DETECCIÓN DE PATRONES DE DISEÑO EN CÓDIGOS ORIENTADOS A OBJETOS

Patricia Zavaleta Carrillo
Abril Ayala Sánchez*

Resumen

Uno de los objetivos más importantes de la Ingeniería de Software es el diseño e instrumentación de métodos y técnicas que permitan producir software de calidad con un mínimo de tiempo de desarrollo. Uno de estos métodos es el desarrollo de software basado en componentes, los cuales han sido previamente construidos y tienen probada su calidad. Un componente debe obedecer a una arquitectura bien diseñada, en la que se vean disminuidos algunos problemas como son las dependencias funcionales, que sea cerrada a las modificaciones y abierta a las extensiones, y que además se utilice frecuentemente. En este caso se puede hablar de patrones de diseño. Si los componentes están organizados como patrones de diseño, se puede asegurar en mayor medida su calidad y también su reusabilidad.

Palabras clave

Patrones de diseño, desarrollo basado en componentes, reusabilidad.

Introducción

Uno de los objetivos más importantes de la Ingeniería de Software es el diseño e instrumentación de métodos y técnicas que permitan producir *software* de calidad con un mínimo de tiempo de desarrollo. Uno de estos métodos es el desarrollo de *software* basado en componentes, los cuales han sido previamente construidos y tienen probada su calidad. Para que los componentes sean reusables deben cumplir con las siguientes características: que obedezcan a una arquitectura bien diseñada, es decir, que tengan la forma de plantillas genéricas, en las que se vean disminuidos algunos problemas como son las dependencias funcionales; que dichas plantillas sean cerradas a las modificaciones y abiertas a las extensiones [MAR96], y que, además, sean recurrentes, es decir, que se utilicen frecuentemente. Cuando existen componentes que cumplen con estas características se puede hablar de patrones de diseño, los cuales se definen [GAM95] como un conjunto de clases que trabajan en colaboración en la solución de un problema que se presenta recurrentemente bajo un mismo contexto, o de problemas similares; y que además han sido analizadas y diseñadas de conformidad a las características antes mencionadas. Si los componentes están organizados como patrones de diseño, se puede asegurar en mayor medida su calidad y también su reusabilidad.

El desarrollo de *software* se está moviendo rápidamente de una forma artesanal hacia el proceso de ingeniería y manufactura a gran escala. Un movimiento hacia componentes de software está siendo manejado, por la urgente necesidad de contar con artefactos de software reusable que cuenten con una mejor calidad y que puedan ser configurados en diferentes aplicaciones, para satisfacer las necesidades de cambio con un mínimo de inversión en costo y esfuerzo [THO95]. Los primeros componentes fueron llamados circuitos integrados de software, los cuales aparecieron como paquetes sellados con un conjunto específico de entradas y salidas, a estos componentes se les conoce como mensajes, interfaces de aplicaciones (API´s) y protocolos [THO95].

La evolución de los modelos del ciclo de vida del *software* ha sido constante, transitando desde los tradicionales modelos de cascada a los modelos más modernos como el de prototipos rápidos, implementados con interfaces interactivas que permiten producir software al nivel de prototipo de manera inmediata hasta lograr la satisfacción de los requerimientos del cliente o del usuario final. Una tendencia tecnológica actual es la definición de nuevos modelos del

*Docentes de la Dependencia Área Ciencias de la Información en la Universidad Autónoma del Carmen.

ciclo de vida del *software*, sobre los cuales se construyan ambientes de desarrollo que incluyan herramientas automáticas o semiautomáticas, visuales e interactivas, que cubran tanto el desarrollo de componentes como la construcción de sistemas basándose en el uso de éstos. Margaret Burnett [BUR95] ha propuesto que los nuevos modelos de desarrollo de *software* contemplen al menos cuatro etapas: 1. Mecanismos para la clasificación y recuperación de componentes, 2. Mecanismos para el encapsulado de nuevos componentes, 3. Mecanismos para la integración de componentes y, 4. Mecanismos para la ejercitación de componentes.

