos, es decir, los datos, las relaciones entre los datos y las restricciones que deben cumplirse sobre los datos (Marqués, 2011). Los modelos de datos se pueden clasificar dependiendo de los tipos de conceptos que ofrecen para describir la estructura de la base de datos, formando una jerarquía de niveles.

Los modelos de datos de alto nivel, o modelos conceptuales, disponen de conceptos muy cercanos al modo en que la mayoría de los usuarios percibe los datos, mientras que los modelos de datos de bajo nivel, o modelos físicos, proporcionan conceptos que describen los detalles de cómo se almacenan los datos en el ordenador. Los conceptos de los modelos físicos están dirigidos al personal informático, no a los usuarios finales.

Entre estos dos extremos se encuentran los modelos lógicos, cuyos conceptos pueden ser entendidos por los usuarios finales, aunque no están demasiado alejados de la forma en que los datos se organizan físicamente. Los modelos lógicos ocultan algunos detalles de cómo se almacenan los datos, pero pueden implementarse de manera directa en un SGBD (Marqués, 2011).

En el libro de Lenguaje Unificado de Modelado definen que los modelos se usan para muchos propósitos (Jacobson, y otros, 2000):

- Para captar y enumerar exhaustivamente los requisitos y el dominio de conocimiento, de forma que todos los implicados puedan entenderlos y estar de acuerdo con ellos.
- Para pensar el diseño de un sistema. Un modelo de un sistema software ayuda a los desarrolladores a explorar varias arquitecturas y soluciones de diseño fácilmente, antes de escribir el código.

- Para capturar decisiones del diseño en una forma mutable a partir de los requisitos. Un modelo de un sistema software puede captar el comportamiento externo de un sistema y la información del dominio del mundo real representado por el sistema. Otro modelo muestra las clases y las operaciones internas, que implementan el comportamiento externo.
- Para organizar, encontrar, filtrar, recuperar, examinar, y corregir la información en grandes sistemas.
- Para explorar económicamente múltiples soluciones.
- Para domesticar los sistemas complejos.

Teniendo en cuenta los conceptos planteados anteriormente, se define que un modelo de datos es un prototipo que representa gráficamente la estructura de una base de datos. Esta representación está compuesta fundamentalmente por entidades, relaciones y restricciones.

1.2. Entidades

Las entidades son elementos que existen y están bien diferenciados entre sí, que poseen propiedades y entre los cuales se establecen relaciones. Por ejemplo, una silla, un automóvil, un empleado, que son elementos concretos; pero también puede ser algo no tangible, como un suceso cualquiera, una cuenta de ahorro, o un concepto abstracto. Entre las propiedades que caracterizan a una entidad u objeto pudieran encontrarse el color, el valor monetario, el nombre. (Mato García, 1999).

Se definen las entidades como la representación de un objeto o concepto del mundo real que se describe en una base de datos. Las entidades se describen en la estructura de la base de datos empleando un modelo de datos. Cada entidad está constituida por uno o más atributos. En el modelo de entidad-relación se emplean dos tipos de entidades: entidad fuerte y entidad débil. Además, en este modelado las entidades están relacionadas entre sí a través de relaciones. Las entidades fuertes tienen atributos claves, en tanto las entidades débiles no tienen atributos claves propios. (Leandro Alegsa, 2016).

Teniendo en cuenta los conceptos anteriores, se define entidad como un objeto o concepto que poseen propiedades llamadas atributos. Entre las entidades se pueden establecer relaciones.

1.3. Doctrine 2.0

Doctrine es un marco trabajo ORM para PHP el cual proporciona persistencia transparente de objetos PHP, situándose en la parte superior de una robusta capa de abstracción de base de datos. La principal tarea de los ORM para PHP es la traducción transparente entre objetos y las filas relacionales de la base de datos.

Una de las principales características de Doctrine es su lenguaje de consulta estructurado llamado Lenguaje de Consulta de Doctrine (DQL, por sus siglas en inglés *Doctrine Query Language*) y el bajo nivel de configuración que se necesita para comenzar un proyecto. Además, DQL difiere ligeramente de SQL en que abstrae considerablemente la asignación entre las filas de la base de datos y objetos, permitiendo a los desarrolladores escribir poderosas consultas de una manera sencilla y flexible. Soporta las operaciones de crear, obtener, actualizar y borrar (CRUD - *Create, Retrieve, Update and Delete*) habituales, desde la creación de nuevos registros a la actualización de los antiguos. Crea de forma manual y automáticamente el modelo de base de datos a implementar. Soporta varios motores de base de datos como MySQL y PostgreSQL (Pacheco, 2011).

Características principales de Doctrine (Guardado, 2010):

- Generación automática del modelo: Cuando se trabaja con ORM, se necesita crear el conjunto de clases que representan el modelo de la aplicación, luego estas clases serán vinculadas al esquema de la base de datos de forma automática con un motor ORM. Doctrine permite generar de forma automática el modelo de clases basándose en el modelo relacional de tablas. Es decir, si se tiene una tabla llamada usuarios, se autogenerará una clase llamada Usuarios cuyas propiedades son las columnas de dicha tabla.
- Posibilidad de trabajar con anotaciones: mecanismo muy utilizado en el lenguaje de programación Java, las aplicaciones Symfony2 pueden hacer uso de estas gracias a una librería desarrollada por el proyecto Doctrine 2.

1.4. Mapeo Objeto-Relacional

El proceso ORM consiste en crear una capa de abstracción intermedia entre el SGBD y la aplicación. Este permite crear un acceso a datos independiente y robusto de un SGBD específico. Constituye una técnica de programación muy utilizada ya que permite mapear o emparejar entidades: por un lado, una base de datos relacional y por otro lado objetos (Saavedra Quevedo , y otros, 2011).

El flujo normal del proceso de mapeo para la creación de un nuevo módulo en el proyecto de Sistema de Gestión e Información del Proceso Electoral (SIGEL), se realiza de la forma siguiente:

1. Los diseñadores del sistema en conjunto con los especialistas elaboran el modelo de datos haciendo uso de Visual Paradigm para UML.
2. Se generan las clases con extensión (.php) a través de la consola con sus respectivos atributos y se cargan en el IDE[2] para su edición.
3. Se editan cada una de las clases obtenidas y se les incorporan las relaciones existentes entre las entidades.
4. Se genera la base de datos haciendo uso de los comandos que brinda la consola de Symfony.

A la hora de realizar las modificaciones al modelo ocurre un proceso similar al descrito anteriormente:

- El Comité de Control de Cambios del proyecto aprueba la modificación al modelo.
- Entre los diseñadores del sistema y los especialistas se elabora la modificación tratando de optimizar y minimizar el impacto del cambio.
- Se actualiza el modelo de datos.
- Se editan las entidades a modificar.
- Se editan las tablas de base de datos que representan las entidades modificadas.

Existen distintos tipos de herramientas ORM en correspondencia con el lenguaje de programación que se desee utilizar para el desarrollo de determinado software; por ejemplo, el ORM Doctrine es utilizado para el lenguaje de programación PHP. Esta es una herramienta que apoya el desarrollo de diversas aplicaciones.

Las herramientas ORM diseñan modelos de datos y generan el código correspondiente. Incluyen operaciones de inserción, actualización, selección, eliminación, además de consultas y llamadas a procedimientos almacenados. También permiten definir relaciones de uno-a-uno, uno-a-muchos, muchos-a-muchos y herencia entre objetos (Saavedra Quevedo , y otros, 2011).

1.5. Herramientas de apoyo al proceso ORM

El empleo de herramientas ORM y la realización del proceso de mapeo relacional de objetos es importante ya que se necesita de una interfaz que traduzca la lógica de los objetos a la lógica relacional, la cual está

[2] Entorno de Desarrollo Integrado, por sus siglas en inglés *Integrated Development Enviroment.*

formada por objetos que permiten acceder a los datos y que contienen en sí mismos el código necesario para hacerlo.

1.5.1. Visual Paradigm para UML 8.0

Visual Paradigm para UML (VP) es una Herramienta de Ingeniería de Software Asistida por Computadora (CASE por sus siglas en inglés Computer Aided Software Engineering,) que permite visualizar, diseñar e integrar diferentes aplicaciones. Además de soportar modelado, ofrece generación de informes, códigos de diagramas y permite la realización de la ingeniería inversa. Comprende el ciclo de vida completo del proceso de desarrollo del software y dentro de sus principales funcionalidades a partir de la versión 7.2 incorpora la generación automática de la capa de acceso a datos para entidades Doctrine 1.0 (Targetware, 2017).

Específicamente en SIGEL se utilizan, para el desarrollo de una nueva versión de la aplicación, Symfony 2.7 como marco de trabajo y Doctrine 2.0 como ORM. Las entidades Doctrine que soporta el VP son incompatibles con la versión de entidades que se utiliza en SIGEL, impidiendo construir la capa de acceso a datos de forma automática a través de la herramienta VP. Por tanto, esta herramienta no puede ser utilizada para la generación de entidades para Doctrine 2.0.

1.5.2. ORM Designer

ORM Designer es una herramienta profesional que permite la generación de entidades para Propel, Doctrine 1.0 y Doctrine 2.0, todas a partir del diseño del modelo de datos (ORM Designer, 2013).

Características de la herramienta (ORM Designer, 2013):

- Interfaz de usuario: está basada en una interfaz de usuario diseñada para la creación de modelos de datos.
- Manipulación del modelo: herramientas avanzadas de búsqueda, que son capaces de buscar a través de las relaciones, las columnas, las regiones y las tablas.
- Los archivos y serialización: se basa en un documento XML muy flexible que se utiliza para almacenar definiciones del modelo de datos y los datos de visualización.

La herramienta ofrece múltiples características que favorecen su utilización, pero presenta algunos inconvenientes para el desarrollo de aplicaciones en la universidad por ser un software privado, los cuales se mencionan a continuación (ORM Designer, 2013):

- Restricciones en el uso (marcadas por licencias).
- Imposibilidad de redistribución.
- El costo de la aplicación es elevado.
- Escasa documentación técnica para su uso.
- El soporte de la aplicación es exclusivo del propietario.

A continuación, se realiza un análisis de las herramientas antes descritas en función de algunas de las características que favorecen la solución propuesta (ver Tabla. 1). Se consideran como características principales:

- C1: Multiplataforma.
- C2: Posee licencia gratuita.
- C3: Genera entidades Doctrine 2.0.
- C4: Permite establecer relaciones entre entidades.
- C5: Soporta todos los tipos de datos de Doctrine 2.0.

Tabla. 1. Comparación entre herramientas ORM

Herramientas	Características				
	C1	**C2**	**C3**	**C4**	**C5**
Visual Paradigm para UML	X			X	
ORM Designer	X		X	X	X

1.5.3. Conclusiones del estudio de las herramientas de apoyo al proceso ORM

Las herramientas presentadas no pueden ser utilizadas debido a que no cumplen con algunos de los requisitos principales requeridos por los especialistas. No obstante, constituyen una base para el diseño e implementación de la propuesta ya que presentan algunas características, como las mostradas en la tabla anterior que debe poseer la herramienta a desarrollar. Además de funcionalidades básicas como: gestionar

diagrama, gestionar entidades, gestionar atributos, gestionar relaciones. En el caso de VP no genera entidades Doctrine 2.0, ni soporta todos los tipos de datos de Doctrine 2.0 a diferencia del ORM Designer. Pero ambas necesitan licencia lo cual dificulta su acceso. Por tanto, se decide desarrollar una aplicación basada en software libre que permitirá facilitar el trabajo de los especialistas, en la creación y mantenimiento del mapeo de las clases de acceso a datos de Doctrine 2.0.

El estudio de las herramientas de apoyo al proceso ORM, permitió identificar un conjunto de funcionalidades básicas de diseñar un modelo de datos, características como las principales propiedades de entidades y sus relaciones, así como componentes esenciales que debe poseer la herramienta a desarrollar. Estas aplicaciones presentan aspectos particulares que, aunque resuelven problemas para los diferentes usuarios que las utilizan, no lo realizan directamente para los especialistas. Ante la necesidad de utilizar una serie de procedimientos, técnicas, herramientas y soporte documental a la hora de desarrollar un producto de software, surgen las metodologías de desarrollo.

1.6. Metodologías de desarrollo

La metodología de desarrollo de software es un enfoque estructurado para el desarrollo de software que incluye modelos de sistemas, notaciones, reglas, sugerencias de diseño y guías de procesos (Sommerville, Ian, 2006). En los últimos años el uso de estas ha incidido positivamente en la calidad del software. Las metodologías de desarrollo de software se dividen en dos vertientes: tradicionales o pesadas y metodologías ágiles.

Las tradicionales llevan una documentación exhaustiva de todo el proyecto. Se utilizan en grandes equipos de desarrollo y necesitan generar una gran documentación para mantener controlado el proceso de desarrollo y gestionar la comunicación entre los departamentos en los que puede dividirse el equipo. Las metodologías ágiles son flexibles ante requisitos cambiantes y son utilizadas en equipos pequeños de desarrollo.

1.6.1. Metodología de desarrollo a emplear

La metodología de desarrollo a emplearse en los proyectos productivos de la Universidad de las Ciencias Informáticas (UCI) es una variación de la metodología “Proceso Unificado Ágil” (AUP por sus siglas en inglés *Agile Unified Process*) en unión con el modelo CMMI-DEV v1.3.

La variante de AUP utilizada en la UCI es una metodología ágil que tiene entre sus objetivos aumentar la calidad del software que se produce, de ahí la importancia de aplicar el Modelo CMMI-DEV v1.3. El cual

constituye una guía para aplicar las mejores prácticas en una entidad desarrolladora. Estas prácticas se centran en el desarrollo de productos y servicios de calidad (Rodríguez Sánchez, 2014).

