Una ontología para la representación de conceptos de diseño de software

An ontology for the representation of software design concepts

Gloria L. Giraldo G., Ph. D., Juan F. Acevedo O., M. Sc. (c) & David A. Moreno N., Ing. (c)[3]
Escuela de Sistemas, Facultad de Minas, Universidad Nacional de Colombia, Medellín, Colombia
{glgiraldog, jacevedoo, damorenon}@unal.edu.co

Recibido para revisión 29 de junio de 2011, aceptado 18 de octubre de 2011, versión final 10 de noviembre de 2011

***Resumen* — En los últimos años se integraron conceptos importantes al diseño de software, tales como nuevos diagramas de modelado y patrones, creados con el fin de evolucionar el diseño de software y mejorar la calidad de las aplicaciones. Los conceptos incorporados se presentan a los desarrolladores en extensa documentación. La cantidad de información resultante se ha incrementado de tal manera que es difícil manejarla. Las investigaciones hacen énfasis en mejorar y crear nuevos conceptos de diseño, pero existen pocas iniciativas orientadas a estructurar este conocimiento. Un claro ejemplo de la necesidad de estructurar este conocimiento son los patrones GoF, pues su éxito no fue su creación sino su recopilación desde los trabajos de otros desarrolladores. En este artículo se propone una ontología para organizar el conocimiento en diseño de software, incorporando conceptos de diagramas de modelado y patrones, Con el fin de recopilar algunos conceptos de diseño a tener en cuenta en un proyecto de software.**

***Palabras Clave* — Diagramas de modelado, patrones, GoF, Ontologia**

***Abstract* — In recent years, important concepts were integrated into the design of software, such as new modeling diagrams and patterns created to evolve the software design and improve the quality of the applications. The concepts embedded developers are presented in extensive documentation. The amount of resulting information has increased so that it is difficult to handle this. Emphasize research on improving and creating new design concepts, but there are few initiatives to structure this knowledge. A clear example of the need to structure this knowledge are the GoF patterns, as the success of its creators was not his creation but the collection of these from other developers work. This article proposes an ontology to organize knowledge in software design, incorporating concepts of modeling diagrams and patterns, to collect some design concepts to consider in a software project.**

***Keywords* — Model Diagrams, Pattern, GoF, Ontology**

I. INTRODUCCIÓN

Las ontologías permiten representar el entendimiento común y compartido de un dominio, facilitando la comunicación entre los miembros de una comunidad o entre sistemas computacionales [8]-[9]-[14].

Por otro lado, el diseño hace parte del ciclo de vida del software y permite que los desarrolladores se enfoquen en la calidad del software, en los plazos de implementación y en los costos asociados. Algunos de los conceptos más importantes en esta fase de diseño son los diagramas y los patrones de diseño. Un Diagrama es una representación gráfica de una colección de elementos de modelado, a menudo dibujada como un grafo conexo que consiste en arcos y vértices [2], [3], [10], [11], [12]. Un patrón de diseño es una abstracción de una solución en alto nivel y solucionan problemas que existen en muchos de estos niveles de abstracción, además, existen patrones que abarcan las distintas etapas del desarrollo [1], [5].

En la literatura actual se considera la necesidad de una ontología que almacene y permita consultar los conceptos relacionados con el diseño de software, además de facilitar la representación de las razones que conllevan a estos diseñadores a elegir un modelo de entre varias opciones [4], [6], [7], [13]. Sin embargo, los trabajos revisados tienen en cuenta muy pocos patrones y diagramas que se pueden utilizar para el diseño.

Por las razones expuestas, en este artículo se propone una ontología para representar el conjunto de diagramas estructurales y de comportamiento, además de los patrones GoF y GRASP que se emplean durante la fase de diseño de software. Esta ontología se puede utilizar en aplicaciones colaborativas que pretendan enseñar conceptos básicos de la fase de diseño de software a estudiantes de ingeniería de sistemas.

Este artículo se organiza de la siguiente manera: la sección 2 contiene el marco teórico, donde se presentan los principales elementos que fundamentan esta ontología; en la sección 3 se realiza una revisión de algunas propuestas para establecer ontologías en diseño de software; en la sección 4 se propone una ontología para la representación de conceptos de diseño de software. Finalmente, en la sección 5 se presentan las conclusiones.

II. MARCO TEÓRICO

A. Ontología

El diccionario define ontología como "la rama de la metafísica que estudia la naturaleza de la existencia". En la práctica, las ontologías son una entidad computacional y no deben ser consideradas como una entidad natural que se descubre, sino como recurso artificial que se crea [9]. Una ontología corresponde a un entendimiento común y compartido de un dominio, que permite una comunicación efectiva entre miembros de una comunidad y también entre sistemas computacionales. En los últimos años, se ha suscitado gran interés en la creación e integración de ontologías debido a que, por su naturaleza, ellas se pueden compartir y reutilizar en diferentes aplicaciones [14]. Una de las definiciones más referenciadas en la literatura es la de Thomas Gruber [8]: "*Una Ontología es una especificación formal y explícita de una conceptualización compartida*". Él explica cada uno de sus términos: *conceptualización* como un modelo abstracto de algún fenómeno del mundo del que se busca representar los conceptos más relevantes, *explícita* se refiere a la necesidad de detallar de forma consciente los diferentes conceptos de la ontología, *formal* se refiere a que se debe utilizar un lenguaje de representación formalizado para su especificación; y *compartida* es que ésta debe contener conocimiento aceptado en el dominio.

B. Fase de diseño de software

Las fases de diseño de software son parte del ciclo de vida del software, en cada una de estas se valida el desarrollo de una aplicación, Esto permite que los errores se detecten lo antes posible y por lo tanto, permite a los desarrolladores concentrarse en la calidad del software. Algunos de los instrumentos utilizados para mejorar la calidad del software, orientan y facilitan el proceso de diseño.

De hecho, para muchos problemas, buscamos la solución a un problema similar en el pasado para intentar aplicarla misma solución. Los patrones son esa herramienta para capturar y comunicar este entendimiento y experiencia, creando conocimiento de diseño persistente entre profesionales de una comunidad. Un diseño de software efectivo requiere considerar asuntos que pueden no ser visibles hasta más allá de su implementación. Reutilizar los patrones de diseño ayuda a prevenir asuntos sutiles que pueden causar problemas mayores, aunque los patrones no son un método o proceso de desarrollo, pueden complementarlos.

Existen un lenguaje gráfico para visualizar, especificar y documentar cada una de las partes que comprende el desarrollo de software, son los diagramas, los cuales son la forma de modelar cosas conceptuales como lo son procesos de negocio y funciones de sistema, además de cosas concretas como lo son escribir clases en un lenguaje determinado, esquemas de base de datos y componentes de software reusables.

La adecuada utilización de los elementos mencionados permiten mejorar la calidad del software, y empiezan a crear la cultura de buenas prácticas de desarrollo, aunque los analista han escuchado de estos elementos en ocasiones no tiene los suficientes conocimientos para aplicarlos, esta ontología quiere difundir el conocimiento en patrones y diagramas de diseño.

Diagramas de la fase de diseño

Un Diagrama es una representación gráfica de una colección de elementos de modelado, a menudo dibujada como un grafo conexo de arcos (relaciones) y vértices (otros elementos del modelo). Un diagrama no es un elemento semántico, un diagrama muestra representaciones de elementos semánticos del modelo, pero su significado no se ve afectado por la forma en que son representados [2] [12].

- **Diagramas de comportamiento:** Los diagramas de comportamiento se emplean para visualizar, especificar, construir y documentar los aspectos dinámicos de un sistema. Estos aspectos involucran cosas tales como el flujo de mensajes a lo largo del tiempo y el movimiento físico de componentes en una red.
- **Diagrama de Casos de uso (UML):** Un caso de uso se puede definir como una secuencia de acciones, incluyendo acciones alternas, que el sistema puede realizar y que producen un resultado concreto para un actor que interactúa con el sistema.
- **Diagrama de Actividades (UML):** Representa los flujos de trabajo paso a paso de negocio y operacionales de los componentes en un sistema. Un Diagrama de Actividades muestra el flujo de control general.
- **Diagrama de Comunicación (UML):** Muestra interacciones organizadas alrededor de los roles. A diferencia de los diagramas de secuencia, los diagramas de comunicación muestran explícitamente las relaciones de los roles.
- **Diagrama de Secuencia (UML):** Un tipo de diagrama usado para modelar interacción entre objetos en un sistema según UML. muestra la interacción de un conjunto de objetos en una aplicación a través del tiempo y se modela para cada caso de uso.
- **Diagrama de Decisión:** El diagrama de decisiones contribuye al análisis de la estrategia tanto en términos de coherencia interna (sucesión lógica de las sucesivas decisiones) como de pertinencia respecto al exterior (consideración de los

elementos contextuales y de las posturas de las partes interesadas) [11].

- **Maquina de estados (UML):** Es un diagrama utilizado para identificar cada una de las rutas o caminos que puede tomar un flujo de información luego de ejecutarse cada proceso. Identifica bajo qué argumentos se ejecuta cada uno de los procesos y en qué momento podrían tener una variación, además, permite visualizar de una forma secuencial la ejecución de cada uno de los procesos.
- **Diagrama de tiempo (UML):** El diagrama de Tiempo define el comportamiento de los diferentes objetos con una escala de tiempo. Provee una representación visual de los objetos cambiando de estado e interactuando a lo largo del tiempo.
- Puede usar diagramas de tiempos para definir componentes de software dirigidos por hardware o embebidos; por ejemplo, aquellos usados en un sistema de inyección de combustible, un controlador de microondas. También puede usar diagramas de tiempo para especificar procesos de negocio dirigidos por tiempo.
- **Revisión de interacciones (UML):** Los diagramas de interacciones muestran la cooperación entre otros diagramas de interacción para reflejar el flujo de control. Como los Diagramas de Interacciones son una variante de los diagramas de actividades, la mayor parte de la notación es similar, al igual que el proceso de construcción del diagrama. Los puntos de decisión, bifurcación, unión, puntos de inicio y final son los mismos. En lugar de actividades se usan elementos rectangulares. Hay dos tipos de estos elementos: Los elementos de interacción muestran un diagrama de interacción en línea, el cual puede ser un diagrama de Secuencias, Comunicaciones, de Tiempos, o de descripción de las Interacciones. Los elementos de ocurrencia de interacción son referenciados a un diagrama de interacción existente.
- **Diagramas estructurales:** Los diagramas estructurales representan elementos y así componen un sistema o una función. Estos diagramas pueden reflejar las relaciones estáticas de una estructura, como lo hacen los diagramas de clases o de paquetes, o arquitecturas en tiempo de ejecución, tales como diagramas de Objetos o de Estructura de Composición.
- **Diagrama de clases (UML)**: Es un tipo de diagrama estático que describe la estructura de un sistema mostrando sus clases, atributos y las relaciones entre ellos. Los diagramas de clases son utilizados durante el proceso de análisis y diseño de los sistemas, donde se crea el diseño conceptual de la información que se manejará en el sistema, y los componentes que se encargaran del funcionamiento y la relación entre uno y otro.
- **Diagrama de componentes (UML):** Representa cómo un sistema de software es dividido en componentes y muestra las dependencias entre estos componentes. Los componentes físicos incluyen archivos, cabeceras, bibliotecas compartidas, módulos, ejecutables, o paquetes.
- **Diagrama de despliegue (UML):** Es un tipo de diagrama del Lenguaje Unificado de Modelado que se utiliza para modelar el hardware utilizado en las implementaciones de sistemas y las relaciones entre sus componentes.
- **Diagrama de estructura de Jackson:** Se basa en el principio de que la base inicial del diseño del programa son los datos del problema y no los requisitos funcionales exigidos. Una vez obtenida una estructura objetiva del problema, que constituye un reflejo del mundo real con el que trata el programa, resulta más fácil asignar las distintas funciones a realizar [10].
- **Diagrama de objetos (UML):** Son utilizados durante el proceso de Análisis y Diseño de los sistemas informáticos en la metodología UML. Los diagramas de objetos utilizan un subconjunto de los elementos de un diagrama de clase.
- **Diagrama entidad relación:** Este modelo representa a la realidad a través de un esquema gráfico empleando los terminología de **entidades**, que son objetos que existen y son los elementos principales que se identifican en el problema a resolver con el diagramado y se distinguen de otros por sus características particulares denominadas **atributos,** el enlace que rige la unión de las entidades está representada por la **relación** del modelo [3].
- **Diagrama de Paquetes (UML):** Los Diagramas de Paquetes se usan para reflejar la organización de los paquetes y sus elementos, y para proveer una visualización de sus correspondientes nombres de espacio.
- **Estructura compuesta (UML):** Un diagrama de Estructura Compuesta refleja la colaboración interna de clases, interfaces o componentes para describir una funcionalidad. Los diagramas de estructura compuesta son similares a los diagramas de clase, a excepción de que estos modelan un uso especifico de la estructura. Los diagramas de clase modelan una vista estática de las estructuras de clase, incluyendo sus atributos y comportamientos. Un diagrama de Estructura Compuesta se usa para expresar arquitecturas en tiempo de ejecución, patrones de uso, y las relaciones de los elementos participantes, los que pueden no estar reflejados por diagramas estáticos.

Patrones

Un patrón de diseño es una abstracción de una solución en un nivel alto. Los patrones solucionan problemas que existen en muchos niveles de abstracción. Hay patrones que abarcan las distintas etapas del desarrollo.

Patrones GRASP

Lo esencial de un diseño de objetos lo constituye el diseño de las interacciones de objetos y la asignación de responsabilidades. Las decisiones que se tomen pueden influir profundamente en la extensibilidad, claridad y mantenimiento del sistema de software de objetos, además en el grado y calidad

Revista Avances en Sistemas e Informática, Vol.8 No. 3, diciembre de 2011 - Medellín. ISSN 1657-7663

de los componentes reutilizables, por esta razón, durante el diseño se deben realizar los casos de usos con objetos basado en los patrones GRASP [1].

- **Alta cohesión:** Es un principio evaluativo que aplica un diseñador mientras evalúa todas las decisiones de diseño. Indica la relación que existe entre los elementos de un mismo módulo. Es la medida de la relación funcional de los elementos de un módulo. El objetivo es organizar estos elementos de manera que los que tengan una mayor relación a la hora de realizar una tarea pertenezcan al mismo módulo, y los elementos no relacionados, se encuentren en módulos separados.
- **Bajo acoplamiento:** Impulsa la asignación de responsabilidades de manera que su localización no incremente el acoplamiento hasta un nivel que nos lleve a los resultados negativos que puede producir un acoplamiento alto. Es el grado de interdependencia entre los módulos. Un buen diseño se caracteriza por un acoplamiento mínimo, es decir, unos módulos tan independientes los unos de los otros como sea posible.
- **Controlador:** Proporciona guías acerca de las opciones generalmente aceptadas y adecuadas para manejar eventos. Es conveniente utilizar la misma clase controlador para todos los eventos del sistema de un caso de uso, de manera que es posible manejar la información acerca del estado del caso de uso en el controlador.
- **Creador:** Guía la asignación de responsabilidades relacionadas con la creación de objetos, una tarea muy común. La intención básica del patrón es encontrar un creador que necesite conectarse al objeto creado en alguna situación.
- **Experto:** Se utiliza con frecuencia en la asignación de responsabilidades; es un principio de guía básico que se utiliza continuamente en el diseño de objetos. Expresa la intuición común de que los objetos hacen las cosas relacionadas con la información que tienen.

Patrones GoF

En el año 1994, apareció el libro "Design Patterns: Elements of Reusable Object Oriented Sofware" escrito por los ahora famosos Gang of Four (Pandilla de los cuatro) integrada por Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides. Estos recopilaron y documentaron 23 patrones de diseño aplicados usualmente por expertos diseñadores de software orientado a objetos. Desde luego ellos no son los inventores ni los únicos involucrados, pero luego de la publicación de ese libro empezó a difundirse con más fuerza la idea de patrones de diseño [5]. Se distinguen tres tipos de patrones GoF: patrones de comportamiento, patrones creacionales y patrones estructurales

Patrones de comportamiento

Definen la comunicación e iteración entre los objetos de un sistema. El propósito de este tipo de patrón es reducir el acoplamiento entre los objetos.

- **Abstract Factory:** Permite trabajar con objetos de distintas familias de manera que las familias no se mezclen entre sí. Además, hace transparente el tipo de familia concreta que se esté usando.
- **Builder:** Abstrae el proceso de creación de un objeto complejo, centralizando dicho proceso en un único punto.
- **Factory Method:** Centraliza en una clase constructora la creación de objetos de un subtipo de un tipo determinado, ocultando al usuario la casuística para elegir el subtipo que crear.
- **Prototype:** Crea nuevos objetos clonándolos de una instancia ya existente.
- **Singleton:** Garantiza la existencia de una única instancia para una clase y la creación de un mecanismo de acceso global a dicha instancia.

Patrones creacionales

Tratan con las formas de crear instancias de objetos. El objetivo de estos patrones es de abstraer el proceso de instanciación y ocultar los detalles de cómo los objetos son creados o inicializados.

- **Adapter:** Adapta una interfaz para que pueda ser utilizada por una clase que de otro modo no podría utilizarla.
- **Bridge:** Desacopla una abstracción de su implementación.
- **Compositive:** Permite tratar objetos compuestos como si se tratara de uno simple.
- **Decorator:** Añade funcionalidad a una clase dinámicamente.
- **Facade:** Provee de una interfaz unificada simple para acceder a una interfaz o grupo de interfaces de un subsistema.
- **Flyweigth:** Reduce la redundancia cuando gran cantidad de objetos poseen idéntica información.
- **Proxy:** Mantiene un representante de un objeto.

Patrones estructurales

Describen cómo clases y objetos pueden ser combinados para formar grandes estructuras y proporcionar nuevas funcionalidades. Estos objetos adicionados pueden ser incluso objetos simples u objetos compuestos.

- **Chain of responsability:** Permite establecer la línea que deben llevar los mensajes para que los objetos realicen la tarea indicada.
- **Command:** Permite encapsular una operación en un objeto, permitiendo ejecutar dicha operación sin necesidad de conocer el contenido de la misma.
- **Interpreter:** Dado un lenguaje, define una gramática para dicho lenguaje, así como las herramientas necesarias para interpretarlo.
- **Iterator:** Proporciona un modo de acceder secuencialmente

a los elementos de un objeto agregado sin exponer su representación interna.

- **Mediator:** Define un objeto para encapsular la interacción en un conjunto de objetos. Promueve un bajo acoplamiento al evitar que los objetos se refieran unos a otros explícitamente, y permite variar la interacción entre ellos de forma independiente.
- **Memento:** Representa y externaliza el estado interno de un objeto sin violar la encapsulación, de forma que éste puede volver a dicho estado más tarde.
- **Observer:** Define una dependencia de uno-a-muchos entre objetos, de forma que cuando un objeto cambie de estado se notifique y actualicen automáticamente todos los objetos que dependen de él.
- **State:** Permite que un objeto modifique su comportamiento cada vez que cambia su estado interno. Parecerá que cambia la clase del objeto.
- **Strategy:** Define una familia de algoritmos, encapsula uno de ellos y los hace intercambiables. Permite que un algoritmo varíe independientemente de los clientes que lo usan.
- **Template Method:** Define en una operación el esqueleto de un algoritmo, delegando en las subclases algunos de sus pasos. Permite que las subclases redefinan ciertos pasos del algoritmo sin cambiar su estructura.
- **Visitor:** Representa una operación sobre los elementos de una estructura de objetos. Permite definir una nueva operación sin cambiar las clases de los elementos sobre los que opera.

III. ANTECEDENTES

En [13] se plantea la necesidad de tener una ontología para almacenar y consultar conceptos relacionados con el diseño de software, además de facilitar la representación de las razones que conllevan a estos diseñadores a elegir un modelo de entre varias opciones. Para ello, se propone una categorización de dichos conceptos, la cual permite que el diseñador comprenda, entre otras ideas, algunas técnicas de diseño. Con esta propuesta se pretende evitar malinterpretaciones de términos ambiguos por parte de los diseñadores y usuarios, quienes tienen diferentes conocimientos acerca del diseño de software. Por otro lado, se sugiere una serie de pasos para desarrollar una ontología que permita construir una base de conocimientos con clases, objetos y atributos. Esta base de conocimientos ayuda a los diseñadores a analizar el sistema de software desarrollado. Los pasos a seguir son: identificar los límites del diseño de software, construir la taxonomía, identificar los atributos de las clases y los valores que puede tomar y, finalmente, definir los axiomas y las reglas para la verificación de restricciones mediante lógica de predicados.

Los trabajos de [6] y [7] plantean que se dispone de una gran cantidad y variedad de conocimiento teórico y práctico en diseño orientado a objetos (DOO), pero aún no se sabe cómo aprovecharlo al máximo. Por ello, proponen una ontología para organizar el conocimiento declarativo y operativo de DOO, ya sea general, tecnológico o de dominio, además del registro de las decisiones tomadas durante el DOO.

Por otro lado, en [4] se plantea que, aunque se reúnen algunos conceptos de paradigmas de programación y su impacto en el diseño de programas simples, este vocabulario es aún pequeño y no tiene un derrotero ni formalismos definidos. Por ello, proponen una estructura conceptual para el diseño de software que permita realizar un acercamiento a un proyecto macro, el cual es generar una ontología completa sobre el tema en cuestión. La abstracción de conceptos se realiza desde tres puntos de vista, a saber, la arquitectura, el detalle y la implementación de software.

Sin embargo, los trabajos revisados tienen en cuenta muy pocos patrones y diagramas que se pueden utilizar para el diseño

IV. DESARROLLO

El dominio de la ontología será la representación del conjunto de diagramas estructurales y de comportamiento, además de algunos patrones (GoF y GRASP) que se emplean durante la fase de diseño de software. Esta ontología se puede utilizar en aplicaciones colaborativas que pretendan enseñar (o introducir) conceptos básicos de la fase de diseño de software a estudiantes de ingeniería de sistemas. Se describirán conceptos como los diversos elementos (clases, objetos, componentes, flujos de control, funcionalidad, etc.) que se pueden representar mediante diagramas de estructura y de comportamiento, además de ciertas características (problema que soluciona, beneficios, etc.) de los patrones de diseño básicos.

Las preguntas de competencia que se diseñaron para esta ontología son:

¿Qué componentes tiene el diagrama de casos de uso?

¿Qué componentes tiene el diagrama de clases?

¿Cuáles diagramas emplean la relación de Asociación?

¿En cuáles diagramas se encuentra el elemento Clase?

¿Cuáles patrones permiten mantener el bajo acoplamiento en el modelo de diseño?

¿Cuáles patrones permiten asignar responsabilidades a los objetos?

¿Qué componentes tiene el diagrama de actividades?

¿Qué componentes tiene el diagrama de colaboración?

¿Qué componentes tiene el diagrama de secuencias?

¿Qué componentes tiene el diagrama de decisión?

Revista Avances en Sistemas e Informática, Vol.8 No. 3, diciembre de 2011 - Medellín. ISSN 1657-7663

¿Qué componentes tiene el diagrama de componentes?
¿Qué componentes tiene el diagrama de despliegue?
¿Qué componentes tiene el diagrama de estructura de Jackson?
¿Qué componentes tiene el diagrama de objetos?
¿Qué componentes tiene el diagrama entidad relación?
¿Cuáles diagramas emplean la relación de Agregación?
¿Cuáles diagramas emplean la relación de Composición?
¿Cuáles diagramas emplean la relación de Dependencia?
¿Cuáles diagramas emplean la relación de Herencia?
¿Cuáles diagramas emplean la relación de Iteración?
¿Cuáles diagramas emplean la relación de Cardinalidad?
¿Cuáles son las relaciones que emplean los casos de uso?
¿En cuáles diagramas se encuentra el elemento Actor?
¿En cuáles diagramas se encuentra el elemento Objeto?
¿En cuáles diagramas se encuentra el elemento Proceso?
¿En cuáles diagramas se encuentra el elemento Nodo?
¿Cuáles patrones permiten mantener alta cohesión en el modelo de diseño?
¿Cuáles patrones se deben tener en cuenta a la hora de crear o instanciar objetos?
¿Cuáles son los patrones GRASP?
¿Cuáles son los patrones GoF?
¿Cuáles son los diagramas más empleados durante la fase de diseño de software?
¿Cuáles patrones se deben tener en cuenta a la hora de diseñar las interfaces del modelo?
¿Cuáles patrones favorecen la reutilización?
¿Cuáles patrones permiten manejar los estados del sistema?
¿Cuáles patrones se deben tener en cuenta a la hora de diseñar las clases del sistema?
¿Cuáles patrones permiten manejar la encapsulación en el sistema?

Para la ontología a desarrollar se hará uso de la clasificación de diagramas realizada en la ontología de Wongthongtham *et al.* [15] y en Zapata *et al.* [16] acerca de la fase de diseño de software, y se extenderá dicha ontología para incluir conceptos acerca de los patrones GRASP y GoF, los cuales también se deben tener en cuenta a la hora de diseñar una aplicación de software.

Los aspectos relevantes para la ontología serán básicamente dos, los diagramas y los patrones. Los diagramas pertenecen a dos grupos, estructurales y de comportamiento. Algunos diagramas estructurales, como clases, componentes y despliegue, entre otros, representan ciertos elementos como las clases y sus interacciones, los objetos, los componentes, las responsabilidades, los nodos físicos, etc. Algunos diagramas de comportamiento, como el diagrama de actividades, de colaboración, de secuencias, etc., representan elementos como el control de flujo entre actividades, objetos y estados, entre otros, además de las interacciones entre los mismos.

Por otro lado, los patrones GRASP son experto, creador, bajo acoplamiento, alta cohesión y controlador, los cuales permiten obtener ciertos beneficios tras su aplicación, como los son el encapsulamiento, la reutilización, claridad y acoplamiento bajo. Los patrones GoF se dividen en 3 categorías, a saber, estructural, creacional y comportamiento. Estos patrones tienen un problema de diseño que atacar y proponen una solución al respecto. La jerarquía de clases se aprecia en la figura 1, y se definió de la forma Top-down.

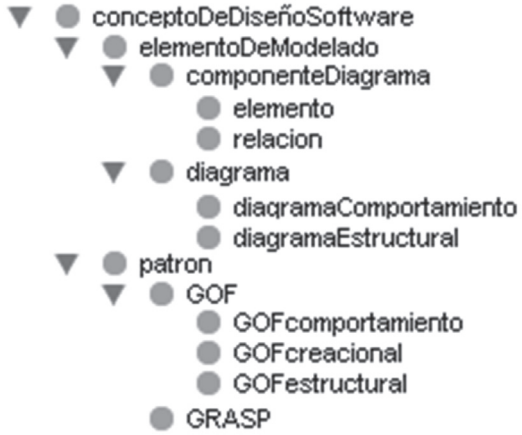

Figura 1. Jerarquía de clases

Como se mencionó anteriormente, el objetivo de la ontología es enseñar conceptos de diseño, por ello la clase "conceptoDeDiseñoSoftware", desde los puntos de vista de los elementos que se emplean para modelar en esta fase, y los patrones que conllevan a las buenas prácticas de la misma (clases "elementoDeModelado" y "Patrón"). Dentro de los elementos de modelado se incluyen los diagramas (case "diagrama") y los componentes ("componenteDiagrama") que los construyen. En los patrones ("Patrón"), se describen los de tipo GRASP y GoF. Por otro lado, los slots definidos hasta el momento se aprecian en la figura 2.

Figura 2. Slots de la ontología

La tabla 1 muestra las facetas de cada uno de los slots definidos anteriormente.

Tabla 1. Facetas de los slots

Slot	Tipo de valor	cardinalidad	Dominio	Rango
Beneficio	String	Single	GRASP	String
Descripción	String	Single	patrón	String
Nombre	String	Single	conceptoDeDiseñoSotware	String
Presente En Diagrama	InstanceOf	Multiple	componenteDiagrama	diagrama
Principio	String	Single	GRASP	String
Problema	String	Single	patrón	String
Representación	Image	Single	**GoF**	.jpg
Solución	String	Single	patrón	String
Tiene Componente	InstanceOf	Single	componenteDiagrama	diagrama

Las instancias de las clases se aprecian en las figuras 3 a 10:

- Accion
- Activacion
- Actividad
- Actor
- Arco
- Atributo
- Camino de concurrencia
- Camino de ejecucion
- Caso de Uso
- Clase
- Colaboración
- Combiar
- Combinacion compleja
- Condicion
- Conexion
- Contexto
- Control
- Decision
- Destroy
- EJB
- Elección
- Entidad
- Entrada
- Enumeracion
- Enviar
- Escala de tiempo
- Estado
- Estructura de datos
- Excepcion
- Exponer interfaz
- Fin
- Final de flujo
- Flujo
- Flujo de mensaje
- Fork
- Frontera
- Historial
- Indicador de subactividad
- Inicio
- Instancia
- Interfaz
- Interrupcion
- Linea de vida
- Linea de vida de valor
- Linea de vida del estado
- Marcador de creacion
- Marcador de destruccion
- Mensaje
- Metodo recursivo
- Nodo
- Nota
- Objeto
- Objeto compuesto
- Paquete
- Parte
- Partición
- Proceso
- Puerta del diagrama
- Puerto
- Punto final del mensaje
- Recibir
- Región
- RMI
- Salida
- Señal
- Sinc
- Sistema
- Sub-Maquina
- Tabla
- Terminar
- Tiempo real
- Union
- Utilidad

Figura 3. Instancias de la clase elemento.

- Agregacion
- Anidamiento
- Asociacion
- Bifuraccion
- Cardinalidad
- Clase asociacion
- Combinación del paquete
- Composicion
- Conector
- Delegar
- Dependencia
- Enlace de roles
- Enlaces
- Ensamble
- Extends
- Flujo de control
- Flujo de interrupción
- Flujo de objeto
- Fusion
- Generalizacion
- Herencia
- Importación del paquete
- Include
- Iteracion
- Ocurrencia
- Particion
- Realización
- Representa
- Secuencia
- Seleccion
- Simple
- Transición

Figura 4. Instancias de la clase relación.

- Diagrama casos de uso
- Diagrama de actividades
- Diagrama de colaboracion
- Diagrama de interacción
- Diagrama de secuencias
- Diagrama de tiempos
- Diagrama maquina de estados
- Diagramas de decision

Figura 5. Instancias de la clase diagramaComportamiento.

- Diagrama de clases
- Diagrama de componentes
- Diagrama de despliegue
- Diagrama de estructura compuesta
- Diagrama de estructura de jackson
- Diagrama de objetos
- Diagrama de paquetes
- Diagrama entidad relacion

Figura 6. Instancias de la clase diagramaEstructural.

- AbstractFactory
- Builder
- FactoryMethod
- Prototype
- Singleton

Figura 7. Instancias de la clase GoFcomportamiento.

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Figura 8. Instancias de la clase GoFcreacional.

- ChainOfResponsabilit
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- TemplateMethod
- Visitor

Figura 9. Instancias de la clase GoFestructural.

- AltaCohesion
- BajoAcoplamient
- Controlador
- Creador
- Experto

Figura 10. Instancias de la clase GRASP.

IV. CONCLUSIONES

En este artículo se presentó una ontología para la representación del conocimiento de diseño de software, en particular del

conjunto de diagramas y patrones de esta fase del desarrollo, la cual permite tener una estructura organizada y unificada de los conceptos mencionados anteriormente, para facilitar su acceso a los estudiantes que deseen aprenderlos.

Dado que el proceso de creación de una ontología es iterativo e incremental, se recomienda continuar los esfuerzos en pro de depurar y mejorar ésta versión, así como también ampliar su dominio de conocimiento, la estructura de la ontología permite fácilmente integrar nuevos patrones y diagramas de diseño, así mismo, pueden integrarse nuevos conceptos.

REFERENCIAS

[1] Alexander C., Murria S., Jacobson M., Fiksdahl-King I. y Angel S., 1977. A pattern Language: Towns, Building, Construction. New York: Oxford University Press, 1171 P.

[2] Booch G., Rumbaugh J. y Jacobson I. 1999. El Lenguaje Unificado de Modelado. Madrid: Addison-wesley iberoamericana, 512 P.

[3] Chen P., 1976. The entity-relationship model toward a unified view of data. *En:* ACM Transactions on Database Systems, Vol. 1, pp. 9 - 36.

[4] Eden A., Turner R., 2005. Towards an ontology of software design: the intension/locality hypothesis. En: 3rd European Conference of Computing and Philosophy, pp. 1 - 4.

[5] Gamma E., Helm R., Johnson J. y Vlissides J. 1994. Design Patterns: Elements of Reusable Object Oriented Sofware. Boston: Addison-Wesley, 416 P.

[6] Garzas J. y Piattini M., 2005. An Ontology for Microarchitectural Design Knowledge, En: IEEE Software, Vol. 22, pp. 28 - 33.

[7] Garzas J. y Piattini M., 2007. An Ontology for Understanding and Applying Object-Oriented, Design Knowledge. En: International Journal of Software Engineering and Knowledge Engineering, Vol. 17, pp. 407 - 421.

[8] Gruber T., 1993. A Translation Approach to Portable Ontologies. En: Knowledge Acquisition, Vol. 5, pp.199 -220.

[9] Mahesh K., 1996. Ontology Development for Machine Translation: Ideology and Methodology, New Mexico: Computing Research Laboratory, 87 P.

[10] Marco M., Marco M., Prieto J. y Segret R., 2010. Escaneando la informática. Catalunya: Editorial UOC, 166 P.

[11] Oficina de publicaciones oficiales de las comunidades europeas. Disponible en: (http://ec.europa.eu/europeaid/evaluation/methodology/tools/too_dcs_def_es.htm). Consultado el 10 de noviembre de 2010.

[12] OMG, Disponible en: http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF. Consultado el 10 de noviembre de 2010.

[13] Saiyd A., Said A. y Neaimi A., 2009. Towards an Ontological Concepts for Domain-Driven Software Design. En: First International Conference on Networked Digital Technologies, pp. 127 - 131.

[14] Steve G., Gangemi A., Pisanelli D., 1998. Ontology Integration: Experiences with Medical terminologies. En: N. Guarino (ed.). pp. 163 - 178.

[15] Wongthongtham P., Chang E. y Dillon T., 2006. Software Design Process Ontology Development, En: Workshops lecture notes in computer science, vol. 4278, pp. 1806 - 1813.

[16] Zapata C., Giraldo G., Portilla B., D. Gómez, Naranjo M. y P. Carmona, 2009. Aproximación a una ontología para lenguajes de modelado gráfico. En: Ingeniería Universidad de los Andes, Vol. 29, pp. 16 - 25.