Un patrón de diseño nombra, abstrae e identifica los aspectos clave de una estructura común de diseño que la hace útil para crear un diseño orientado a objetos reusable. El patrón de diseño identifica las clases participantes y sus instancias, sus funciones y colaboraciones y la distribución de sus responsabilidades. Cada patrón de diseño se enfoca a un tópico o problema de diseño orientado a objetos en particular. Describe cuando se aplica, si puede o no ser aplicado en vista de otras restricciones de diseño y las consecuencias y negociaciones de su uso. El catálogo de Gamma et.al.[GAM95] clasifica 23 patrones de diseño, de acuerdo a dos criterios: propósito y alcance, entre los que están el *Factory method, adapter, interpreter, template method, abstract factory, builder, adapter, bridge, command, strategy, visitor*, entre otros.

Investigaciones relacionadas con los patrones de diseño

De los trabajos relacionados con el término de patrones de diseño, se tienen varios enfoques, entre los que hay dos direcciones principales. una de ellas se enfoca a la implementación de patrones de diseño en el desarrollo de sistemas de *software*, ya sea incluyendo los patrones de diseño como parte de los lenguajes de programación o implementados como parte del diseño del sistema de *software*; la otra es en la investigación en la reingeniería acerca de patrones de diseño, la cual se enfoca en dos áreas: una de ellas se refiere a la identificación de patrones en códigos existentes y la otra es aplicar refactorización para incluir patrones en los sistemas, con el fin de poder obtener componentes reusables de estos sistemas y poder incluirlos en *frameworks* donde se puedan controlar, evolucionar y administrar estos componentes.

Investigación de patrones de diseño implementados en el desarrollo de *software*

Dentro de este enfoque existen varias investigaciones, como los trabajos desarrollados por F.J. Budinsky et. al. [BUD96] que desarrollaron una herramienta que automatiza la implementación de patrones de diseño; esto es realizado al proporcionarle al sistema algunas piezas de información, nombres de aplicación específicos para los participantes en un patrón, se crea la declaración y definiciones de clases que implementan el patrón; posteriormente, el usuario tiene que adicionar este código al resto de la aplicación.

Otra investigación está enfocada a la inclusión de patrones de diseño como lenguaje de construcción, donde Jan Bosh de la Universidad de Kariskrona / Ronneby [BOS98] presenta un modelo de objeto en capas, que provee un lenguaje de soporte para la representación explícita de los patrones de diseño en el lenguaje de programación.

En otro enfoque, se utilizan los patrones de diseño ya existentes para solucionar nuevos problemas, como describen Ted Foster y Hiping Zhao [FOS98] en el modelado de un sistema de transportes, donde utilizan patrones de diseño que proveen un formalismo para capturar componentes concurrentes, frecuentemente dentro de los modelos de transportes público.

Baumgartner, Läufer y Russo [CHA00] analizaron patrones de diseño desde muchas fuentes, e identificaron pocas características clave que podrían adicionarse a C++ para hacer los patrones de diseño más fáciles de expresar (interesantemente) todas estas características del lenguaje están presentes en algunos lenguajes.

Investigación de patrones de diseño en la reingeniería

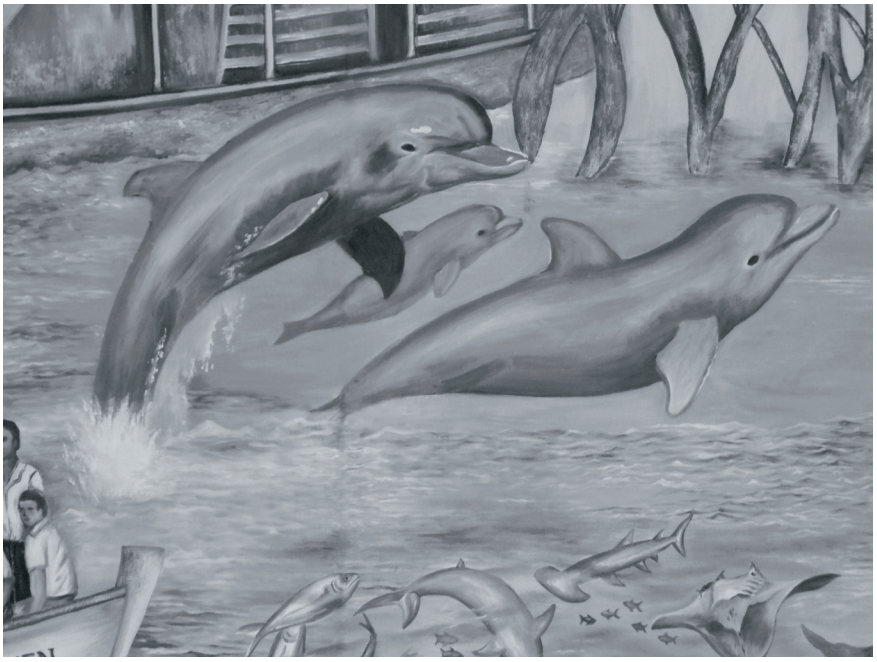
Dentro de este enfoque se trata la recuperación de los patrones de diseño en sistemas de *software* existentes, esto es con el fin de obtener un mejor entendimiento de los sistemas y poder hacer de ellos una base de componentes que tengan las características necesarias para que puedan ser reusados. Así tenemos diferentes investigaciones que se describen a continuación:

BACKDOOR (*Backwards Architecture Concerned with Knowledge Discovery of OO Relationships*) [SHU96] es un método inductivo[1] (no automático) que ayuda a descubrir patrones de diseño en

[1]Se le denomina inductivo, ya que primero localiza patrones de diseño potenciales, posteriormente realiza una comparación de ellos con los patrones de diseño existentes con la finalidad de hacer una mejor detección, además si encuentra patrones nuevos, los anexa al conjunto de patrones de referencia.

sistemas de *software* orientados a objetos. Proporciona un conjunto de procedimientos definidos rigurosamente para que puedan ser repetibles y utilizables por personas que no están familiarizadas con procesos de ingeniería inversa. La principal salida del proceso al aplicar el método, es una base de conocimientos que describe que patrones han sido usados a la fecha por una organización. Este método consta de seis pasos secuenciales e interactivos. Su principal desventaja es que se trata de un método manual, por lo tanto requiere de mucha intervención por parte del usuario.

Recuperación de diseño mediante búsqueda automática de patrones de diseño estructurales en software orientado a objetos [KRA96], en esta investigación se desarrollo una herramienta denominada PAT (Program Análisis Tool), en esta se extrae información de diseño directamente de los archivos de cabecera de C++ y se almacenan en un repositorio. Una simple consulta de Prolog es usada para la búsqueda de todos los patrones. En esta herramienta únicamente se reconocen instancias de algunos patrones de diseño estructurales, los cuales son: *adapter, bridge, composite, decorator y proxy.*



Kyle Brown describe el desarrollo de su tesis de maestría [BRO97], en la que demuestra la viabilidad de desarrollo de programas para detectar el uso de patrones de diseño de *software* en programas realizados en lenguaje *Smalltalk*. Para este fin examina la estructura de los patrones de diseño, determina la naturaleza de lo que hace que un patrón de diseño se detecte por medios automatizados y bosqueja algoritmos por medio de los cuales un pequeño conjunto de patrones de diseño pueden ser detectados. Documenta el desarrollo de una herramienta de *software* (KT) la cual recupera información de diseño desde código *Smalltalk*, y la usa para detectar algunos patrones de diseño como son: *composite, decorator, y template method*. Muestra el diagrama de clases del código analizado, utilizando la herramienta CASE Rational ROSE.

La herramienta DP++ [BAN98] automatiza la detección, identificación y clasificación de patrones de diseño estructurales y algunos de los patrones de comportamiento en programas C++. Esta herramienta se basa en las relaciones estructurales entre las clases y objetos, para identificar usos de patrones de diseño en programas orientados a objetos; relaciones estructurales como: clases abstractas, clases base y subclases, plantillas de clases (*template classes*), relaciones de herencia, agregación por contención física de variables de instancia y agregación por referencia/apuntador a variables de instancia. Esta herramienta utiliza un algoritmo de reconocimiento para cada patrón de diseño reconocido en ella. Además despliega una ventana con el modelo de clases de la estructura del programa y una ventana con una vista de árbol que despliega la jerarquía de clases del proyecto.

Ingeniería inversa de componentes de diseño basada en patrones [KEL99], es una investigación con enfoque principal a la recuperación de información referente a lo racional detrás de las decisiones de diseño de grandes sistemas de *software*. Aquí se presenta un medio ambiente para la ingeniería inversa de componentes de diseño basado en las descripciones estructurales de patrones de diseño.

Implementación de gramática del lenguaje C++ y reconocimiento de dos patrones de diseño en el IPADIC++

El IPADIC++ es un sistema que detecta patrones de diseño en códigos hechos en lenguaje C++, pero no considera la gramática completa del lenguaje C++ en la detección de patrones de diseño. Esta limitante conduce a que el analizador del código que se explora no reconozca todas las secuencias o cadenas de código aunque éstas sean bien formadas de conformidad con las reglas de la gramática estándar ANSI-ISO del lenguaje C++; por lo que se planteó como una necesidad, incluir en el analizador del IPADIC++ la gramática estándar de C++, además incluir el reconocimiento de otros dos patrones de diseño.

Análisis del IPADIC++

Se realizó un análisis del IPADIC++, para determinar cómo implementar la gramática del lenguaje C++ a la herramienta, de tal análisis se concluyó que era necesario utilizar algunos de los conceptos como la definición del modelo canónico y la definición del autómata de reconocimiento para los patrones de diseño, así como de la estructura general de entradas y salidas del sistema.

El sistema está formado por dos módulos. En el primer módulo se obtiene la forma canónica del código fuente. En este módulo se implementó un analizador sintáctico que realiza la compilación del código fuente con respecto a la gramática completa del lenguaje C++. El segundo módulo realiza el reconocimiento de patrones de diseño. A este módulo se le añadieron las acciones correspondientes para la detección de los patrones de diseño *Strategy* y *Template method*, también se modificó el funcionamiento de la interfaz con el usuario para optimizar el tiempo de ejecución del sistema.

Implementación de la gramática del lenguaje C++

La gramática del lenguaje C++ se encuentra implementada en un analizador sintáctico desarrollado en *JavaCC* por *Sreenivasa Viswanadha* [VIS96], el cual se analizó para implementarlo en el primer módulo del sistema. Esta herramienta está formada por clases y métodos de Java que ayudan en la determinación de los alcances de las variables que forman el programa analizado, la clase CPPParser es la que comienza la ejecución del análisis, abriendo el archivo con extensión CPP a analizar. El anali-

zador sintáctico no reconoce el salto de línea y las directivas del preprocesador, dentro de las derivaciones de declaración es donde se extrae la información referente a las clases y las relaciones de herencia, agregación, asociación e instanciación. Y dentro de las derivaciones de la declaración de función se extrae la información referente a las funciones como es su nombre, el alcance que tiene cada función y la localización de estructuras *if* o *switch* anidadas.

Implementación de reconocimiento de los patrones de diseño a identificar en el sistema

Para el reconocimiento de los patrones de diseño se requiere de la siguiente información, para el ***Factory method***, las clases cliente, creador_abstracto, creador_concreto [n], producto_abstracto y producto_concreto[n], las relaciones herencia, instanciación y asociación, se requiere de encontrar al menos una clase derivada (creador_concreto[n]) de otra clase abstracta (creador_abstracto) y que esta clase derivada efectué la instanciación de un objeto de clase (producto_concreto[n]) la cual debe ser derivada de la clase abstracta (producto_abstracto); o encontrar solo una clase que instancie objetos de otra clase y hacer la observación de que se pueden incluir las clases base de cada una de ellas para que se cuente con el patrón. Del patrón **Strategy** clases participantes (context, strategy que es una clase abstracta y concrete_strategy [n]), relaciones (herencia y agregación), para el reconocimiento de este patrón hay dos opciones: Si se encuentra la instrucción *switch* o *if* (anidados en más de tres niveles), se puede sugerir la implementación de este patrón poniendo el código correspondiente a cada opción del *switch* en un método interfaz dentro de una clase concrete_strategy. