GENERACIÓN AUTOMÁTICA DE PROTOTIPOS FUNCIONALES A PARTIR DE ESQUEMAS PRECONCEPTUALES

JOHN JAIRO CHAVERRA MOJICA

Ingeniero de Sistemas e Informática

Universidad Nacional de Colombia
Facultad de Minas – Escuela de Sistemas
Maestría en Ingeniería – Ingeniería de Sistemas
Área de Ingeniería de Software
Medellín, Colombia.
2011

GENERACIÓN AUTOMÁTICA DE PROTOTIPOS FUNCIONALES A PARTIR DE ESQUEMAS PRECONCEPTUALES

JOHN JAIRO CHAVERRA MOJICA

Trabajo de investigación presentado como requisito parcial para optar al Título Magister en Ingeniería – Ingeniería de Sistemas.

Este documento tiene únicamente propósitos de evaluación y no debería ser consultado o referido por cualquier persona diferente a los evaluadores.

Director: CARLOS MARIO ZAPATA JARAMILLO, Ph.D.

Universidad Nacional de Colombia
Facultad de Minas – Escuela de Sistemas
Maestría en Ingeniería – Ingeniería de Sistemas
Área de Ingeniería de Software
Medellín, Colombia.
2011

DEDICATORIA

A mis padres, Jairo y Blanca,

por su amor incondicional y apoyo en el deseo de cumplir mis sueños.

A mis hermanos, Yeferson y Cirly,

por el apoyo y el respaldo que siempre me brindan.

John J.

AGRADECIMIENTOS

Estas son algunas de las personas y entidades que hicieron realidad este logro que inicialmente parecía una locura, para todos ellos muchas gracias.

Mi mayor agradecimiento para el Doctor Carlos Mario Zapata Jaramillo, quien puso a mi disposición todo su apoyo y conocimientos, no solo académico sino también vivenciales para llevar a cabo este trabajo, enseñándome el día a día la tarea de la Investigación. Agradezco por su enorme paciencia para conmigo, por sus sabios consejos en la hora precisa. Además porque se convirtió en más que un director de Tesis en un amigo, un padre.

Agradezco las enseñanzas, apoyo y colaboración de los profesores Gloria Lucía Giraldo Gómez, Juan David Velásquez y John William Branch, quienes me brindaron su experiencia y conocimientos para llevar a feliz término esta meta.

Adicionalmente, quiero extender este agradecimiento a los alumnos Katherine Villamizar, Santiago Londoño, Heidy Villa, Edward Naranjo y Bryan Zapata, por sus aportes y críticas constructivas hechas en el proceso de investigación. Además agradezco el apoyo recibido por el proyecto de investigación "TRANSFORMACIÓN SEMIAUTOMÁTICA DE LOS ESQUEMAS CONCEPTUALES, GENERADOS EN UNC-DIAGRAMADOR, EN PROTOTIPOS FUNCIONALES" financiado por la Vicerrectoría de Investigación de la Universidad Nacional de Colombia.

Finalmente agradezco a la Universidad Nacional de Colombia, Sede de Medellín por poner a disposición de los alumnos de Maestría toda una infraestructura. Agradezco por la financiación parcial en la divulgación de nuestra investigación en la Novena Conferencia Iberoamericana en Sistemas, Cibernética e Informática: CISCI 2010. Orlando, Florida.

TABLA DE CONTENIDO

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

ABREVIATURAS

- JSP: *Java Server Pages*
- PHP: *Hypertext Preprocessor*
- HTML: *HyperText Markup Language*
- XHTML: *eXtensible Hypertext Markup Language*
- XMI: *eXtensible Markup Language*
- CASE: *Computer-Aided Software Engineering*
- OMG: *Object Management Group*
- UML: *Unified Modeling Language*
- MVC: *Model View Controller*
- RAD: *Rapid Application Development*
- GUI: *Graphical User Interface*
- SQL: *Structure Query Language*
- DDL: *Data definition Language*
- WYSIWIG: *What you see is what you get*
- *RADD*: *Rapid Application and Database Development*
- OMT: *Object Modeling Technique*
- IDE: *Integrated Development Environment*
- NLU: *Natural Language Understander*
- DMG: *Data Model Generator*
- OMT: *Object Modeling Technique*
- KAOS: *Knowledge Acquisition in Automated Specification*
- IDE: *Integrated Development Environment*
- EP: Esquema Preconceptual
- ACG: *Activity Chain Graphs*

RESUMEN

La Educción de Requisitos es una tarea compleja, dado que es en este proceso donde se establecen los elementos del aplicativo de software a desarrollar. Esta tarea, frecuentemente, presenta problemas de comunicación debido a las diferentes formaciones que tienen los analistas e interesados. Usualmente, la información recolectada en las entrevistas se suele plasmar en esquemas conceptuales, generalmente de UML. Aunque estos diagramas son estándar no permiten una validación del interesado debido a su complejidad, ya que son cercanos al lenguaje técnico del analista.

Una vez finalizada la Educción de Requisitos, se procede con la generación del código fuente de la aplicación. Con el fin de mejorar y agilizar este proceso existen varios métodos de desarrollo de software que impulsan la generación automática de código. Para tal fin, se utilizan las herramientas CASE convencionales, pero aún están muy distantes de exhibir un proceso automático y muchas de estas herramientas se complementan con algunos trabajos que se alejan de los estándares de modelado. La mayoría de estas herramientas CASE generan parte del código fuente, pero no generan completamente la aplicación de software funcional.

Con el fin de solucionar estos problemas, en esta Tesis, se propone un conjunto de reglas heurísticas para generar, automáticamente, una aplicación de software totalmente funcional a partir de Esquemas Preconceptuales bajo el patrón arquitectónico MVC, utilizando como lenguaje de programación PHP 5.x con XHTML. Además, se propone un conjunto de reglas heurísticas para generar, automáticamente, el diagrama entidad-relación y las sentencias DDL para el gestor de base de datos MySQL. Al utilizar los Esquemas Preconceptuales se mejora la comunicación con el interesado, dada la cercanía con el lenguaje natural que poseen estos esquemas. Adicionalmente, se mejora la calidad de las aplicaciones de software ya que es posible obtener una validación del interesado para dicho

diagrama durante todas las fases del desarrollo. Esta Tesis se complementa con la elaboración de una herramienta CASE en la cual se incorporan todas las reglas heurísticas definidas para la generación automática del código. El funcionamiento de esta herramienta se ejemplifica con un caso de laboratorio.

ABSTRACT

Requirements elicitation is a complex task, because in this process the elements of the software to-be-made are established. Frequently, this task is affected by communication problems, due to the fact that both analysts and stakeholders (the main actors of this task) have differences in training. Commonly, the information gathered during interviews is reflected into conceptual schemas, mainly UML diagrams. Even though UML diagrams are standardized, they are barely validated by stakeholders, because the UML diagrams are complex and nearer to the analyst technical language.

Once requirements elicitation task is completed, source code of the application can be developed. In order to improve and speed up this process, several software development methods searching for the automated generation of code are proposed. So, well-known CASE tools are employed, but they are far away from automated processes and, sometimes, they are non-standard modeling proposals. Most of these tools partially generated source code, but the resulting application is barely functional.

Trying to fix the above problems, in this Thesis I propose a set of heuristic rules for automatically generating a fully-functional software application from pre-conceptual schemas. Both the MVC architectural pattern and the XHTML-based PHP 5.x language are selected for this process. Also, I propose a set of heuristic rules for automatically generating the entity-relationship diagram and the DDL commands for constructing and using the MySQL database management system. The usage of pre-conceptual schemas improves the analyst-stakeholder communication process, because such schemas are closer to the natural language. Also, the stakeholder validation of pre-conceptual schemas we can achieve during all the phases of software development lifecycle improves the quality of the software application. This M. Sc. Thesis is complemented by the elaboration of a CASE

tool which includes all the defined heuristic rules for automated code generation. A lab case is used to exemplify the functioning of the above mentioned CASE tool.

I. INTRODUCCIÓN

La Educción de Requisitos es la tarea más compleja e importante del desarrollo de software. Por tanto, es, en este proceso, donde se establecen todos los elementos que harán parte del producto a desarrollar (Leite, 1987; Bennett, 1997; Kotonya y Sommerville, 1998). Mediante la Educción de Requisitos se procura capturar, analizar, sintetizar y convertir a esquemas conceptuales las características que un interesado describe acerca de un dominio. Esta tarea, que se realiza interactivamente mediante entrevistas en lenguaje natural entre el analista e interesado, suele presentar problemas de comunicación originados en la diferencia de especialidades de los participantes (Leffingwell y Widrig, 2003) y en el desconocimiento del interesado en relación con la idea específica de lo que necesita. Además, el interesado no posee una cultura que le permita simplificar la información, lo que conlleva a que el analista cometa errores de interpretación y, por ende, estos errores se reflejan en el producto final (Sommerville, 2007). Tradicionalmente, estos problemas, en Ingeniería de Software, se suelen solucionar empleando métodos de desarrollo tales como XP (*Extreme Programming*), RUP (*Rational Unified Process*) y AUP (*Agile Unified Process*), entre otros.

Comúnmente, a partir de las entrevistas con los interesados, los analistas presentan la información obtenida en artefactos denominados esquemas conceptuales. Los que más se utilizan son los diagramas de UML (*Unified Modeling Language*), que es un lenguaje gráfico para visualizar, especificar, construir y documentar sistemas (Fowler y Scott, 1999) con el fin de obtener la documentación técnica de un aplicativo de software y facilitar el desarrollo, pruebas y mantenimiento del mismo. Estos diagramas requieren, para su elaboración, lectura, análisis y validación, ciertos conocimientos técnicos que, usualmente, no posee el interesado, lo cual dificulta la validación de la información en las primeras etapas del desarrollo de software (Zapata *et al.*, 2006).

Cuando se completa la fase de Educción de Requisitos, se procede con la generación del código fuente. Con el fin de agilizar y mejorar este proceso, diferentes organizaciones emplean métodos de desarrollo de software que promueven la generación automática del código fuente. Para tal fin, los analistas vienen utilizando con mayor frecuencia las herramientas CASE (*Computer-Aided Software Engineering*) convencionales, las cuales, usualmente, emplean los diagramas UML como punto de partida, pero estas herramientas aún están muy distantes de tener un proceso automático (Zapata *et al.*, 2007). Aunque el proceso es semiautomático, persisten errores debido a la interpretación subjetiva del dominio que debe realizar el analista. Además, muchas de estas herramientas CASE se complementan con algunos trabajos que se alejan de los estándares de modelado. Los trabajos que incentivan la generación automática y semiautomática de código fuente, se pueden agrupar en cuatro categorías.

Existe un primer grupo de proyectos que utiliza las especificaciones formales como punto de partida para lograr la generación automática del código fuente de una aplicación de software. En este grupo existe un subgrupo que complementa sus propuestas con desarrollo de herramientas CASE, las cuales utilizan como punto de partida discursos el lenguaje natural o controlado para la generación de esquemas conceptuales. Esta tendencia surge de la necesidad de apoyar al analista en la comprensión de las necesidades del interesado, que se suelen expresar mediante discursos en lenguaje natural, algunas de estas herramientas son: NL-OOPS (*Natural Language Object-Oriented*), LIDA (*Linguistic assistant for Domain Analysis*), CM-Builder (*Class Model Builder*), RADD (*Rapid Application and Database Development*) y NIBA. Estas herramientas generan parte de una aplicación de software (los esquemas conceptuales, normalmente los de UML) y, aunque estos esquemas son estándar en el desarrollo de software, no son suficientes para tener una aplicación totalmente funcional. Además, estas herramientas aún no se ligan con las herramientas

CASE convencionales para generar el código correspondiente. En el caso de NIBA es posible obtener la estructura básica del código (las clases).

Otro grupo de proyectos se enfoca en generar, automáticamente, el código fuente desde esquemas conceptuales (clases, casos de uso, entidad-relación y máquinas de estados, entre otros), pero lo hacen para lenguajes específicos y, también, se afectan con los problemas de consistencia que pueden acarrear los esquemas conceptuales de partida. Generalmente, estos esquemas no contienen toda la información necesaria para generar un prototipo funcional, además de que estos diagramas no son de fácil comprensión para un interesado, lo cual dificulta la validación de la información en las primeras etapas del desarrollo de software.

Un tercer grupo, además de generar la estructura básica del código fuente, genera las interfaces gráficas de usuario (GUI). Para tal fin, estos proyectos utilizan diagramas tales como casos de uso, escenarios y componentes de patrones de diseño. También, existen las herramientas denominadas RAD (*Rapid Application Development*) en las cuales se construye una interfaz rápidamente mediante el paradigma WYSIWIG (*What you see is what you get*). En estas herramientas se pueden cometer errores debido a que las interfaces están sujetas a la decisión del diseñador, sin tener en cuenta ningún patrón de diseño.

Por último, existe un grupo de proyectos que, además de generar la estructura básica de la aplicación, genera el cuerpo de los métodos. Algunos de ellos utilizan diagramas intermedios tales como secuencias, actividades y máquina de estados, entre otros, y otros acuden a procesos predefinidos tales como: *log-in* y funciones básicas para crear, editar o eliminar entidades (conceptos). Pese a estos esfuerzos, la generación del cuerpo de los métodos sigue siendo deficiente en cuanto al detalle, pues la sintaxis y semántica definida

en estos diagramas no contiene la suficiente información para la especificación total del cuerpo del método.

Con el fin de agilizar y mejorar el proceso de Educción de Requisitos de manera tal que el interesado pueda interactuar constantemente con los analistas y pueda validar los requisitos, Zapata *et al.* (2006) desarrollaron los Esquemas Preconceptuales. Mediante estos esquemas, por ser un lenguaje de modelado gráfico, se permite plasmar el discurso del interesado de forma cercana al lenguaje natural, facilitando el entendimiento y la validación de la información allí contenida. Los Esquemas Preconceptuales se conciben como un paso intermedio entre el discurso natural del interesado y los esquemas conceptuales del UML.

En esta Tesis se define un conjunto de reglas heurísticas para generar, automáticamente, prototipos ejecutables a partir de Esquemas Preconceptuales. Para tal fin, se genera código fuente bajo el patrón MVC (*Model View Controller*) en lenguajes de programación PHP 5.x con XHTML, y JSP, además de generar el diagrama entidad-relación y las correspondientes sentencias DDL (*data definition language*) para el gestor de base de datos MySQL

Los principales aportes de esta Tesis son:

- Definición gráfica para representar los tipos de datos en el Esquema Preconceptual.
- Definición gráfica para representar conceptos obligatorios en el Esquema Preconceptual.
- Definición gráfica para especificar el cuerpo de los métodos de una aplicación, en el Esquema Preconceptual.
- Definición de reglas heurísticas para obtener el diagrama entidad-relación y sus correspondientes sentencias DDL a partir de Esquemas Preconceptuales.

- Implementación de los artefactos descritos en una herramienta CASE, que se encarga de validar el Esquema Preconceptual y generar, automáticamente, una aplicación funcional bajo el patrón de programación MVC para lenguajes de programación PHP 5.x con XHTML y JSP.

Con estos aportes se pretende reducir el tiempo requerido para la elaboración de las aplicaciones de software y generar un código fuente con las reglas estándar de programación que, además, sea consistente con los diagramas resultantes. Finalmente, se pretende el mejoramiento de la calidad de las aplicaciones de software, la cual se entiende como la carencia de errores en corrección (la utilización de la sintaxis adecuada) y consistencia (la representación de un mismo elemento en diferentes diagramas) (Leffingwell y Widrig, 2003).

Al utilizar los Esquemas Preconceptuales se mejora la comunicación entre analistas e interesados, acercando el lenguaje técnico del analista al lenguaje natural del interesado. De esta manera, se posibilita obtener una validación de la documentación técnica del interesado durante todas las fases del desarrollo de software, lo cual conlleva a obtener un producto de software de alta calidad.

Esta Tesis se complementa con la elaboración de una herramienta CASE en la cual se incorporan todas las reglas heurísticas definidas y, además, se incorporan las reglas definidas en Zapata *et al.* (2007) para la obtención automática del diagrama de clases. Esta herramienta CASE se desarrolla en PHP 5.x con JavaScript, jQuery y JGraph**.**

El caso de laboratorio se refiere al desarrollo de un aplicativo de software denominado InnSoftware (Software para el diligenciamiento de encuestas con el fin de medir la capacidad de innovación en las empresas de software, que corresponde a un proyecto

conjunto entre la Universidad Nacional de Colombia, Antójate de Antioquia, Colciencias e InterSoftware).

Esta Tesis se estructura de la siguiente forma: en el Capítulo II se define el marco teórico que agrupa los conceptos de este dominio; en el Capítulo III se realiza un análisis descriptivo de los aportes más relevantes en generación automática de código, complementado con un análisis crítico de la literatura; en el Capítulo IV se realiza la Definición del Problema de Investigación que motiva esta Tesis; en el Capítulo V se presenta la propuesta de solución encontrada al problema planteado; en el Capítulo VI se presenta el caso de laboratorio relacionado con un sistema de encuestas. Finalmente, en el Capítulo VII se resumen los hallazgos y se presenta el trabajo futuro que se puede derivar de esta propuesta.

II. MARCO CONCEPTUAL

2.1 Educción de Requisitos

El análisis de requisitos lo componen la educción, modelado, validación y especificación de requisitos. En la educción de requisitos se definen las características que tendrá el producto software a desarrollar, en el modelado se organizan de forma coherente, en la validación se priorizan y se solucionan los conflictos encontrados. Por último se deben plasmar los requisitos en el documento de especificación (Leite, 1986).

El IEEE (*Institute of Electrical and Electronics Engineers*) define el estándar 830, el cual establece las características que debe tener una especificación de software. De acuerdo con esto, la especificación debe ser (IEEE, 1998):

- Correcta (cada requisito se debe encontrar en el software).
- No ambigua (un requisito solo tiene un único significado).
- Completa (todos los elementos se deben especificar).
- Consistente (los requisitos no pueden ser contradictorios).
- Jerarquizada (se pueden clasificar los requisitos por su importancia).
- Verificable (se puede demostrar que el sistema cumple con los requisitos).
- Modificable (los cambios en los requisitos se pueden estructurar fácilmente).
- Trazable (se puede identificar el origen y los elementos del sistema que lo satisfacen).

La educción de requisitos se considera una de las tareas más delicadas del desarrollo de software, dado que los interesados suelen expresar los requisitos de manera muy general o en su propia terminología. También, es común que el interesado no tenga claridad de lo que

desea y cómo lo desea. Esto ocasiona especificaciones inadecuadas o redundantes (Leffingwell y Widrig, 2003). Comúnmente, esta tarea se realiza mediante un diálogo en lenguaje natural entre el analista y el interesado, para determinar las características que tendrá el producto software a desarrollar (Leite, 1987).

2.2 UML (*Unified Modeling Language*)

UML es una notación estándar para el modelado de sistemas. Según sus creadores, UML es una unificación, pero no define una semántica precisa para esa notación (Molina, 2009). Tal y como lo especifica el OMG (2010), la especificación recoge sólo la notación, pero su uso puede generar toda una diversidad de usos de los mismos diagramas.

Fowler (2004) define UML como una "familia de notaciones gráficas, apoyadas en un metamodelo sencillo, que facilita la descripción y diseño de sistemas de software, particularmente sistemas de software construidos usando el paradigma de programación orientado a objetos". La jerarquía de los diagramas de la versión UML 2.0 (OMG, 2006) se muestra en la Figura 1.

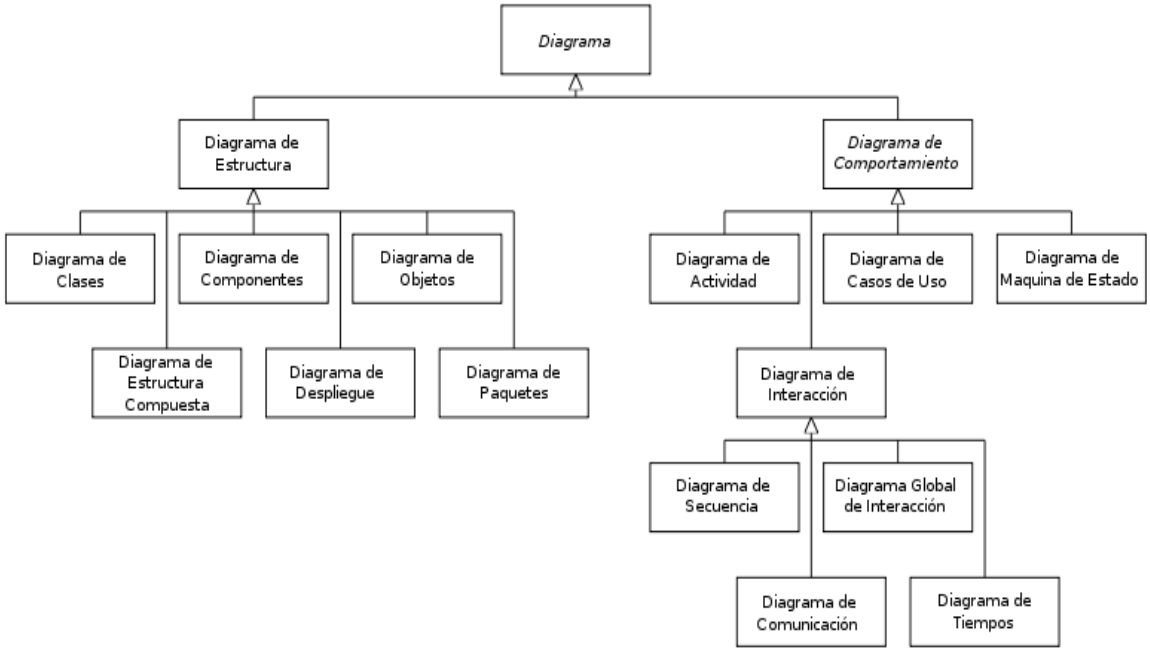


Figura 1. Jerarquía de Diagramas de la Superestructura de UML 2.0 (Fowler, 2004).

2.3 Herramienta CASE (*Computer-Aided Software Engineering*)

Conjunto de aplicaciones informáticas que dan asistencia a los analistas y desarrolladores, durante todo el ciclo de vida del software. Estas herramientas se destinan a aumentar la productividad, mejorar la calidad en el desarrollo de software y reducir costos (Burkhard y Jenster, 1989).

Las herramientas CASE se pueden clasificar teniendo en cuenta los siguientes parámetros (aunque no existe una única forma de clasificarlas):

- Las plataformas que soportan.
- Las fases del ciclo de vida que cubren en el desarrollo de sistemas.
- La arquitectura de las aplicaciones que producen.
- Su funcionalidad.

La siguiente clasificación es la más habitual basada en las fases del ciclo de desarrollo que cubren (Booch *et al.*, 1999):

- *Upper* CASE (U-CASE): herramientas que ayudan en las fases de planificación y análisis de requisitos, usando, entre otros, diagramas UML.
- *Middle* CASE (M-CASE): herramientas para automatizar tareas en el análisis y diseño.
- *Lower* CASE (L-CASE): herramientas que soportan la generación semiautomática de código, la depuración de programas y la realización de pruebas. Aquí, se pueden incluir las herramientas de desarrollo rápido de aplicaciones.

Existen otros nombres que se les pueden dar a este tipo de herramientas, en una clasificación que no es excluyente con la anterior (Booch *et al.*, 1999):

- *Integrated* CASE (I-CASE): herramientas que engloban todo el proceso de desarrollo software, desde el análisis hasta la implementación.
- MetaCASE: herramientas que permiten la definición de técnicas personalizadas de modelado, con elementos propios, restricciones y relaciones posibles.
- CAST (*Computer-Aided Software Testing*): herramientas de soporte al proceso de pruebas del software.
- IPSE (*Integrated Programming Support Environment*): herramientas que soportan todo el ciclo de vida, incluyen componentes para la gestión de proyectos y gestión de la configuración.

2.4 MVC (*Model View Controller*)

MVC es un patrón de diseño que considera dividir una aplicación en tres módulos claramente identificables y con funcionalidad bien definida: el Modelo, las Vistas y el Controlador (Reenskaug, 1979). El Modelo corresponde a la información, las Vistas corresponden a la presentación o interacción con el usuario y el Control corresponde al comportamiento. La comunidad de desarrolladores de Smalltalk-80 fue quien popularizó, inicialmente, este patrón.

2.4.1.1 Model

El Modelo es la representación específica de la información del dominio con la cual el sistema opera. El modelo es la estructura básica del sistema y puede ser tan simple como un número entero (un contador, por ejemplo) o tan complejo como una instancia de una subclase (Krasner y Pope, 1998).

2.4.1.2 View

La vista representa la interfaz gráfica de usuario, que es la encargada de la interacción con el usuario final. Además, puede solicitar y presentar los datos del modelo que requiere el usuario final para realizar una acción (botón o *link*). No sólo contienen los componentes necesarios para la visualización, sino que también puede contener subvistas y formar parte de supervistas. La supervista ofrece la posibilidad de realizar transformaciones gráficas entre los niveles de esta subvista (Krasner y Pope, 1998).

2.4.1.3 Controller

Los controladores contienen la interfaz de sus modelos asociados e interfaces gráficas que controlan los dispositivos de entrada (teclado, mouse). Generalmente, responden a las acciones del usuario mediante la interfaz gráfica. Los controladores también se ocupan de la programación de interacciones con otros controladores. Las opciones de menú se pueden considerar como dispositivos de entrada (Krasner y Pope, 1998).

2.5 Entidad-Relación

En 1976 Peter Chen publicó el Modelo Entidad-Relación original, que proveía un enfoque visual fácil de usar del diseño lógico de la base de datos (Chen, 1976). El modelo elude las complicaciones de almacenamiento y consideraciones de eficiencia, las cuales se reservan para el diseño físico de la base de datos. Algunos autores extendieron el Modelo entidad-relación, incrementando sus capacidades y haciéndolo más apropiado para sus requisitos particulares. La extensión más comprehensiva incluye iconos dinámicos como una adaptación al modelado de bases de datos orientadas a objetos (Saiedian, 1995).

El esquema de negocio, que se expresa como un diagrama entidad-relación, es un diseño conceptual de la base de datos y es una representación pura del mundo real. Además, es independiente del almacenamiento y las consideraciones de eficiencia.

2.5.1 Componentes del Modelo Entidad-Relación

- Entidad: Representada por un rectángulo con el nombre del tipo-entidad dentro de él.

- Relación: Representada por un diamante, con el nombre de la relación dentro. Los tipos-entidad relacionados se conectan al diamante con líneas rectas. Cada línea se marca con un "1", "N" o "M" para indicar relaciones del tipo 1:1, 1:N o M:N.
- Atributo: Representado por un círculo con el nombre del tipo de dato dentro, conectado con una flecha a su tipo-entidad.

Cada entidad debe tener un único identificador que la distinga de las demás entidades. El identificador único pude ser un atributo ya en uso, como el nombre del empleado, o puede ser un atributo agregado por su unicidad, como el número de identificación del empleado. Chen compara el identificador de la entidad con el concepto de *clave primaria* en las bases de datos convencionales

En la Figura 2 se presenta un ejemplo de un diagrama entidad-relación para una base de datos universitaria.

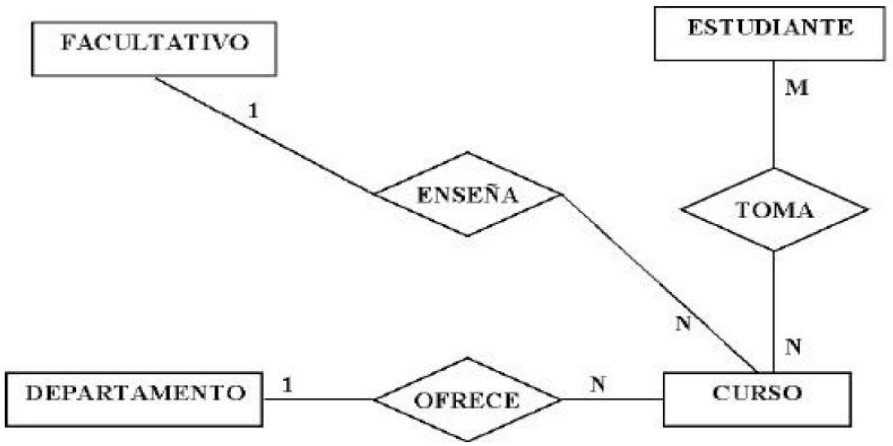


Figura 2. Un diagrama entidad-relación para una base de datos universitaria (Saiedian, 1995)

2.6 Esquema Preconceptual

Los Esquemas Preconceptuales son representaciones gráficas de los discursos que se expresan en UN-Lencep (Universidad Nacional de Colombia—Lenguaje Controlado para la Especificación de Esquemas Preconceptuales) y que permiten generar, automáticamente, cinco diagramas de UML 2.0: clases, comunicación, máquinas de estados, secuencias y casos de uso, además del diagrama de objetivos de KAOS (*Knowledge Acquisition in Automated Specification*) y los diagramas de OO-Method. Los Esquemas Preconceptuales posibilitan la frecuente interacción entre personas de orientación técnica (analistas) y los conocedores del dominio (interesados) (Zapata, 2007).

Según Zapata (2007), se prefiere el nombre "Esquema Preconceptual" pues representa un dominio de aplicación, expresando el conocimiento previo y definiendo la terminología propia del área, en lugar de establecer el conjunto de características previas de diseño de una solución informática. Así, un Esquema Preconceptual se constituye en un facilitador de la traducción del discurso inicial sobre el dominio y los diferentes esquemas conceptuales que se elaboran en las fases de análisis y diseño de una aplicación.

Los elementos que conforman el Esquema Preconceptual se clasifican en dos categorías, los elementos básicos para la representación del dominio de un área y los elementos avanzados, lo cuales se utilizan para realizar la especificación de cada proceso del sistema. A continuación se especifican cada uno de los elementos del Esquema Preconceptual:

2.6.1 Elementos Básicos

2.6.1.1 Concepto

Según Zapata *et al.* (2006a, 2006b y 2006c), los conceptos son sustantivos o frases nominales del discurso del interesado que representan un actor, categoría,

característica o subcaracterística del dominio del problema. Toda interacción con un concepto se hará con una conexión (Zapata y Arango, 2007). La representación gráfica de un concepto se muestra en la Figura 3 y en la Figura 4 se puede apreciar un ejemplo de su uso.

Figura 3. Representación gráfica de un Concepto (elaboración propia).

Figura 4. Ejemplo del elemento Concepto (elaboración propia).

2.6.1.2 Relación Estructural

En dicha relación se expresan los verbos "tiene" y "es", los cuales generan una relación de dependencia entre los conceptos involucrados, formando lo que se denomina una tríada estructural (Zapata, 2007).

La representación gráfica de una Relación Estructural se muestra en la Figura 5 y en la Figura 6 se puede apreciar un ejemplo de su uso.

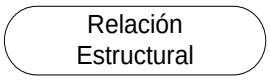


Figura 5. Representación gráfica de una Relación Estructural (elaboración propia).

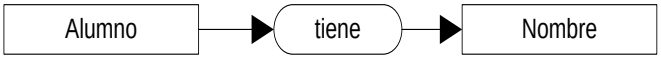


Figura 6. Ejemplo de una Tríada Estructural (elaboración propia).

2.6.1.3 Relación Dinámica

Son verbos conjugados que demuestran las acciones, operaciones o funciones que se realizan en el dominio del problema. Para este elemento se suelen utilizar verbos de acción o ejecución, por ejemplo '*Crea*' o '*Envia*' (Zapata y Arango, 2007).

La representación gráfica de una Relación Dinámica se muestra en la Figura 7 y en la Figura 8 se puede apreciar un ejemplo de su uso.

Figura 7. Representación gráfica de una Relación Dinámica (elaboración propia).

Figura 8. Ejemplo de una Relación Dinámica (elaboración propia).

2.6.1.4 Nota

Una Nota permite especificar una lista de posibles valores que puede tomar un concepto, en el marco del dominio del problema. Representa las instancias o características fijas de un elemento (Zapata y Arango, 2007).

En la Figura 9 se observa la representación gráfica de una nota y en la Figura 10 se puede apreciar un ejemplo de su uso.

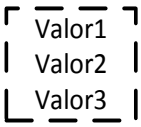


Figura 9. Representación gráfica de una Nota (elaboración propia).

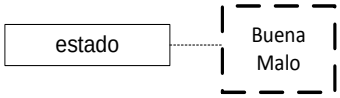


Figura 10. Ejemplo de una Nota (elaboración propia).

2.6.1.5 Condicional

Estructura conformada por una expresión que puede contener conceptos y operadores entre ellos, como condición para que se lleve a cabo una tríada dinámica (Zapata y Arango, 2007).

En la Figura 11 se observa la manera de representar un condicional y en la Figura 12 se puede apreciar un ejemplo de su uso.

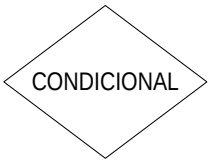


Figura 11. Representación gráfica de un Condicional (elaboración propia).

Figura 12. Ejemplo de un Condicional (elaboración propia).

2.6.1.6 Conexión

Son flechas que unen los conceptos con relaciones estructurales o relaciones dinámicas, creando asociación entre ellos y denotan la conformación de la frase

(Zapata y Arango, 2007). La representación gráfica de la Conexión se puede observar en la Figura 13.

Figura 13. Representación gráfica del elemento Conexión (elaboración propia).

2.6.1.7 Implicación

Representa una relación de consecuencia o causa-efecto entre tríadas dinámicas o entre condicionales y tríadas dinámicas, siendo la primera tríada dinámica el antecedente y la segunda el consecuente (Zapata, 2007). La secuencia muestra el orden en que se deben realizar dichas acciones (Zapata y Arango, 2007).

La representación gráfica de una Implicación se puede observar en la Figura 14 y en la Figura 15 se puede apreciar un ejemplo de su uso.

Figura 14. Representación gráfica del elemento Implicación (elaboración propia).

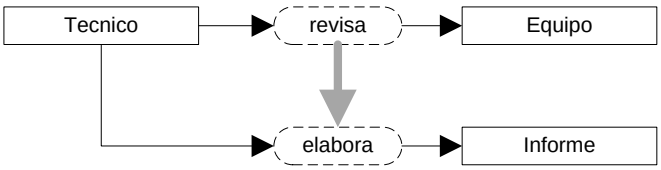


Figura 15. Ejemplo de una Implicación (elaboración propia).

2.6.1.8 Referencia

Debido a que en el Esquema Preconceptual que representa el dominio de un problema puede llegar a ser muy grande, es necesario utilizar el elemento "referencia" para ligar conceptos y/o relaciones distantes en el esquema (Zapata, 2007).

La representación gráfica de una Referencia se puede observar en la Figura 16 y en la Figura 17 se puede apreciar un ejemplo de su uso.

Figura 16. Representación gráfica del elemento Referencia (elaboración propia).

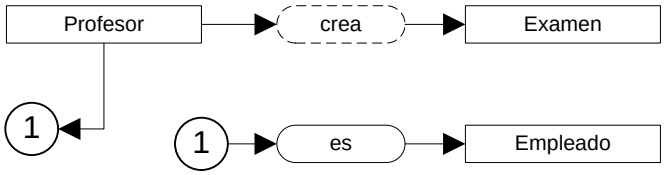


Figura 17. Ejemplo de una Referencia (elaboración propia).

2.6.1.9 Marco

Mediante un Marco, se pueden agrupar conceptos, como características de un concepto. También se pueden agrupar triadas dinámicas (Zapata *et al.*, 2010).

La representación gráfica de un Marco se puede observar en la Figura 18 y en la Figura 19 se puede apreciar un ejemplo de su uso.

Figura 18. Representación gráfica del elemento Marco (elaboración propia).

Figura 19. Ejemplo de un Marco (elaboración propia).

En la Figura 19 se aprecia un Marco que agrupa las relaciones dinámicas: *registra*, *edita* y *elimina*. Esta agrupación significa que un Profesor puede realizar estas tres acciones sobre un Alumno. De manera similar se interpreta que el Profesor tiene las características (nombre y teléfono) descritas en el Marco de la Figura 19.

2.6.2 Elementos Avanzados

2.6.2.1 Tiene_Unico

El TIENE_UNICO (véase la Figura 20) se refiere aquella relación estructural de la cual se puede obtener un identificador único (concepto destino) del concepto origen. Uno o más conceptos pueden componer dicho identificador único (identificador único compuesto) (Ruiz, 2008).

En la Figura 20 se puede apreciar un ejemplo de su uso.

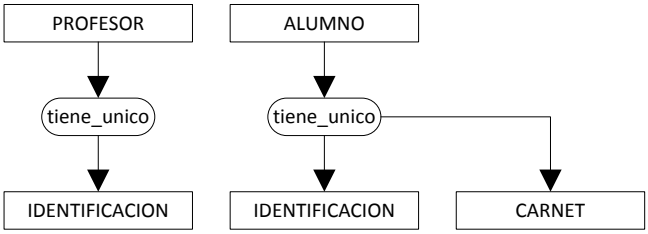


Figura 20. Ejemplo del elemento TIENE_UNICO (elaboración propia).

2.6.2.2 Atributo Compuesto

El atributo compuesto se utiliza para referirse a atributos pertenecientes a un concepto específico, dado que en el Esquema Preconceptual se pueden repetir conceptos, pero con propósitos diferentes (Zapata *et al.*, 2010).

La representación gráfica de un Atributo Compuesto se puede observar en la Figura 21 y en la Figura 22 se puede apreciar un ejemplo de su uso.

Figura 21. Representación gráfica del elemento Compuesto (elaboración propia).

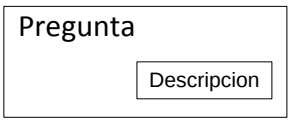


Figura 22. Ejemplo de una Compuesto (elaboración propia).

III. REVISIÓN DE LITERATURA

En este capítulo se presenta un análisis descriptivo de distintas aproximaciones relacionadas con la generación automática de código fuente. El objetivo de este análisis es contextualizar adecuadamente la Tesis, destacando sus aportes frente a las propuestas existentes.

La discusión se categoriza de acuerdo con las falencias identificadas en cada uno de los trabajos relacionados con generación automática de código.

3.1 Lenguaje controlado o especificaciones formales como punto de partida

La generación automática de código a partir de especificaciones formales conforma dos grupos importantes. Un primer grupo, en la parte académica, donde se definen conjuntos de reglas heurísticas y se presentan casos de estudio mediante los cuales evidencian sus ventajas. El otro grupo, además de las reglas heurísticas, desarrolla herramientas CASE como apoyo al proceso para agilizar y evitar errores en la aplicación de las reglas heurísticas.

Gomes *et al.* (2007) y Mammar y Laleau (2006) proponen una metodología para la generación automática de código. Gomes *et al.* (2007) lo hacen para el lenguaje de programación Java y, Mammar y Laleau lo hacen para el lenguaje de programación C. Ambos grupos utilizan como punto de partida las especificaciones formales, utilizando el Método-B. Este método provee un formalismo para el refinamiento de las especificaciones

y se estructura en módulos que se clasifican de acuerdo con su nivel de abstracción: máquina, refinamiento e implementación.

El desarrollo de sistemas de información requiere mecanismos para: la construcción modular, la reutilización de código y para el encapsulamiento de la información. Métodos formales como B (Abrial, 1996), Z (Spivey, 1998) y VDM (Jones, 1990), proveen dichos mecanismos.

El método B es el más reciente de dichos métodos. Lo diseñó Jean-Raymond Abrial, quien participó también en la concepción del método Z. El método B es una notación estándar que utiliza las *máquinas abstractas* para especificar, diseñar e implementar sistemas. Una máquina abstracta es una entidad que guarda información y provee mecanismos de servicios. En terminología del método B, la información se guarda en *variables* y los servicios son *operaciones*. Las operaciones constituyen la interfaz de una máquina, es decir diferentes maquinas interactúan únicamente por medio de sus operaciones.

En la Figura 23 se puede observar una máquina abstracta, la cual hace parte de una especificación formal de un sistema de transporte.

```
MACHINE Transport
VARIABLES
    balance
INVARIANT
    balance : NAT
INITIALISATION
    balance := 0
OPERATIONS
    addCredit (cr) =
      PRE
        cr : NAT &
        cr > 0 &
        balance + cr <= MAX_DATA
      THEN
        balance := balance + cr
      END
END
```

Figura 23. Especificación formal de un sistema de transporte (Gomes *et al.,* 2007)

Utilizando la especificación formal definida en la Figura 23 y, aplicando las reglas definidas en Gomes *et al.* (2007), se obtiene el siguiente código en lenguaje de programación Java (véase la Figura 24).

```
import javacard.framework.*;
public class Transport extends Applet {
    //The current amount of credit

   private short balance;
   //The INS code of addCredit method
   public static final ADD_CREDIT = 0x01;

   /** Constructor method */
   private Transport(byte[] bArray, short bOffset, byte bLength) {
       balance = 0;
       register();
   }
   /** Invoked by the JCRE, install is the applet entry point. It
       creates an applet instance and registers it in the
       JCRE through the invocation of the applet constructor method. */
   public static void install(byte[] bArray, short bOffset, byte bLength) {
       new Transport(bArray, bOffset, bLength);
   }
   /** process receives an APDU object and selects the instruction
       specified in its INS filed. */
   public void process (APDU apdu) {
       byte[] buffer = apdu.getBuffer();
       switch (buffer[ISO7816.OFFSET_INS]) {
           case ADD_CREDIT:
               addCredit(apdu);
   }
   /** The method addCredit adds some data to the balance attribute. */
   public final void addCredit(APDU apdu) {
       byte[] buffer = apdu.getBuffer();
       byte bytesRead = (byte) apdu.setIncomingAndReceive();
```

Figura 24. Código resultante de aplicar las reglas definidas a una especificación formal definida en el método B (Gomes *et al.,* 2007)

Ramkarthik y Zhang (2006) definen un conjunto de reglas heurísticas para generar, automáticamente, la estructura básica (clases) de una aplicación en Java, utilizando como punto de partida las especificaciones formales escritas en subconjunto del lenguaje Object-Z (notación para las especificaciones formales). Object-Z es una extensión orientada a objetos del lenguaje de especificaciones formales Z. A éste se le añaden las nociones de clases, objetos, herencia y polimorfismo. En la Figura 25 se presenta una descripción en Object-Z y en la Figura 26 las reglas que definieron Ramkarthik y Zhang para obtener las clases en Java a partir de las especificaciones en Object-Z.

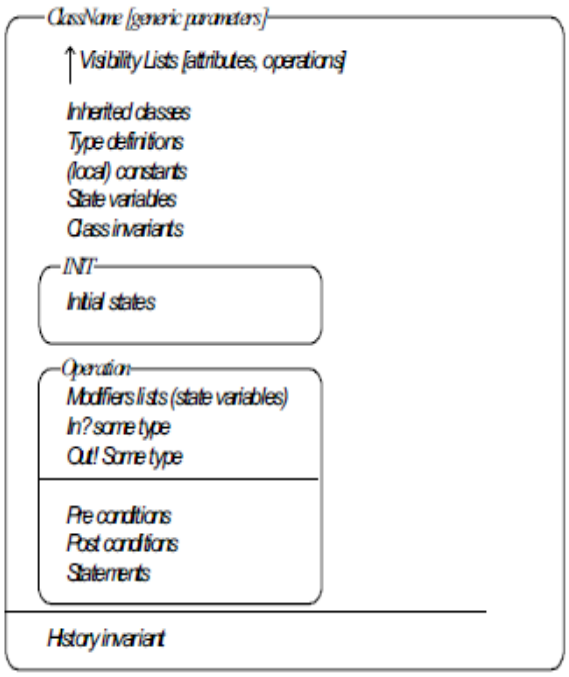


Figura 25. Especificación gráfica en Object-Z (Ramkarthik y Zhang, 2006)

Mapping Object Z into Java Skeletal Class	
Object Z	**Java skeletal code**
Class Schema Name	Public class "classname"
Class Constants	Final class constant values
Class variables	Class variables
Class invariant	Code for checking the invariant at object initialization and before and after each method invocation.
Initialization schema	Constructor of the Java class
Operations schema 1. Modifiers List 2. Pre conditions 3. Post conditions	Methods in the class 1. List of variables that does or doesn't change 2. Code for checking pre-conditions that need to be satisfied before the method gets executed. 3. Code for checking post-conditions that must be guaranteed after the method gets executed.
Visibility List	"public, private, protected" modifiers for methods and variables

Figura 26. Reglas de transformación de un subconjunto en Object-Z a Java (Ramkarthik y Zhang, 2006).

Gangopadhyay (2001) propone una herramienta CASE para la obtención del diagrama entidad-relación a partir de un lenguaje controlado, la cual se muestra en la Figura 27. Esta

herramienta emplea un diagrama de dependencias conceptuales como representación intermedia a partir del lenguaje controlado y un *parser* basado en una red de transición aumentada (una especie de diagrama de máquina de estados) para el procesamiento de las diferentes palabras. Este *parser* emplea un lexicón para la identificación de las diferentes categorías gramaticales en las cuales se pueden clasificar las palabras. Esta propuesta únicamente obtiene el diagrama entidad-relación, el cual, por sí solo, no permite determinar todas las características del software por construir.

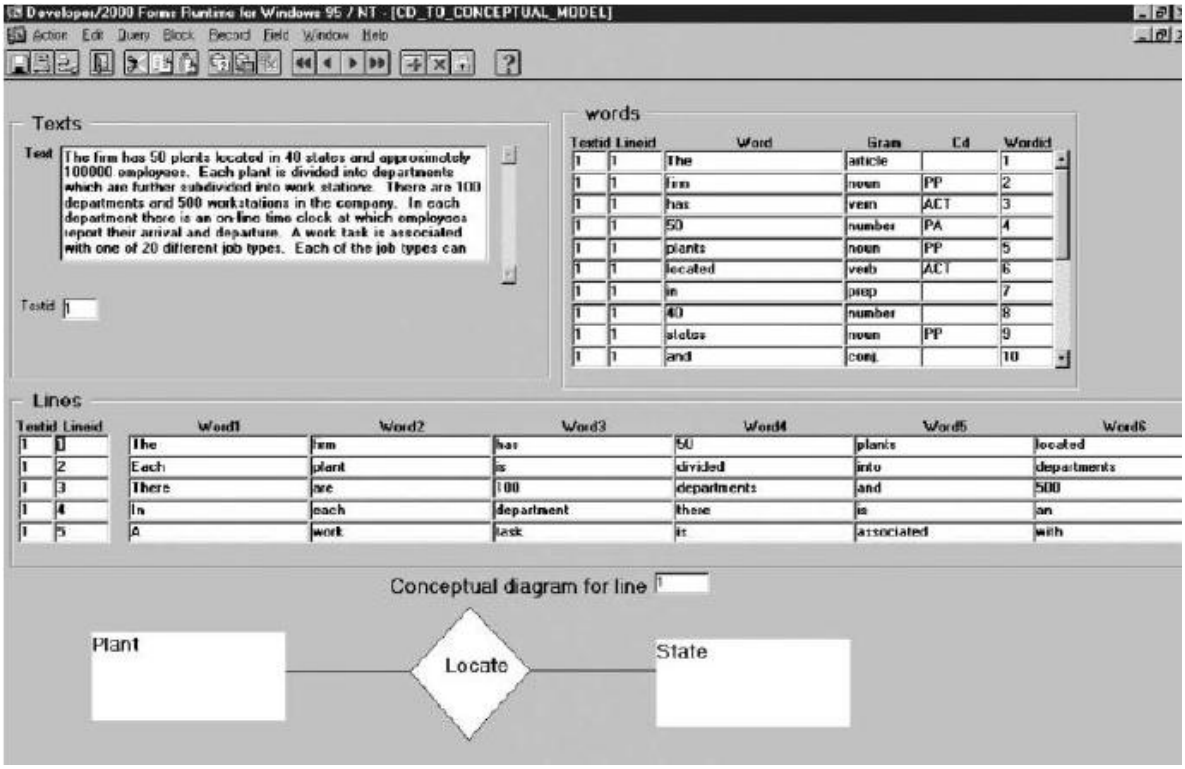


Figura 27. Herramienta CASE para generar el diagrama entidad-relación desde lenguaje controlado (Gangopadhyay, 2001).

Con el fin de mejorar el proceso de generación automática de código, diferentes autores vienen desarrollado un conjunto de herramientas CASE, las cuales procuran ayudar al analista en el proceso de conceptualización de los diagramas, es decir, en la interpretación del discurso del interesado y su traducción a los diferentes diagramas que de ese discurso pueden surgir. Alrededor de este tema, algunos de los principales proyectos son los siguientes:

- NL-OOPS (*Natural Languaje Object-Oriented Production System*) (Mich, 1996). Es una herramienta CASE basada en un sistema de procesamiento del lenguaje natural denominado LOLITA *(Large-scale Object-based Language Interactor, Translator and Analyser)*, el cual contiene una serie de funciones para el análisis del lenguaje natural (Mich, 1999) para apoyar el desarrollo del sistema desde las primeras fases de análisis de requisitos. NL-OOPS entrega, como resultado, una versión preliminar del diagrama de clases de OMT (*Object Modeling Technique*), un lenguaje de modelado previo a UML. El diagrama de OMT incluye un conjunto de las clases candidatas, posibles atributos y métodos de las mismas.

- LIDA (*Linguistic assistant for Domain Analysis*): es una herramienta CASE que es capaz de analizar un texto en lenguaje natural y hacer una clasificación en tres categorías gramaticales: sustantivo, verbo y adjetivo (Overmyer *et al.*, 2000). Con esta información, el analista debe asignar a cada palabra, manualmente, un elemento del diagrama de clases (por ejemplo, clase, atributo, operación o rol) de UML y, de esta manera, LIDA permite trazar el diagrama de Clases. En la Figura 28 se presenta una interfaz gráfica de la herramienta LIDA en la cual se realiza el procesamiento de un discurso y el resultado obtenido de las clases candidatas con sus respectivos atributos.

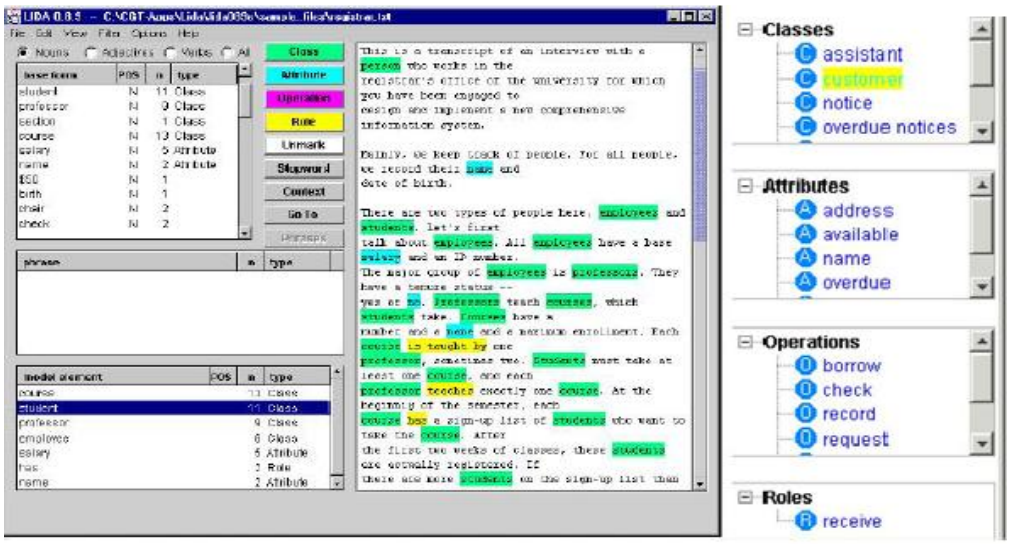


Figura 28. LIDA. Herramienta CASE para el procesamiento de Lenguaje Natural (Overmyer *et al.*, 2000).

- CM-Builder (*Class Model Builder*) (Harmain y Gaizauskas, 2000) es una herramienta CASE que permite la elaboración del diagrama de clases a partir de textos en inglés, utilizando como modelo intermedio una red semántica (Harmain y Gaizauskas, 2000). CM-Builder produce tres tipos de salidas: una lista de las clases candidatas, una lista de las relaciones entre las clases candidatas y un modelo conceptual en un formato llamado CDIF (*CASE Data Interchange Format*), el cual se puede ingresar directamente en la herramienta CASE *Select Enterprise Modeler* para el refinamiento del diagrama preliminar.

  En la Figura 29 se aprecian los pasos del enfoque que emplea CM-Builder para el procesamiento del Lenguaje natural, y, en la Figura 30 se muestra un ejemplo de la versión preliminar del diagrama de clases que genera CM-Builder.

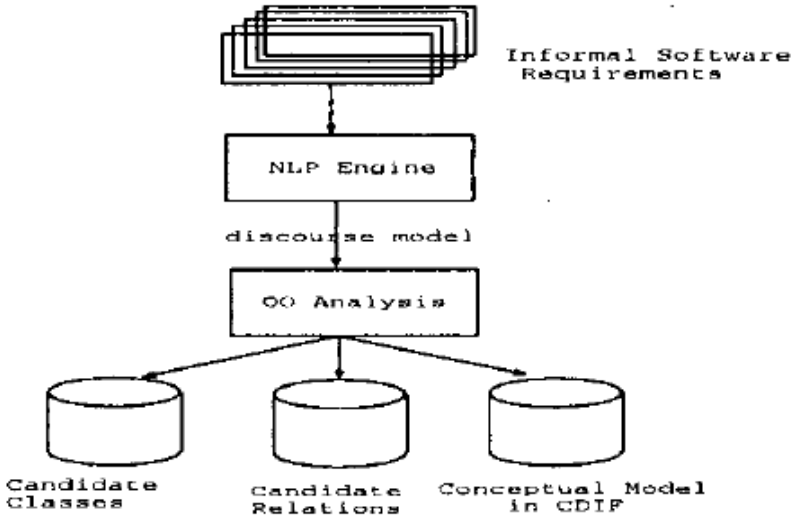


Figura 29. Enfoque de CM-Builder (Harmain y Gaizauskas, 2000).

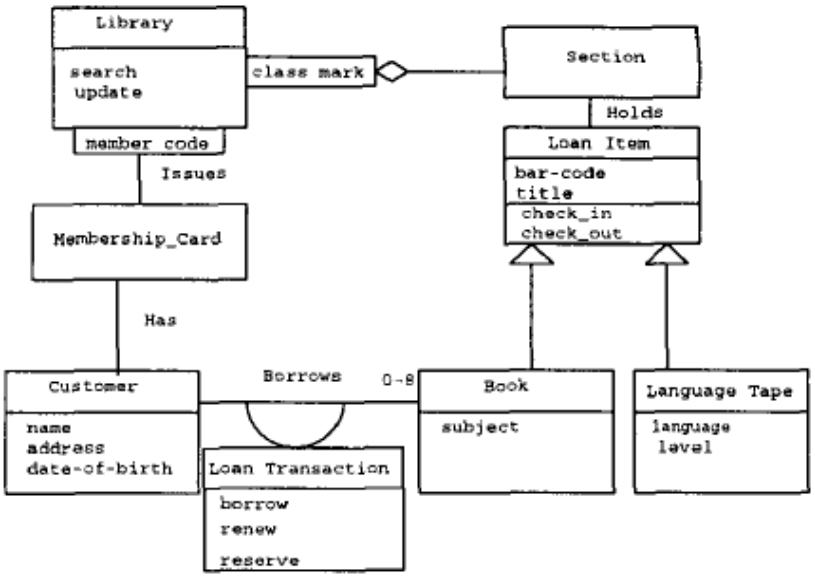


Figura 30. Versión preliminar del diagrama de Clases generado en CM-Builder (Harmain y Gaizauskas, 2000).

- RADD (*Rapid Application and Database Development*): es una herramienta CASE que se enfoca en la obtención del diagrama entidad-relación a partir de un lenguaje controlado. Además, la herramienta emplea una "Herramienta de Diálogo moderado" que posibilita la comunicación con el diseñador de la base de datos en lenguaje natural (Buchholz y Düsterhöft, 1994).

  En RADD, el proceso se inicia con un texto en lenguaje natural, en alemán, que se ingresa a un analizador sintáctico, el cual identifica los diferentes elementos gramaticales presentes en el texto y elabora un árbol gramatical que clasifica las palabras en diferentes categorías, denominadas "sintagmas".

- NIBA busca la elaboración de diferentes diagramas UML (especialmente clases y actividades, aunque que se podrían obtener otros como secuencias y comunicación), a partir de un lenguaje controlado. Para obtener los diagramas UML, NIBA emplea para un conjunto de esquemas intermedios denominados KCPM (*Klagenfurt Conceptual Predesign Model*) (Fliedl y Weber, 2002). KCPM posee formas diferentes de representación del conocimiento para los diferentes diagramas de UML. NIBA, además de generar los diferentes diagramas UML, genera el código fuente en C++.

  En la Figura 31 se aprecia un texto en el lenguaje controlado que emplea NIBA. A partir de este discurso se obtiene el esquema KCPM que se presenta en la Figura 32. Finalmente, en la Figura 33 se puede observar el diagrama de clases obtenido en NIBA.

(1) Publishers publish books. (2) Authors write books. (3) A book can be written by several authors.(4) An author has a unique name, a birth date and an address. (5) A book has a unique title and a prize. (6) An ISBN number identifies a book (7) Each book is published by exactly one publisher. (8) A publisher has a unique denomination and an address. (9) A publisher has employees. (10) An employee has a unique social insurance number, a name and a birth date.

Figura 31. Lenguaje controlado empleado en NIBA (Fliedl y Weber, 2002).

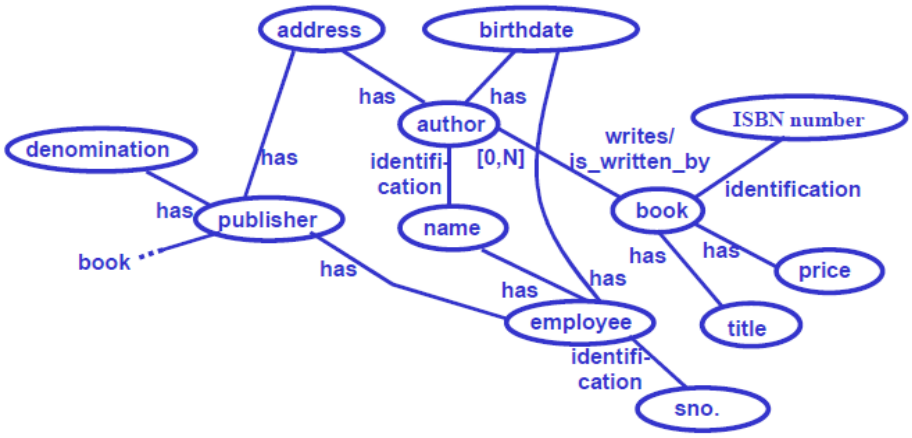


Figura 32. Esquema KCPM obtenido en NIBA (Fliedl y Weber, 2002).

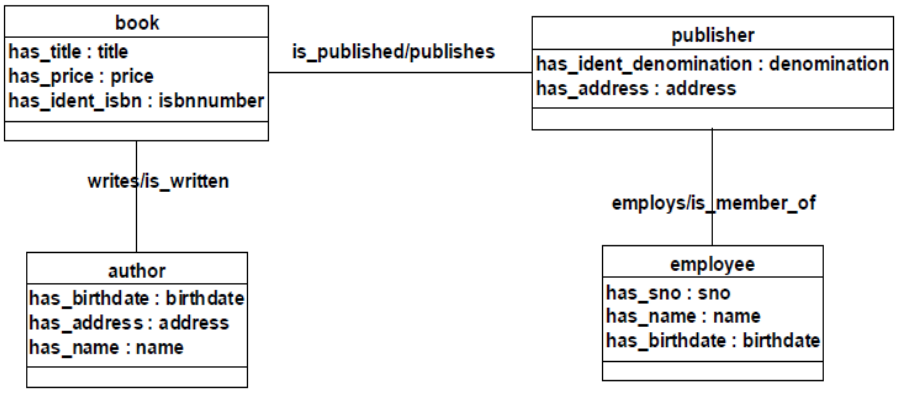


Figura 33. Diagrama de clases generado por NIBA (Fliedl y Weber, 2002).

Aunque en este grupo de herramientas CASE, algunas no generan código (excepto NIBA), se estudian en esta Tesis, dado que utilizan el lenguaje natural o controlado como punto de partida para obtener resultados preliminares. De esta manera, acercan el lenguaje técnico del analista al lenguaje natural del interesado.

3.2 Esquemas conceptuales como punto de partida

El diagrama de Clases de UML se referencia, frecuentemente, como punto de partida cuando se pretende generar automáticamente el código fuente de una aplicación. Generalmente, los proyectos académicos y las herramientas CASE que utilizan en diagrama de Clases como punto de partida, obtienen un código fuente muy similar. Se obtiene la estructura básica de las clases con sus atributos y el encabezado de los métodos.

Muñetón *et al.* (2007), Pilitowski y Dereziska (2007), Regep y Kordon (2000) y, Gessenharter (2008) definen reglas heurísticas con el fin de obtener el código fuente de una aplicación de software a partir del diagrama de Clases. Además de la estructura básica del código, Muñetón *et al.* (2007) y, Regep y Kordon (2000) complementan el código resultante, tomando como referencia el diagrama de secuencias y máquina de estados. Las reglas se definen en lógica de primer orden, permitiendo una especificación donde se evitan las ambigüedades y la necesidad de aprender un lenguaje de programación específico. Cabe anotar que las reglas definidas se pueden adaptar a cualquier lenguaje de programación orientado a objetos como Java o C#.

En la Figura 34 se presenta un diagrama de Clases del servicio a domicilio de un restaurante y, en la Figura 35 se muestra el código fuente obtenido al aplicar las reglas definidas en Muñetón *et al.* (2007) sobre la clase Pedido. Nótese que únicamente se genera la estructura básica (clase, atributos y el encabezado de los métodos).

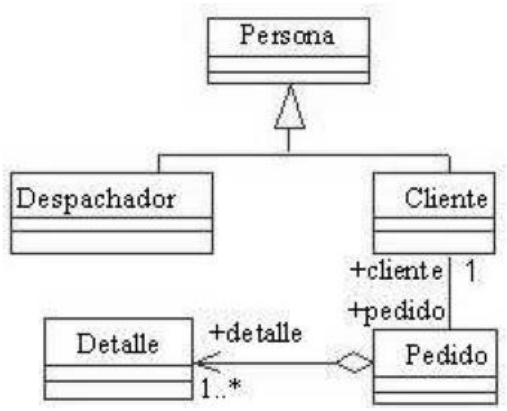


Figura 34. Diagrama de Clases del servicio a domicilio de un Restaurante (Muñetón *et al.*, 2007).

```
public class Pedido{
    //atributos
    private String number;
    private Cliente cliente;

    //operaciones
    public void agregarDetalle(){
    //método
    }
}
```

Figura 35. Resultado de aplicar las reglas sobre la clase "Pedido" (Muñetón *et al.*, 2007).

En la Figura 36 se presenta el diagrama de Secuencias del servicio a domicilio de un restaurante, mediante el cual se complementa en cuerpo del método "agregarDetalle" con las reglas definidas en Muñetón *et al.* (2007) y en la Figura 37 el código resultante.

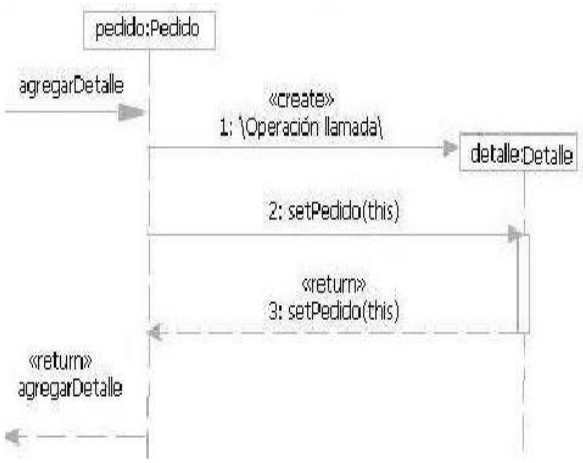


Figura 36. Diagrama de Secuencias del servicio a domicilio de un restaurante (Muñetón *et al.*, 2007).

```
public void agregarDetalle()
{
    //creación de objeto
    Detalle detalle = new Detalle();
    //llamada a método
    detalle.setPedido(this);
}
```

Figura 37. Resultado de aplicar las reglas definidas sobre el diagrama de secuencias (Muñetón *et al.*, 2007).

Together™ (Together, 2006), Rose™ (IBM, 2006) y Fujaba® (Fujaba, 2006; Geiger y Zundoft, 2006) son herramientas CASE orientadas a lenguajes de tercera generación que, además de la estructura básica, complementan el código con el comportamiento de los objetos. En Together™ se logra empleando el diagrama de secuencias y, en Fujaba®, los *Story Diagrams*, creados por el grupo Fujaba®. Estos diagramas surgen de la combinación del diagrama de actividades con el diagrama de comunicación.

En la Figura 38 se aprecia un *Story Diagram* mediante el cual se obtiene el cuerpo del método *doDemo* de la clase *House* (Fischer *et al.*, 2000).

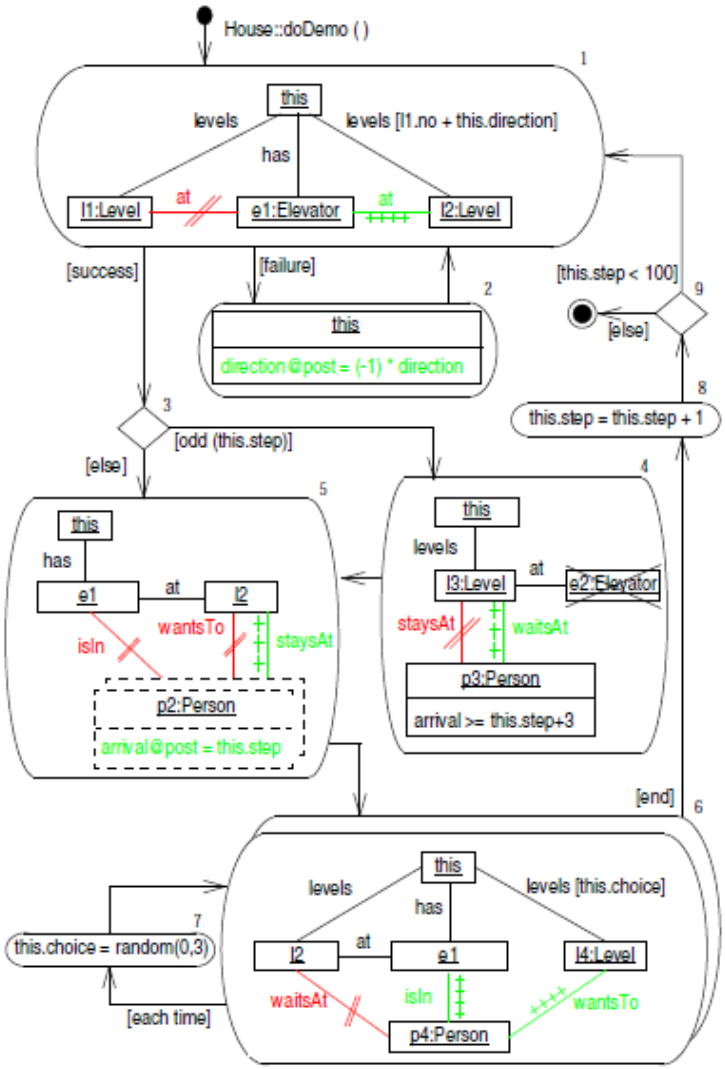


Figura 38. *Story Diagram* para el método *doDemo* de la clase *House* (Fischer *et al.*, 2000).

Las redes de Petri también se utilizan cuando se quiere abordar la generación automática de código. Normalmente, se utilizan con el fin de que el interesado pueda comprender la descripción de dominio y, por ende, pueda realizar una validación de la documentación técnica. Yao y He (1997), Mortensen (1999, 2000) y Chachkov y Buchs (2001) definen un

conjunto de reglas para la conversión a código fuente en cualquier lenguaje de programación orientado objetos a partir de las redes de Petri.

LOOPN++ (Lakos y Kenn, 1994) es compatible con una especie de alto nivel con Redes de Petri orientado a objetos. Contiene un traductor que puede generar código C++ y Java (Lewis, 1996), código que posteriormente se compila en código nativo. LOOPN++ es sólo un compilador y no contiene un simulador.

Groher y Schulze (2003), Beier y Kern (2002), Zapata *et al.* (2010) y, Bennett *et al.* (2009) proponen una serie de reglas heurísticas que permiten generar automáticamente código orientado a Aspectos. Para tal fin, Groher y Schulze (2003), Beier y Kern (2002) utilizan como punto de partida el diagrama de Clases para representar los aspectos como un paquete en UML y Zapata *et al.* (2010) utilizan los denominados Esquemas Preconceptuales y complementan su trabajo con una herramienta CASE. Bennett *et al.*, (2009), además de de las reglas heurísticas, desarrollaron un *framework* que apoya el proceso.

En Zapata *et al.* (2010) los Aspectos se representan de dos maneras en el Esquema Preconceptual. La primera forma es identificar aquellos aspectos implícitos en el sistema, es decir, en términos del Esquema Preconceptual, las tríadas dinámicas que se repiten, por ejemplo, en “profesor *registra* alumno” y “administrador *registra* asignatura”, la operación *registra* se identifica como un aspecto implícito en el sistema. En la Figura 39 se aprecia parte de un esquema Preconceptual en el cual es posible identificar aspectos implícitos en el sistema, y, en la Figura 40, se presenta el código fuente resultante de aplicar las reglas definidas.

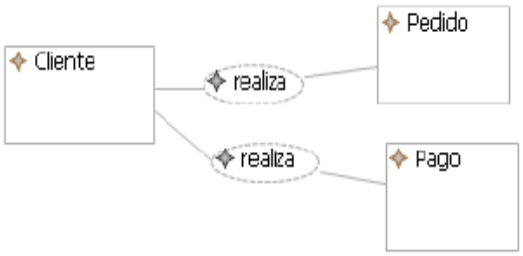


Figura 39. Esquema Preconceptual con Aspectos implícitos (Zapata *et al.*, 2010).

```
public aspect Realiza {

    private Pago pago = new Pago();
    private Pedido pedido = new Pedido();

    after (Pedido p): target(p)
      && call(void pedido.realiza()) {
    }
    after (Pago p): target(p)
      && call(void pago.realiza()) {
    }

    after(Pedido c): endTiming(c) {}

    after(Pago c): endTiming(c) {}
}
```

Figura 40. Código fuente del Aspecto "Realiza" (Zapata *et al.*, 2010).

La segunda forma permite al analista identificar manualmente aspectos desde las primeras fases del desarrollo. En el Esquema Preconceptual, un aspecto se puede representar mediante una agrupación asociada con una restricción.

En la Figura 41 se presenta la identificación de un aspecto candidato en el Esquema Preconceptual y, en la Figura 42, el código resultante del aspecto en cuestión.

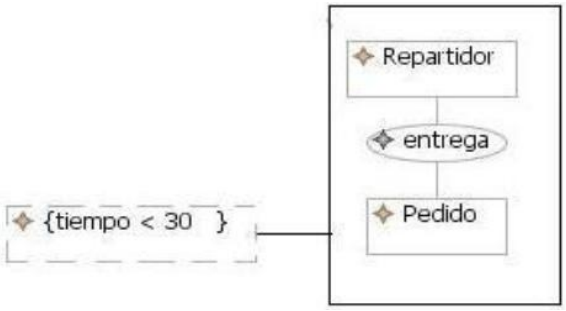


Figura 41. Identificación manual de Aspectos candidatos en el Esquema Preconceptual (Zapata *et al.*, 2010).

```
public aspect Entrega {

    private Pedido pedido = new Pedido();

    after (Pedido p): target(p) &&
      call(void pedido.entrega()) {

    }

    after(Pedido c): endTiming(c) {
      pedido.tiempo <= 30
    }
}
```

Figura 42. Código fuente del aspecto "Entrega" (Zapata *et al.*, 2010).

Bennett *et al.* (2009) emplean el diagrama de Clases. Así, definen los Aspectos como una clase y, mediante un conjunto de reglas heurísticas, transforman dicho diagrama en el código fuente. Estos autores, por defecto, definen procesos como autenticación, *login* y persistencia entre los objetos y la base de datos. En la Figura 43 se presenta un diagrama de Clases que representa una transacción bancaria y en la Figura 44 se presenta parte de la herramienta CASE que desarrollaron estos autores.

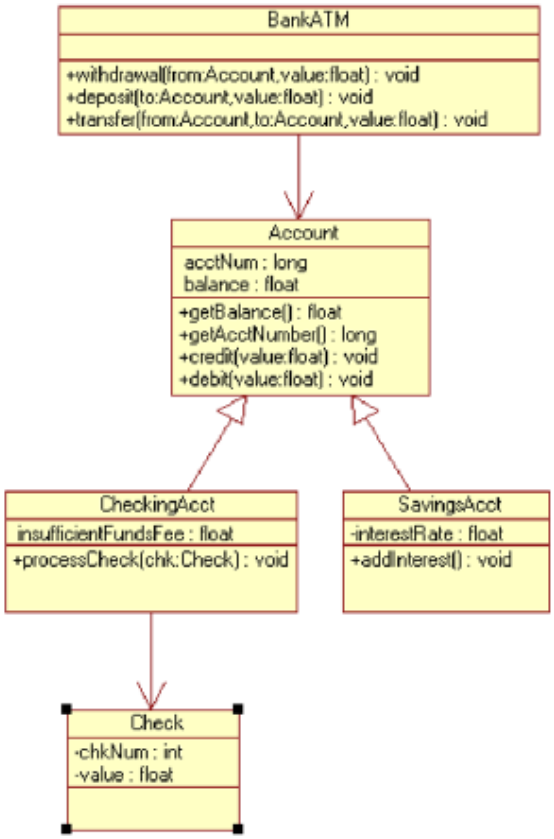


Figura 43. Diagrama de Clases para una transacción bancaria (Bennett *et al.*, 2009)

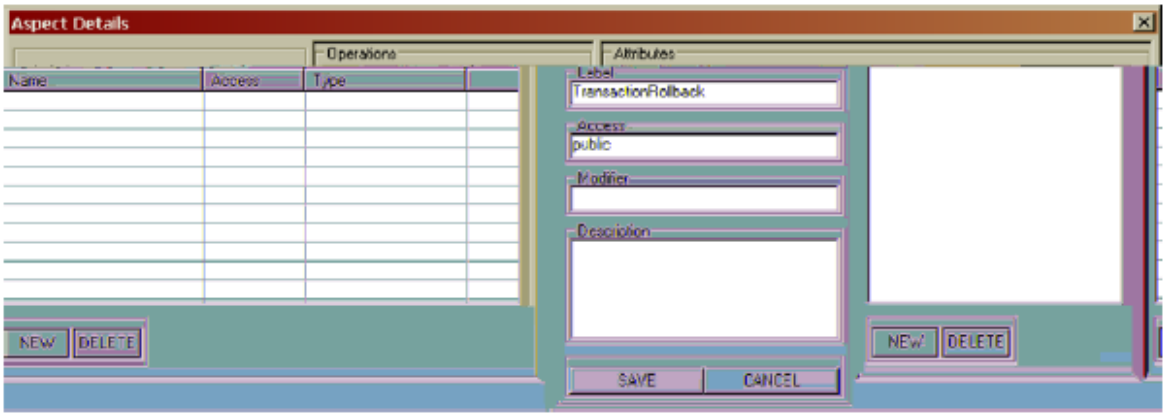


Figura 44. Herramienta CASE para la identificación de Aspectos (Bennett *et al.*, 2009)

Génova *et al.* (2003) analizan los principios de la relación entre los diagramas UML y el código fuente de una aplicación, prestando especial atención a la multiplicidad, la navegabilidad y la visibilidad, dado que los lenguajes de programación orientado a objetos expresan bien la clasificación (generalización), pero no contienen ni sintaxis ni semántica para expresar directamente las asociaciones. Estos autores definen reglas heurísticas para obtener, a partir del diagrama de Clases de UML, las clases en el lenguaje de programación Java. Para tal fin, desarrollaron una herramienta denominada JUMLA (*Java Code Generator For Unified Modeling Language Associations*). Esta herramienta lee un modelo UML, almacenado en formato XMI. La herramienta presenta las clases y las asociaciones encontradas en el modelo y el usuario debe seleccionar de qué asociaciones quiere que se genere el código.

En la Figura 45 se muestra un diagrama de ejemplo y la Figura 46 muestra cómo se presenta la ventana principal de JUMLA. El panel izquierdo muestra las clases contenidas en el modelo, en una estructura de árbol correspondiente a la estructura de paquetes del modelo en el código fuente. El panel derecho muestra las asociaciones contenidas en el modelo. Para cada asociación se presenta: clase de origen y clase de destino, nombre del rol, multiplicidad, navegabilidad y visibilidad

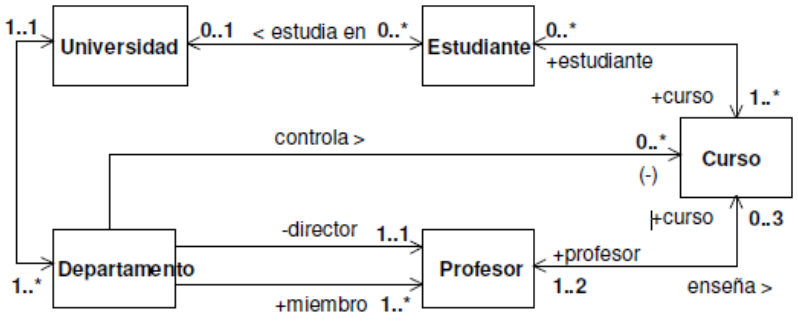


Figura 45. Diagrama de Clases en JUMLA (Génova *et al.*, 2003)

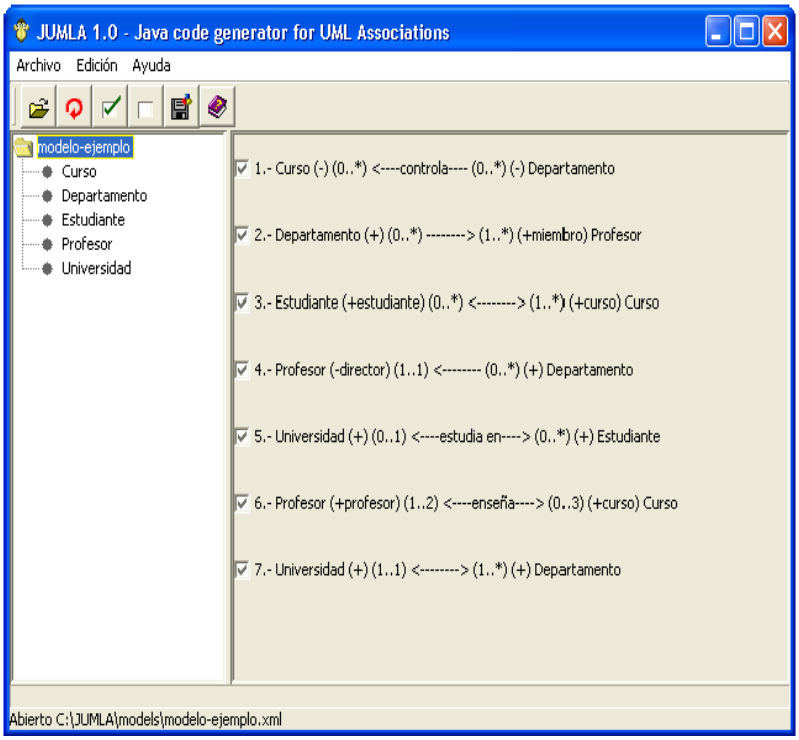


Figura 46. Herramienta JUMLA (Génova *et al.*, 2003)

En Nassar *et al.* (2009) se propone una herramienta llamada VUML, la cual apoya el proceso de análisis y diseño. El objetivo principal es almacenar y entregar información de acuerdo con el punto de vista del usuario. VUML apoya el cambio dinámico de las vistas, y ofrece mecanismos para describir la dependencia entre las vistas. Estos autores definen un conjunto de reglas en ATL (ATL *Transformation Language*), las cuales se aplican al diagrama de Clases en VUML para obtener, automáticamente, el código fuente en lenguaje de programación Java.

En la Figura 47 se observa un diagrama de clases en VUML y, en la Figura 48 se presenta el código fuente generado para la clase *MedicalForm* con las reglas definidas en Nassar *et al.* (2009).

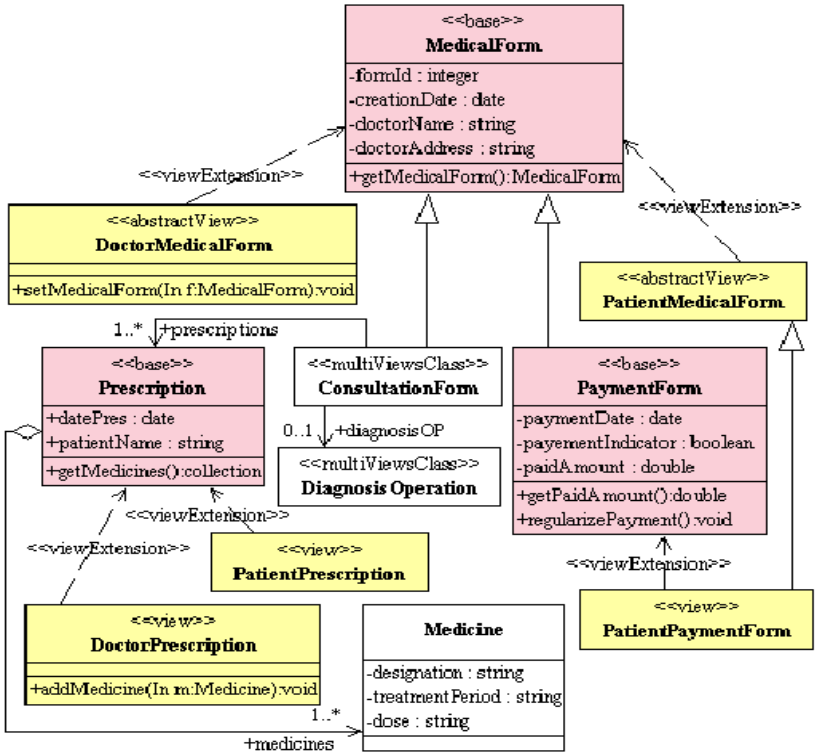


Figura 47. Diagrama de Clases en VUML (Nassar *et al.*, 2009).

```
package modelVUML;
private class MedicalForm {
private View_MedicalForm currentView;
private vector viewsList = new vector();
private Base_MedicalForm _base;
public  MedicalForm() {
  currentView=new View_MedicalForm(this);
}
private Base_MedicalForm getBase() {
  return _base;
}
private View_MedicalForm getCurrentView() {
  return currentView;
}
private void setCurrentView(String  view) {
  currentView=view;
}
public MedicalForm getMedicalForm(User U) {
  setCurrentView(U.getView());
  getCurrentView.getMedicalForm();
}
public void setMedicalForm(User U,MedicalForm f) {
  setCurrentView(U.getView());
  getCurrentView.setMedicalForm(f);
}
}
```

Figura 48. Clase "MedicalForm" generada del diagrama de Clases en VUML (Nassar *et al*, 2009)

Una de las herramientas más consolidadas en el mercado es GeneXus, de la compañía Artech. En GeneXus se obtiene un prototipo totalmente funcional, diseña y crea de modo automático una base de datos en la tercera forma normal, como se propone en la teoría de bases de datos relacionales, a partir de simples diagramas que representan el punto de vista del cliente. Según Artech (2010), GeneXus es una herramienta basada en el conocimiento, cuyo objetivo es ayudar a los analistas de sistemas para implementar aplicaciones en menos tiempo y con la mayor rapidez posible y ayudar a los usuarios en todo el ciclo de vida de las aplicaciones

3.3 Generación automática de interfaces gráficas de usuario

Lozano *et al.* (2002), Ramos *et al.* (2002), Almendros-Jiménez e Iribarne (2005) y Kantorowitz *et al.* (2003) proponen un entorno para el desarrollo de interfaces de usuario basado en modelos en el marco del desarrollo de software orientado a objetos. Para tal fin, emplean los casos de uso y el análisis de tareas. Así, proponen un modelo de interfaz, que incluye, además, la generación de diagramas gráficos para representar los diferentes aspectos de la interfaz gráfica y permite generarla de forma automática a partir de los modelos definidos.

Díaz *et al.* (2000), Elkoutbi *et al.* (1999) y, Elkoutbi y Keller (2000) desarrollaron herramientas CASE que soportan la obtención de interfaces de usuario a partir de escenarios, generando una especificación formal del sistema en la forma de diagramas de transición de estados. Esta especificación se incluye dentro de un entorno de ejecución, pudiendo animar cada uno de los prototipos. Los escenarios se describen mediante *message sequence charts*, enriquecidos con información referente a la interfaz de usuario. A partir de estos, se genera un formulario por caso de uso y un modelo de navegación entre formularios.

Díaz *et al.* (2000) utilizan el diagrama de casos de uso, los diagramas de transición de estados y los MSC (una variante al diagrama de secuencias) mediante los cuales, a partir de una serie de reglas heurísticas obtienen las interfaces gráficas de usuario. Los MSC se consideran equivalentes a los diagramas de interacción de UML. La Figura 49 contiene la notación de los MSC.

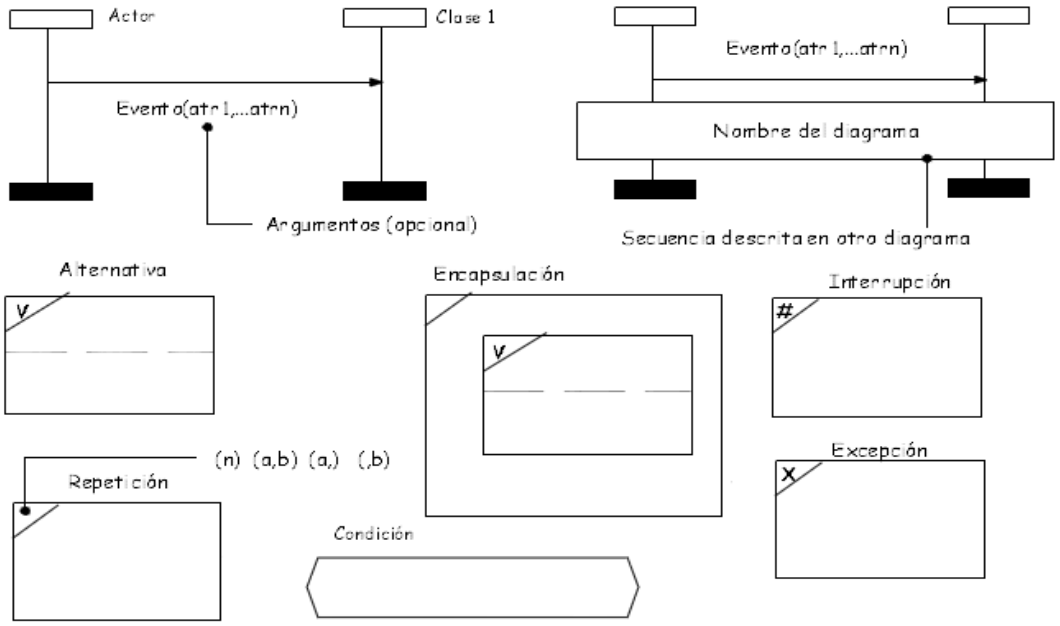


Figura 49. Elementos de los MSC (Díaz *et al.*, 2000).

En la Figura 50 se presenta un diagrama MSC para buscar clientes y, en la Figura 51 se muestra la interfaz gráfica de usuario que se obtiene a partir del MSC para la búsqueda de clientes.

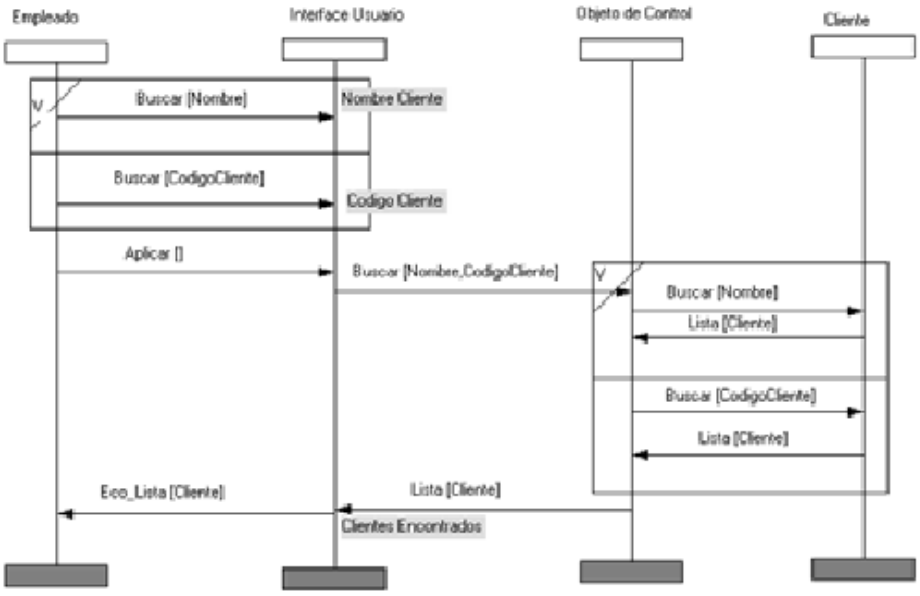


Figura 50. Diagrama MSC para búsqueda de clientes (Díaz *et al.*, 2000).

Figura 51. Interfaz gráfica obtenida a partir del MSC para búsquedas de clientes (Díaz *et al.*, 2000).

Los escenarios se pueden utilizar para diferentes propósitos, tales como el análisis de la interacción entre humanos y un computador (Nardi, 1992), la generación de la especificación (Anderson y Durney, 1993), el análisis orientado a objetos y el diseño (Booch, 1994) y la ingeniería de requisitos (Hsia *et al.*, 1994). Aprovechando estas ventajas de los escenarios, Elkoutbi *et al.* (1999) definen un conjunto de reglas heurísticas para obtener, automáticamente, las interfaces gráficas de usuario.

En la Figura 52, se presenta un escenario "*regularLoan*" y, en la Figura 53, se puede observar la interfaz gráfica obtenida.

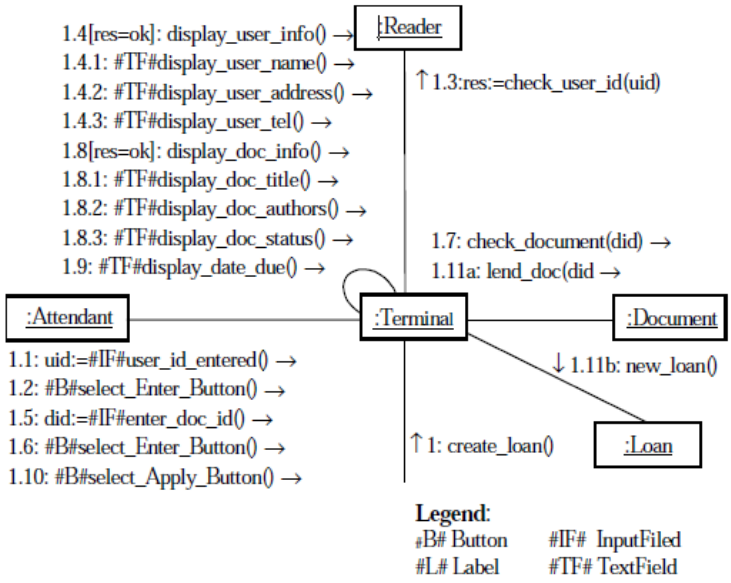


Figura 52. Escenario "*regularLoan*" (Elkoutbi *et al.*, 1999).

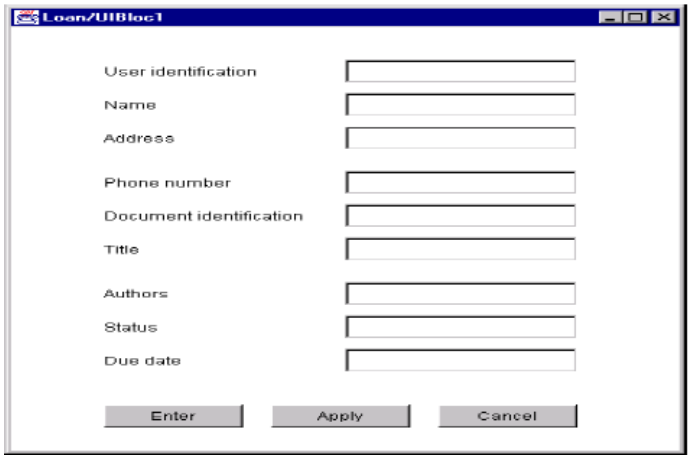


Figura 53. Interfaz gráfica para "*regularLoan*" (Elkoutbi *et al.*, 1999).

En el mundo de las aplicaciones comerciales se pueden distinguir dos tipos de aproximaciones para la construcción de interfaces gráficas de usuario. Las herramientas RAD (*Rapid Application Development*) y las herramientas basadas en modelos, las cuales intentan aumentar el nivel de abstracción (Kerr y Hunter, 1994).

Entre las herramientas RAD se destacan algunas como: *Visual Basic* (Microsoft, 2009), *Power Builder* (Sybase, 2009), *Delphi* (CodeGear, 2009), *Macromedia Flash y Macromedia Dreamweaver* (Adobe System, 2009). En estas herramientas, la construcción de la interfaz gráfica de usuario es muy rápida en comparación con otras herramientas y lenguajes de tercera generación, ya que un IDE (*integrated development environment*) (*Eclipse o Visual Studio.NET*) apoya el proceso. Las interfaces se construyen mediante el paradigma WYSIWIG (*What you see is what you get*), eligiendo los componentes de la interfaz gráfica de usuario según su necesidad. De esta manera, la calidad y la validación de la interfaz gráfica de usuario dependen de la experiencia y el criterio del diseñador.

Las herramientas basadas en modelos, son aquellas que utilizan como punto de partida los diagramas, generalmente los de UML, para la generación automática de la interfaz gráfica de usuario. Entre ellas se encuentran:

- Genova 8.0 (Genera, 2007) es un módulo de extensión para Rational Rose®. A partir de modelos UML construidos en Rational Rose®, Genova permite construir Modelos de Diálogo y de Presentación de modo parcial, que representan la interfaz de usuario como un árbol de composición de AIO (*Abstract Interaction Object*) (Bodart y Vanderdonck, 1996). Genova incluye el concepto de *Object Selectors* para establecer correspondencia entre AIO y CIO. Algunos aspectos de diseño pueden ser parametrizables (véase la Figura 54 y Figura 55).

  El desarrollador selecciona las clases que desea utilizar desde un modelo de clases en Rational Rose® y selecciona la guía de estilos utilizar. La guía de estilo asegura la uniformidad de la presentación de la interfaz gráfica generada. Genova soporta la generación automática de interfaces gráficas para C++, Visual Basic y HTML.

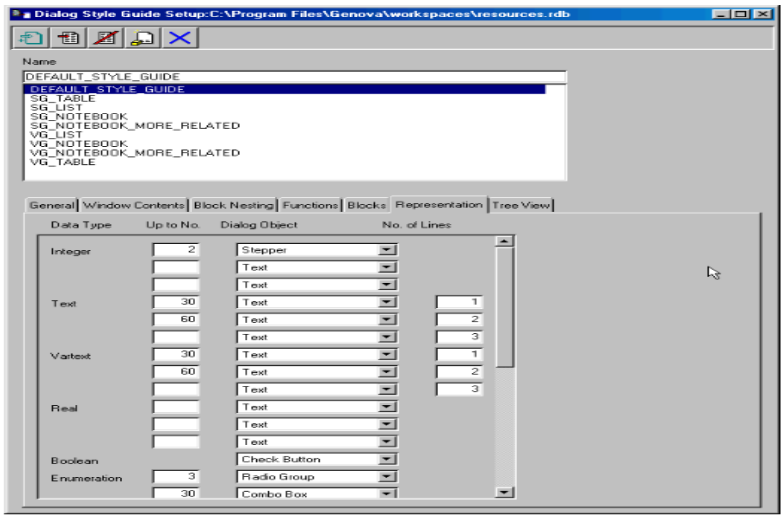


Figura 54. Parámetros de diseño en Genova 1/2 (Genera, 2007).

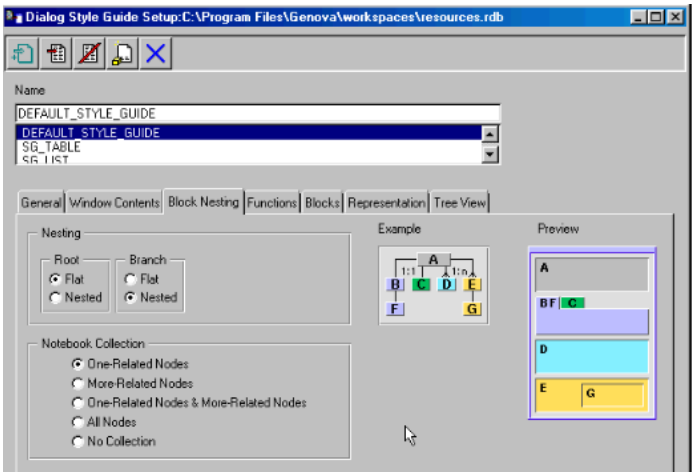


Figura 55. Parámetros de diseño en Genova 2/2 (Genera, 2007).

- Cool:Plex es una herramienta comercial utilizada para el diseño de aplicaciones cliente servidor independientemente de la plataforma. Se basa en el diagrama entidad-relación extendido (véase la Figura 56) para la especificación estática del sistema. Cool:Plex

Usa componentes y patrones de diseño que se traducen en la fase de generación de código a un lenguaje en particular (Gamma *et al*., 1995). De la interfaz gráfica, sólo se obtiene el diseño de las ventanas estáticas; es decir: no se conectan con controladores o funciones que interpreten sus acciones.

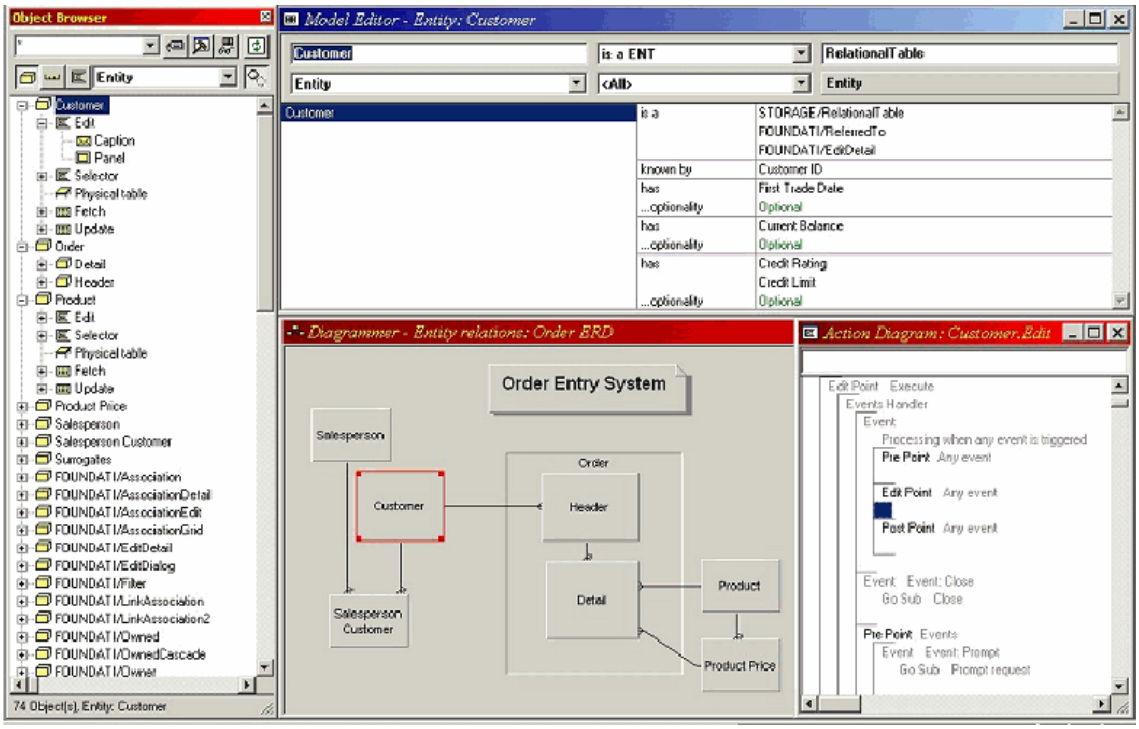


Figura 56. Herramienta Cool:Plex (Gamma *et al*., 1995).

- Together (Together, 2010) es una herramienta de modelado orientado a objetos que da soporte a la generación de clases y permite mantener una sincronización entre el código fuente y el modelo de diseño. En la última versión incluye características de tipo RAD, con lo cual es posible diseñar la interfaz de usuario para el lenguaje JAVA.

- CEM (*CSCW Environments Methodology*) (Penichet *et al.*, 2005) es una metodología que se fundamenta en MDA (*Model-driven Architecture*) como iniciativa para el desarrollo de sistemas mediante modelos de datos y procesos. Esta metodología parte de las especificaciones de aspectos estáticos y dinámicos, pero no se tiene en cuenta la interacción entre el interesado y la interfaz gráfica de usuario.

- CIAM (*Collaborative Interactive Applications Methodology*) (Molina *et al.*, 2006) permite el desarrollo de interfaces gráficas de usuario en aplicaciones *groupware* mediante la adopción de distintos puntos de vista a la hora de abordar la creación de modelos conceptuales para este tipo de sistemas.

3.4 Generación automática del cuerpo de los métodos

Comúnmente, cuando se quiere generar el cuerpo de los métodos de las clases en una aplicación orientada a objetos, se toma como punto de partida el diagrama de clases con el fin de generar la estructura básica del código y, luego, el cuerpo del método se complementa con el diagrama de secuencias.

Usman *et al.* (2008, 2009) y Pilitowski y Dereziñska (2007) desarrollaron una herramienta que, además de generar la estructura básica del código y de agregar el comportamiento de los objetos a partir del diagrama de secuencias, hacen un complemento con el diagrama de actividades. Long *et al.* (2005), además de generar el código básico y complementar el código con el diagrama de secuencias, proponen un algoritmo para comprobar la coherencia de los diagramas. De igual forma, desarrollan un algoritmo para generar código rCOS (*Relational Calculus of Object Systems*) a partir del diagrama de clases, verificando consistencia con el de secuencias. rCOS es un lenguaje orientado a objetos que soporta orientación semántica y cálculo de refinamiento.

Por el contrario, Niaz y Tanaka (2003, 2004, 2005) no complementan el cuerpo de los métodos con el diagrama de secuencias, sino con el diagrama de máquinas de estados de UML, ya que el diagrama de máquinas de estado permite especificar el comportamiento dinámico de los objetos. Los estados se representan como objetos y cada objeto define el comportamiento que se asocia con el estado.

3.5 Generación automática del diagrama Entidad-Relación

RADD (*Rapid Application and Database Development*) (Buchholz y Düsterhöft, 1994), ER-Converter (Omar *et al*, 2004), E-R Generator (Gomez *et al*, 1999) y ANNAPURNA (Eick y Lockemann, 1985) procuraron elaborar un diagrama entidad-relación a partir de especificaciones en lenguaje natural y, luego, promover la completitud del diagrama obtenido mediante un diálogo controlado con el usuario.

ER-Converter tiene como finalidad la obtención automática del diagrama entidad relación desde el lenguaje natural, para ello, se utilizan dos tipos de reglas: normas vinculadas a la semántica y normas genéricas que identifican entidades y sus atributos. ER-Converter emplea los siguientes pasos:

- Marcación de texto en MBSP (*Memory-Based Shallow Parser*).
- Identificación de entidades y sus atributos en el texto marcado.
- Intervención humana para realizar correcciones.
- Fijación de los atributos a las entidades correspondientes.
- Fijación de las relaciones entre entidades.
- Fijación de la cardinalidad entre las entidades.
- Resultado final.

En la Figura 57 se presenta un discurso en lenguaje natural en Inglés y, en la Figura 58, se observa el diagrama entidad-relación que se obtiene en ER-Converter.

The company is organized into departments. Each department has a unique department name and a unique number. A department may manage many employees but an employee can only be managed by one department. The employee name and employee id are recorded. A department may have several locations.

Figura 57. Discurso de entrada a ER-Converter (Omar *et al*, 2004).

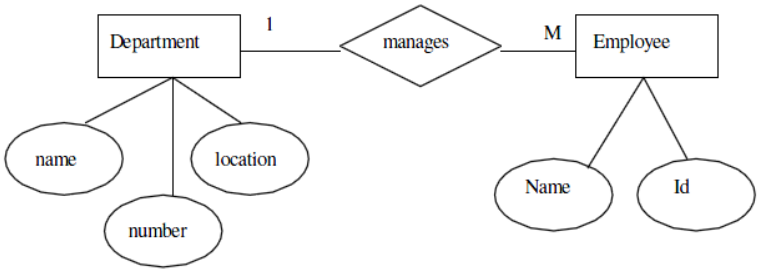


Figura 58. Diagrama entidad-relación obtenido en ER-Converter (Omar *et al*, 2004).

E-R Generator es un sistema basado en reglas heurísticas para el procesamiento del lenguaje natural que genera los elementos del diagrama Entidad-Relación, como entidades, atributos y relaciones. E-R Generator es parte de un sistema mayor, que también consta de un analizador semántico y el intérprete conocido como NLU (*Natural Language Understander*). En algunos casos, es necesario que el analista intervenga para resolver las ambigüedades tales como la fijación de los atributos y las relaciones con las demás entidades.

ANNAPURNA define un conjunto de reglas semánticas con el fin de extraer, desde el lenguaje natural, las entidades, atributos, relaciones y cardinalidad entre las entidades. Luego, estas entidades se representan en *S-diagram*, que es un modelo de datos gráfico utilizado para especificar las entidades, atributos y las conexiones entre entidades. *S-diagram* funciona mejor cuando la complejidad es pequeña.

En la Figura 59 se presenta un ejemplo en *S-diagram*.

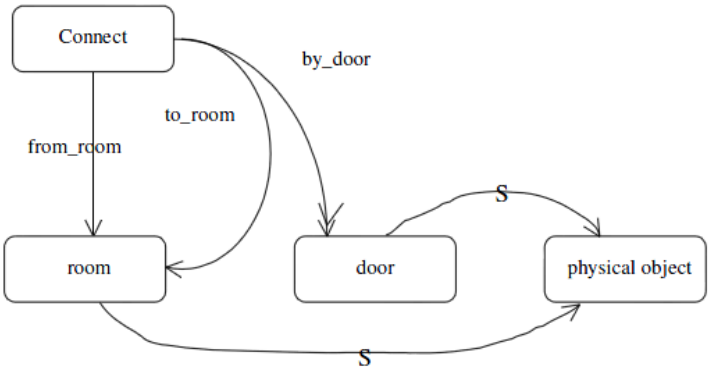


Figura 59. Ejemplo en *S-diagram.* (Eick y Lockemann, 1985)

TRIDENT (*Tools foR an Interactive Development EnvironmeNT*) (Bodart *et al*, 1993, 1994, 1995) emplea análisis de requisitos funcionales y análisis de tareas. A partir del análisis de los requisitos funcionales se obtiene un diagrama entidad-relacion extendido. El análisis de tareas se obtiene construyendo un ACG (*Activity Chain Graphs*) que ligan las tareas interactivas del usuario con la funcionalidad del sistema.

Gangopadhyay (2001) continuó con la tendencia proponiendo una herramienta CASE para la obtención automática del diagrama entidad-relación a partir de un lenguaje controlado.

Esta herramienta emplea un diagrama de dependencias conceptuales como representación intermedia a partir del lenguaje controlado y un *parser* basado en una red de transición aumentada para el procesamiento de las diferentes palabras. Esta propuesta se implementó en un prototipo usando Oracle™.

Tjoa y Berger (1993) proponen una herramienta CASE llamada DMG (*Data Model Generator*), que transforma las especificaciones de los requisitos en lenguaje natural en conceptos de un diagrama entidad-relación. DMG utiliza como texto de entrada el idioma Alemán. DMG emplea un lexicón para la identificación gramatical de cada palabra. Si una palabra no está en el lexicón el usuario debe intervenir para hacer la respectiva corrección.

3.6 Análisis crítico

Las propuestas que utilizan el lenguaje natural o algún lenguaje controlado como punto de partida para generar: diagramas, interfaces gráficas o código fuente tienen el gran problema de la ambigüedad que presenta el lenguaje natural. Empero, se reconoce que estas propuestas son interesantes, dado que acercan el lenguaje técnico del analista al leguaje natural del interesado. En los casos en que generan diagramas existe el problema de la complejidad de dichos diagramas para un interesado. Es por ello que el resultado final (diagramas o código) no es posible que lo valide un interesado.

En las propuestas donde el punto de partida para generar código fuente son los esquemas conceptuales, si bien en es cierto que generan parte sustancial del código fuente de la aplicación, aún no es un código totalmente funcional, dado que las propuestas se enfocan en la generación de “plantillas” de código independientes, como un apoyo a los programadores.

En algunos donde se generan prototipos funcionales, se hace dispendiosa la labor para el analista, dado que, generalmente, emplean diagramas propios de la metodología, lo cual deja de ser estándar.

A pesar del avance significativo que tiene la generación automática de código, aún persisten problemas tales como: comunicación con el interesado, diagramas de difícil comprensión y enfoques con reglas heurísticas hacia un único diagrama.

Por las razones mencionadas, en esta Tesis se propone un conjunto de reglas heurísticas hacia los lenguajes de programación orientados a objetos (PHP y JSP, en este caso), a partir de un Esquema Preconceptual, el cual, por su cercanía al lenguaje natural del interesado, permite validar su contenido. Al permitirle al interesado una validación desde las primeras etapas del desarrollo se incrementa la calidad de las aplicaciones.

En la Tabla 1 se presenta un análisis de varias características para todos los trabajos analizados en esta Tesis:

1. **Intervención del analista:** Se determina si es necesaria la intervención del analista para tomar decisiones en el transcurso del proceso y poder continuar con la generación de los diagramas o del código fuente. También se tiene en cuenta si es necesaria la intervención del desarrollador para realizar retoques manuales al código fuente.

2. **Punto de partida:** Se establece para cada proyecto, cuál es el punto de partida (diagramas, redes de Petri, lenguaje natural, etc.).

3. **Generación de diagramas:** En este ítem, se identifican qué proyectos generan diagramas, sin importar si son o no estándar.

4. **Generación de interfaces gráficas de usuario:** Se establece quiénes generan interfaces gráficas de usuario.

5. **Tipificación de atributos:** Se determinan qué propuestas que generan la estructura básica del código fuente, incluyendo el tipo de dato de cada concepto.

6. **Persistencia con base de datos:** Se determina qué proyectos generan persistencia entre los modelos y la base de datos

7. **Generación de procesos:** Se determina qué proyectos generan el código fuente del cuerpo de los métodos

8. **Lenguaje de programación:** Se identifica cuál es lenguaje de programación para el cual se genera el código fuente.

Tabla 1 (parte 1/5). Resumen de los trabajos en generación automática de código.

Autor(es)	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**Lenguaje de Programación.**			
								JAVA	**PHP**	**.NET**	**SQL**
Gomes *et al.* (2007), Mammar *et al.* (2006)	X	Método-B			X		X	X			

Tabla 2 (parte 2/5). Resumen de los trabajos en generación automática de código.

Autor(es)	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**Lenguaje de Programación.**			
								JAVA	**PHP**	**.NET**	**SQL**
Peckham *et al.* (2001)		L. Natural Controlado			X		X	X		X	
Ramkarthik *et al.* (2006)	X	Especificaciones Formales			X		X	X			
Gangopadhyay (2001)	X	Lenguaje Controlado	X								
NL-OOPS		Lenguaje Natural	X								
Harmain y Gaizauskas, (2000)		Lenguaje Natural	X								
CM-Builder		Textos en Inglés	X								
RADD		Lenguaje Controlado	X			X					X
NIBA		KCPM	X							X	
Muñeton *et al.* (2007)	X	D. Clases			X		X	X			
Pilitowski yn Dereziska, (2007).	X	D. Clases			X		X			X	
Regep *et al.* (2006)	X	D. Clases					X			X	

Tabla 3 (parte 3/5). Resumen de los trabajos en generación automática de código.

Autor(es)	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**Lenguaje de Programación.**			
								JAVA	**PHP**	**.NET**	**SQL**
Together	X	D. Clases					X	X			
Rose	X	D. Clases			X			X			
								JAVA	**PHP**	**.NET**	**SQL**
Fujaba	X	D. Clases					X	X			
Engels *et al.* (1999) y Samuel *et al.* (2007)	X	D. Comunicación			X		X	X			
Yao y He (1997), Mortensen (1999, 2000)	X	Redes de Petri			X		X			X	
Chachkov y Buchs (2001)	X	Redes de Petri			X		X	X			
Groher y Schulze (2003)	X	D. Clases(Ampliado)						X		X	
Beier y Kern (2002)	X	D. Clases(Ampliado)								X	
Génova *et al.* (2003)	X	D. Clases						X			
Nassar *et al* (2009)	X	D. Clases			X			X		X	

Tabla 4 (parte 4/5). Resumen de los trabajos en generación automática de código.

Autor(es)	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**Lenguaje de Programación.**			
								JAVA	**PHP**	**.NET**	**SQL**
Bennett *et al.* (2009)	X	D. Clases			X			X			
Lozano *et al,* (2002), Ramos *et al* (2002) y Almendro-Jiménez e Iribarne (2005)	X	Casos de Uso		X				X			
Kantorowitz *et al.*	X	Casos de Uso		X							
Elkoutbi *et al.* (1999) y, Elkoutbi y Keller (2000)	X	Escenarios		X							
Genova *et al* (2003)	X	WYSIWYG	X	X				X	X	X	
Cool:Plex	X	WYSIWYG		X			X	X		X	
Visual Basic, Power Builder, Delphi	X	WYSIWYG		X			X			X	
Usman, Nadeem y Kim (2008, 2009)	X	D. Clases, Secuencias y Actividades			X		X	X			
Macromedia Dreamweaver	X	WYSIWYG		X			X		X		

Tabla 5 (parte 5/5). Resumen de los trabajos en generación automática de código.

Autor(es)	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**Lenguaje de Programación.**			
								JAVA	**PHP**	**.NET**	**SQL**
Pilitowski y Dereziska (2007)	X	D. Clases, Secuencias y Actividades			X		X			X	
Long *et al.* (2005)	X	D. Clases y Secuencias	X		X		X	X			
Niaz y Tanaka (2003,2004 y 2005)	X	D. Clases y Máquinas de Estado			X						

Una síntesis adicional de esta revisión se presenta en la Figura 60 mediante un Esquema Preconceptual. Con esta representación se fundamentan los problemas, conclusiones y el trabajo futuro, que se proponen en la Sección 7 y 8 respectivamente.

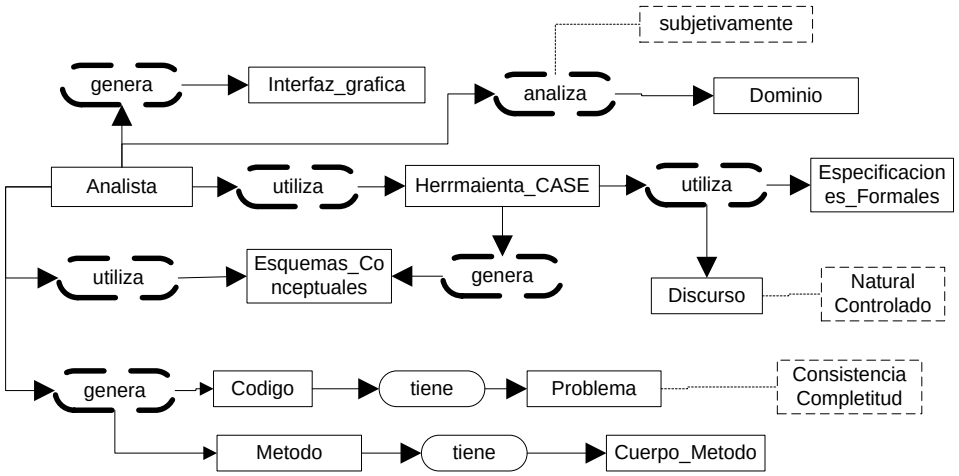
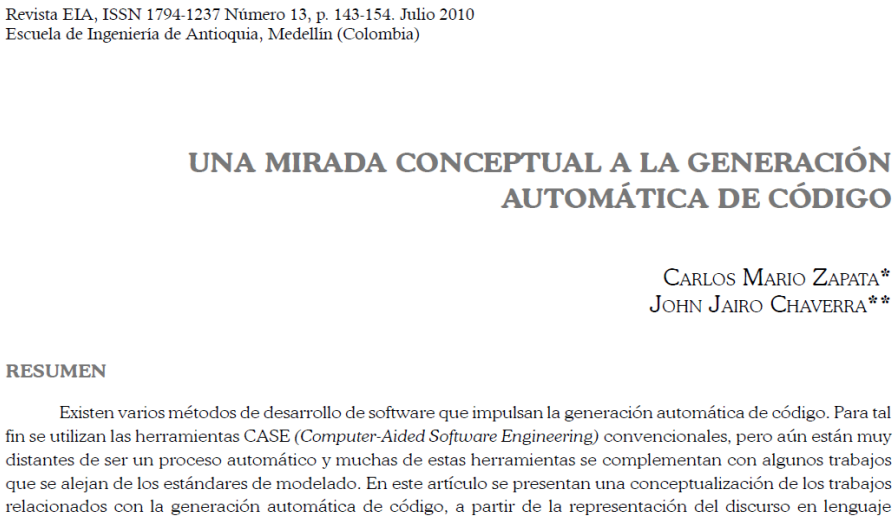


Figura 60. Síntesis de la revisión en literatura en generación automática de código

De esta revisión de literatura se publicó un artículo el cual lleva por nombre "UNA MIRADA CONCEPTUAL A LA GENERACIÓN AUTOMÁTICA DE CÓDIGO", en la Revista EIA. En la Figura 61 se aprecia una imagen de la primera página de este artículo.



Revista EIA, ISSN 1794-1237 Número 13, p. 143-154. Julio 2010
Escuela de Ingeniería de Antioquia, Medellín (Colombia)

UNA MIRADA CONCEPTUAL A LA GENERACIÓN AUTOMÁTICA DE CÓDIGO

CARLOS MARIO ZAPATA*
JOHN JAIRO CHAVERRA**

RESUMEN

Existen varios métodos de desarrollo de software que impulsan la generación automática de código. Para tal fin se utilizan las herramientas CASE *(Computer-Aided Software Engineering)* convencionales, pero aún están muy distantes de ser un proceso automático y muchas de estas herramientas se complementan con algunos trabajos que se alejan de los estándares de modelado. En este artículo se presentan una conceptualización de los trabajos relacionados con la generación automática de código, a partir de la representación del discurso en lenguaje

Figura 61. UNA MIRADA CONCEPTUAL A LA GENERACIÓN AUTOMÁTICA DE CÓDIGO

IV. DEFINICIÓN DEL PROBLEMA DE INVESTIGACIÓN

4.1 Limitaciones encontradas

Consecuentemente con la literatura especializa, los problemas que afectan la comunicación entre analista e interesado interfieren en el proceso de desarrollo del software. Para solucionar estos problemas se encuentran propuestas que disminuyen su impacto y mejoran el proceso. No obstante, no se logra una solución efectiva, pues se deben sacrificar algunos elementos a fin de obtener los resultados esperados.

Pese a los diferentes avances tecnológicos y a los esfuerzos de la comunidad científica para mejorar la relación entre analistas e interesados y disminuir los defectos en la comunicación, estos persisten y generan errores en la especificación (Shahidi y Mohd, 2009).

De otro lado, a estos problemas de comunicación se suma la dificultad del interesado para expresar sus necesidades, pues, en algunas ocasiones, no posee una idea clara de lo que desea o introduce información innecesaria en su discurso. Todo esto, genera especificaciones erróneas que se convierten en defectos y pueden producir sobrecostos en el desarrollo de una aplicación de software (Leffingwell y Widrig, 2003).

Adicionalmente a los problemas de comunicación, no existen métodos de desarrollo de software que permitan obtener prototipos totalmente funcionales desde las primeras etapas del desarrollo y que permitan que el interesado valide desde las primeras etapas y no al final cuando ya los errores se reflejan en el sistema. Generalmente, el interesado valida sólo hasta que puede interactuar con las interfaces gráficas de usuario. No tener una validación del interesado desde las primeras etapas es poco conveniente, dado que en la etapa final cualquier error que se detecte implicará retrasos en el desarrollo y un aumento significativo en los costos del producto.

A continuación, se discuten los problemas recurrentes en la generación automática de código.

4.1.1 Intervención del analista

En la mayoría de los trabajos científicos en los cuales se utiliza como base el Lenguaje Natural o un Lenguaje Controlado, se requiere una alta participación del analista, con el fin de tomar decisiones de diseño pertinentes a la generación de código, las cuales se deberían automatizar. La generación automática de código se percibe con el fin de aliviar carga al analista en decisiones de análisis y diseño. Así, se busca optimizar el tiempo de desarrollo y minimizar los posibles errores, también con el fin de que el analista se centre más en el análisis subjetivo del dominio.

4.1.2 Intervención del programador

La intervención del programador se hace necesaria cuando el código fuente generado no cumple con los requisitos inicialmente planteados. Generalmente, se hace necesaria la intervención del programador para complementar: las relaciones entre clases, el cuerpo de los métodos y las interfaces gráficas, siendo estos, puntos críticos en el desarrollo del software.

4.1.3 Punto de partida

En la mayoría de los trabajos tecnológicos como las herramientas CASE, se suelen utilizar, como punto de partida, algunos de los diagramas UML (clases, actividades, secuencias). Si bien, este punto de partida permite la generación automática de código, sólo se puede hacer después de un análisis subjetivo del problema, lo cual requiere tiempo y, puede tener implícitos errores de análisis cometidos por los analistas. Si se

procura la generación automática de código desde las etapas tempranas de desarrollo, se debería hacer desde la descripción del problema y no desde la solución como tal.

4.1.4 Objetivo final

Diferentes propuestas de la comunidad científica utilizan como punto de referencia el Lenguaje Natural o Controlado. Generalmente, estas propuestas se enfocan en obtener, de manera automática o semiautomática, diagramas UML o propios de las metodologías. Aunque los diagramas desempeñan un papel fundamental en el desarrollo de software, no constituyen, por sí solos, una aplicación funcional.

Otras propuestas utilizan como punto de partida los diagramas UML para generar el código fuente, pero, habitualmente, generan únicamente la estructura básica del código: clases, atributos y el encabezado de los métodos. Algunas propuestas exploran la posibilidad de complementar el cuerpo de los métodos, pero este código aún sigue siendo incompleto debido a que estos diagramas no poseen una estructura sintáctica ni semántica que permita representar completamente el comportamiento de los métodos.

Dada estas limitantes, no es posible obtener un código fuente totalmente funcional, ya que no se integran todas las fases del desarrollo.

4.1.5 Validación desde las primeras etapas

La mayoría de los trabajos utilizan los esquemas conceptuales como punto de partida para la generación de código. Muchos de estos esquemas no los interpreta fácilmente el interesado, lo cual impide una validación desde las primeras fases del desarrollo de software. Dicha validación es necesaria con el fin de obtener un código fuente consistente con los requisitos del interesado.

En la Figura 62 se presenta un diagrama Causa-Efecto que detalla las limitaciones encontradas en la literatura consultada.

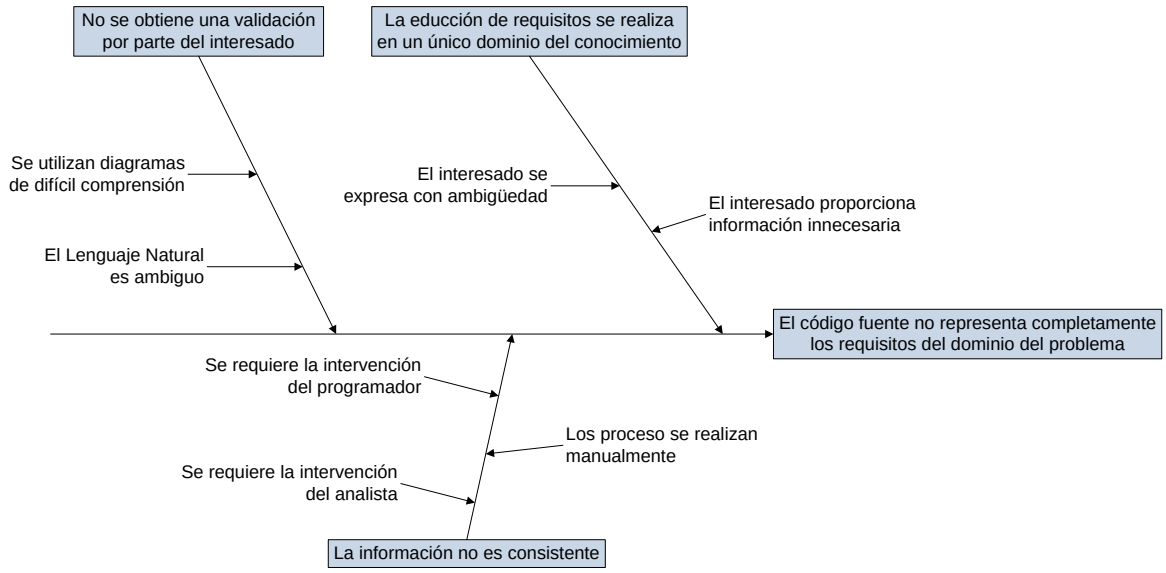


Figura 62.Diagrama Causa-Efecto del problema estudiado.

4.2 Objetivos

4.2.1 Objetivo general

Definir un conjunto de reglas heurísticas para generar, automáticamente, prototipos funcionales en lenguaje de programación PHP y JSP a partir del Esquema Preconceptual.

4.2.2 Objetivos específicos

- Obtener automáticamente el diagrama entidad-relación y, las correspondientes sentencias DDL para el gestor de base de datos MySQL a partir de Esquemas Preconceptuales.

- Obtener automáticamente código fuente bajo el patrón MVC en lenguaje de programación PHP y JSP a partir de Esquemas Preconceptuales.

- Representar los tipos de datos de cada concepto en el Esquema Preconceptual.

- Definir reglas heurísticas para complementar el código fuente con el tipo de dato de cada concepto.

- Definir elementos que permitan representar el comportamiento de las relaciones dinámicas en el Esquema Preconceptual.

- Definir reglas heurísticas que permitan obtener el cuerpo de los métodos

- Implementar una herramienta CASE que valide el Esquema Preconceptual, aplique las reglas definidas para la generación automática del código fuente y genere el diagrama de clases.

Para cumplir con estos objetivos y tomando en consideración las limitaciones de la literatura, se conciben algunos objetivos deseables para la solución propuesta. La Figura 63

representa estas características. Por simplicidad, se omiten los actores que deberían acompañar cada requisito.

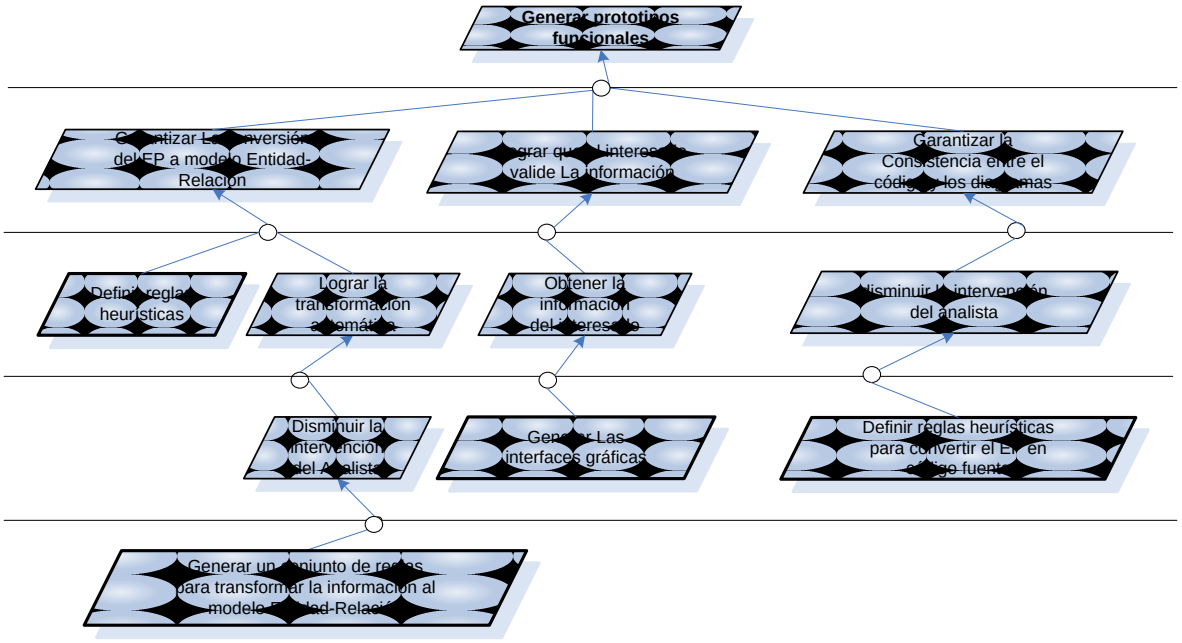


Figura 63. Diagrama de Objetivos

V. PROPUESTA DE SOLUCIÓN

De acuerdo con los planteamientos del capítulo anterior, en esta Tesis se define un conjunto de reglas heurísticas con el fin de obtener, automáticamente, prototipos funcionales en lenguajes de programación orientados a objetos (en este caso, PHP y JSP) a partir de la descripción de un dominio, expresada en Esquemas Preconceptuales, reduciendo al mínimo la intervención del programador para completar el código fuente e incluyendo al interesado en todas las etapas del desarrollo.

Esta Tesis utiliza como punto de partida los Esquemas Preconceptuales. Para ello, se definieron nuevos elementos de modo tal que se permita representar tanto el dominio del problema como de la solución.

Con el fin de obtener automáticamente un código fuente completo y totalmente funcional, se establecen dos tipos de reglas:

- **Reglas Tipo A:** Estas reglas tienen como precondición elementos propios del Esquema Preconceptual.

- **Reglas Tipo B:** Estas reglas tienen como precondición elementos propios del diagrama de Clases, además de los elementos propios del Esquema Preconceptual.

5.1 Conceptos Obligatorios

Los Esquemas Preconceptuales carecen de elementos que permitan especificar, en los conceptos, la obligatoriedad de los datos, los cuales se deben validar en etapas iniciales del desarrollo de software. Por ello, se proponen en esta Tesis algunas modificaciones a los Esquemas Preconceptuales con el fin de representar esta característica en cada concepto.

Para definir la obligatoriedad de un concepto, se propone utilizar una flecha con terminación doble unidireccional, conservando la conexión original de los Esquemas Preconceptuales para conceptos no obligatorios. Estas conexiones parten desde una relación estructural "tiene" y terminan en un concepto. En la Figura 64 se presenta un concepto "B" obligatorio y un concepto "C" no obligatorio, los cuales pertenecen al concepto "A".

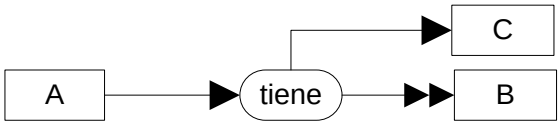


Figura 64. Obligatoriedad de un Concepto

Es conveniente modificar la conexión original del Esquema Preconceptual, ya que un concepto puede ser obligatorio únicamente para ciertos conceptos del dominio.

En la Figura 65, se presenta un concepto "B" el cual es obligatorio para el concepto "A" y no obligatorio para el concepto "C".

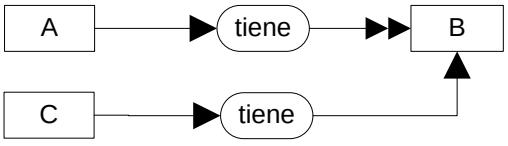


Figura 65. Ejemplo de conexión obligatoria

5.2 Tipos de Datos

Actualmente, los Esquemas Preconceptuales no poseen elementos que permitan identificar el tipo de dato de un concepto. Para ello, se proponen en esta Tesis cuatro símbolos, asociados con cuatro tipos de datos diferentes (numérico, fecha, booleano, email). Además, se define el tipo de dato "texto" como representación por defecto.

El símbolo se ubica dentro de la casilla del concepto siguiendo la estructura: *nombre_del_concepto:símbolo*. Si no se usa ningún símbolo se asume que el concepto que se representa es de tipo "texto". El tipo de dato sólo se define en los conceptos hoja.

A continuación se explica brevemente cada uno de los símbolos:

5.2.1 Concepto tipo texto

Los conceptos de tipo "texto" suelen ser los más comunes en una aplicación de software. Es por ello que, para simplificar la complejidad del esquema, se mostrará el nombre del concepto sin el uso de ningún símbolo adicional.

En la Figura 66 se presenta la forma de uso de este tipo.

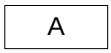


Figura 66. Representación de un concepto tipo texto.

5.2.2 Concepto tipo numérico

Se define el símbolo "#" para la representación de un concepto numérico. Con este símbolo se agrupan datos enteros, reales o de cualquier otro tipo numérico. La

especificación detallada del tipo de número ni el tamaño del mismo se tiene en cuenta en esta propuesta.

En la Figura 67 se presenta la forma de uso de este tipo

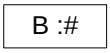


Figura 67. Representación de un concepto tipo número.

5.2.3 Concepto tipo fecha

Para representar los datos de tipo "fecha" se propone el uso del símbolo "//".

En la Figura 68 se presenta la forma de uso de este tipo.

Figura 68. Representación de un concepto tipo fecha.

5.2.4 Concepto tipo Booleano

Para representar los datos de tipo "booleano" se propone el uso del símbolo "?". Este concepto sólo puede tomar dos posibles valores (*true*, *false*).

En la Figura 69 se presenta la forma de uso de este tipo.

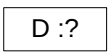


Figura 69. Representación de un concepto tipo Booleano.

5.2.5 Concepto tipo E-mail

Para representar los datos de tipo "*e-mail*" se propone el uso del símbolo "@". Aunque el tipo de dato no figura entre los diferentes motores de base de datos, ya que se considera como un tipo texto, sí se suele utilizar para validar información en la interfaz gráfica de usuario. Es por ello que se decidió agregar este tipo de dato en los Esquemas Preconceptuales.

En la Figura 70 se presenta la forma de uso de este tipo.

Figura 70. Representación de un concepto tipo Email.

En la Tabla 2 se resumen de los tipos de datos definidos en el Esquema Preconceptual.

Tabla 6. Representación de tipos de datos en Esquemas Preconceptuales

TIPO DE DATO	REPRESENTACIÓN	EJEMPLO
Texto	CONCEPTO	Nombre
Fecha	CONCEPTO://	Fecha_Venta://
Número	CONCEPTO:#	Cantidad:#
Booleano	CONCEPTO:?	Activo:?
Email	CONCEPTO:@	Emal:@

Cabe anotar que, para efectos de esta Tesis, los conceptos numéricos no distinguen longitudes. En lugar de la longitud se define un tamaño por defecto.

5.3 Atributos Derivados

Son aquellos cuyo valor se calcula a partir uno o más atributos, es decir, existen como función de otros valores. Generalmente, los atributos derivados se calculan con operaciones matemáticas. Se propone en esta Tesis que los atributos derivados se representan al interior de una Nota. Se emplea, además, una notación gráfica como la que se define en Zapata *et al.* (2010).

En la Figura 71 se presenta un ejemplo del cálculo de un atributo derivado en los Esquemas Preconceptuales.

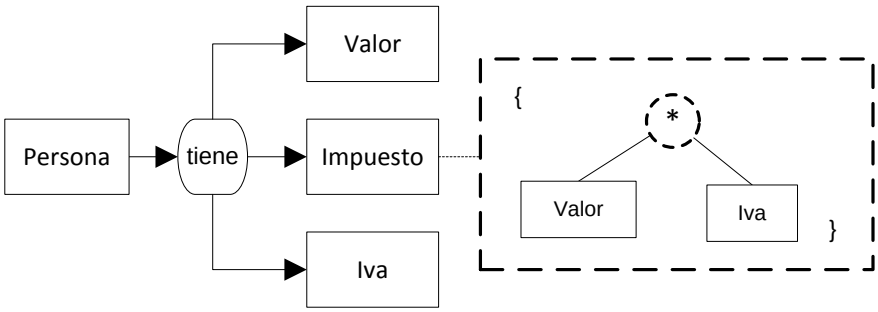


Figura 71. Ejemplo de atributo derivado.

De la Figura 71 se puede apreciar que el impuesto de un producto equivale al valor del producto por el iva. El valor del impuesto se debe reflejar en dos partes: en la interfaz gráfica para informar al usuario de dicho valor, aunque no se debe permitir modificar, y, en el modelo (clase) a modo de cálculo con el fin de almacenarlo en la base de datos.

5.4 Roles de Usuario

En toda aplicación de software es pertinente identificar los roles de usuario para determinar qué actividades puede realizar cada usuario del sistema.

Para ello, se determina que todos aquellos conceptos que ejecutan alguna tríada dinámica constituyen los diferentes roles del sistema. Por ejemplo. "*profesor registra nota*", "*profesor edita alumno*". En este caso el rol es el profesor, quien ejecuta la tríada dinámica.

En la Figura 72 se presenta un ejemplo donde se identifica el rol de usuario.

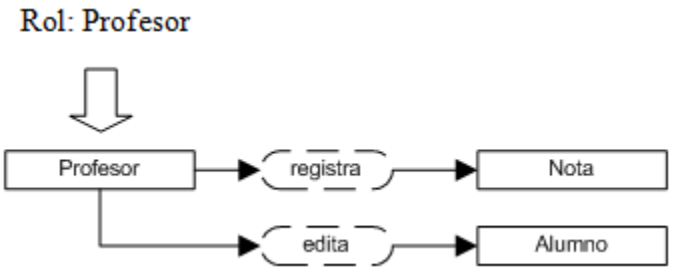


Figura 72. Identificación de un rol de usuario.

5.5 Menús de Usuarios

Una vez se identifican los roles del sistema, es necesario identificar y agrupar las acciones que ejecuta cada rol. Para ello, se determina la siguiente estructura:

- Menú: Concepto
  - Sub Menú: Acción 1.
  - Sub Menú: Acción 2.

Donde el Menú, es el Concepto sobre el cual se ejecutan las acciones (relaciones dinámicas) y, los Submenús son las acciones que se ejecutan sobre un Concepto determinado. Para el nombre del Submenú se debe utilizar el plural del nombre de la relación dinámica, por ejemplo, en la tríada dinámica "*Profesor registra Alumno*", el Menú sería Alumnos y el Sub Menú sería Registrar.

En la Figura 74 se presenta un ejemplo de Menú a partir del Esquema Preconceptual de la Figura 73.

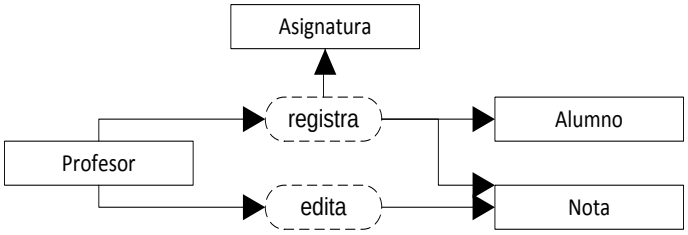


Figura 73. Esquema Preconceptual.

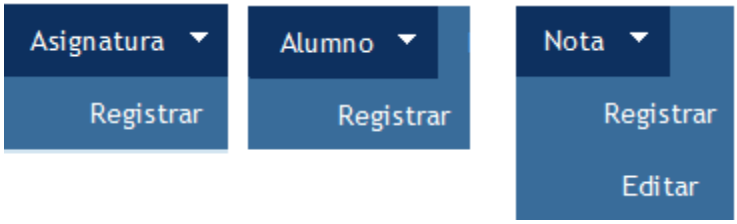


Figura 74. Menús de usuario para el Rol Profesor

5.6 Obtención automática del diagrama Entidad-Relación

Para la obtención del diagrama entidad relación, también se tiene en cuenta los tipos de datos representados en el Esquema Preconceptual, por ello en las Tablas 3 y 4 se observa que todos los atributos son de tipo texto (representación por defecto) dado que los conceptos no están acompañados de un símbolo adicional. Por simplicidad sólo se presentan los las reglas con atributos tipo texto, de igual forma aplica para los demás tipos de datos (*int*, *date*, *bynary*).

5.6.1 Reglas Tipo A

En la Tabla 3 se presentan las reglas Tipo A para obtener el diagrama entidad-relación. En la primera columna se presenta la condición que se debe cumplir en el Esquema Preconceptual, en la segunda columna el diagrama entidad-relación que se obtiene y, en la tercera columna la sentencia DDL para el gestor de base de datos MySQL correspondiente al diagrama entidad-relación.

Tabla 7 (parte 1/2). Reglas tipo A para obtener el diagrama entidad-relación.

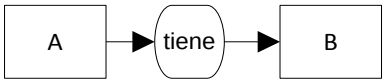
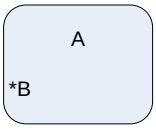
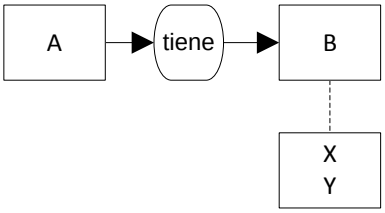
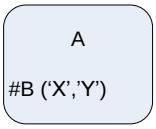
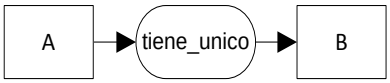
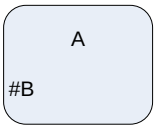
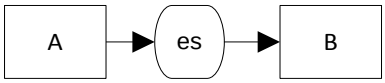
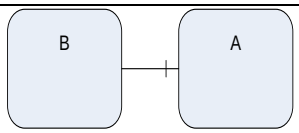
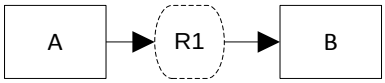
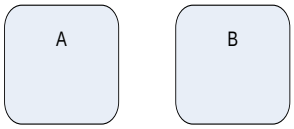
EP	ENTIDAD-RELACIÓN	SQL
		CREATE TABLE **A** (**B** varchar(30) default NULL,)
		CREATE TABLE **A** (**B** ENUM('X','Y'))

Tabla 8 (parte 2/2). Reglas tipo A para obtener el diagrama entidad-relación.

EP	ENTIDAD-RELACIÓN	SQL
		CREATE TABLE **A** (**B** varchar(30) default NULL, UNIQUE (B))
		CREATE TABLE **B** ( ) CREATE TABLE **A** UNDER **B** ( )
		CREATE TABLE **A** ( ) CREATE TABLE **B** ( )

5.6.2 Reglas Tipo B

En la Tabla 4 se presentan las reglas Tipo B para obtener el diagrama entidad-relación. En la primera columna se presenta la condición que se debe cumplir en el Esquema Preconceptual, en la segunda columna la precondición en el diagrama de clases, en la tercera columna el diagrama entidad-relación que se obtiene y, en la cuarta columna la sentencia DDL para el gestor de base de datos MySQL correspondiente al diagrama entidad-relación.

Tabla 9 (parte 1/2). Reglas tipo B para obtener el diagrama Entidad-Relación

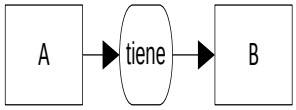
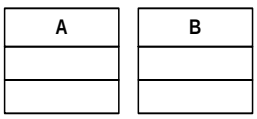
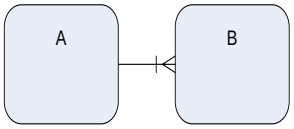
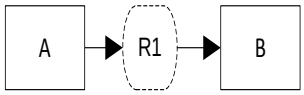
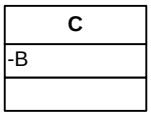
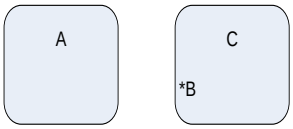
EP	PRECONDICIÓN	ENTIDAD-RELACIÓN	SQL
			CREATE TABLE **A** ( ) CREATE TABLE **B** (**a_id** int(11) FOREIGN KEY (**A**))

Tabla 10 (parte 2/2). Reglas tipo B para obtener el diagrama Entidad-Relación

EP	PRECONDICIÓN	ENTIDAD-RELACIÓN	SQL
			CREATE TABLE **A** ( ) CREATE TABLE **C** (**B** varchar(30) default NULL)

5.7 Representación del comportamiento de las relaciones dinámicas (cuerpo de los métodos)

Con el fin de representar en detalle el cuerpo de los métodos, se propone el uso del concepto nota asociado con la relación dinámica para especificar su comportamiento.

En la Figura 75 se presenta un ejemplo de cómo se utilizaría la especificación de una relación dinámica. Al interior de esta nota se permite representar cualquier estructura perteneciente a un esquema Preconceptual o, incluso, un árbol como se establece en Zapata *et al.* (2010).

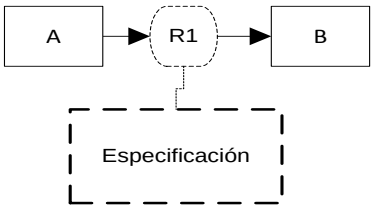


Figura 75. Ejemplo de especificación de una relación dinámica

Con el fin de representar en detalle el comportamiento de una relación dinámica, a continuación se presentan varios elementos que se deben tener en cuenta:

5.7.1 Restricciones

Una restricción puede estar asociada a un concepto o a una Nota. Esta restricción se debe enmarcar en una nota. Adicionalmente, el contenido debe estar entre llaves ("{" y "}"). Este elemento se usa de la forma en que lo presentan Zapata *et al.* (2010).

En la Figura 76 se presenta un ejemplo de una restricción.

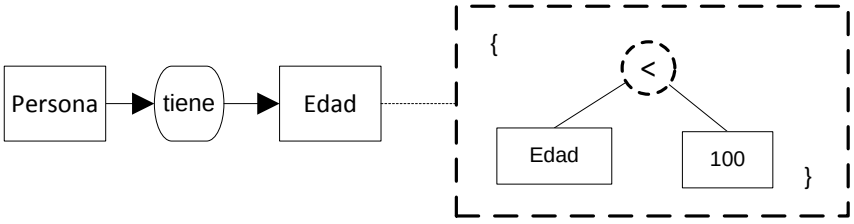


Figura 76. Ejemplo de una restricción

En la Figura 76 se identifica una restricción asociada al concepto Edad. Esta restricción significa que ninguna persona puede tener una edad mayor o igual a 100.

5.7.2 Ciclos

Los ciclos son iteraciones definidas e indefinidas. Se representan mediante una Nota asociada a una restricción, la cual, en el código fuente, se traduce en la condición de parada del ciclo.

En la Figura 77 se presenta un ejemplo, en el cual para todos los detalles se realizan las acciones que se encuentra al interior de la Nota. Para este ejemplo, detalle es un concepto.

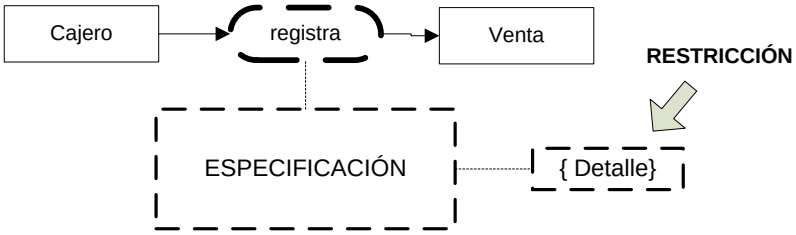


Figura 77. Ejemplo No 1 de un Ciclo.

La restricción no necesariamente debe ser un concepto, también puede ser una operación matemática.

En la Figura 78 se presenta un segundo ejemplo de un ciclo, para el cual, la condición de parada es que el total de la compra sea menor a 100

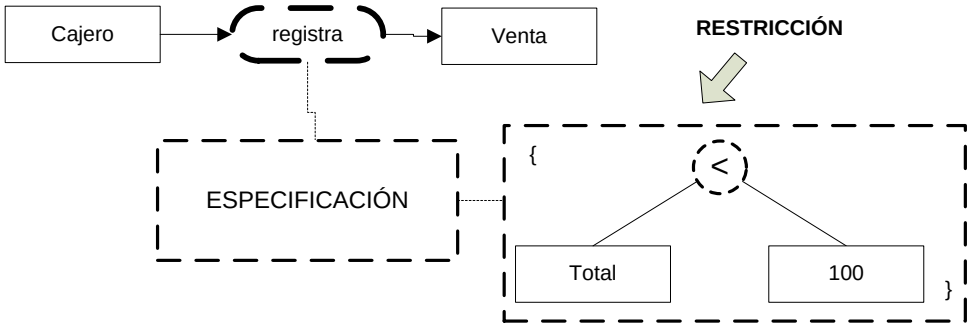


Figura 78. Ejemplo No 2 de un Ciclo.

5.7.3 Asignaciones

Mediante una asignación es posible darle valores a un determinado concepto, bien sea mediante operaciones entre otros conceptos u operaciones matemáticas. Para las asignaciones se utiliza un círculo con línea continua que une dos conceptos. La forma de lectura del árbol es de izquierda a derecha.

En la Figura 79 se presenta un ejemplo de una asignación.

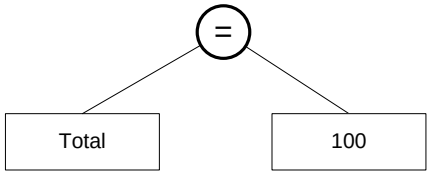


Figura 79. Ejemplo de asignación

En la Figura 79 se aprecia como al concepto Total se le asigna el valor de 100.

5.7.4 Operaciones Matemáticas

Las operaciones matemáticas entre conceptos se realizan mediante símbolos que se enmarcan en un círculo con línea punteada. Una operación matemática entre conceptos no puede existir por sí sola si su resultado no se almacena en un concepto.

En la Figura 80 se presenta un ejemplo.

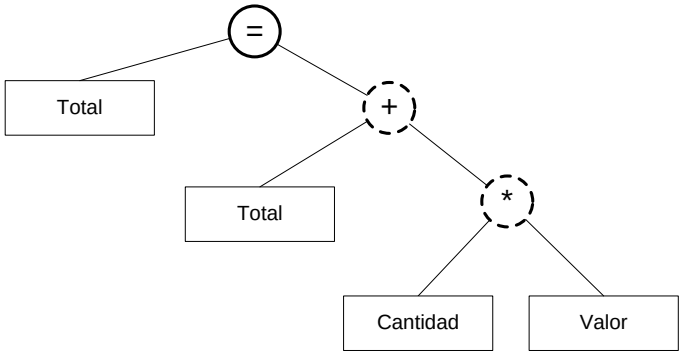


Figura 80. Ejemplo de operaciones matemáticas

5.8 Operaciones Atómicas

Con el fin de establecer unos procedimientos básicos sobre las clases, se definen las siguientes operaciones atómicas. Estas operaciones sólo se pueden realizar sobre los conceptos que previamente han sido identificados como clases.

5.8.1 Lista

Esta operación se define con el fin de obtener todo el listado de los registros existentes en la base de datos de un concepto específico. La sentencia SQL incluirá todos los atributos del modelo en cuestión y los atributos de los conceptos a los cuales se asocia con cardinalidad 1:1.

En la Figura 81 se presenta un ejemplo de su uso. El resultado de aplicar esta regla en la Figura 81 sería: el listado de todos los Exámenes existentes en la base de datos y con su respectivo Grupo.

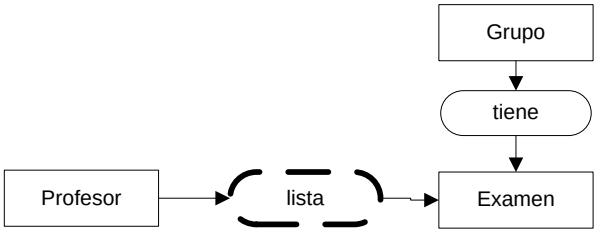


Figura 81. Ejemplo de operación atómica “lista”.

La forma correcta de leer el Esquema Preconceptual correspondiente a la Figura 81 es: “Un Grupo tiene muchos Exámenes”, “Un Examen pertenece a un Grupo”.

El SQL correspondiente al Esquema Preconceptual de la Figura 81 se puede apreciar en la Figura 82.

```
SELECT Examen.*, Grupo.*
FROM examenes AS Examen
JOIN grupos AS Grupo ON (Grupo.id = Examen.grupo_id)
```

Figura 82. SQL correspondiente a listar exámenes.

En caso tal que no se desee listar la totalidad de los Exámenes, es posible agregar restricciones a la búsqueda. Esto se logra combinando esta regla con una restricción.

En la Figura 83 se presenta un ejemplo de la operación atómica "listar" con restricciones. En este caso se listan únicamente los exámenes que pertenezcan al mismo Grupo al cual pertenece el Alumno que está consultado.

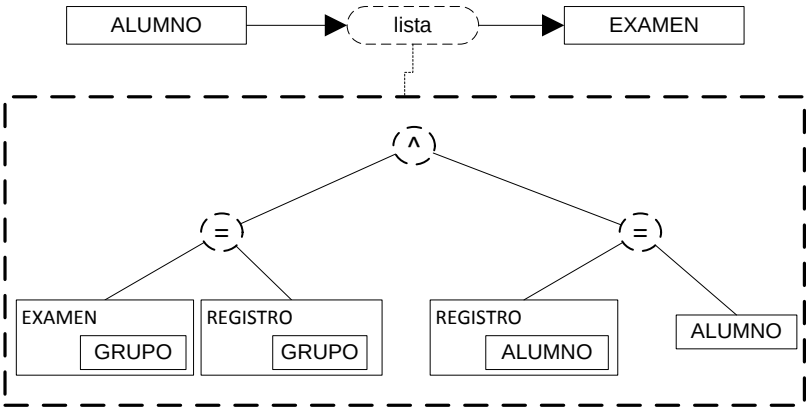


Figura 83. Operación atómica "lista" con restricciones.

El SQL correspondiente al Esquema Preconceptual de la Figura 83 se puede apreciar en la Figura 84, donde "alumnoId" es el ID del alumno que está listando los exámenes.

```
SELECT Examen.*
FROM examenes AS Examen
JOIN grupos AS Grupo ON (Grupo.id = Examen.grupo_id)
JOIN registros AS Resgitro ON (Registo.grupo_id = Grupo.id)
JOIN alumnos AS Alumno ON (Alumno.id = Registro.alumno_id)
WHERE Examen.grupo_id = Registro.grupo_id AND Registro.alumno_id = alumnoId
```

Figura 84. SQL correspondiente a listar exámenes con restricciones.

5.8.2 Inserta

La operación atómica "insertar" se define con la intención de guardar toda la información perteneciente a un concepto.

En la Figura 85 se aprecia un ejemplo de su uso.

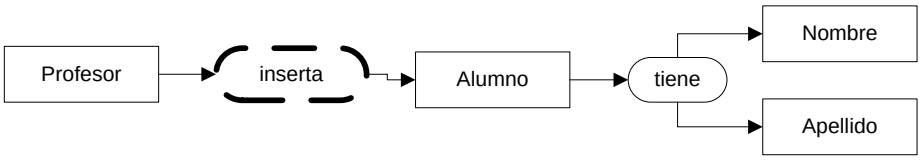


Figura 85. Ejemplo de operación atómica "inserta"

El SQL correspondiente al Esquema Preconceptual de la Figura 85 se puede apreciar en la Figura 86.

```
INSERT INTO alumnos VALUES ("nombre","apellido")
```

Figura 86. SQL correspondiente a insertar un alumno.

5.8.3 Edita

La operación atómica edita, se define con la intención de actualizar la información perteneciente a un concepto. Para poder ejecutar esta operación atómica, se debe conocer el ID del concepto en la base de datos.

En la Figura 87 se aprecia un ejemplo de su uso.

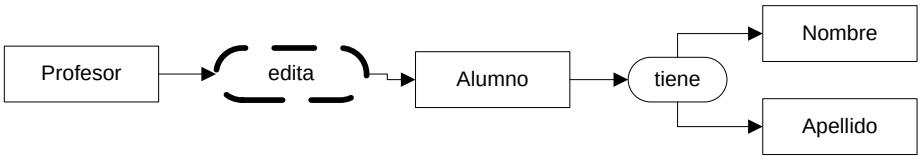


Figura 87. Ejemplo de operación atómica “edita”

El SQL correspondiente al Esquema Preconceptual de la Figura 87 se puede apreciar en la Figura 88. Nótese que aparece un número “10”, el cual corresponde al ID identificado previamente

```
UPDATE alumnos SET nombre = "María" , apellido = "López" WHERE id = 10
```

Figura 88. SQL correspondiente a editar un alumno

5.8.4 Elimina

La operación atómica eliminar se define con la intención de eliminar el registro perteneciente a un concepto. Para poder ejecutar esta operación atómica, se debe conocer previamente el ID del concepto en la base de datos.

En la Figura 89 se aprecia un ejemplo de su uso.

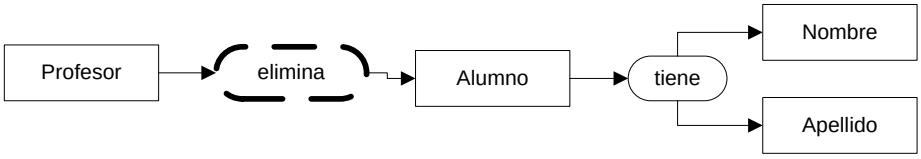


Figura 89. Ejemplo de operación atómica "elimina"

El SQL correspondiente al Esquema Preconceptual de la Figura 89 se puede apreciar en la Figura 90. Nótese que el aparece un número "10", el cual corresponde al ID identificado previamente

```
DELETE FROM alumnos WHERE id = 10
```

Figura 90. SQL correspondiente a eliminar un alumno

5.8.5 Selecciona

La operación atómica "Selecciona" es muy semejante a la operación atómica "Lista", dado que se traen de la base de datos todas las tuplas pertenecientes a un modelo, pero en esta operación únicamente se trae a modo de lista, el ID y el Nombre del concepto. Por defecto se establecen estos dos atributos, en caso de que se requiera traer en vez del Nombre otro atributo debe especificarse mediante un atributo compuesto.

Esta operación atómica se define con la intención de presentar en la interfaz gráfica de usuario esta información a modo de lista desplegable.

En la Figura 91 se presenta un ejemplo. En este caso se trae de la base de datos el ID y el Nombre (regla por defecto)

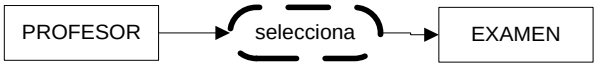


Figura 91. Ejemplo de operación atómica “selecciona”

El SQL correspondiente al Esquema Preconceptual de la Figura 91 se puede apreciar en la Figura 92.

```
SELECT id, nombre FROM examenes
```

Figura 92. SQL correspondiente a seleccionar un examen.

En la Figura 93 se aprecia un ejemplo de la operación atómica selecciona, en el cual en vez del ID y el Nombre del examen se trae el ID y el Porcentaje del examen.

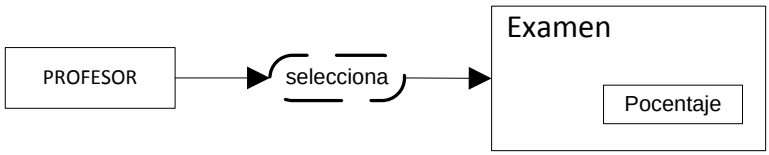


Figura 93. Ejemplo 2 de operación atómica “selecciona”.

El SQL correspondiente al Esquema Preconceptual de la Figura 93 se puede apreciar en la Figura 94.

```
SELECT id, porcentaje FROM examenes
```

Figura 94. SQL correspondiente a seleccionar el porcentaje de un examen.

En la Figura 95, se presentan un ejemplo donde se especifica el comportamiento de la relación dinámica calificar examen. Esto se logra combinando todos los elementos anteriormente definidos.

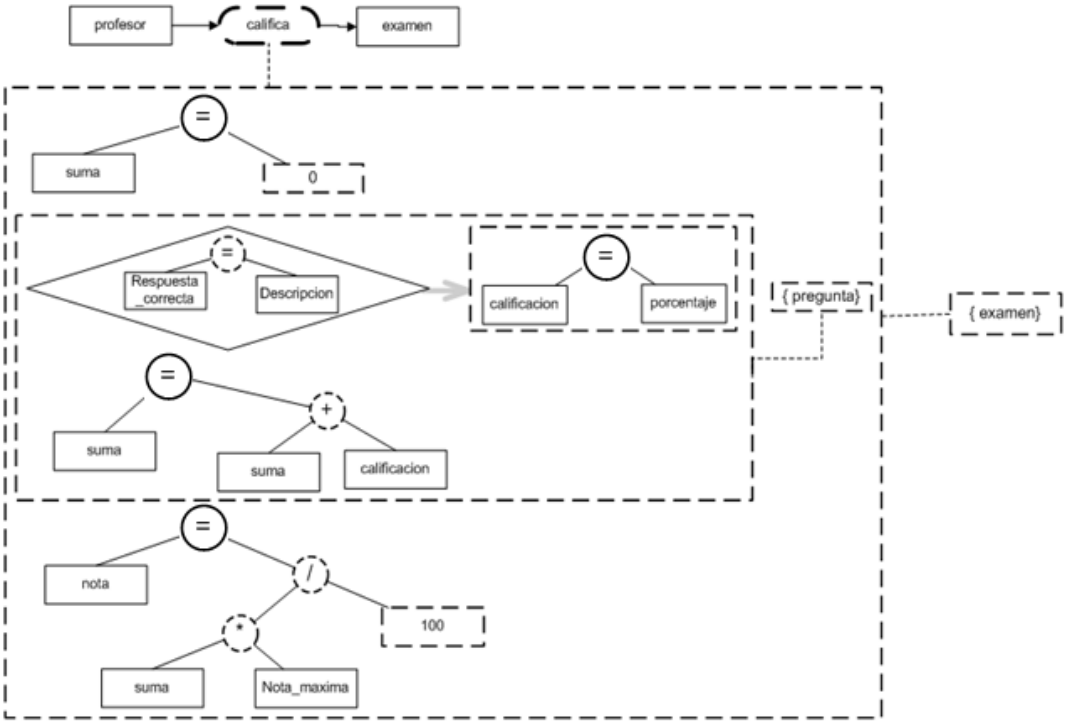


Figura 95. Detalle de calificar examen.

Nótese que en la Figura 95 la nota del examen depende de la calificación que tenga el estudiante en todas las preguntas pertenecientes al examen en cuestión.

En la Figura 96 se presenta un ejemplo del registro de una venta en un supermercado.

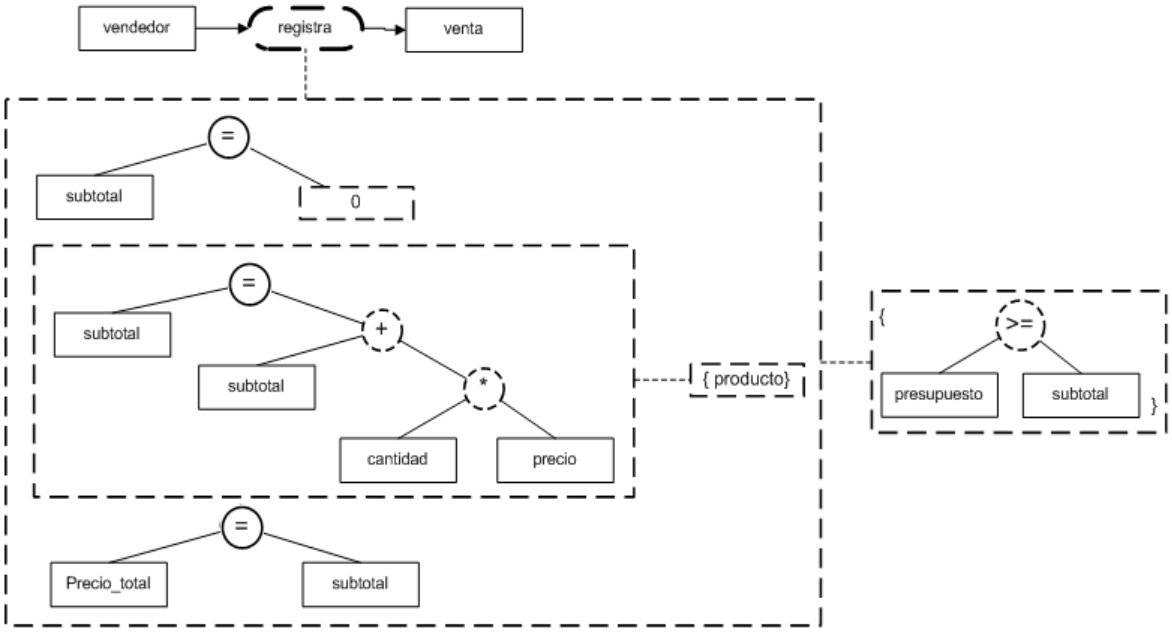


Figura 96. Registro de una venta en un supermercado

A diferencia de la relación dinámica calificar examen, en la cual el proceso se realiza para todas las preguntas del examen, en la operación registrar venta, el proceso únicamente se realiza hasta que el subtotal sea mejor o igual que el presupuesto o, lo que es lo mismo, hasta que el presupuesto sea mayor o igual al subtotal.

5.9 Obtención del código fuente bajo el patrón MVC

5.9.1 Modelo

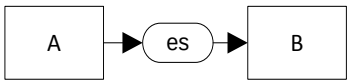
Para la definición de los modelos en JSP, se introduce un elemento "TIPO" el cual sirve para representar cualquiera de los tipo de datos (*String*, *int*, *date*, *email*) definidos en el capítulo anterior.

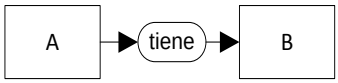
5.9.1.1 Reglas tipo A

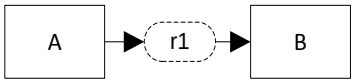
En la Tabla 5 se presentan las reglas Tipo A para la obtención automática del modelo. En la primera columna se presenta la condición que se debe cumplir en el Esquema Preconceptual, en la segunda columna el código fuente en JSP que se genera y, en la tercera columna el código fuente en PHP.

Tabla 11. Reglas tipo A para la obtención del modelo

EP	JSP	PHP

	public class **A** extends **B**{
	
	}
		<?php
		class **A** extends **B**{
		
		}
		?>

	public class **A** {
	private TIPO **b**;
	}
		<?php
		class **A** {
		var **$b**;
		
		}?>

	public class **A** {
	
	}
	
	public class **B** {
	public TIPO **R1**(){
	
	}
	}
		<?php
		class **A** {
		
		}
		
		class **B** {
		function **R1**(){
		
		}
		}? >

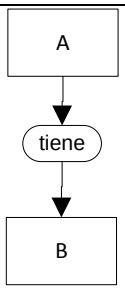
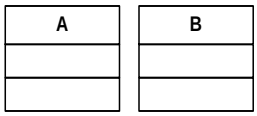
5.9.1.2 Reglas tipo B

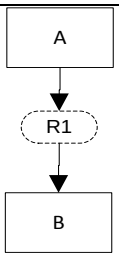
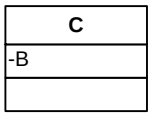
En la Tabla 6 se presentan las reglas Tipo B para la obtención automática del modelo. En la primera columna se presenta la condición que se debe cumplir en el

Esquema Preconceptual, en la segunda columna la precondición que se debe cumplir en el diagrama de clases, en la tercera columna el código fuente en JSP que se genera y, en la cuarta columna el código fuente en PHP.

Tabla 12. Reglas tipo B para la obtención del modelo

EP	Precondición	JSP	PHP

		public class **A** {
		  private **B** b;
		}
		
		public class **B** {
		
		}
			<?php
			class **A** {
			  var $b = new **B**();
			}
			
			 class **B** {
			
			}
			? >

		public class **C** {
		  private TIPO **b**;
		
		  public TIPO
		  R1(TIPO **b**){
		  }
		}
			<?php
			 class **C** {
			  var **$b**;
			
			  function **R1**($**b**){
			  }
			}?>

Para cada uno de los conceptos del Esquema Preconceptual que previamente se identificaron como clase, se determinan, automáticamente, las siguientes operaciones básicas: inserta, edita, elimina, lista, busca y selecciona. La función listar se define con la intención de obtener todos los registros de una determinada clase y, la función buscar, con la intención de obtener la información de una determinada clase. En la Tabla 7 se presentan las funciones básicas para cada modelo en PHP y, en la Tabla 8 para JSP.

En la definición de las operaciones básicas se incluye la palabra *MODELO*, para referirse al modelo en cuestión, por ejemplo (Persona, Auto). Nótese, nuevamente, que la generación de estos elementos sigue siendo automática desde el Esquema Preconceptual, pues los elementos intermedios también se determinan con base en reglas heurísticas predefinidas para los diagramas particulares que se emplean (como los diagramas de clases, comunicación y máquina de estados), que se definen en Zapata *et al.* (2006a).

Tabla 13 (parte 1/2). Operaciones básicas para los modelos en PHP

Función	**PHP**

Lista	public function lista($condicion = “1”) {
	$consulta = $this->query('SELECT * FROM *MODELO* WHERE $condicion');
	return $consulta;
	}

Busca	public function buscar($id) {
	$consulta = $this->query('SELECT * FROM *MODELO* WHERE id=$id');
	return $consulta;
	}

Selecciona	public function selecciona($campo = “nombre”) {
	$consulta = $this->query('SELECT id, $campo FROM *MODELO*');
	return $consulta;
	}

Inserta	function inserta($atributo1, $atributo2…){
	$sql = "INSERT INTO *MODELO* (atributo1, atributo2...) VALUES ('$atributo1, $atributo2….)";
	$consulta = $this->query($sql);
	}

Edita	function edita($id, $atributo1, $atributo2…){
	$sql="UPDATE MODELO SET atributo1= '$atributo1', atributo2= '$atributo2' …WHERE id=$id";
	$consulta = $this->query($sql);
	}

Tabla 14 (parte 2/2). Operaciones básicas para los modelos en PHP

Función	**PHP**

Elimina	function elimina($id){
	$sql = "DELETE FROM MODELO WHERE id= '$id' ";
	$consulta = $this->query($sql);
	}

Tabla 15. Operaciones básicas para los modelos en JSP

Función	**JSP**

Lista	public List lista(condicion) {
	List consulta = this.query('SELECT * FROM *MODELO* WHERE condicion');
	return consulta;
	}

Busca	public List buscar(id) {
	consulta = this.query('SELECT * FROM *MODELO* WHERE id=id');
	return consulta;
	}

Selecciona	public List selecciona(campo = “nombre”) {
	consulta = this.query('SELECT id, $campo FROM *MODELO*');
	return consulta;
	}

Inserta	function inserta(atributo1, atributo2…){
	String sql = "INSERT INTO *MODELO* (atributo1, atributo2...)
	VALUES (atributo1, atributo2….)";
	consulta = this.query(sql);
	}

Edita	function edita(id, atributo1, atributo2…){
	sql="UPDATE MODELO SET atributo1= 'atributo1', atributo2= 'atributo2' …WHERE id=id";
	consulta = this.query(sql);
	}

Elimina	function elimina(id){
	sql = "DELETE FROM MODELO WHERE id= 'id' ";
	consulta = this.query(sql);
	}

5.9.2 Vista

El nombre del archivo que contiene la interfaz gráfica lo determina la relación dinámica (acción) sobre un concepto específico. Es decir, existe una relación dinámica (r1) que une los conceptos A y B, entonces la interfaz llevará el nombre de "B/r1.php", "B/r1.java".

A continuación se presenta de manera propuesta el contenido de la interfaz gráfica. Se presentan algunas reglas que contienen diferentes opciones de diseño, las cuales se podrían presentar como opciones alternativas al interesado cuando se genere automáticamente la interfaz, de forma que pueda elegir la más conveniente de acuerdo con sus preferencias. Para las interfaces gráficas únicamente se presenta el código en XHTML dado que PHP y JSP lo emplean para el despliegue de las interfaces gráficas.

5.9.2.1 Regla 1.

En una relación estructural de tipo "tiene" que une dos conceptos A y B, el concepto A es un objeto y el concepto B un atributo. Esto se traduce en la interfaz gráfica de usuario en una etiqueta y su correspondiente valor al interior de la interfaz que contiene el objeto A.

En la Tabla 9 se observa el código fuente y su interfaz resultante.

Precondición:

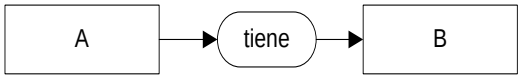


Tabla 16. Atributos en la vista (regla 1)

XHTML	RESULTADO
<label for="B">B:</label> <input name="B" type="text" />	B:

5.9.2.2 Regla 2.

En una relación estructural de tipo "es" que une dos conceptos A y B, y existe un concepto D el cual se identificó previamente como un atributo de A y un concepto C que se identificó como un atributo de B, la interfaz debe contener los atributos de A y de B en forma de etiquetas y con sus respectivos valores.

Por ejemplo, si se quiere registrar un profesor y la clase profesor hereda de la clase persona, entonces en la interfaz registrar profesor debe aparecen tanto la información de profesor como de persona.

En la Tabla 10 se observa el código fuente y su interfaz resultante.

Precondición:

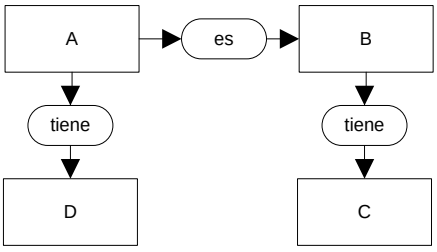


Tabla 17. Herencia en la vista (regla 2)

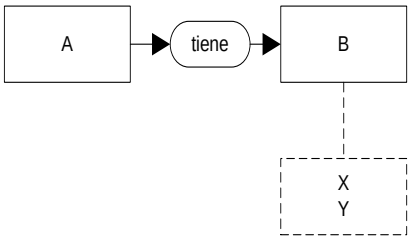
XHTML	RESULTADO
<label for="C">C:</label> <input name="C" type="text" /> <label for="D">D:</label> <input name="D" type="text" />	C: D:

5.9.2.3 Regla 3.

En una relación estructural de tipo "tiene" que une dos conceptos A y B, donde "X" y "Y" son los posibles valores de B, se pueden presentar diferentes opciones de diseño de la interfaz gráfica de usuario. El diseñador deberá seleccionar la interfaz gráfica con el usuario final, de acuerdo con sus preferencias.

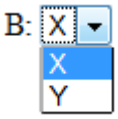
En la Tabla 11 se observa el código fuente de la primera opción y su interfaz resultante y, en la Tabla 12, se observa la segunda opción.

Precondición:



Opción 1:

Tabla 18. Posibles valores en la vista (regla 3). Opción 1

XHTML	RESULTADO
<select id="B"> <option>X</option> <option>Y</option> </select>	

Opción 2:

Tabla 19. Posibles valores en la vista (regla 3). Opción 2

XHTML	RESULTADO
<label for="B">B:</label> <label for="x">X</label> < input type=radio name="b" value="X"> <label for="y">Y</label> <INPUT type=radio name="b" value="Y">	

5.9.2.4 Regla 4.

En una relación estructural de tipo “tiene” que une dos conceptos A y C, los dos conceptos se identificaron, previamente, como clases. A tiene un atributo B y C tiene un atributo D.

Se quiere generar un formulario para la clase C, entonces la interfaz deberá contener la información de su atributo D y el valor del atributo “Nombre” del concepto A. Si

el concepto A no tiene un atributo "Nombre", entonces se deberá especificar cuál es el atributo a mostrar mediante un atributo derivado.

En la Tabla 13 se observa el código fuente y su interfaz resultante.

Precondición:

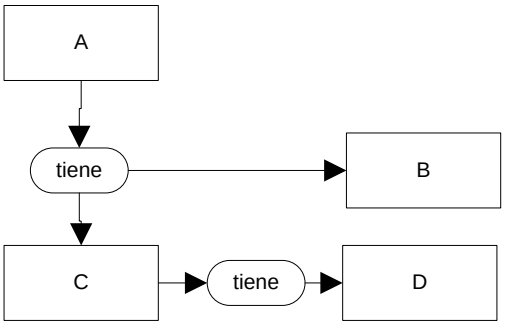


Tabla 20. Conceptos con relación 1:1

XHTML	RESULTADO
<label for="D">D:</label> <input name="D" type="text" /> <label for="A">A:</label> <select id="A"> <option>A1</option> <option>A2</option> <option>A3</option> </select>	D: A: A1 A1 A2 A3

Cabe anotar, que la información de la lista desplegable A (A1, A2, A3) corresponde a los valores de la entidad A que estén en la base de datos.

5.9.2.5 Regla 5.

En una relación dinámica de nombre "lista" que une dos conceptos A y B se determina, por defecto, que se deben listar todos los atributos del concepto B. Esta regla se define con la intención de generar el "index" de un concepto.

En la Figura 97 se presenta un ejemplo de la operación "lista".

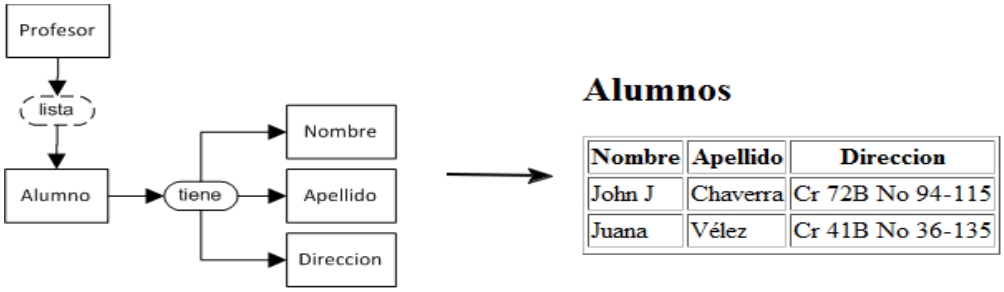


Figura 97. Ejemplo de operación "lista".

En la Tabla 14 se presenta el código fuente en JSP y en PHP correspondiente al Esquema Preconceptual de la Figura 97.

Tabla 21. Operación "listar" (regla 5)

JSP	**PHP**

<table>
<tr>
<th>Nombre</th>
<th>Apellido</th>
<th>Direccion</th>
</tr>
 	
<%
List
alumnos=(List)request.getAttribute("alumnos");
Iterator alumno = alumnos.iterator();
while(alumno.hasNext()) {
out.print("<tr>");
out.print("<td>"+alumno.nombre+"</td>");
out.print("<td>"+alumno.apellido+"</td>");
out.print("<td>"+alumno.direccion+"</td>");
out.print("</tr>");
}
%>
</table>
	<table>
	<tr>
	<th>Nombre</th>
	<th>Apellido</th>
	<th>Direccion</th>
	</tr>
	<?php
	$i=0;
	foreach ($alumnos as $alumno):
	?>
	<tr>
	<td><?php echo
	$alumno['Alumno']['nombre']; ?>
	</td>
	<td><?php echo
	$alumno['Alumno']['apellido']; ?>
	</td>
	<td><?php echo
	$alumno['Alumno']['direccion']; ?>
	</td>
	</tr>
	<?php endforeach; ?>
	</table>

Por defecto, en cada operación dinámica, el formulario contendrá todos los atributos pertenecientes a la clase en cuestión. Nótese que en la Figura 98 aparece un botón "Registrar Alumno". Este nombre se logra combinando el nombre de la relación dinámica con el nombre del concepto en cuestión.

En la Figura 98 se presenta un ejemplo de un formulario con todos sus atributos.

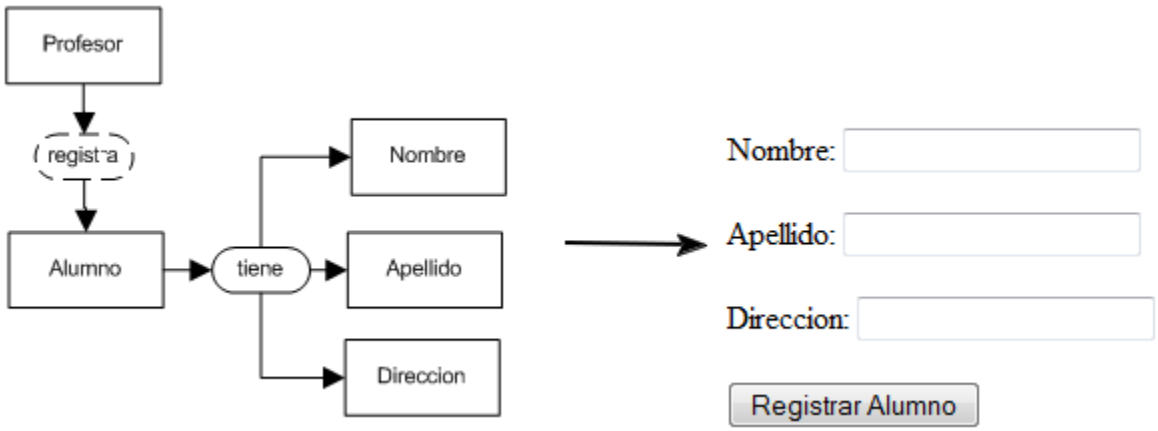


Figura 98. Formulario por defecto

En caso de que no se requiera mostrar toda la información correspondiente al modelo en cuestión, se debe especificar el comportamiento de la relación dinámica mediante una Nota. Para tal fin se define la relación dinámica "ingresa", mediante esta operación se determinan los elementos a mostrar en la interfaz gráfica.

En la Figura 99 se presenta un ejemplo de la relación dinámica "ingresa".

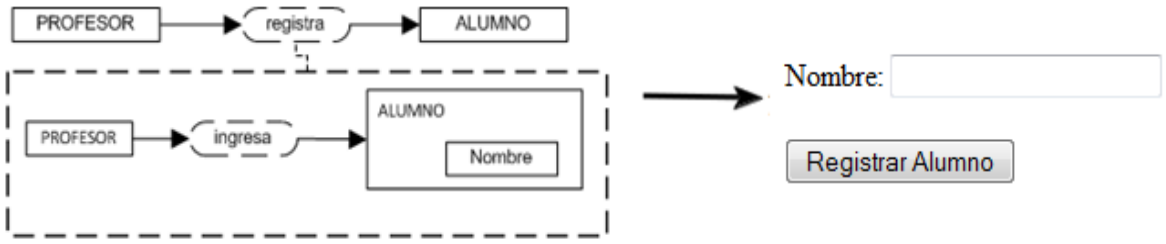


Figura 99. Ejemplo de la relación dinámica "ingresa".

5.9.3 Controlador

En este caso, sólo se cuenta con precondiciones basadas en el Esquema Preconceptual y se genera sólo una opción de diseño para cada caso. Nótese que los diferentes elementos se obtienen sustituyendo los valores de las variables A, R1 y B en el fragmento correspondiente al código.

En la Tabla 15 se presenta el código fuente de los controladores en PHP y, en la Tabla 16 se presentan los descriptores "Servlets" en JSP y los "HttpServlet" que atienden la acción del formulario.

Tabla 22. Controladores en PHP.

EP	**PHP**

	<?php
	class **AController** {
	}
	?>
	
	<?php
	class **BController** {
	}
	?>

EP	**PHP**

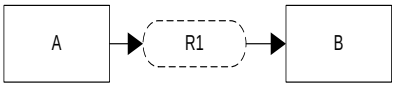
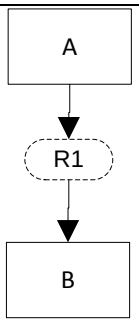
	<?php
	class BController {
	function R1( ) { }
	}
	?>

Tabla 23. Controladores en JSP.

EP	Descriptor	Controller

	<web-app>
	
	<servlet>
	<servlet-name>R1B</servlet-name>
	<servlet-class>action.R1B</servlet-class>
	</servlet>
	
	<servlet-mapping>
	<servlet-name>R1B</servlet-name>
	<url-pattern>/R1B.do</url-pattern>
	</servlet-mapping>
	
	</web-app>
		public class **R1B** extends HttpServlet{
		
		public void doPost(
		HttpServletRequest request,
		HttpServletResponse response)
		throws ServletException {
		
		}

En la definición de las operaciones básicas para los controladores se incluye la palabra *MODELO*, para referirse al modelo en cuestión, por ejemplo (Persona, Auto) y, la palabra *modelos* para referirse a un listado (personas, autos) del respectivo modelo.

En la Tabla 17 se definen por defecto las funciones básicas (lista, inserta, edita, elimina) para cada controlador en lenguaje de programación PHP y, en la Tabla 18 se presentan los "HttpServlet" en JSP que atienden las diferentes acciones de los formularios, siendo A en nombre de la clase correspondiente.

Tabla 24. Funciones básicas para el controlador en PHP

Función	**PHP**

Lista	function lista() {
	$MODELO = new MODELOModel();
	$modelos = $MODELO->lista();
	$data['modelos'] = $modelos;
	$this->view->show("MODELO/listar.html", $data);
	}

Inserta	function inserta() {
	$data = $_POST;
	if(!empty($data)){
	$atributo1 = $_POST['atributo1'];….
	$MODELO = new MODELOModel();
	$MODELO-> inserta($atributo1…..);
	$this->lista( );
	}
	}

Edita	function edita($id) {
	$data = $_POST;
	if(!empty($data)){
	$id = $_POST['id'];
	$atributo1 = $_POST['atributo1'];
	….
	$MODELO = new MODELOModel();
	$MODELO->edita($id, $atributo1…..);
	$this->lista( );
	} else{
	$MODELO = new MODELOModel();
	$modelos = $MODELO->busca($id);
	$data['modelos'] = $modelos;
	$this->view->show("MODELO/editar.html", $data);
	}
	}

Elimina	function elimina($id) {
	$ MODELO = new MODELOModel();
	$ MODELO->elimina($id);
	$this->lista( );
	}

Tabla 25 (parte 1/2). Funciones básicas para el controlador en JSP/

Función	**JSP**

Lista	public class **ListaMODELO** extends HttpServlet{
	
	public void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException {
	
	MODELO modelo = new MODELO();
	List modelos = modelo.lista();
	
	request.setAttribute("modelos", modelos);
	RequestDispatcher view = request.getRequestDispatcher("lista.jsp");
	view.forward(request, response);
	
	}

Inserta	public class **InsertaMODELO** extends HttpServlet{
	
	public void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException {
	
	Tipo atributo1 = request.getParameter("atributo1");
	Tipo atributo2 = request.getParameter("atributo2");
	
	MODELO modelo = new MODELO();
	modelo.inserta(atributo1, atributo2..);
	
	RequestDispatcher view = request.getRequestDispatcher("lista.jsp");
	view.forward(request, response);
	
	}

Tabla 26 (2/2). Funciones básicas para el controlador en JSP

Función	**JSP**

Edita	public class **EditaMODELO** extends HttpServlet{
	
	public void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException {
	
	Tipo id = request.getParameter("id");
	Tipo atributo1 = request.getParameter("atributo1");
	Tipo atributo2 = request.getParameter("atributo2");
	
	MODELO modelo = new MODELO();
	modelo.edita(id, atributo1, atributo2..);
	
	RequestDispatcher view = request.getRequestDispatcher("lista.jsp");
	view.forward(request, response);
	
	}

Elimina	public class **EliminaMODELO** extends HttpServlet{
	
	public void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException {
	
	Tipo id = request.getParameter("id");
	
	MODELO modelo = new MODELO();
	modelo.elimina(id);
	
	RequestDispatcher view = request.getRequestDispatcher("lista.jsp");
	view.forward(request, response);
	
	}

5.10 UNC-PSCode: Una herramienta CASE basada en Esquemas Preconceptuales para la generación automática de código

La herramienta CASE UNC-PSCoder procura resolver algunas de las limitaciones que aún subsisten en la generación automática de código fuente y la obtención automática de Esquemas Conceptuales de UML 2.0.

Para el desarrollo de esta herramienta se aprovecharon las ventajas de la tecnología JavaScript y del *framework* ExtJs, combinándolas con las posibilidades gráficas de mxGraph.

UNC-PSCoder realiza las siguientes funciones:

- Permite diagramar los Esquemas Preconceptuales, incluidos los nuevos elementos.
- Valida el Esquema Preconceptual.
- Obtiene automáticamente el código fuente en lenguaje de programación PHP para el *framework* CakePhp y JSP.
- Obtiene automáticamente el diagramas de clases y casos de uso a partir de las reglas definidas en Zapata *et al.* (2007).
- Obtiene automáticamente el diagrama entidad-relación
- Obtiene automáticamente las sentencias SQL correspondientes al diagrama entidad-relación.

En la Figura 100, se presenta una imagen de la herramienta CASE.

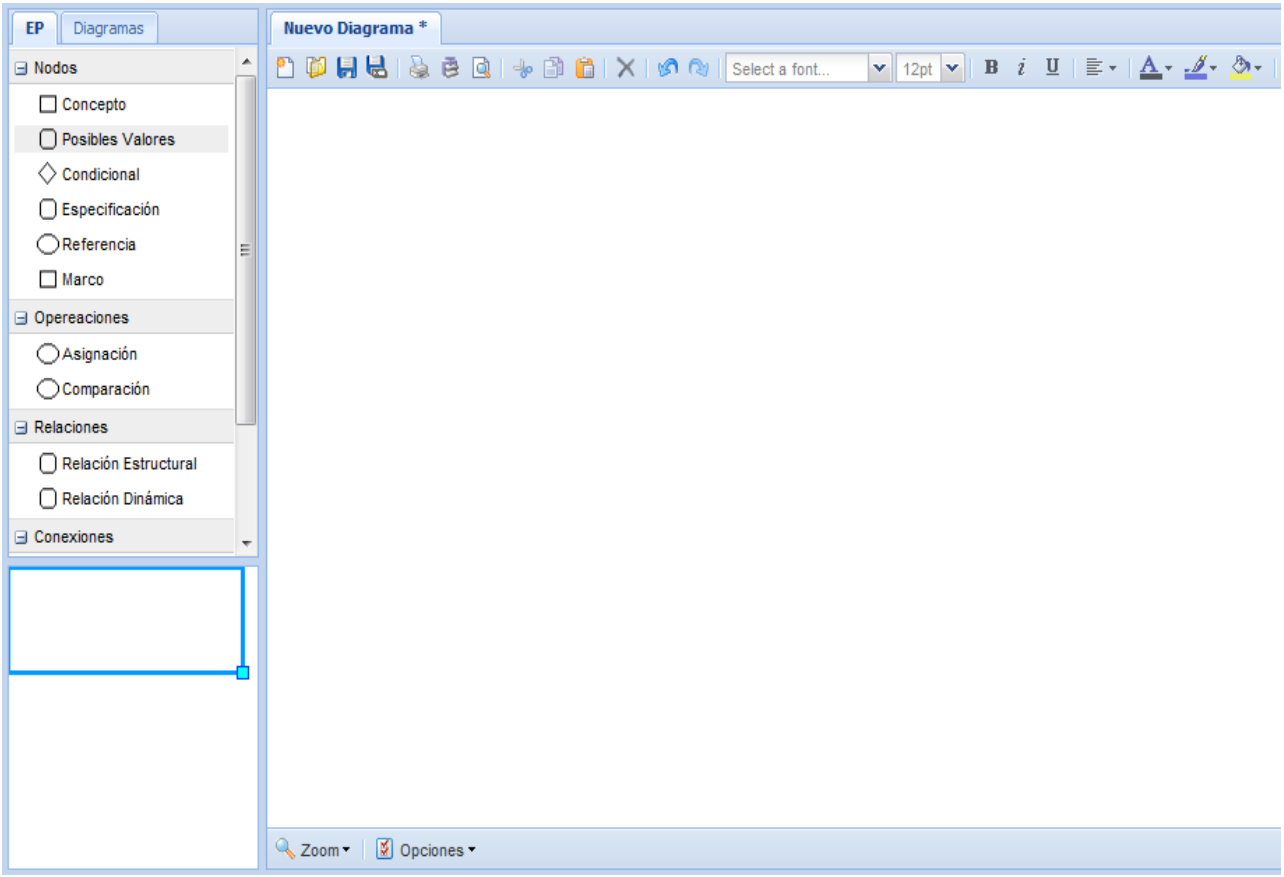


Figura 100. Herramienta CASE

En la Figura 101 se presenta un ejemplo de un Esquema Preconceptual en la herramienta CASE UNC-PSCoder.

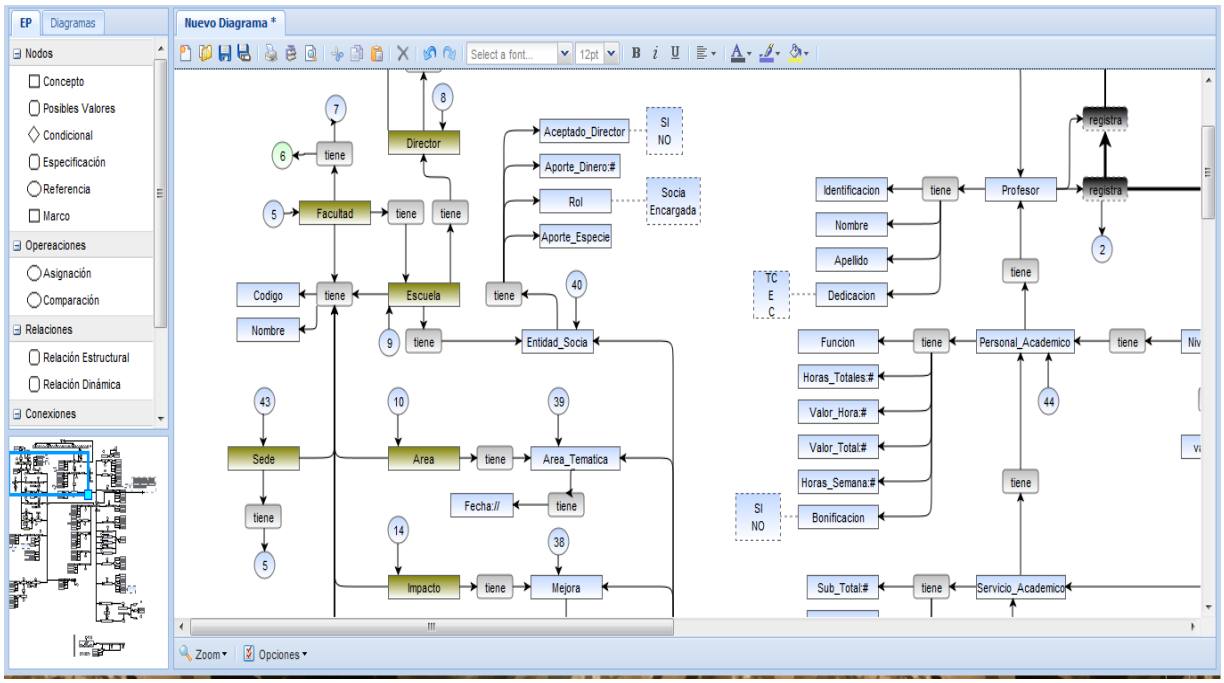


Figura 101. Un Esquema Preconceptual en UNC-PSCoder.

Una vez se tiene el Esquema Preconceptual en esta herramienta en Menú Opciones se da *click* en generar código. La clase EPtoPHP.php se encarga de analizar el XML del Esquema Preconceptual que se genera y, generar el código fuente. En la Figura 102 se presenta un XML generado en esta herramienta.

```
<mxGraphModel>
  <root>
    <mxCell id="1" parent="0"/>
    <mxCell id="13" value="" style="vertical;type=conexion;exitX=0.5;exitY=1;entryX=0;entryY=0.5" parent="1" source="8" target="6" edge="1">
      <mxGeometry x="1340" y="290" width="100" height="100" as="geometry">
        <mxPoint x="1430" y="260" as="sourcePoint"/>
        <mxPoint x="1460" y="260" as="targetPoint"/>
        <Array as="points">
          <mxPoint x="1510" y="350"/>
        </Array>
      </mxGeometry>
    </mxCell>
    <mxCell id="15" value="" style="vertical;type=conexion;exitX=0.5;exitY=1;entryX=0;entryY=0.5" parent="1" source="8" target="7" edge="1">
      <mxGeometry x="1340" y="290" width="100" height="100" as="geometry">
        <mxPoint x="1430" y="260" as="sourcePoint"/>
        <mxPoint x="1460" y="260" as="targetPoint"/>
        <Array as="points">
          <mxPoint x="1520" y="380"/>
        </Array>
      </mxGeometry>
    </mxCell>
    <mxCell id="16" value="Participante" parent="1" vertex="1">
      <mxGeometry x="1350" y="410" width="80" height="20" as="geometry"/>
    </mxCell>
    <mxCell id="17" value="Cargo" parent="1" vertex="1">
      <mxGeometry x="1550" y="410" width="80" height="20" as="geometry"/>
    </mxCell>
    <mxCell id="18" value="Meses_Totales:#" parent="1" vertex="1">
      <mxGeometry x="1550" y="440" width="80" height="20" as="geometry"/>
    </mxCell>
  </root>
</mxGraphModel>
```

Figura 102. XML de un Esquema Preconceptual en UNC-PSCoder

VI. CASO DE LABORATORIO

Con el fin de ejemplificar las reglas definidas en los capítulos anteriores, se presenta parte del software "Innsoftware". Es un software para el diligenciamiento de encuestas para medir las capacidades de innovación en las empresas desarrolladoras de software. Este software se implementó con la herramienta desarrollada.

A continuación se presentan fragmentos del Esquema Preconceptual y su interfaz gráfica correspondiente. Cabe anotar que los estilos (css) que se emplearon en ese software no se generaron con la herramienta CASE.

En la Figura 103 se ejemplifica la operación atómica "lista".

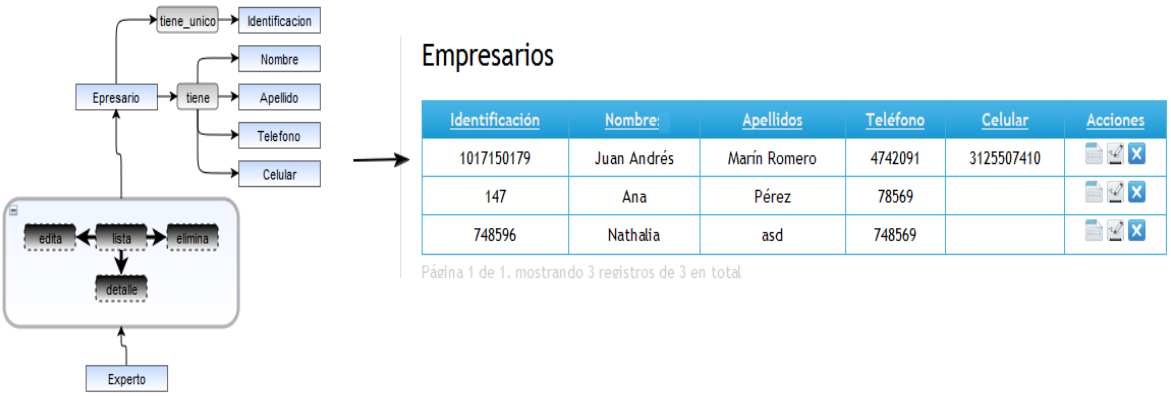


Figura 103. Innsoftware. Operación lista

Nótese que, en la Figura 103 aparece una Columna con unos íconos, de izquierda a derecha (detalle, edita, elimina), que corresponden a las implicaciones en el Esquema Preconceptual.

Es decir; para editar un empresario, se deben haber listado los empresarios, para conocer el ID del empresario a editar.

De esta interfaz el único retoque que tuvo que hacer el programador fue poner tildes y poner los íconos en las relaciones de implicación.

En la Figura 104 se ejemplifica la operación atómica "registrar", nótese que en el Esquema Preconceptual aparecen conceptos obligatorios, los cuales en la interfaz gráfica se reflejan en las etiquetas con terminación en "*". En esta interfaz también se evidencia el uso de los posibles valores para el concepto "Tipo_Identificacion".

Figura 104. Innsoftware. Operación registra.

En la Figura 105 se puede apreciar que, al darle *click* al botón "Registrar Empresa", aparece un texto en rojo "Obligatorio", el cual indica que este campo no puede quedar vacio.

Figura 105. Innsoftware. Operación registra con validación.

En la Figura 106 se ejemplifica el uso de la relación estructural de tipo "tiene_unico", donde el concepto NIT es único. Por lo tanto al intentar registrar una empresa con un NIT ya registrado, aparece un mensaje "El NIT ya existe!" y no permite registrar esta empresa.

Figura 106. Innsoftware. Ejemplo de Tiene único.

Debido a que en el Esquema Preconceptual no pueden haber relaciones M:M, es necesario especificar sus "intersecciones". En la Figura 107 se presenta un ejemplo de intersección. En el formulario "Registrar Adaptación Encuesta" se observa que aparece una lista desplegable con las encuestas y otra con las empresas.

Figura 107. Innsoftware. Ejemplo de intersección entre conceptos

Nótese que la lista desplegable que contiene las encuestas muestra el nombre de las encuestas y la lista de las empresas, muestra el nombre de todas las empresas (regla por defecto). Esto se logra mediante la especificación del comportamiento de la relación dinámica "registrar".

VII. CONCLUSIONES

7.1 Conclusiones

El uso de los Esquemas Preconceptuales, para la generación automática de código, mejora la comunicación entre el analista y el interesado. Además, disminuye los errores de comunicación que plantea Ongallo (2007). Dado que los Esquemas Preconceptuales son cercanos al Lenguaje Natural del interesado, es posible obtener una validación de la especificación de los requisitos desde las primeras etapas del desarrollo, lo cual mejora ostensiblemente la calidad e imprime celeridad en el proceso de desarrollo de software.

Además, con el uso de los aportes realizados en trabajos anteriores como Zapata *et al*. (2006), Zapata y Arango (2007) y Zapata *et al*. (2007), entre otros, es posible obtener diferentes artefactos UML 2.0 y diagramas de objetivos de KAOS, que se utilizan en la especificación, modelado y diseño de una aplicación de software. De esta forma, se automatizan estos procesos, disminuyendo la intervención del analista, con lo cual se reducen los posibles errores humanos y se mantiene la consistencia entre las necesidades del interesado y la especificación de requisitos.

La definición de reglas heurísticas para la generación automática de código a partir de Esquemas Preconceptuales garantiza la consistencia de la información. Además, se plantea la posibilidad de incluir nuevas reglas que permitan generar los diagramas que aún no se generan desde el Esquema Preconceptual y que ayuden a refinar la generación de los diagramas existentes. También, se plantea la posibilidad de definir nuevas reglas heurísticas mediante las cuales sea posible obtener prototipos funcionales, por ejemplo para dispositivos móviles.

La herramienta CASE que se implementó ofreció, entre otras, las siguientes ventajas: la posibilidad de validar el Esquema Preconceptual en su nivel mínimo, es decir; verificar que no hayan conceptos o conexiones aisladas, conceptos en plural o conceptos tipo clase repetidos. También ofrece la posibilidad de generar, automáticamente, el diagrama de clases de UML 2.0 mediante las reglas definidas en Zapata *et al.* (2007) y el diagrama entidad-relación. Al ser una herramienta WEB, mejora las condiciones de trabajo de quienes la utilicen, debido a que es posible trabajar conjuntamente entre varios analistas.

Con esta herramienta CASE se buscan la reducción en el tiempo de elaboración de los diferentes diagramas y la generación del código fuente, además de la consistencia entre los mismos. Estos son dos de los factores críticos de éxito en el desarrollo de productos de software.

De otro lado, esta herramienta se puede usar de forma pedagógica, ayudando al estudiante en la identificación de las necesidades del interesado, ya que puede obtener un prototipo totalmente funcional desde las primeras etapas. De esta manera, el estudiante, con la ayuda del interesado, puede validar la información contenida en las interfaces gráficas de usuario y puede complementar el Esquema Preconceptual. Por otro lado, el estudiante desarrolla la habilidad de mantener la atención del interesado en el tema de discusión.

7.2 Contribuciones

Esta Tesis generó un conjunto de resultados directos, que se traducen en publicaciones en revistas y congresos a nivel nacional e internacional. A continuación se presenta una breve descripción de cada uno de ellos:

7.2.1 Publicaciones relacionadas

Título	Una mirada conceptual a la generación automática de código
Autores	Carlos Mario Zapata Jaramillo John Jairo Chaverra Mojica
Revista	Revista EIA, Categoría B
Estado	Publicado

Resumen:

Existen varios métodos de desarrollo de software que impulsan la generación automática de código. Para tal fin se utilizan las herramientas CASE *(Computer-Aided Software Engineering)* convencionales, pero aún están muy distantes de ser un proceso automático y muchas de estas herramientas se complementan con algunos trabajos que se alejan de los estándares de modelado. En este artículo se presenta una conceptualización de los trabajos relacionados con la generación automática de código, a partir de la representación del discurso en lenguaje natural o controlado o de esquemas conceptuales. Así, se concluye que la generación automática de código suele partir de representaciones de la solución del problema y no desde la representación del dominio. Además, estos puntos de partida son de difícil comprensión para el interesado, lo que impide que se tenga una validación en etapas previas del desarrollo.

Título	Generación automática de código a partir del lenguaje controlado UN-Lencep
Autores	Carlos Mario Zapata Jaramillo John Jairo Chaverra Mojica Bryan Zapata Ceballos
Revista	Novena Conferencia Iberoamericana en Sistemas, Cibernética e Informática: CISCI 2010. Orlando, Florida.
Estado	Publicado

Resumen:

La captura de requisitos de software se realiza entre el analista y el interesado mediante una entrevista en Lenguaje Natural. De esta entrevista surgen unas especificaciones de la aplicación por construir, las cuales los analistas suelen representar en esquemas conceptuales. Estos esquemas se pueden elaborar en varias de las herramientas CASE (*Computer Aided Software Engineering*) convencionales, que incluso generan automáticamente parte del código de la aplicación, pero requieren que el analista interprete subjetivamente el dominio, que elabore manualmente los esquemas conceptuales y que haga una verificación manual del código fuente y los diagramas generados. Además, los esquemas que se emplean no los comprende fácilmente el interesado, lo que implica que no se tenga una validación en tiempo real. Para solucionar parcialmente estos problemas, en este artículo se definen reglas heurísticas para convertir en código Java y PHP un discurso en Un-Lencep (Universidad Nacional de Colombia—Lenguaje Controlado para la Especificación de Esquemas Preconceptuales). Esta propuesta se ejemplifica con un caso de estudio.

Título	Generación automática de interfaces gráficas de usuario a partir de Esquemas Preconceptuales
Autores	Carlos Mario Zapata Jaramillo John Jairo Chaverra Mojica
Revista	Quinto Congreso de Computación Colombiano (5CC). Cartagena, Colombia
Estado	Publicado

Resumen:

El principal objetivo cuando se desarrolla una aplicación de software es facilitar a los usuarios finales alcanzar sus objetivos de forma efectiva y eficiente. Por esta razón, es necesario incorporar técnicas que ayuden a mejorar las interfaces gráficas de usuario para mejorar la interacción entre el usuario y la aplicación. Existen varias aproximaciones a la generación automática de interfaces gráficas de usuario que se basan en los modelos conceptuales, mejorando los tiempos en el desarrollo del software. También, existen las herramientas RAD (*Rapid Application Development*), en las cuales se pueden obtener muy rápidamente las interfaces de usuario mediante el paradigma WYSIWIG, pero en estas herramientas se pierde toda calidad y consistencia con los demás modelos. Por lo general, estas propuestas suelen utilizar como punto de partida los diagramas UML, los cuales no son de fácil comprensión por parte del interesado, impidiendo una validación en tiempo real. Para solucionar parcialmente estos problemas, en este artículo se propone un método basado en reglas heurísticas para obtener las interfaces gráficas de usuario a partir de Esquemas Preconceptuales.

Título	Generación automática de código bajo el patrón MVC a partir de Esquemas Preconceptuales
Autores	Carlos Mario Zapata Jaramillo Gloria Lucia Giraldo Gómez John Jairo Chaverra Mojica
Revista	Revista Facultada de Ingeniería Universidad de Antioquia. Categoría A1.
Estado	En evaluación.

Resumen:

Existen diferentes métodos que incentivan la generación automática de código. Para tal fin, se utilizan las herramientas CASE (*Computer-Aided Software Engineering*) convencionales. Sin embargo, estas herramientas están muy distantes de automatizar completamente el proceso de generación de código, lo cual requiere una alta participación del programador para completar el código fuente. Además, el código fuente que se genera no refleja una arquitectura por capas, tal como el patrón MVC (*Model View Controller*), que es altamente conveniente para separar las reglas del negocio de la funcionalidad de la aplicación. Finalmente, muchas herramientas CASE utilizan como punto de partida diagramas que no son de fácil interpretación para un interesado, lo cual impide que el código fuente obtenido refleje las características del dominio del problema. Con el fin de solucionar parcialmente estos problemas, en este artículo se propone un conjunto de reglas heurísticas para obtener automáticamente código fuente en lenguaje de programación PHP bajo el patrón de programación MVC, a partir de Esquemas Preconceptuales. Esta propuesta se ejemplifica con un caso de estudio.

Título	Especificación del tipo y la obligatoriedad de los datos en Esquemas Preconceptuales
Autores	Carlos Mario Zapata Jaramillo John Jairo Chaverra Mojica Edward Antonio Naranjo Guzmán
Revista	Revista Ingeniería y Ciencia (EAFIT). Categoría C.
Estado	En evaluación.

Resumen:

La especificación del tipo y la obligatoriedad de los conceptos es una tarea fundamental en el desarrollo de una aplicación de software, esta tarea suele exclusiva del desarrollador sin tener presente la validación del interesado. Algunas metodologías de desarrollo de software, para esta tarea, utilizan diagramas técnicos tales como clases, entidad-relación y grafos conceptuales. Estos diagramas dificultan la interacción con el interesado, ya que son cercanos al lenguaje técnico del analista. En este artículo se propone, como posible solución a este problema, la representación del tipo de dato y su obligatoriedad para cada concepto en los Esquemas Preconceptuales, los cuales se acercan al lenguaje natural del interesado, permitiendo así su validación en las etapas iniciales del desarrollo de software.

Título	Un entorno para la generación automática de código bajo el patrón MVC a partir de Esquemas Preconceptuales
Autores	Carlos Mario Zapata Jaramillo John Jairo Chaverra Mojica
Revista	Revista Ingeniería y Ciencia (EAFIT). Categoría C.
Estado	En evaluación.

Resumen:

Cada día las herramientas CASE (*Computer-Aided Software Engineering)* han sido de mayor apoyo en las diferentes fases del desarrollo de software, a tal punto, que hoy en día es casi imposible pensar en un desarrollo de software sin el apoyo de una herramienta CASE. Una de las características más deseadas de estas herramientas es la generación automática de código. Por ello, en este artículo se propone una herramienta CASE para generar automáticamente código ejecutable bajo el patrón de programación MVC (*Model View Controller*), en el lenguaje de programación PHP, a partir de los Esquemas Preconceptuales.

7.2.2 Productos de Software

Con el uso de la herramienta CASE UNC-PSCoder se desarrolló un aplicativo funcional. A continuación se relata una breve descripción:

- **InnSoftware:** Software para el diligenciamiento de encuestas con el fin de medir la capacidad de innovación en las empresas de software. Proyecto conjunto entre Universidad Nacional, Antójate de Antioquia, Colciencias e

InterSoftware. La herramienta CASE generó el 90% de la aplicación, se tuvo que programar manualmente la parte de reportes y estadísticos en gráficos.

VIII. TRABAJO FUTURO

Pese a los diversos aportes realizados en esta Tesis aún persisten algunas limitantes, de las cuales se pueden derivar diferentes líneas de investigación. A continuación se listan algunas de ellas:

- Definir nuevas reglas heurísticas que permitan general el código fuente en otros lenguajes de programación como, Python, C#, ASP.NET y Perl.

- Definir nuevos elementos en el Esquema Preconceptual que permitan identificar la longitud en el tipo de dato de los conceptos.

- Definir nuevos elementos en el Esquema Preconceptual que permitan incorporar parámetros de usabilidad en la interfaz gráfica de usuario sin perder de vista la validación del interesado.

- Definir nuevos elementos en el Esquema Preconceptual que permitan generar procedimientos automáticos (*triggers*) para las diferentes bases de datos.

- Definir reglas heurísticas que permitan generar, automáticamente, código fuente para dispositivos móviles.

- Definir reglas heurísticas que permitan generar, automáticamente, los diagramas UML 2.0 que aún no se generan desde el Esquema Preconceptual y refinar las reglas existentes.

- Definir mecanismos que permitan mantener sincronizado el código fuente con el Esquema Preconceptual, de modo tal que los cambios realizados en el código fuente se reflejen en el Esquema Preconceptual y viceversa. Esto se realizaría con el fin de mantener actualizada toda la documentación correspondiente al desarrollo.

REFERENCIAS

Almendros-Jiménez, J. e Iribarne, L. (2005). Designing GUI components from UML uses cases". *Proc. 12th IEEE Int. Conf. and Workshop on the Engineering of Computer Based Systems (ECBS'05)*. IEEE Computer Society Press, pp 210-217.

Beier, G. y Kern, M. (2002). Aspects in UML models from a code generation perspective. *In 2nd Workshop on Aspect-Oriented Modelling with UML*, Dresden, Germany: 2002

Bennett, J. Cooper, K. y Dai, L. (2009). Aspect-oriented model-driven skeleton code generation: A graph-based transformation approach. *Science of Computer Programming.*

Bodart, F. y Vanderdonckt, J. (1996).Widget Standardization through Abstract Interaction Objects. *Proceedings of 1st Int.Conf. on Applied Ergonomics ICAE'96*. pp 300-305.

Booch, G., Rumbaugh, J. y Jacobson, I. (1998). The Unified Modeling Language User Guide. Addison Wesley. 512 p.

Buchholz, E. y Düsterhöft, A. (1994). Using Natural Language for Database Design. *Proceedings Deutsche Jahrestagung für Künstliche Intelligenz*.

Burkhard, D. y Jenster, P. V. (1989). Applications of Computer-Aided Software Engineering Tools: Survey of Current and Prospective Users, Data Base. *ACM SIGMIS Database*. Vol. 20, No. 3, pp 28–37.

Chachkov, S. y Buchs, D. (2001). From formal specifications to ready-to-use software components: the concurrent object oriented Petri Net approach. *International*

Conference on Application of Concurrency to System Design, Newcastle, IEEE Computer Society Press. pp 99-110.

Chen, P. English sentence structure and entity relationship diagrams, Information Sciences. Vol. 29. No. 2. (1983). pp 127-149.

Delphi. 2009. Disponible en http://delphi.com/

Elkoutbi, M. y Keller, R. (2000). User Interface Prototyping based on UML Scenarios and High-level Petri Nets. *Application and Theory of Petri Nets (21 ATPN). Springer-Verlag*

Elkoutbi, M. Khriss, I. y Keller, R. (1999). Generating user interface prototypes from scenarios. *RE'99, Fourth IEEE International Symposium on Requirements Engineering*, pp 150-158.

Engels, G. Hücking, R. Sauer, S. y Wagner. (1999). A: UML Collaboration Diagrams and Their Transformation to Java. *LNCS*, Vol. 1723, pp. 473-488. Springer, Berlin

Fliedl, G. y Weber, G. (2002). Niba-Tag. A Tool for Analyzing and Preparing German Texts. *Management Information Systems*, Vol. 6. pp 331-337.

Fowler, M. (2004). UML Distilled: A brief guide to the Standard Object Modeling Language. *Addison – Wesley*, tercera edición. 208 p.

Fowler, M. y Scott, K. (1999). *UML gota a gota*. Ciudad de Mexico, Mexico: Adison Wesley.

FUJABA, University of Paderborn, Software Engineering Group. Fujaba Tool Suite. Disponible En: http://wwwcs.uni-paderborn.de/cs/Fujaba/index.html

Gamma, E. Helm, R. Johnson, R y Vlissides, J. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. *Addyson Wesley*.

Gangopadhyay, A. (2001). Conceptual modelling from natural language functional specifications, *Artificial Intelligence in Engineering*. Vol. 15. No. 2, pp 207–218.

Geiger, L. y Zündorf, A. (2006). TOOL modelling with fujaba. *Electronic Notes in Theoretical Computer Science*. No 148, pp 173–186.

Genera. (Genova 7.0), 2000. Disponible en http://www.genera.no/2052/tilkunde/0904/default.asp

Génova, G. Del Castillo, C.R. y Llorens, J. (2003). Mapping UML associations into Java code. *Journal of Object Technology*. Vol. 2. No. 5. pp 135-162.

Gessenharter, D. (2008). Mapping the UML2 Semantics of Associations to a Java Code Generation Model. *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*. 2008. pp 813-827.

Gomes, B. Moreira, A. y Deharbe, D. (2007). Developing Java Card Applications with B. *Electronic Notes in Theoretical Computer Science*. Vol. 184. pp 81-96.

Groher. I, y Schulze. S. (2003). Generating aspect code from UML models. *The 4th AOSD Modeling With UML Workshop*, Citeseer, 2003.

Harmain, H. y Gaizauskas, R. (2000). CM-BUILDER: An Automated NL-based CASE Tool. *Proceedings of the fifteenth IEEE International Conference on Automated Software Engineering (ASE' 00)*. Grenoble. pp 45-53.

IEEE. 1998, Recommended Practice for Software Requirements Specifications Software, Standard 830, Engineering Standards Committee of the IEEE Computer Society.

Booch, G. Jacobson, I. y Rumbaugh, J. (1999). The unified software development process. 1999. *Addison-Wesley.*

Kantorowitz, E. Lyakas, A. y Myasqobsky, A. (2003). Use case-oriented software architecture. *Proc. of Workshop 11, Correctness of Model-based Software Composition (ECOOP'2003).*

Kotonya, G. y Sommerville, I. (1998). Requirements engineering with viewpoints. *Software Engineering Journal*. Vol. 11, No. 1, 1196, pp 5-18.

Krasner, G. y Pope, S. (1998). A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of Object Oriented Programming.* Vol. 1 No. 3, pp 26-49.

Leffingwell, D. y Widrig, D. (2003), Software Requirements: A Use Case Approach. Boston, USA: Pearson.

Leite, J. y Gilvaz, A. (1996). Requirements Elicitation Driven by Interviews: The Use of Viewpoints. *En: 8th International Workshop on Software Specification and Design*. pp. 85–94. Schloss Velen, Alemania.

Leite, J. (1987). A survey on requirements analysis. *Advanced Software Engineering Project*. Reporte Técnico RTP–071. University of California at Irvine.

Long, Q. Liu, Z. Li, X. y Jifeng, H. (2005). Consistent code generation from uml models. *Australia Conference on Software Engineering (ASWEC).* Australia, *IEEE Computer Sciety*. 2005

Lozano. M, González. P, Ramos. I, Montero. F, y Pascual. J. (2002). Desarrollo y generación de interfaces de usuario a partir de técnicas de análisis de tareas y casos de uso. *INTELIGENCIA ARTIFICIAL*, Vol. 6, pp. 83-91.

Macromedia, 2009. Disponible en http://www.macromedia.com.

Mammar, A. y Laleau, R. (2006). From a B formal specification to an executable code: applicational to the relational database domain. *Information and Software Technology*. Vol. 48, No. 4, pp 253-279.

Mich, L. (1996). NL-OOPS: From Natural Lenguaje to Object Oriented Ruirements using the Natural Languaje Processing System LOLITA. *Journal of Natural Language Engineering.* Cambridge University Press. Vol. 2. No. 2. pp 161-187.

Microsoft. 2009. Disponible en http://msdn.microsoft.com/es-co/default.aspx.

Mortensen, K.H. (1999). Automatic Code Generation from Coloured Petri Nets for an Access Control System. *Second Workshop on Practical Use of Coloured Petri Nets.* 1999. pp 41-58.

Mortensen, K.H. (2000). Automatic Code Generation from Coloured Petri Nets for an Access Control System". *Lecture Notes in Computer Science.* 2000. pp 367-386.

Nassar, M. Anwar, A. Ebersold, S. Elasri, B. Coulette, y Kriouile, A. (2009). Code Generation in VUML Profile: A Model Driven Approach. *IEEE/ACS International Conference on Computer Systems and Applications*, Rabat, Marruecos. pp 412-419.

Niaz, I. A. y Tanaka, J. (2003). Code Generation from UML Statecharts. *in Proc. 7 th IASTED International Conf. on Software Engineering and Application SEA*, pp 315-321.

Niaz, I. y Tanaka, J. (2004). Mapping UML statecharts to Java code. *In Proc. IASTED International Conf. on Software Engineering (SE 2004)*. pp 111–116.

Niaz, I. y Tanaka, J. (2005). An Object-Oriented Approach To Generate Java Code From UML Statecharts. *International Journal of Computer & Information Science*, Vol. 6, No. 2, pp 315–321.

Muñeton, A. Zapata, C. M. y Arango, F. (2007). Reglas para la generación automática de código definidas sobre metamodelos simplificados de los diagramas de clases, secuencias y máquinas de estados de UML 2.0. *Dyna*, vol 74. N153, pp 267-283.

OMG. (2010) UML 2.0 Superestructure Specification. Disponible en línea en http://www.omg.org/uml/.

Ongallo, C. (2007). Manual de comunicación. Madrid, España: Dykinson.

Overmyer, S.P. Lavoie, B. y Rambow, O. (2001). Conceptual Modeling through linguistic analysis using LIDA. *Proceedings of ICSE,* Washington, DC, 2001. pp 401 - 410.

Peckham, J. y MacKellar, B. (2001). Generating code for engineering design systems using software patterns. In *Artificial Intelligence in Engineering*. Vol. 15. No 2, pp. 219-226.

Pilitowski, P. y Dereziska, A. (2007). Code Generation and Execution Framework for UML 2.0 Classes and State Machines**.** *Innovations and Advanced Techniques in Computer and Information Sciences and Engineering.* Springer Netherlands. pp 421-427.

PoweBuilder. 2009. Disponible en http://www.mtbase.com/productos/desarrollo/powerbuilder.

Ramos, I. Montero, F. y Molina, J.P. (2002). Desarrollo y generación de interfaces de usuario a partir de técnicas de análisis de tareas y casos de uso. *Revista Iberoamericana*, Vol. 16, pp 83-91.

Ramkarthik, S. y Zhang, C. (2006). Generating Java Skeletal Code with Design Contracts from Specifications in a Subset of Object Z. In *5th IEEE/ACIS International Conference on Computer and Information Science, 2006 and 2006 1st IEEE/ACIS International Workshop on Component-Based Software Engineering, Software Architecture and Reuse. ICIS-COMSAR 2006* pp. 405-411. IEEE Computer Society, 2006.

ROSE, IBM Corporation. Rational Rose ArchitectTM. En: http://www-306.ibm.com/software/awdtools/architect/swarchitect/index.html

RUIZ, L. (2008). Reglas para la generación de modelos de OO-Method a partir de los esquemas preconceptuales. Tesis Universidad Nacional de Colombia. Medellín, Colombia.2008.

Saiedian, H. (1997). Una evaluación del modelo entidad relación extendido. *Information and Software Technology*. Vol. 39, pp 449-462.

Samuel, P. Mall, R. y Kanth, P. (2007). Automatic test case generation from UML communication diagrams. *Information and Software Technology*, Vol. 49. No. 2, pp 158–171.

Shahidi, S. y Mohd, Z. (2009). Using Ethnography Techniques in Developing a Mobile Tool for Requirements Elicitation. *En: International Conference on Information Management and Engineering*. Kuala Lumpur, Malaysia. pp. 510-513.

Sommerville, I. (2007). Software Engineering. *Londres, Inglaterra: Editorial Pearson*.

TOGETHER™, BORLAND Software Corporation. Borland Together Architect. En: http://www.borland.com/us/products/together/index.html.

Usman, M. Nadeem, A. y Kim, T. (2008). UJECTOR: A tool for Executable Code Generation from UML Models. *Advanced Software Engineering & Its Aplications*, 2008, pp. 165-170.

Usman, M. y Nadeem, A. (2009). Automatic Generation of Java Code from UML Diagrams using UJECTOR. *International Journal of Software Engineering and Its Applications (IJSEIA).* Vol 3. No. 2. pp 21-37.

Yao, W. y He, X. (1997). Mapping Petri nets to concurrent programs in CC++. *Information and Software Technology*, Vol *39.* No. 7, pp 485-495.

Zapata, C. Gelbukh, A. y Arango, F. (2006). Pre-conceptual Schema: a UML Isomorphism for Automatically Obtaining UML Conceptual Schemas. *Research in Computing Science: Advances in Computer Science and Engineering*, Vol. 19, pp. 3–13.

Zapata, C. M. Ruiz, L. M. y Villa, F. A. (2007). UNC - Diagramador una herramientas *upper* CASE para la obtención de diagramas UML desde esquemas Preconceptuales. *Revista universidad EAFIT*, Vol. 43, No 147, pp. 68-80.

Zapata, C. M. Gelbukh, A. y Arango, F. (2006). UN-Lencep: Obtención Automática de Diagramas UML a partir de un Lenguaje Controlado. Taller de Tecnologías del Lenguaje, San Luis Potosí, septiembre, 2006, pp. 1-6

Zapata, C. M. Manjarrés, R. González. G. (2010). Reglas para la Obtención de la Especificación Declarativa a partir de la Especificación Gráfica de un Metamodelo. *Novena Conferencia Iberoamericana en Sistemas, Cibernética e Informática: CISCI 2010*. Orlando, Florida