Esta metodología cuenta con tres **fases** de desarrollo (Rodríguez Sánchez, 2014):

1. **Inicio:** Durante el inicio del proyecto se llevan a cabo las actividades relacionadas con la planeación del proyecto. En esta fase se realiza un estudio inicial de la organización cliente que permite obtener información fundamental acerca del alcance del proyecto, realizar estimaciones de tiempo, esfuerzo, costo y decidir si se ejecuta o no el proyecto.

2. **Ejecución:** En esta fase se ejecutan las actividades requeridas para desarrollar el software, incluyendo el ajuste de los planes del proyecto considerando los requisitos y la arquitectura. Durante el desarrollo se modela el negocio, se obtienen los requisitos, se elaboran la arquitectura y el diseño, se implementa y se libera el producto. Durante esta fase el producto es transferido al ambiente de los usuarios finales o entregado al cliente. Además, en la transición se capacita a los usuarios finales sobre la utilización del software.

3. **Cierre:** En esta fase se analizan tanto los resultados del proyecto como su ejecución y se realizan las actividades formales de cierre del proyecto.

La variante de AUP definida por la UCI propone siete disciplinas que se describen a continuación:

1. **Modelado de negocio:** Es la disciplina destinada a comprender los procesos de negocio de una organización. Se comprende cómo funciona el negocio que se desea informatizar para tener garantías de que el software desarrollado va a cumplir su propósito.

2. **Requisitos:** El esfuerzo principal en la disciplina Requisitos es desarrollar un modelo del sistema que se va a construir. Esta disciplina comprende la administración y gestión de los requisitos funcionales y no funcionales del producto. Existen tres formas de encapsular los requisitos [Casos de Uso del Sistema (CUS), Historias de usuario (HU) y Descripción de requisitos por proceso (DRP)], agrupados en cuatro escenarios condicionados por el Modelado de negocio.

3. **Análisis y diseño:** En esta disciplina, si se considera necesario, los requisitos pueden ser refinados y estructurados para conseguir una comprensión más precisa de estos, y una descripción que sea fácil de mantener y ayude a la estructuración del sistema (incluyendo su arquitectura). Además, en esta disciplina se modela el sistema y su forma (incluida su arquitectura) para que soporte todos los

requisitos, incluyendo los requisitos no funcionales. Los modelos desarrollados son más formales y específicos que el de análisis.

4. **Implementación:** En la implementación, a partir de los resultados del Análisis y Diseño se construye el sistema.
5. **Pruebas internas:** En esta disciplina se verifica el resultado de la implementación probando cada construcción, incluyendo tanto las construcciones internas como intermedias, así como las versiones finales a ser liberadas. Se deben desarrollar artefactos de prueba como: diseños de casos de prueba, listas de chequeo y, componentes de prueba ejecutables para automatizar las pruebas.
6. **Pruebas de liberación:** Pruebas diseñadas y ejecutadas por una entidad certificadora de la calidad externa, a todos los entregables de los proyectos antes de ser entregados al cliente para su aceptación.
7. **Pruebas de Aceptación:** Es la prueba final antes del despliegue del sistema. Su objetivo es verificar que el software está listo y que puede ser usado por usuarios finales para ejecutar aquellas funciones y tareas para las cuales el software fue construido.

A partir de que el Modelado de negocio propone tres variantes a utilizar en los proyectos (CUN[3], DPN[4] o MC[5]) y existen tres formas de encapsular los requisitos (CUS, HU, DRP), surgen cuatro escenarios para modelar el sistema en los proyectos:

1. Proyectos que modelen el negocio con CUN solo pueden modelar el sistema con CUS.
2. Proyectos que modelen el negocio con MC solo pueden modelar el sistema con CUS.
3. Proyectos que modelen el negocio con DPN solo pueden modelar el sistema con DRP.
4. Proyectos que no modelen negocio solo pueden modelar el sistema con HU.

En el caso del desarrollo del Generador visual de entidades para Doctrine 2.0 se selecciona el escenario número dos que aplica a los proyectos que hayan evaluado el negocio a informatizar y como resultado

[3] Casos de Uso del Negocio
[4] Descripción de Proceso de Negocio
[5] Modelo Conceptual

obtengan que no es necesario incluir las responsabilidades de las personas que ejecutan las actividades, de esta forma modelarían exclusivamente los conceptos fundamentales del negocio. Se recomienda este escenario para proyectos donde el objetivo primario es la gestión y presentación de información.

1.7. Herramientas y lenguajes de programación

Para el desarrollo de software es fundamental el empleo de un conjunto de herramientas y lenguajes que apoyen las fases de la metodología seleccionada. En los siguientes subepígrafes se mencionan las características de las herramientas y lenguajes que serán utilizados en la construcción del Generador visual de entidades para Doctrine 2.0.

1.7.1. Lenguaje Unificado de Modelado (UML) 2.0

Un Lenguaje Unificado de Modelado es una notación (esquemática en su mayor parte) con que se construyen sistemas por medio de conceptos orientados a objetos (Larman, 2003). Este lenguaje empleado en el Proceso Unificado de Desarrollo de Software da la posibilidad de obtener una abstracción del sistema junto con sus componentes a través de la confección de diagramas y modelos necesarios para la construcción del producto en cuestión.

No hay ninguna línea entre los diferentes conceptos y las construcciones en UML, pero, por conveniencia, se dividen en varias vistas. Una vista es simplemente un subconjunto de UML que modela construcciones que representan un aspecto de un sistema. Una o dos clases de diagramas proporcionan una notación visual para los conceptos de cada vista (Jacobson, y otros, 2000).

Se emplea el lenguaje UML, con el fin de realizar los diagramas de: casos de uso, clases, componentes y, además, se diseñaron el modelo de dominio y el modelo de diseño.

1.7.2. Lenguaje de programación

Un “lenguaje de programación” es un lenguaje diseñado para describir el conjunto de acciones consecutivas que un equipo debe ejecutar. (Lenguajes de programación, 2016)

Java 8.0

El lenguaje para la programación en Java, es un lenguaje orientado a objeto, de una plataforma independiente, desarrollado por la compañía Sun Microsystems (la cual fue adquirida por la compañía Oracle). Java permite la modularidad, por tanto, se pueden crear rutinas individuales que sean usadas por

más de una aplicación. La programación en Java, permite tanto el desarrollo de aplicaciones bajo el esquema de Cliente-Servidor, como de aplicaciones distribuidas, lo que lo hace capaz de conectar dos o más computadoras u ordenadores, ejecutando tareas simultáneamente, y de esta forma logra distribuir el trabajo a realizar. (Lenguajes de programación, 2016)

Se decide como lenguaje de programación Java en su versión 8 pues además de ser propuesto por el proyecto de SIGEL y ser multiplataforma, esta versión tiene incluida nuevas características a diferencia de versiones anteriores. Las ventajas de Java en su versión 8 son (javaHispano, 2014):

- Las expresiones lambda[6].
- Provoca importantes incrementos del rendimiento de aplicaciones mediante la reducción de código repetitivo, la mejora de las colecciones y anotaciones y la simplificación de modelos de programación.

1.7.3. Entorno de desarrollo integrado: NetBeans IDE 8.0

Un Entorno de Desarrollo Integrado (IDE) es una aplicación visual que sirve para la construcción de aplicaciones a partir de componentes (Ramos Salavert, y otros, 2000). Entre sus principales componentes se encuentran un editor de código, un compilador, un depurador y un constructor de interfaz gráfica.

NetBeans IDE 8.0 es gratuito y de código abierto. Permite el uso de un amplio rango de tecnologías de desarrollo tanto para escritorio, como aplicaciones Web o para dispositivos móviles. Da soporte a las siguientes tecnologías: Java, PHP, C/C++, HTML. Además puede instalarse en varios sistemas operativos: Windows, Linux, Mac OS. (NetBeans, 2016)

Características principales (NetBeans, 2016):

- Posee un editor de código robusto, multilenguaje, con el habitual coloreado y sugerencias de código, acceso a clases a través de enlaces en el código, control de versiones, ubicación de la clase actual, comprobaciones sintácticas, semánticas y plantillas de código.

[6] Es una función anónima. Básicamente es un método abstracto que solo está definido en una interfaz, pero puede ser implementado en cualquier clase sin heredar de la interfaz. Permite que el código sea más conciso y significativo.

- Simplifica la gestión de grandes proyectos con el uso de diferentes vistas, asistentes de ayuda y estructurando la visualización de manera ordenada, lo que ayuda en el trabajo diario.
- Contiene herramientas para depurado de errores: el depurador que incluye el IDE es bastante útil para definir puntos de ruptura en la línea de código que se desee, y monitorizar en tiempo real los valores de propiedades y variables.
- Optimización de código.

1.8. Conclusiones del capítulo

El estudio de los principales conceptos relacionados con el mapeo objeto-relacional y la descripción de las principales características que posee Doctrine permitió comprender el proceso de generación de entidades para Doctrine 2.0. Al realizar un análisis crítico de herramientas ORM como Visual Paradigm para UML 8.0 y ORM Designer, se determinó que ninguna cumple con las características necesarias para generar entidades Doctrine 2.0 en el proyecto SIGEL. No obstante, permitió identificar un conjunto de funcionalidades básicas de la herramienta a desarrollar como gestionar entidades, gestionar atributos y gestionar relaciones. El estudio de un conjunto de herramientas y metodologías de software permitió seleccionar Visual Paradigm para UML8.0 para la confección de modelos de diseño y diagramas de clases a través del UML, y la variación de AUP definida en la UCI como metodología para guiar el proceso de desarrollo de software empleando el lenguaje de programación Java en su versión 8, y NetBeans 8.0 como IDE.

Capítulo 2: Análisis y diseño del Generador visual de entidades para Doctrine 2.0

Está basado en las tareas realizadas por las disciplinas de Análisis y Diseño definidas por la variación de la metodología AUP definida en la UCI. Se incluye el modelo de dominio el cual refleja los conceptos relacionados en el dominio del problema. Son identificados los requisitos funcionales y no funcionales que se deben tener en cuenta para la implementación de la solución. Se define la arquitectura para modelar su estructura, se realizan diagramas de casos de uso del sistema y clases del diseño aplicando los patrones correspondientes.

2.1. Modelo de dominio

Este modelo es una visualización de los conceptos del dominio, es una representación de las clases conceptuales del mundo real (Larman, 2003). La Fig. 1 muestra el modelo de dominio para la generación de entidades para Doctrine 2.0.

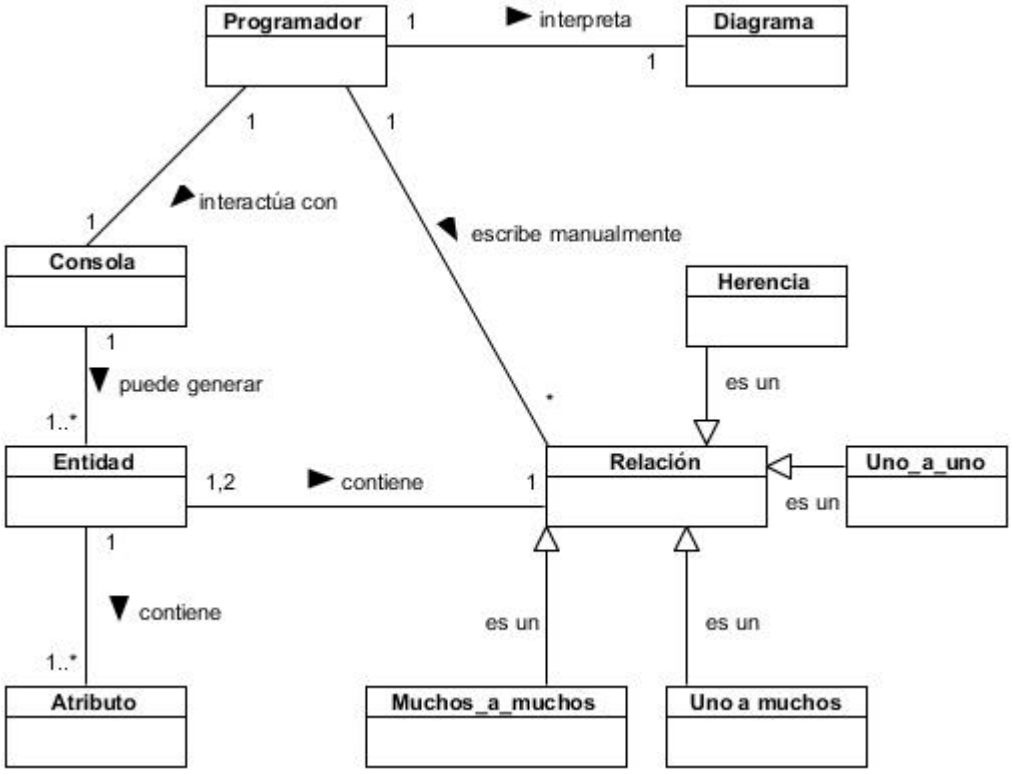


Fig. 1. Modelo de domino del Generador visual de entidades para Doctrine 2.0.

Descripción de los objetos del Modelo del Dominio del Generador visual de entidades para Doctrine 2.0:

- **Programador:** Es aquella persona que escribe, depura y mantiene el código fuente de un programa informático.
- **Consola:** es una interfaz que se utiliza para interactuar con un computador a través de código.
- **Diagrama:** Representa entidades con sus atributos y relaciones que conforman el diseño del modelo de datos.
- **Entidad:** Conceptos que poseen propiedades y entre los cuales se establecen relaciones.
- **Atributo:** Propiedad o característica de una entidad.
- **Relación:** Conexión que se puede establecer entre dos entidades.
- **Uno a uno**: Relación entre dos entidades donde una instancia de la primera puede estar asociada con una única instancia de la segunda y esta puede estar asociada con una única instancia de la primera.
- **Uno a muchos**: Relación entre dos entidades donde una instancia de la primera puede estar asociada con muchas instancias de la segunda y una instancia de la segunda puede estar asociada con una única instancia de la primera.
- **Muchos a muchos**: Relación entre dos entidades donde una instancia de la primera puede estar asociada con muchas instancias de la segunda y una instancia de la segunda puede estar asociada con muchas instancias de la primera.
- **Herencia**: Relación entre dos entidades que comparten características en común que son agrupadas en la primera entidad llamada padre, mientras que la otra nombrada hija contiene las características diferentes y tiene acceso a las del padre.

2.2. Especificación de los requisitos del sistema

Los requisitos para un sistema son la descripción de los servicios proporcionados por el sistema y sus restricciones operativas (Sommerville, Ian, 2006). Estos requisitos permiten confirmar la viabilidad de la

solución, gestionándolos de forma correcta se pueden transformar en un software funcional ya que determinan qué hará este y definen las restricciones de su operación e implementación.

2.2.1. Requisitos funcionales

Los requisitos funcionales son declaraciones de las funcionalidades que debe proporcionar el sistema, de la manera en que este debe reaccionar a entradas particulares y de cómo se debe comportar en situaciones particulares (Sommerville, Ian, 2006). A continuación, se muestran los requisitos funcionales identificados mediante la técnica entrevista no estructurada:

RF 1: Adicionar entidad: adiciona una nueva entidad al modelo de datos.

RF 2: Modificar entidad: modifica una entidad disponible en el modelo de datos.

RF 3: Eliminar entidad: elimina la entidad seleccionada del modelo de datos.

RF 4: Adicionar paquete de entidades: adiciona un nuevo paquete de entidades al modelo de datos.

RF 5: Modificar paquete de entidades: modifica un paquete de entidades disponible en el modelo de datos.

RF 6: Eliminar paquete de entidades: elimina un paquete de entidades en el modelo de datos.

RF 7: Adicionar atributo de la entidad: adiciona atributos a la entidad seleccionada creada por el especialista.

RF 8: Modificar atributo de la entidad: modifica un atributo disponible en la entidad.

RF 9: Eliminar atributo de la entidad: elimina un atributo de la entidad.

RF 10: Listar atributos de la entidad: lista los atributos pertenecientes a la entidad seleccionada.

RF 11: Adicionar relación uno a uno entre entidades: permite adicionar una relación de uno a uno asociada a dos entidades.

RF 12: Modificar relación uno a uno entre entidades: modifica la relación de uno a uno existente entre dos entidades.

RF 13: Eliminar relación uno a uno entre entidades: elimina la relación de uno a uno entre dos entidades.

RF 14: Adicionar relación uno a muchos entre entidades: permite adicionar una relación uno a muchos entre dos entidades.

RF 15: Modificar relación uno a muchos entre entidades: modifica la relación de uno a muchos existente entre dos entidades.

RF 16: Eliminar relación uno a muchos entre entidades: elimina la relación de uno a muchos entre dos entidades.

RF 17: Adicionar relación muchos a muchos entre entidades: permite adicionar una relación de muchos a muchos entre dos entidades.

RF 18: Modificar relación muchos a muchos entre entidades: modifica la relación de muchos a muchos existente entre dos entidades.

RF 19: Eliminar relación muchos a muchos entre entidades: elimina la relación de muchos a muchos entre dos entidades.

RF 20: Adicionar relación de herencia entre entidades: permite adicionar una relación de herencia entre dos entidades.

RF 21: Modificar relación de herencia entre entidades: modifica la relación de herencia existente entre dos o más entidades.

RF 22: Eliminar relación de herencia entre entidades: elimina la relación de herencia entre dos entidades.

RF 23: Adicionar nuevo diagrama: crea un nuevo modelo de datos.

RF 24: Abrir diagrama: permite crear un modelo de datos a partir de un archivo (dentro de un directorio) que tenga la extensión (.orm).

RF 25: Guardar diagrama: permite guardar el modelo de datos en un archivo con la extensión propia del programa (.orm).

RF 26: Exportar diagrama a ORM: permite exportar un modelo de datos como entidades php con el mapeo relacional de objetos del marco de trabajo Doctrine 2.0.

RF 27: Exportar diagrama como imagen: permite exportar una imagen del modelo de datos diseñado.

RF 28: Visualizar árbol de objetos: permite visualizar un árbol con la relación entre paquetes, entidades y atributos.

RF 29: Visualizar diagrama: permite actualizar la representación visual del diagrama cada vez que se realiza algún cambio en el modelo de datos.

2.2.2. Requisitos no funcionales

Los requisitos no funcionales (RNF) son restricciones de los servicios o funciones ofrecidos por el sistema. Incluyen restricciones de tiempo, sobre el proceso de desarrollo y estándares. A menudo se aplican al sistema en su totalidad (Sommerville, Ian, 2006).

Durante la presente investigación se definen 14 RNF que cumplen con las características establecidas en el artefacto especificación de requisitos de software, estas características son: apariencia o interfaz externa, usabilidad, portabilidad, requisitos de software, diseño e implementación y requisitos de hardware. En el artefacto especificación de requisitos de software del Generador visual de entidades para Doctrine 2.0 se definen todos los RNF. A continuación, se muestran cada uno de estos.

Requisitos de apariencia o interfaz externa:

RNF 1: Prevalecen los colores azul y blanco.

RNF 2: El sistema será en idioma inglés.

Requisitos de usabilidad:

RNF 3: El usuario debe poseer conocimientos de diseño de clases o de diseño de base de datos.

RNF 4: El sistema posee un acceso rápido y fácil ya que presenta como máximo tres niveles para acceder a una funcionalidad.

RNF 5: El sistema debe contar con un manual de usuario estructurado adecuadamente que describa su funcionamiento y uso.

Requisitos de portabilidad:

RNF 6: La herramienta desarrollada deberá ser multiplataforma teniendo un buen funcionamiento tanto en Linux como en Windows.

Requisitos de software:

RNF 7: Se utilizará la máquina virtual de Java en su versión 8.

RNF 8: Sistema operativo deberá ser Microsoft Windows 7 o superior o cualquier distribución de Linux que soporte la máquina virtual de Java en su versión 8.

Requisitos del diseño y de implementación:

RNF 9: Constituye una aplicación de escritorio.

RNF 10: El diseño está basado en los principios de programación orientada a objeto.

RNF 11: El sistema se implementará usando NetBeans 8.0 IDE.

RNF 12: El lenguaje de programación a utilizar será Java en su versión 8.

Requisitos de hardware

RNF 13: Como mínimo se recomienda de memoria RAM 512 MB.

RNF 14: Se necesita como mínimo un procesador Dual-Core.

2.3. Validación de los requisitos funcionales

Una vez concluido el levantamiento de los requisitos del sistema para el Generador visual de entidades para Doctrine 2.0 se aplicó la técnica prototipado para garantizar que fueran correctos y que cumplieran con las necesidades del cliente.

Prototipo de la interfaz de usuario (IU): se diseñaron las IU de cada requisito del sistema y fueron mostradas al cliente para conocer su criterio. Luego de ser revisados, se les efectuaron algunos cambios a los prototipos de IU para ajustarlos a las necesidades del cliente. Con esta técnica se comprobó que el sistema cubre las necesidades planteadas por el cliente, viendo reflejadas cada una de las funcionalidades en los prototipos. (Ver Fig. 2).

Inicialmente cuando el usuario abre la aplicación se muestra en la pantalla una interfaz (Ver Fig. 2) donde aparecen todas las posibles funcionalidades de la misma. También se ve la existencia de un paquete que se crea por defecto cada vez que se abre la aplicación. En la región izquierda aparece un árbol de objetos el cual se va actualizando a medida que se modifica el modelo de datos.

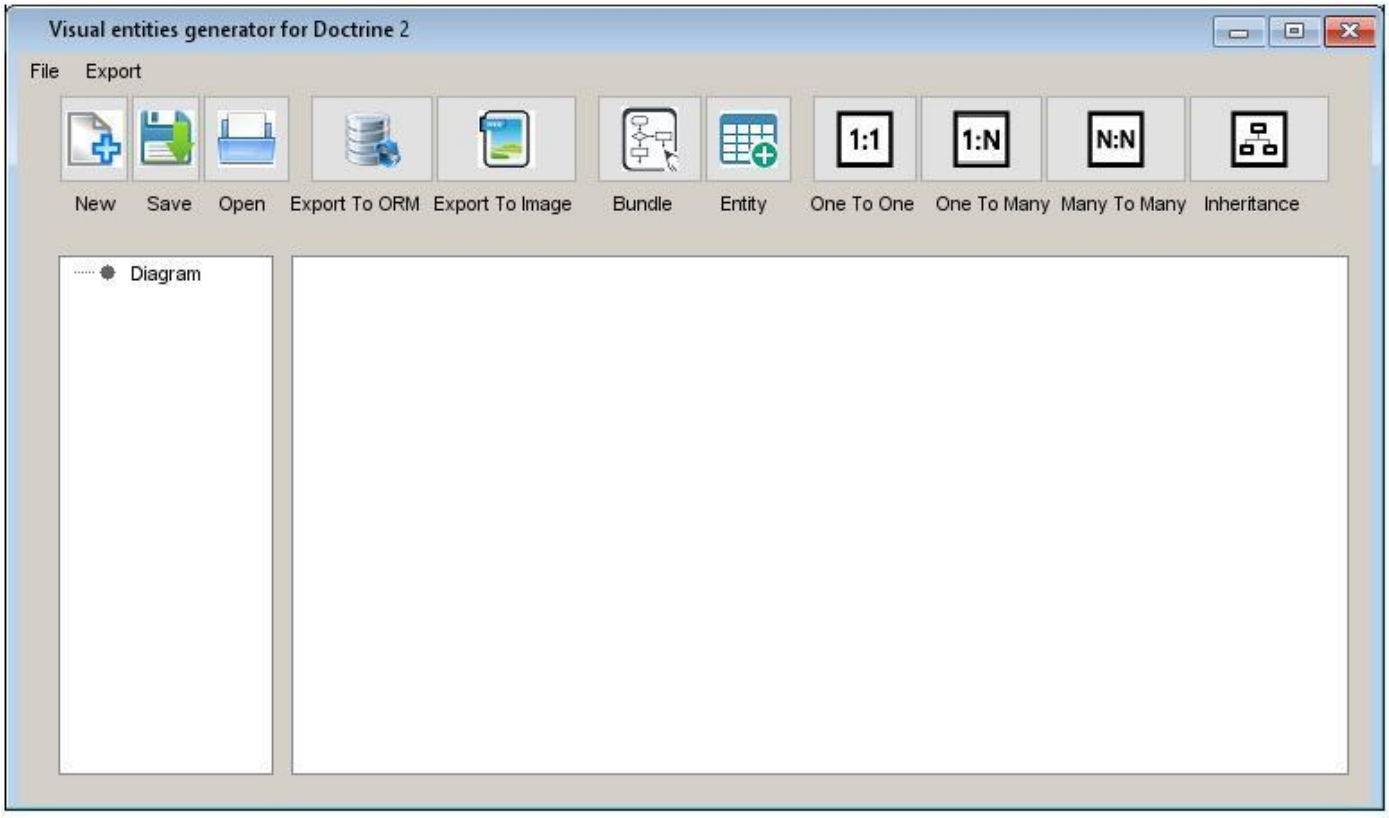


Fig. 2. Prototipo de la pantalla principal del Generador visual de entidades para Doctrine 2.0.

Entre las diversas opciones que brinda la herramienta se encuentra “Exportar diagrama a ORM” funcionalidad la cual se encarga de convertir en código PHP los elementos del diagrama. (Ver Fig. 3)

Fig. 3. Exportar diagrama a ORM en el Generador visual de entidades para Doctrine 2.0.

2.4. Modelo de casos de uso del sistema

El modelo de casos de uso del sistema permite que los desarrolladores del software y los clientes lleguen a un acuerdo sobre los requisitos, es decir, sobre las condiciones y posibilidades que debe cumplir el sistema. Es un modelo que contiene actores, casos de uso y sus relaciones. Además, describe lo que hace el sistema para cada tipo de usuario.

2.4.1. Diagrama de casos de uso del sistema

La vista de los casos de uso modela la funcionalidad del sistema según lo perciben los usuarios externos, llamados actores. Un caso de uso es una unidad coherente de funcionalidad, expresada como transacción entre los actores y el sistema. El propósito de la vista de casos de uso es enumerar a los actores y los casos de uso, y demostrar qué actores participan en cada caso de uso (Jacobson, y otros, 2000). El diagrama de casos de uso refleja cómo los actores interactúan con los casos de uso. (Ver Fig. 4)

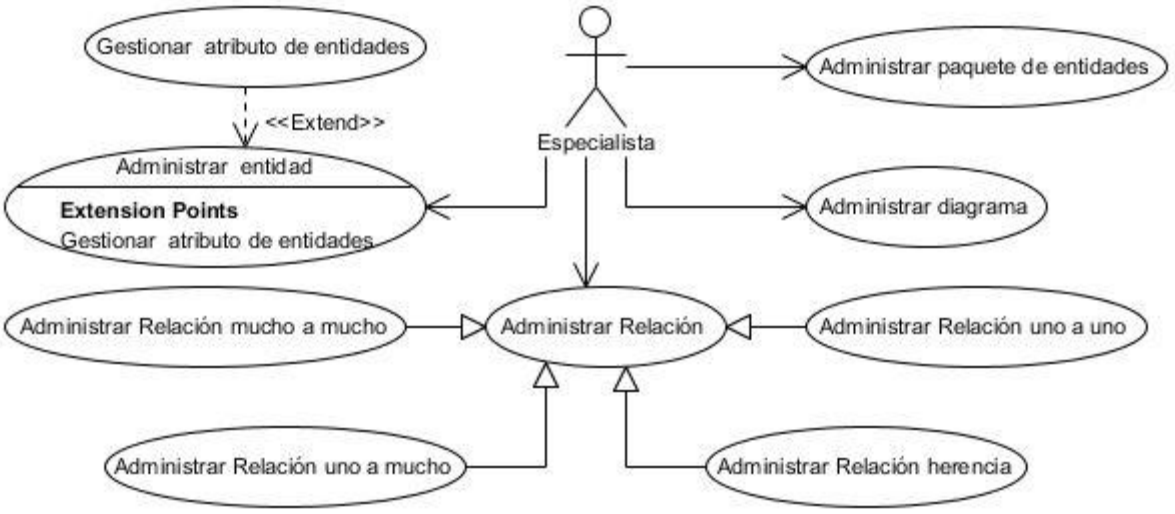


Fig. 4. Exportar diagrama a ORM en el Generador visual de entidades para Doctrine 2.0.

2.4.2. Patrones de caso de uso del sistema utilizados

Los patrones de casos de uso ofrecen una solución a problemas comunes en el modelado de casos de uso. El uso de estas técnicas permite que el modelo sea mantenible, reusable y entendible.

EL patrón Relación de Extensión relaciona a un caso de uso de extensión con un caso de uso base, que especifica cómo el comportamiento definido por el caso de uso de extensión puede insertarse dentro del comportamiento definido por el caso de uso base. Este comportamiento es condicional u opcional. En la Fig. 5 se muestra la evidencia de este patrón en el diagrama de casos de uso del sistema.

Fig. 5. Patrón de casos de uso Relación de Extensión.

El patrón Generalización – Especificación es un patrón de concordancia que contiene casos de uso del mismo tipo. Estos son modelados como una especialización de casos de uso de tipo uso común. En este caso, todas las acciones del caso de uso padre son heredadas por los casos de uso hijos, donde otras acciones pueden ser adicionadas o acciones heredadas pueden ser especializadas. (Ver Fig. 6)

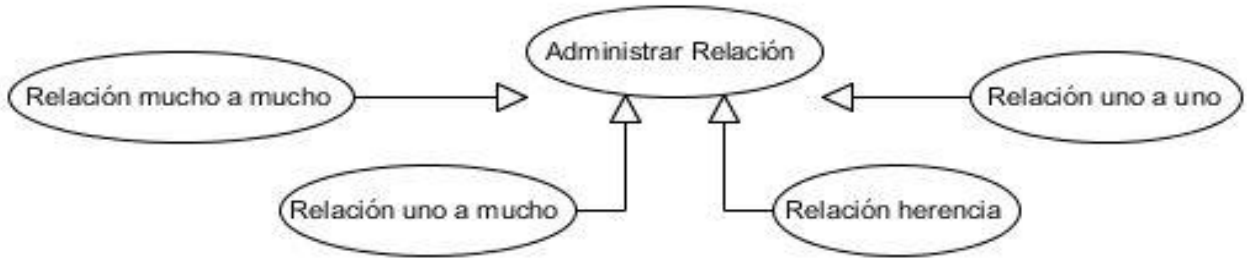


Fig. 6. Patrón de casos de uso Generalización-Especificación.

2.4.3. Descripción textual de los casos de uso del sistema

Las descripciones de casos de uso se utilizan para especificar requisitos con el objetivo de documentar la información relativa a los requisitos funcionales y no funcionales de un sistema de forma detallada. Dichas descripciones se clasifican en formales, semiformales e informales. Existen técnicas para estas descripciones, tales como:

- Documento de Especificación de Requisitos
- Descripciones textuales de casos de uso

- Historia de usuarios

A continuación, se describen formalmente el caso de uso “Administrar diagrama” aplicando la técnica Descripciones textuales de casos de uso.

Tabla. 2. Descripción de casos de uso Administrar diagrama

Objetivo	Visualizar árbol de objetos, crear, abrir, guardar, exportar y visualizar diagrama.	
Actores	Especialista: (Inicia).	
Resumen	Administrar diagrama incluye las opciones de adicionar, abrir, guardar, exportar y visualizar diagrama además de visualizar árbol de objetos.	
Complejidad	Alta	
Prioridad	Alta	
Precondiciones	Se debe haber abierto la aplicación.	
Poscondiciones	No procede	
Flujo de eventos		
Flujo básico: Administrar diagrama		
	Actor	**Sistema**
1.	Abre la aplicación.	
2.		Muestra un diagrama que incluye un paquete por defecto. Permite hacer diferentes acciones: - Crear un nuevo diagrama. Ver sección 1: Crear nuevo diagrama. - Abrir diagrama. Ver sección 2: Abrir diagrama. - Guardar diagrama. Ver sección 3: Guardar diagrama. - Exportar diagrama como imagen. Ver sección 4: Exportar diagrama como imagen. - Exportar diagrama a ORM: Ver sección 5: Exportar diagrama a ORM. - Visualizar árbol de objetos. Ver sección 6:

		Visualizar árbol de objetos. - Visualizar diagrama. Ver sección 7: Visualizar diagrama.
3.		Finaliza el caso de uso.
Sección 1: "Crear nuevo diagrama"		
Flujo básico "Crear nuevo diagrama"		
	Actor	**Sistema**
1.	Selecciona el botón "New".	
2.		El sistema verifica si el diagrama actual está guardado.
3.		Se crea un nuevo diagrama.
Flujos alternos		
Nº 2: "Diagrama actual sin guardar"		
	Actor	**Sistema**
2.1.		Se ejecuta la sección 3: Guardar diagrama.
Sección 2: "Abrir diagrama"		
Flujo básico "Abrir diagrama"		
	Actor	**Sistema**
1.	Selecciona el botón "Open".	
2.		El sistema verifica si el diagrama actual está guardado.
3.		Muestra una ventana con directorios y/o archivos con extensión (.orm) para seleccionar el fichero que se desea abrir.
4.	Selecciona el fichero deseado.	
5.	Selecciona el botón "Open".	
6.		Visualiza el diagrama seleccionado.
Flujos alternos		
Nº 2: "Diagrama actual sin guardar"		
	Actor	**Sistema**

2.1.		Se ejecuta la sección 3: Guardar diagrama.
Nº 5: "Se cancela la acción"		
	Actor	**Sistema**
5.1.	Selecciona el botón "Cancel".	
5.2		No se visualiza el diagrama seleccionado.
Sección 3: "Guardar diagrama"		
Flujo básico "Guardar diagrama"		
	Actor	**Sistema**
1.	Selecciona el botón "Save".	
2.		Muestra una ventana con directorios para seleccionar en cuál se desea guardar el diagrama.
3.	Selecciona el directorio deseado.	
4.	El usuario introduce el nombre del fichero.	
5.	Selecciona la opción "Save".	
6.		Se crea un archivo con el nombre introducido por el usuario y extensión (.orm) donde se guardan los cambios realizados al diagrama.
Flujos alternos		
Nº 5: "Se cancela la acción"		
	Actor	**Sistema**
5.1.	Selecciona el botón "Cancel".	
5.2.		No se guarda el archivo deseado.
Sección 4: "Exportar diagrama como imagen"		
Flujo básico: "Exportar diagrama como imagen"		
	Actor	**Sistema**
1.	Selecciona la opción "Export To Image".	

2.		Muestra una ventana con directorios para seleccionar en cuál se desea guardar la imagen.
3.	Selecciona el directorio deseado.	
4.	Introduce el nombre deseado para la imagen.	
5.	Selecciona la opción “Save”.	
		Guarda una imagen con extensión (.jpeg) en el directorio seleccionado.
Flujos alternos		
Nº 5: “Se cancela la acción”		
	Actor	**Sistema**
5.1.	Selecciona el botón “Cancel”.	
5.2.		No se guarda la imagen.
Sección 5: “Exportar diagrama a ORM”		
Flujo básico: “Exportar diagrama a ORM”		
	Actor	**Sistema**
1.	Selecciona la opción “Export To ORM”.	
2.		Muestra una ventana con directorios para seleccionar en cuál se desea guardar los ficheros.
3.	Selecciona el directorio deseado.	
4.	Selecciona la opción “Save”.	
5.		Guarda un conjunto de archivos con extensión (.php) en el directorio seleccionado. Se crea una carpeta por cada bundle del diagrama nombrada como el bundle correspondiente. Dentro de esta se crean las carpetas Entity que contiene un archivo por cada entidad perteneciente al bundle, Repository que posee un

		archivo repositorio por cada entidad y un archivo con el nombre del bundle.
Flujos alternos		
Nº 5: “Se cancela la acción”		
	Actor	**Sistema**
5.1.	Selecciona el botón “Cancel”.	
5.2.		No se guarda el archivo deseado.
Sección 6: “Visualizar árbol de objetos”		
Flujo básico: “Visualizar árbol de objetos”		
	Actor	**Sistema**
1.		Muestra los paquetes que contiene el diagrama.
2.	Desglosa la carpeta paquete.	
3.		Muestra las entidades correspondientes al paquete.
4.	Desglosa la carpeta de entidad.	
5		Muestra los atributos y relaciones correspondientes a la entidad.
Sección 7: “Visualizar diagrama”		
Flujo básico: “Visualizar diagrama”		
	Actor	**Sistema**
1.	Realiza alguna acción sobre el diagrama.	
2.		Dibuja cada uno de los elementos contenidos en el diagrama.

Relaciones	**CU incluidos**	No incluye otros casos de uso.
	CU extendidos	No extiende otros casos de uso.
Referencias	**RF23, RF24, RF25, RF26, RF27, RF28, RF29**	
Prototipo elemental de interfaz gráfica de usuario		

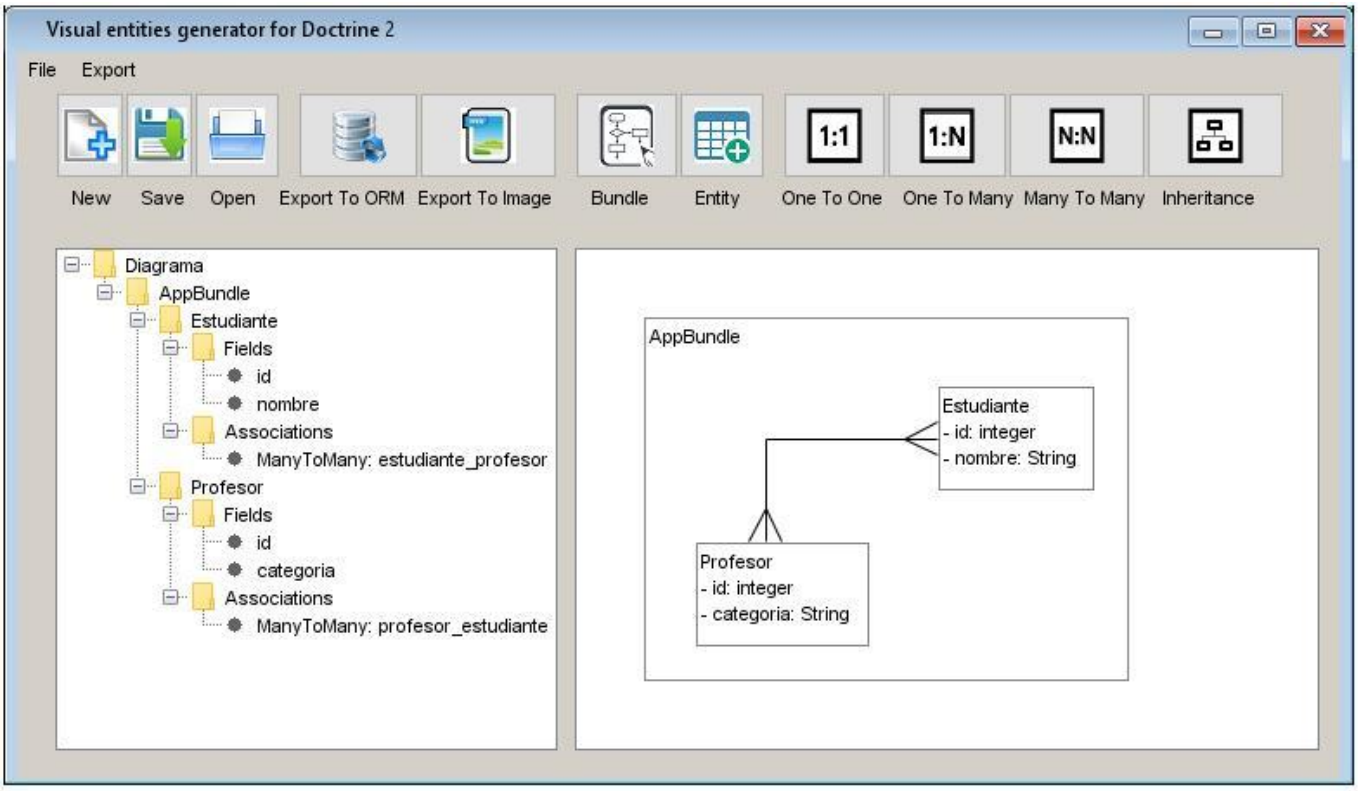


Fig. 7. Prototipo del caso de uso Administrar diagrama.

2.5. Arquitectura de software

La arquitectura de software debe modelar la estructura de un sistema y la manera en que los datos y los componentes colaboran unos con otros (Pressman, 2010). La arquitectura no es el software operativo, sino una representación que permite que un ingeniero del software analice la efectividad del diseño para cumplir con los requisitos establecidos. Permite considerar opciones arquitectónicas en una etapa en que aún resulta relativamente fácil hacer cambios al diseño; así como reducir los riesgos asociados con la construcción del software (Pressman, 2010).

La arquitectura del Generador visual de entidades para Doctrine 2.0 fue definida a partir del patrón arquitectónico Modelo – Vista – Controlador (MVC). Pues permite dividir el código de una manera más organizada; el código de la presentación se guarda en la vista, el código de manipulación de datos se guarda en el modelo y la lógica de procesamiento de las peticiones constituye el controlador.

Este patrón consta de tres componentes que se muestran en la Fig. 8.

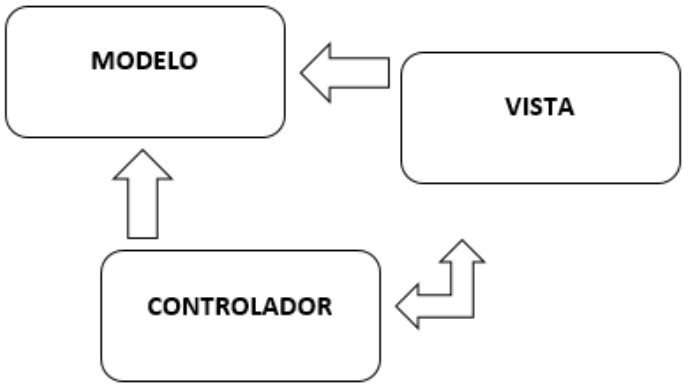


Fig. 8. Patrón arquitectónico Modelo-Vista-Controlador.

Modelo: Es el componente que se encarga de la lógica. Gestiona la información para que pueda ser utilizada por la vista y el controlador. Envía a la vista aquella parte de la información que en cada momento se le solicita para que sea mostrada (típicamente a un usuario). Las peticiones de acceso o manipulación de información llegan al modelo a través del controlador.

Controlador: Responde a eventos e invoca peticiones al modelo cuando se hace alguna solicitud sobre la información. También modifica la vista si se solicita un cambio en la forma en que se presenta el modelo (por ejemplo, cuando se agrega o modifica algún elemento del diagrama es necesario actualizarlo en la vista). Por tanto, se podría decir que el controlador hace de intermediario entre la vista y el modelo.

Vista: Presenta el modelo en un formato adecuado para interactuar con el usuario; por tanto, requiere de dicho modelo la información que debe representar como salida.

2.6. Modelo de diseño

El modelo de diseño es un refinamiento y formalización adicional del modelo de análisis[7], donde se toman en cuenta las consecuencias del ambiente de implementación. El resultado del modelo de diseño son especificaciones muy detalladas de todos los objetos, incluyendo sus operaciones y atributos. El modelo de diseño se basa en el diseño por responsabilidades. Generalmente, la información de gestión del modelo de diseño se representa en diagramas de clases (Larman, 2003).

[7] El modelo de análisis es una abstracción, o generalización, del diseño. Describe objetos del software que colaboran con responsabilidades.

2.6.1. Diagrama de clases del diseño

El diagrama de clases describe gráficamente las especificaciones de las clases de software (Larman, 2003). En otras palabras, se evidencian las clases con sus atributos y relaciones. Una clase es la descripción de un concepto del dominio de la aplicación o de la solución de la aplicación. Las clases son el centro alrededor del cual se organiza la vista de clases; otros elementos pertenecen o se unen a estas (Jacobson, y otros, 2000).

En la Fig. 9 se muestra el diagrama de clases correspondiente al caso de uso “Administrar diagrama”. Está compuesto por los paquetes gui, controller y model correspondientes a los componentes de la arquitectura vista, controlador y modelo respectivamente. Además, se representa el paquete utils que contiene clases auxiliares.

El paquete gui está compuesto por las clases DiagramPanel, CustomFileChooser, MainForm y CustomTree. Estas conforman la interfaz gráfica con la cual interactúa el usuario.

El paquete controller contiene las clases que procesarán las acciones del usuario. Ante alguna acción realizada las clases DiagramListener y DiagramController solicitan los datos al modelo y se los envía a las vistas.

El paquete model contiene las clases que almacenan la información persistente, agrupando un conjunto de funcionalidades que responden a la lógica del negocio del sistema. Este está compuesto por las clases Diagram, Bundle, Entity, Element, Attribute, Relation, OneToOne, OneToMany, ManyToMany, Inheritance y Point.

Finalmente, el paquete utils contiene aquellas clases que brindan un conjunto de funcionalidades auxiliares. En el presente caso de uso se usan las clases ExportDiagram y DiagramIntance.

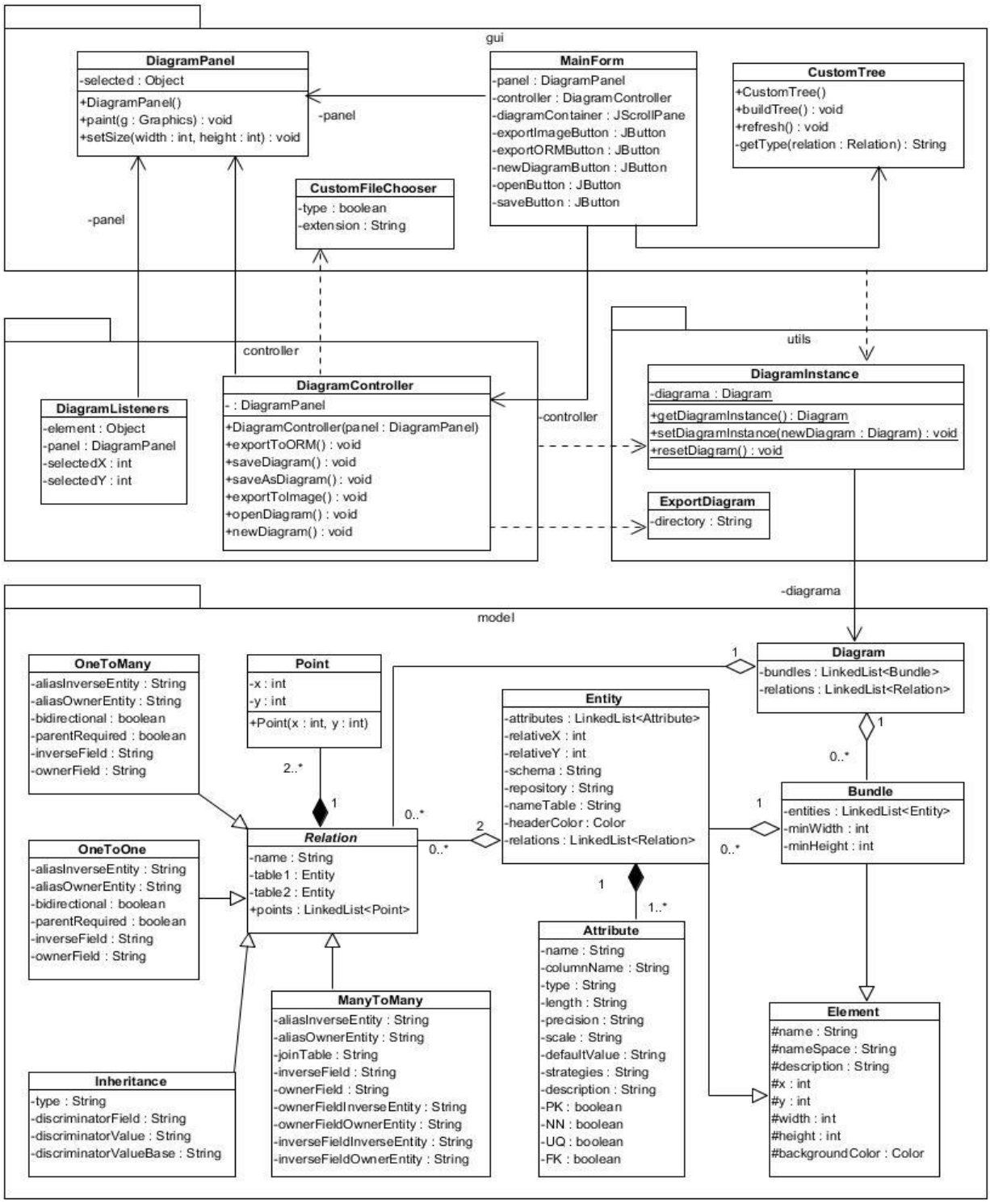


Fig. 9. Diagrama de clases del diseño del Generador visual de entidades para Doctrine 2.0.

2.7. Patrones de diseño e implementación

Los patrones de diseño proveen un esquema para refinar los subsistemas o componentes de un sistema de software, o las relaciones entre estos. Describen la estructura recurrente de los componentes en comunicación, que resuelve un problema general de diseño en un contexto determinado (Gamma, et al., 1994). Por esta razón uno de los pasos a tener en cuenta en la construcción de un software es identificar qué patrones pueden ser utilizados.

Entre los patrones de diseño más conocidos están los Patrones Generales de Asignación de Responsabilidades de Software (GRASP por sus siglas en inglés *General Responsibility Assignment Software Patterns*) (Larman, 2003). Para lograr un diseño eficaz se utilizaron algunos que se describen a continuación.

Patrón Experto: Es asignar una responsabilidad al experto en información. El experto en información es la clase que posee la información necesaria para cumplir con dicha responsabilidad (Larman, 2003). Este patrón es usado en todas las clases ya que cada una posee la información para realizar la tarea que le corresponde. Un ejemplo se evidencia en la clase *Relation*, que es responsable de relacionar dos entidades. Para establecer una relación es necesario saber las tablas que se desean relacionar y que tipo de relación se va a establecer, informaciones que posee esta clase.

Patrón Creador: Este patrón guía la asignación de responsabilidades relacionadas con la creación de objetos. El propósito fundamental de este patrón es encontrar un creador que se debe conectar con el objeto producido en cualquier evento. (Larman, 2003). En la solución se evidencia la utilización de este patrón en la clase *Relation*, donde se crean las instancias de la clase *Point*.

Patrón Alta Cohesión: La cohesión es la medida de la fuerza con la que se relacionan las responsabilidades de un elemento. Un elemento con responsabilidades altamente relacionadas, y que no hace una gran cantidad de trabajo, tiene alta cohesión (Larman, 2003). Para el diseño de las clases se tuvo en cuenta este patrón para lograr un software más robusto. Por ejemplo, las responsabilidades de la clase *DiagramPanel* están estrechamente relacionadas entre ellas, pues el objetivo de todas es la visualización del diagrama.

Patrón Controlador: Permite asignar la responsabilidad de recibir o manejar un mensaje de evento del sistema a una clase que representa una de las siguientes opciones (Larman, 2003):

- Representa el sistema global, dispositivo o subsistema (controlador de fachada). Este caso se evidencia en la clase *RelationController* que contiene las principales funcionalidades de los distintos tipos de relaciones.

- Representa un escenario de caso de uso en el que tiene lugar el evento del sistema (controlador de sesión o de caso de uso). Se evidencia en la clase *EntityController* que maneja los principales eventos relacionados con una entidad.

Otros patrones de diseño muy utilizados son los de la Banda de los Cuatro (GoF[8] por sus siglas en inglés *Gang of Four)* de los cuales en la solución se evidencian dos de ellos (Larman, 2003).

Patrón Singleton (instancia única): Este patrón garantiza que una clase sea instanciada de forma única, utilizando un mecanismo de acceso global a dicha instancia. Por ejemplo, creando una instancia estática de un objeto en su clase correspondiente. Este patrón se evidencia en la clase *DiagramInstance* la cual crea una instancia única de la case *Diagram*.

Patrón Facade (Fachada): Este patrón facilita una interfaz simple a partir de la cual se puedan acceder a una interfaz o grupo de interfaces de un subsistema. Se evidencia en la clase *Relation*, la cual se utiliza como fachada para englobar un conjunto de características en los distintos tipos de relaciones.

2.8. Conclusiones del capítulo

Se definieron 29 requisitos funcionales y 14 requisitos no funcionales sirviendo como base para el desarrollo de un software útil y funcional. Para apoyar la posterior implementación se realizaron los artefactos que propone la metodología tales como diagrama de casos de uso, descripción textual de casos de uso, arquitectura de software y diagrama de clases del diseño. La utilización de los patrones de diseño GRASP (Experto, Creador, Alta Cohesión, Controlador) y patrones de implementación GoF (*Singleton, Facade*) permitió dar respuestas a algunos problemas de diseño e implementación.

[8] En reconocimiento al importante trabajo que realizaron los cuatro autores del libro *Design Patterns: Elements of Reusable ObjectOriented Software*, se les llama afectivamente Gang of Four.

Capítulo 3: Implementación y Pruebas del Generador visual de entidades para Doctrine 2.0

Se abordarán los principales elementos para el desarrollo de las disciplinas de implementación y pruebas. Se muestra el diagrama de componentes y son descritos los estándares de codificación empleados en la implementación. Se evidencian las pruebas aplicadas a la solución con el objetivo de comprobar las funcionalidades del Generador visual de entidades para Doctrine 2.0.

3.1. Modelo de Implementación

A partir de los resultados obtenidos en la disciplina “Análisis y diseño” se realiza la implementación del sistema generándose el modelo de implementación.

La vista de implementación modela los componentes de un sistema a partir de los cuales se construye la aplicación, así como las dependencias entre los componentes, para poder determinar el impacto de un cambio propuesto. También modela la asignación de clases y de otros elementos del modelo a los componentes (Jacobson, y otros, 2000).

3.1.1. Diagrama de componentes

Un componente es un elemento funcional de un programa que incorpora la lógica del procesamiento, las estructuras internas de los datos para implementar dicha lógica, y una interfaz que permita la invocación del componente y el paso de los datos (Pressman, 2010). Un diagrama de componentes modela cómo un sistema de software se divide en componentes y representa las dependencias entre estos (Pressman, 2010).

En la Fig. 10 se muestra el Diagrama de componente del CU Administrar diagrama perteneciente al Generador visual de entidades para Doctrine 2.0.

En el paquete gui se encuentra el componente DiagramPanel.java el cual se encarga de representar los elementos del diagrama. CustomFileChooser.java permite seleccionar un directorio cuando se necesita abrir o guardar algún archivo. MainForm.java es la clase principal de la aplicación la cual contiene todo el diseño de la pantalla principal. Esta está asociada con el componente CustomTree.java que es el encargado de gestionar el árbol de objetos del diagrama.

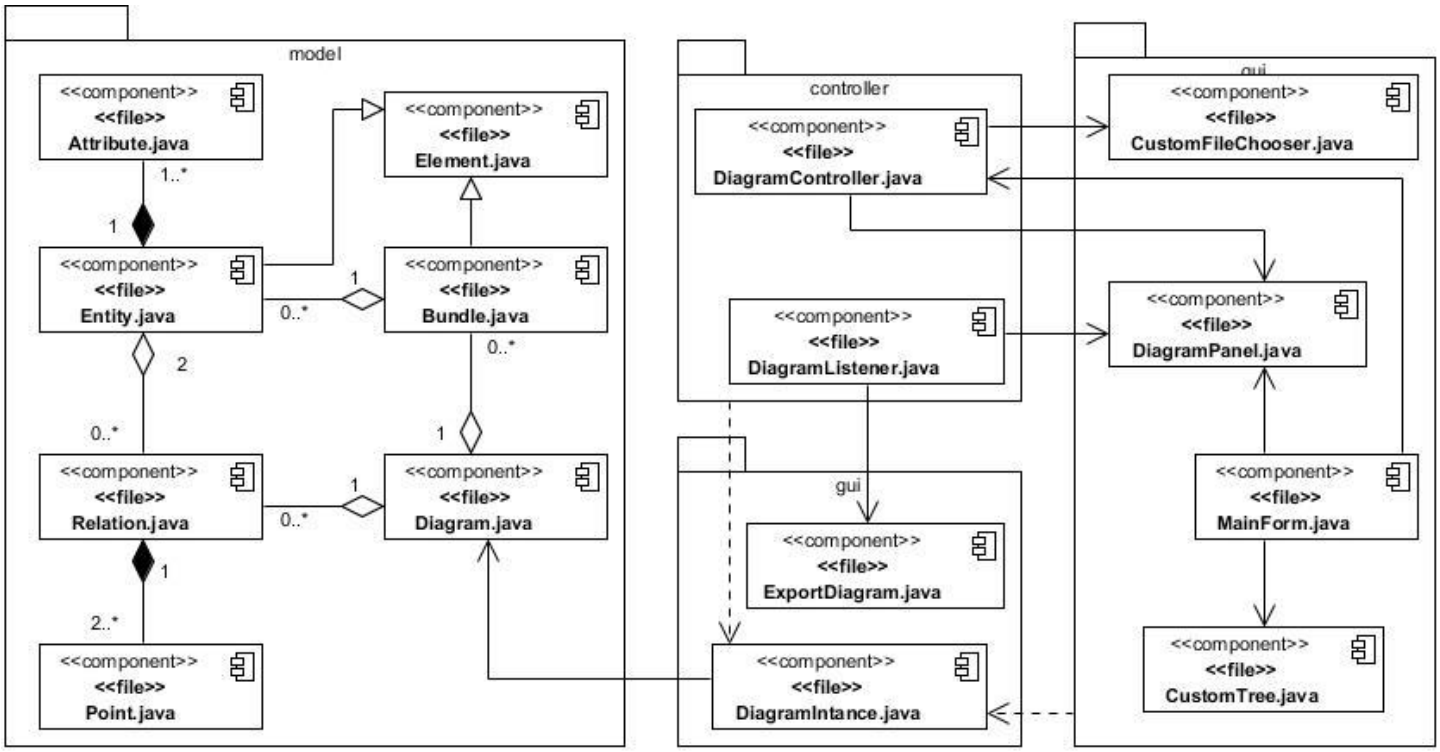


Fig. 10. Diagrama de Componentes del CU Administrar diagrama.

El paquete *controller* contiene el componente *DiagramController*.java, el cual es el encargado de procesar las principales funcionalidades del caso de uso Administrar diagrama. *DiagramListener*.java atiende todos los eventos del *mouse* que tienen lugar en el diagrama representado.

Model es el paquete que contiene todos los componentes que modelan la solución. El componente *Diagram*.java representa el diagrama en general, este contiene un listado de paquetes y relaciones entre entidades. *Bundle*.java representa a los paquetes del diagrama en el cual están contenidas las entidades. *Entity*.java está compuesto por un conjunto de atributos y contiene las relaciones asociadas a esta. Los atributos y funcionalidades comunes entre los componentes *Entity*.java y *Bundle*.java se agrupan en el componente Element.java el cual funciona como padre de estos. El componente *Relation*.java agrupa las funcionalidades de las relaciones que se establecen entre las entidades. El componente *Point*.java representa los distintos puntos que componen la representación de una relación.

Utils es un paquete que contiene los componentes auxiliares como *DiagramInstance*.java que contiene una instancia única de la clase *Diagram* y funciona como puente para enlazar los paquetes del controlador y la vista con el modelo. Además, contiene el componente *ExportDiagram*.java el cual se encarga de confeccionar los ficheros que se exportan con código ORM.

3.2. Código fuente

La legibilidad del código fuente repercute directamente en lo bien que un programador comprende un sistema de software. La mantenibilidad del código es la facilidad con que el sistema de software puede modificarse para añadirle nuevas características, modificar las ya existentes, depurar errores, o mejorar el rendimiento. Aunque la legibilidad y la mantenibilidad son el resultado de muchos factores, una faceta del desarrollo de software en la que todos los programadores influyen especialmente es en la técnica de codificación (Microsoft , 2017). El mejor método para asegurarse de que un equipo de programadores mantenga un código con calidad es establecer un estándar de codificación sobre el que se efectuarán luego revisiones de rutinas.

3.3. Estándares de codificación

Un estándar de codificación completo comprende todos los aspectos de la generación de código. Un código fuente completo debe reflejar un estilo armonioso, como si un único programador hubiera escrito todo el código de una sola vez. Cuando el proyecto de software incorpore código fuente previo, o bien cuando realice el mantenimiento de un sistema de software creado anteriormente, el estándar de codificación debería establecer cómo operar con la base de código existente (Microsoft , 2017). Un estándar de codificación o estilo de programación homogéneo en un proyecto permite que todos los integrantes de este, diferentes al autor, lo puedan entender en menos tiempo y favorece considerablemente el mantenimiento del código.

3.3.1. Convenciones de nombre

A continuación, se describen algunas convenciones usadas para los nombres de clases, variables, atributos, métodos, entre otros (ver Tabla. 3). De esta forma se logra que el código tenga una mayor consistencia, siendo la clave para obtener un código mantenible.

Tabla. 3. Uso y sintaxis de nombre

Identificador	Convención
Archivo fuente	Se escribe en Pascal Case[9]. El nombre de la clase y el fichero son el mismo.

[9] *Pascal Case*: Una palabra con la primera letra capitalizada y la primera letra de cada palabra o parte de palabra subsecuente capitalizada.

Clase o estructura	Se escribe en Pascal Case. Se agrega un sufijo apropiado cuando hereda o es subclase de otra clase. Ejemplo: public class OneToOneRelation extends Relation {…}

Método	Se escribe en Camel Case.
	Se utiliza un verbo o un par verbo-sustantivo. Ejemplo:
	public String getName () {
	return name;
	}

Atributo de clase	Se escriben en Camel Case[10]. Ejemplo: private String nameTable;
Variable	Se escribe en Camel Case. Se enumeran variables como table1, table2, etc.
Parámetro	Se escribe en Camel Case.

3.3.2. Estilo de Código

Es la manera de dar formato al código con el objetivo de crear una implementación más legible, clara, consistente y de fácil mantenimiento. A continuación, se mencionan algunas de las convenciones de estilo de código más relevantes en la solución.

- Un archivo fuente contiene una única clase.
- Las llaves de comienzo ({) se coloca a continuación del nombre del método o clase y la llave de fin (}) se coloca en una nueva línea.
- Siempre se usan las llaves de comienzo y fin en sentencias condicionales.
- Se usa el margen adicional de 4 espacios.
- Las variables se declaran independientes en líneas nuevas, no en las mismas sentencias.
- Las implementaciones de las clases tendrán el siguiente orden:

[10] Camel Case: Una palabra con la primera letra en minúscula y la primera letra de cada palabra o parte de palabra subsecuente capitalizada.

1. Atributos de la clase
2. Constructores
3. Métodos

- Las declaraciones de atributos tendrán el siguiente orden según sus modificadores de acceso:
  1. Privados
  2. Protegidos
  3. Públicos
- Los atributos relacionados se declaran en una misma línea. El resto se declara en líneas separadas.
- Los comentarios están declarados en el mismo idioma, gramaticalmente correctos y contienen los signos de puntuación apropiados.
- Para los comentarios se usa // o /*…*/.

3.4. Definición de la estrategia de prueba

En esta disciplina se verifica el resultado de la implementación probando cada construcción, incluyendo tanto las construcciones internas como intermedias, así como las versiones finales a ser liberadas (Rodríguez Sanchez, 2015).

En el proceso de prueba del Generador visual de entidades para Doctrine 2.0 se aplicarán tres iteraciones, en caso de que se mantenga la existencia de no conformidades el equipo seguirá trabajando hasta solucionarlas. Se realizarán pruebas unitarias y pruebas funcionales para comprobar el cumplimiento de los requisitos iniciales del sistema planteados por el cliente. La primera se efectuará al código a través del método de caja blanca para determinar que las funciones internas del Generador visual de entidades para Doctrine 2.0 se ejecutan de forma correcta; esta prueba se realizará con la técnica del camino básico y la complejidad ciclomática existente. La segunda se realizará por escenarios correspondientes a los casos de pruebas. El diseño de casos de prueba es una parte de las pruebas de componentes y sistemas en las que se diseñan los casos de prueba (entradas y salidas esperadas) para probar el sistema. El objetivo del proceso de diseño de casos de prueba es crear un conjunto de casos de prueba que sean efectivos

descubriendo defectos en los programas y muestren que el sistema satisface sus requerimientos (Sommerville, 2005).

3.4.1 Método de Caja Blanca

Pruebas con acceso al código fuente (datos y lógica). Se trabaja con entradas, salidas y el conocimiento interno (it-Mentor , 2006). Permite probar que las rutas independientes se ejecutan por lo menos una vez, se verifiquen los lados verdaderos y falsos de todas las decisiones lógicas y sean ejecutados todos los bucles dentro de sus límites operacionales.

Para comprobar el código del Generador visual de entidades para Doctrine 2.0 será utilizada la técnica camino básico. Su objetivo es probar cada camino de ejecución independiente en un componente o programa. El punto de partida de este tipo de pruebas es un grafo de flujo. Un camino independiente es aquel que recorre al menos una arista en el grafo de flujo el cual consiste en nodos que representan decisiones y aristas que muestran el flujo de control. Se construye reemplazando las sentencias de control del programa por diagramas equivalentes. Para encontrar el número de caminos independientes en un programa se puede calcular la complejidad ciclomática del grafo del flujo. Después de descubrir el número de caminos independientes en el código, se necesita diseñar casos de prueba para ejecutar cada uno de estos caminos. El número mínimo de casos de pruebas necesarios para probar todos los caminos del programa es igual a la complejidad ciclomática (Sommerville, 2005).

Primer paso: Realizar el grafo de flujo.

Se realizó como ejemplo el grafo de flujo del método editEntity() (ver Fig. 11) perteneciente a la clase DiagramController que se encuentra en el paquete orm.controller el cual contiene todas la clases controladoras de la solución.

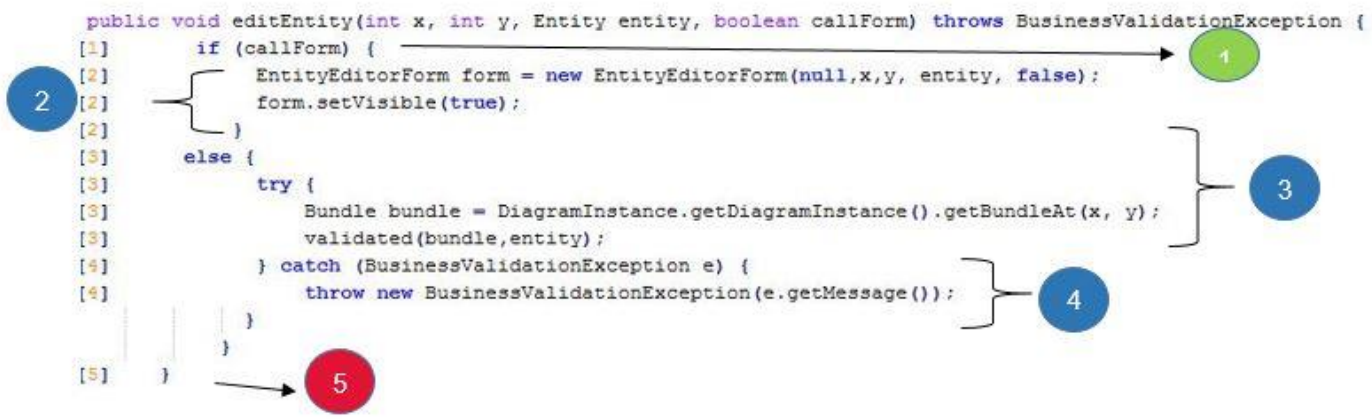


Fig. 11. Código del método editEntity().

Después de analizado el código fuente del método, se construyó el siguiente grafo de flujo (Ver Fig. 12)

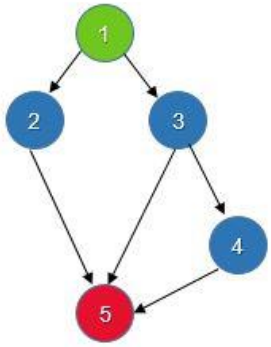


Fig. 12. Grafo de flujo del método editEntity().

Segundo paso: Calcular la complejidad ciclomática

La complejidad ciclomática es igual a la cantidad de nodos predicados[11] más uno. En el grafo de la Fig. 12 se pueden observar dos nodos predicados (el uno y el tres). Por tanto, la complejidad ciclomática es tres. Esto significa que se necesitan realizar como mínimo tres casos de prueba para probar cada uno de los caminos independientes del grafo.

[11] Nodo que representa a una condición. Por tanto, emergen de él dos o más nodos.

Tercer paso: Determinar un conjunto básico de caminos independientes

Cada camino independiente debe recorrer al menos una arista que no haya sido recorrida con anterioridad. En el grafo anterior se reflejan tres caminos:

A. 1, 2, 5
B. 1, 3, 5
C. 1, 3, 4, 5

Cuarto paso: Preparar y realizar los casos de prueba, con los caminos básicos. (Ver Tabla. 4)

Tabla. 4. Casos de prueba con camino básico del método editEntity()

Camino	Entrada	Descripción de la entrada	Respuesta esperada	Respuesta del sistema
1	true	Si la variable callForm es true	Se debe mostrar el formulario para editar la entidad.	Se muestra el formulario para la edición de la entidad.
2	false	Si la variable callForm es false y los datos son válidos	Se debe validar los datos introducidos por el usuario y guardarlos.	Se validaron los datos introducidos por el usuario y fueron guardados.
3	false	Si la variable callForm es false y los datos no son válidos	Se debe validar los datos introducidos por el usuario y no guardarlos.	Se validaron los datos introducidos por el usuario, se lanza una excepción y no se guardan los datos.

Al ejecutar los casos de pruebas diseñados para cada camino básico y comparar los resultados obtenidos con los esperados se comprobó que todas las sentencias del código se ejecutan al menos una vez.

3.4.2 Método de Caja Negra

Se refiere a las pruebas que se llevan a cabo sobre la interfaz del software, por lo que 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 una salida correcta, así como que la integridad de la información externa se

mantiene. Esta prueba examina algunos aspectos del modelo fundamentalmente del sistema sin tener mucho en cuenta la estructura interna del software.

El método de **caja negra** verifica la aplicación y sus procesos internos mediante la interacción con la interfaz gráfica de usuario GUI[12] (por sus siglas en inglés *Graphical User Interface*) y analiza los resultados (Jorge Hernan Abad Londoño, 2005).

La prueba de Caja Negra no es una alternativa a las técnicas de prueba de la Caja Blanca, sino un enfoque complementario que intenta descubrir diferentes tipos de errores a los encontrados en los métodos de la Caja Blanca.

Se ejecuta cada caso de uso, flujo de caso de uso, o función, usando datos válidos e inválidos, para verificar lo siguiente:

- Que los resultados esperados ocurran cuando se usen datos válidos.
- Que sean desplegados los mensajes apropiados de error y precaución cuando se usen datos no válidos.
- Todos los defectos que se identificaron se han tenido en cuenta.

La técnica empleada en el Generador visual de entidades para Doctrine 2.0 es **Particiones de equivalencia o dominios.** Estas particiones son conjuntos de datos donde todos los miembros deberán ser procesados de forma equivalente. Luego de haber identificado las particiones se pueden elegir casos de prueba de cada una de estas (Sommerville, 2005).

Fueron realizadas una serie de casos de prueba para determinar así que los requisitos estaban parcial o completamente satisfactorios. La evidencia de los diseños de casos de pruebas realizados al Generador visual de entidades para Doctrine 2.0 puede ser consultada en los documentos agrupados en la carpeta Prueba almacenada en el expediente de proyecto. En la Tabla. 5 es mostrado el diseño de casos de prueba realizado a la solución.

Tabla. 5. Diseño de Caso de Prueba del CU Administrar diagrama

Escenario	Descripción	Directorio	Nombre de archivo/carpeta	Archivos de tipo	Respuesta del sistema	Flujo central
		V	V	V		

[12] La prueba de interfaz de usuario verifica la interacción del usuario con el software. El objetivo es asegurar que la interfaz tiene apropiada navegación a través de las diferentes funcionalidades.

EC1. Abrir diagrama con campos correctos.	Crea un nuevo diagrama a partir de un modelo de datos existente.	Desktop	newDiagram.orm	Doctrine 2 ORM Diagram(*.orm)	Se abre un nuevo diagrama.	Se da clic en el botón Open. Se muestra un formulario para buscar el directorio donde se encuentra el archivo perteneciente al diagrama que se desea abrir. Se selecciona el botón Open una vez llenado los campos de dicho formulario. Y se abre un diagrama en pantalla.
EC2. Abrir diagrama con campos incompletos.	Intenta abrir un nuevo diagrama sin llenar campos obligatorios.	I	V	V	No permite abrir ningún diagrama.	Se da clic en el botón Open. Se muestra un formulario para buscar el directorio donde se encuentra el archivo perteneciente al diagrama que se desea abrir. Se selecciona el botón Open una vez llenado los campos de dicho formulario. Al dejar campos vacíos no se permitirá abrir un diagrama.
			newDiagram.orm	Doctrine 2 ORM Diagram(*.orm)		
		V	I	V		
		Desktop		Doctrine 2 ORM Diagram(*.orm)		
		I	I	V		
				Doctrine 2 ORM Diagram(*.orm)		
EC3. Guardar diagrama con campos correctos.	Guarda un modelo de datos en el directorio seleccionado.	V	V	V	Se guarda un nuevo diagrama.	Se da clic en el botón Save. Se muestra un formulario para obtener los datos necesarios y guardar
		Desktop	newDiagram.orm	Doctrine 2 ORM Diagram(*.orm)		

						el diagrama actual. Se selecciona el botón Save una vez llenado dicho formulario. Y el sistema guarda un archivo con extensión (.orm).
EC4. Guardar diagrama con campos incompletos.	Intenta guardar un diagrama sin llenar campos obligatorios.	I	V	V	No permite guardar el diagrama.	Se da clic en el botón Save. Se muestra un formulario para obtener los datos necesarios y guardar el diagrama actual. Se selecciona el botón Save una vez llenado dicho formulario. Al dejar campos vacíos no se permitirá guardar el diagrama.
			newDiagram.orm	Doctrine 2 ORM Diagram(*.orm)		
		V	I	V		
		Desktop		Doctrine 2 ORM Diagram(*.orm)		
		I	I	V		
				Doctrine 2 ORM Diagram(*.orm)		
EC5. Exportar diagrama con campos correctos.	Exporta un modelo de datos como imagen o archivo con extensión (.orm) en un directorio seleccionado.	V	V	V	Se exporta un diagrama como imagen con extensión (.jpeg).	Se da clic en el botón Export to image/ Export to ORM. Se muestra un formulario para obtener los datos necesarios y exportar el diagrama actual. Se selecciona el botón Save una vez llenado dicho formulario. Y el sistema exporta el diagrama al formato
		Desktop	image.jpeg	Image file (*.jpeg)		

						según la selección anterior.
EC6. Exportar diagrama con campos incompletos.	Intenta exportar un diagrama sin llenar campos obligatorios.	I	V	V	No permite exportar el diagrama.	Se da clic en el botón Export to image/ Export to ORM. Se muestra un formulario para obtener los datos necesarios y exportar el diagrama actual. Se selecciona el botón Save una vez llenado dicho formulario. Y el sistema exporta el diagrama al formato según la selección anterior.
			image.jpeg	Image file (*.jpeg)		
		V	I	V		
		Desktop		Image file (*.jpeg)		
		I	I	V		
				Image file (*.jpeg)		

Resultados de las Pruebas

El Generador visual de entidades para Doctrine 2.0 fue probado en tres iteraciones de pruebas de caja negra. En la primera iteración fueron detectadas nueve no conformidades: seis de ortografía y tres de funcionalidad. Mientras que en la segunda iteración se detectaron dos no conformidades de funcionalidad. Las no conformidades fueron resueltas luego de ser identificadas, permitiendo que la solución pasara a una tercera iteración en la que no fue encontrada ninguna no conformidad. Estos resultados se muestran gráficamente en la Fig. 13.

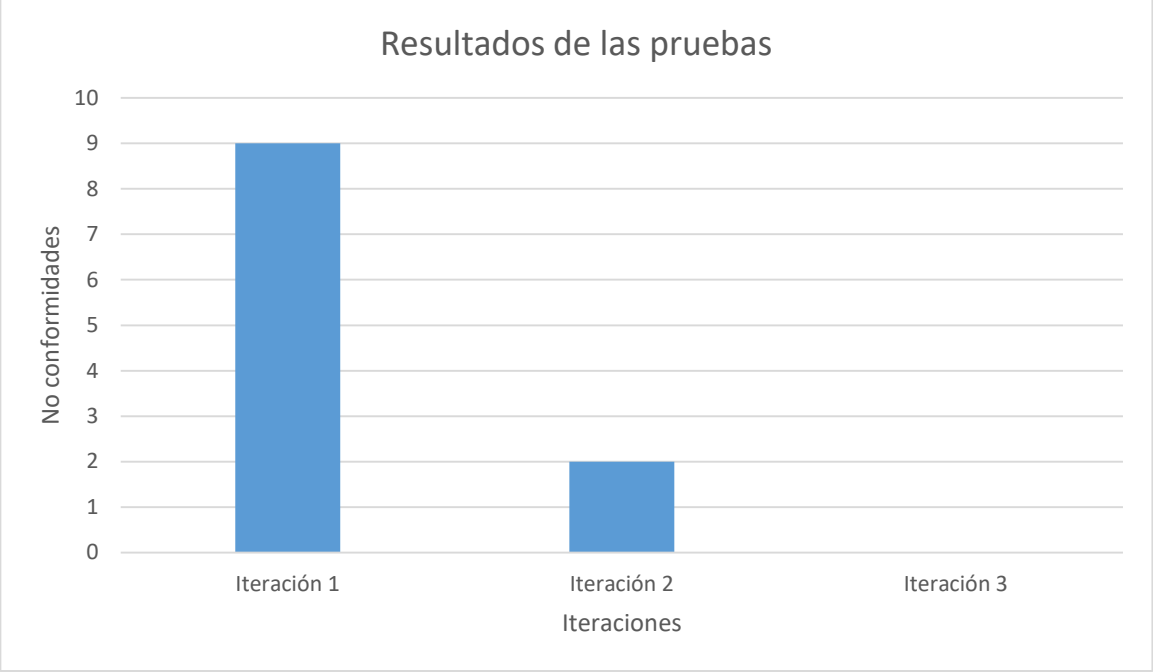


Fig. 13. Resultados de las pruebas.

3.5. Pruebas de aceptación

Es la prueba final antes del despliegue del sistema. Su objetivo es verificar que el software está listo y que puede ser usado por usuarios finales para ejecutar aquellas funciones y tareas para las cuales el software fue construido (Rodríguez Sanchez, 2015). El arquitecto de software del proyecto SIGEL, el ingeniero Rene Leandro Cruz Laguna fue el encargado de validar las funcionalidades del Generador visual de entidades para Doctrine 2.0. Los resultados fueron satisfactorios ya que cumplieron las expectativas del cliente.

3.6. Conclusiones parciales

Con el diseño del diagrama de componente se modeló cómo el sistema de software se divide en componentes y representa las dependencias entre estos para facilitar la implementación del Generador visual de entidades para Doctrine 2.0. Con el uso de los estándares de codificación se estableció un lenguaje común entre los programadores, lo cual facilita el mantenimiento del código, permitiendo de esta manera una implementación satisfactoria. En el desarrollo de las pruebas de caja blanca y caja negra se detectaron un total de nueve no conformidades en una primera iteración, dos en la segunda, quedando resueltas en la última. El Generador visual de entidades para Doctrine 2.0 quedó funcional una vez concluido dicho proceso

ya que cumplió todas las expectativas del cliente, quedando demostrado en la prueba de aceptación donde se verifica que el software está listo y que puede ser usado por usuarios finales.

Conclusiones generales

Para la realización del presente trabajo se trazaron varios objetivos y tareas que tributan a un objetivo general y que permitieron arribar a las siguientes conclusiones:

- El análisis crítico de herramientas ORM como Visual Paradigm para UML 8.0 y ORM Designer evidenció la necesidad de implementar el Generador visual de entidades para Doctrine 2.0.
- La técnica de entrevista no estructurada aplicada permitió obtener un total de 29 requisitos funcionales los cuales fueron validados con el cliente.
- El diseño de modelos a partir de los requisitos funcionales definidos posibilitó la implementación del sistema Generador visual de entidades para Doctrine 2.0.
- El empleo de las pruebas de caja blanca, caja negra y la prueba de aceptación con el cliente permitió verificar el funcionamiento de los requisitos funcionales y no funcionales implementados en el Generador visual de entidades para Doctrine 2.0.

Recomendaciones

Una vez desarrollada la herramienta y siendo presentada a los especialistas del proyecto SIGEL se recomienda que para futuras versiones:

- El sistema permita el trabajo con múltiples diseños de modelos de datos a la vez.

Referencias Bibliográficas

Bauer, Christian and King, Gavin. 2004. *Practical Object/Relational Mapping.* s.l. : Manning Publications, 2004.

Free 64 bit programs. [Online] [Cited: 11 21, 2016.] http://www.64bitprogramlar.com/portable-orm-designer-1-4-3-454-12268.html.

Gamma, Erich, et al. 1994. *Design Patterns.* s.l. : KevinZhang, 1994.

2015. GESPRO Suite de Gestión de Proyectos. *GESPRO Suite de Gestión de Proyectos.* [Online] 2015. [Cited: 10 25, 2016.] https://gespro.datec.prod.uci.cu/.

Guardado, Iván. 2010. Utilizando Doctrine como ORM en PHP. *Utilizando Doctrine como ORM en PHP.* [Online] Ontus.com, 07 06, 2010. [Cited: 02 5, 2017.] http://web.ontuts.com/tutoriales/utilizando-doctrine-como-orm-en-php/.

it-Mentor . 2006. *Pruebas de Software. Capacitación y guia para el desarrollo de software.* 2006.

Jacobson, Ivar, Booch, Grady y Rumbaugh, James. 2000. *El proceso unificado de desarrollo de software.* Madrid : Addison Wesley, 2000. ISBN: 84-7829-036-2.

javaHispano. 2014. javaHispano. *javaHispano.* [Online] 2014. [Cited: 05 10, 2017.] http://www.javahispano.org/portada/2014/3/22/novedades-y-nuevas-caracteristicas-de-java-8.html.

Jorge Hernan Abad Londoño. 2005. Ingeniería de software.TIPOS DE PRUEBAS DE SOFTWARE. *Ingeniería de software.TIPOS DE PRUEBAS DE SOFTWARE.* [Online] 2005. [Cited: Abril 27, 2017.] http://ing-sw.blogspot.com/2005/04/tipos-de-pruebas-de-software.html.

Larman, Craig. 2003. *UML y Patrones. Una introducción al análisis y diseño orientado a objetos y al proceso unificado.* México : Prentice Hall, 2003.

Leandro Alegsa. 2016. ALEGSA.com.ar. *ALEGSA.com.ar.* [Online] ALEGSA, julio 14, 2016. [Cited: abril 25, 2017.] http://www.alegsa.com.ar/Dic/entidad.php.

Lenguajes de programación. 2016. Lenguajes de programación. [Online] 2016. [Cited: 11 25, 2016.] http://www.lenguajes-de-programacion.com/lenguajes-de-programacion.shtml.

Marqués, Mercedes. 2011. *Bases de datos.* Castellón de la Plana : Publicacions de la Universitat Jaume I, 2011. ISBN: 978-84-693-0146-3.

—. 2011. *Bases de Datos.* Castellón de la Plana : Universidad Jaume I, 2011.

Mato García, Rosa María. 1999. *Diseño de bases de datos.* 1999.

—. 1999. *Diseño de bases de datos.* 1999.

Microsoft . 2017. Developer Network. *Developer Network.* [En línea] 2017. [Citado el: 4 de abril de 2017.] https://msdn.microsoft.com/es-es/library/aa291591(v=vs.71).aspx.

Microsoft. Revisiones de código y estándares de codificación. [Online] Microsoft. [Cited: marzo 30, 2017.] https://msdn.microsoft.com/es-es/library/aa291591%28v=vs.71%29.aspx.

NetBeans. 2016. NetBeans. *NetBeans.* [Online] 2016. [Cited: 11 25, 2016.] https://netbeans.org/community/releases/80/.

ORM Designer. 2013. ORM Designer. [Online] 2013. https://www.orm-designer.com/.

Pacheco, Nacho. 2011. gitnacho(Nacho) Repositories. GitHub. *gitnacho(Nacho) Repositories. GitHub.* [Online] 2011. [Cited: 02 05, 2017.] https://github.com/gitnacho?utf8=%E2%9C%93&tab=repositories&q=%20Doctrine%202%20ORM%20Documentation%20&type=&language=.

—. 2011. Ontuts. *Ontuts.* [Online] 2011. [Cited: 02 05, 2017.] http://web.ontuts.com/tutoriales/introduccion-a-object-relational-mapping-orm/..

2016. PHP.net. *PHP.net.* [Online] The PHP Group, 2016. [Cited: 11 25, 2016.] http://php.net/manual/es/intro-whatcando.php.

Pressman, Roger S. 2010. *Softwqre Engineering. A Practitioner´s Approach.Seventh Edition.* s.l. : McGraw-Hill Companies, 2010. ISBN: 978-0-07-337597-7.

Pressman, Roger S. 2010. *Ingeniería de Software. Un enfoque práctico.* 2010.

Ramos Salavert, Isidro and Lozano Pérez, María Dolore. 2000. *Ingeniería del software y bases de datos: tendencias actuales.* Cuenca : Universidad de Castilla La Mancha, 2000. ISBN 8484270777.

Rodríguez Sánchez, Tamara . 2014. *Metodología de desarrollo para la Actividad productiva en la UCI.* La Habana : s.n., 2014.

Rodríguez Sanchez, Tamara. 2015. *Metodología de desarrollo para la Actividad de la UCI.* La Habana : s.n., 2015.

Saavedra Quevedo , Bernardita , Valenzuela Parada , Victor and Alvial Cid, Raquel. 2011. *ORM - Object Relational Mapping.* 2011.

Silberschatz, Abraham, Korth, Henry and Sudarshan, S. 2002. *FUNDAMENTOS DE BASES DE DATOS.* Madrid : Mc Graw Hill, 2002. ISBN: 0-07-228363-7.

Sommerville, Ian. 2005. *Ingeniería del software. Septima edición.* Madrid : Addison Wesley (Pearson Educación,SA), 2005. ISBN: 84-7829-074-5.

Sommerville, Ian. 2006. *Software Engineering.* United Kingdom : Addison Wesley, 2006. ISBN-10: 0321313798.

Targetware. 2017. software.com.ar. *software.com.ar.* [Online] Targetware 2007-2017, 2017. [Cited: 05 16, 2017.] http://www.software.com.ar/p/visual-paradigm-para-uml.

Universidad de las Ciencias Informáticas. 2012. Universidad de las Ciencias Informáticas. *Universidad de las Ciencias Informáticas.* [Online] 2012. [Cited: 11 02, 2016.] http://www.uci.cu/?q=mision.

2016. Visual Paradigm. [Online] 11 21, 2016. http://www.visualparadigm.com/features/.

Bibliografía

Bauer, Christian and King, Gavin. 2004. *Practical Object/Relational Mapping.* s.l. : Manning Publications, 2004.

Free 64 bit programs. [Online] [Cited: 11 21, 2016.] http://www.64bitprogramlar.com/portable-orm-designer-1-4-3-454-12268.html.

Gamma, Erich, et al. 1994. *Design Patterns.* s.l. : KevinZhang, 1994.

2015. GESPRO Suite de Gestión de Proyectos. *GESPRO Suite de Gestión de Proyectos.* [Online] 2015. [Cited: 10 25, 2016.] https://gespro.datec.prod.uci.cu/.

Guardado, Iván. 2010. Utilizando Doctrine como ORM en PHP. *Utilizando Doctrine como ORM en PHP.* [Online] Ontus.com, 07 06, 2010. [Cited: 02 5, 2017.] http://web.ontuts.com/tutoriales/utilizando-doctrine-como-orm-en-php/.

it-Mentor . 2006. *Pruebas de Software. Capacitación y guia para el desarrollo de software.* 2006.

Jacobson, Ivar, Booch, Grady y Rumbaugh, James. 2000. *El proceso unificado de desarrollo de software.* Madrid : Addison Wesley, 2000. ISBN: 84-7829-036-2.

javaHispano. 2014. javaHispano. *javaHispano.* [Online] 2014. [Cited: 05 10, 2017.] http://www.javahispano.org/portada/2014/3/22/novedades-y-nuevas-caracteristicas-de-java-8.html.

Jorge Hernan Abad Londoño. 2005. Ingeniería de software.TIPOS DE PRUEBAS DE SOFTWARE. *Ingeniería de software.TIPOS DE PRUEBAS DE SOFTWARE.* [Online] 2005. [Cited: Abril 27, 2017.] http://ing-sw.blogspot.com/2005/04/tipos-de-pruebas-de-software.html.

Larman, Craig. 2003. *UML y Patrones. Una introducción al análisis y diseño orientado a objetos y al proceso unificado.* México : Prentice Hall, 2003.

Leandro Alegsa. 2016. ALEGSA.com.ar. *ALEGSA.com.ar.* [Online] ALEGSA, julio 14, 2016. [Cited: abril 25, 2017.] http://www.alegsa.com.ar/Dic/entidad.php.

Lenguajes de programación. 2016. Lenguajes de programación. [Online] 2016. [Cited: 11 25, 2016.] http://www.lenguajes-de-programacion.com/lenguajes-de-programacion.shtml.

Marqués, Mercedes. 2011. *Bases de datos.* Castellón de la Plana : Publicacions de la Universitat Jaume I, 2011. ISBN: 978-84-693-0146-3.

—. 2011. *Bases de Datos.* Castellón de la Plana : Universidad Jaume I, 2011.

Mato García, Rosa María. 1999. *Diseño de bases de datos.* 1999.

—. 1999. *Diseño de bases de datos.* 1999.

Microsoft . 2017. Developer Network. *Developer Network.* [En línea] 2017. [Citado el: 4 de abril de 2017.] https://msdn.microsoft.com/es-es/library/aa291591(v=vs.71).aspx.

Microsoft. Revisiones de código y estándares de codificación. [Online] Microsoft. [Cited: marzo 30, 2017.] https://msdn.microsoft.com/es-es/library/aa291591%28v=vs.71%29.aspx.

NetBeans. 2016. NetBeans. *NetBeans.* [Online] 2016. [Cited: 11 25, 2016.] https://netbeans.org/community/releases/80/.

ORM Designer. 2013. ORM Designer. [Online] 2013. https://www.orm-designer.com/.

Pacheco, Nacho. 2011. gitnacho(Nacho) Repositories. GitHub. *gitnacho(Nacho) Repositories. GitHub.* [Online] 2011. [Cited: 02 05, 2017.] https://github.com/gitnacho?utf8=%E2%9C%93&tab=repositories&q=%20Doctrine%202%20ORM%20Documentation%20&type=&language=.

—. 2011. Ontuts. *Ontuts.* [Online] 2011. [Cited: 02 05, 2017.] http://web.ontuts.com/tutoriales/introduccion-a-object-relational-mapping-orm/..

2016. PHP.net. *PHP.net.* [Online] The PHP Group, 2016. [Cited: 11 25, 2016.] http://php.net/manual/es/intro-whatcando.php.

Pressman, Roger S. 2010. *Softwqre Engineering. A Practitioner´s Approach.Seventh Edition.* s.l. : McGraw-Hill Companies, 2010. ISBN: 978-0-07-337597-7.

Pressman, Roger S. 2010. *Ingeniería de Software. Un enfoque práctico.* 2010.

Ramos Salavert, Isidro and Lozano Pérez, María Dolore. 2000. *Ingeniería del software y bases de datos: tendencias actuales.* Cuenca : Universidad de Castilla La Mancha, 2000. ISBN 8484270777.

Rodríguez Sánchez, Tamara . 2014. *Metodología de desarrollo para la Actividad productiva en la UCI.* La Habana : s.n., 2014.

Rodríguez Sanchez, Tamara. 2015. *Metodología de desarrollo para la Actividad de la UCI.* La Habana : s.n., 2015.

Saavedra Quevedo , Bernardita , Valenzuela Parada , Victor and Alvial Cid, Raquel. 2011. *ORM - Object Relational Mapping.* 2011.

Silberschatz, Abraham, Korth, Henry and Sudarshan, S. 2002. *FUNDAMENTOS DE BASES DE DATOS.* Madrid : Mc Graw Hill, 2002. ISBN: 0-07-228363-7.

Sommerville, Ian. 2005. *Ingeniería del software. Septima edición.* Madrid : Addison Wesley (Pearson Educación,SA), 2005. ISBN: 84-7829-074-5.

Sommerville, Ian. 2006. *Software Engineering.* United Kingdom : Addison Wesley, 2006. ISBN-10: 0321313798.

Targetware. 2017. software.com.ar. *software.com.ar.* [Online] Targetware 2007-2017, 2017. [Cited: 05 16, 2017.] http://www.software.com.ar/p/visual-paradigm-para-uml.

Universidad de las Ciencias Informáticas. 2012. Universidad de las Ciencias Informáticas. *Universidad de las Ciencias Informáticas.* [Online] 2012. [Cited: 11 02, 2016.] http://www.uci.cu/?q=mision.

2016. Visual Paradigm. [Online] 11 21, 2016. http://www.visualparadigm.com/features/.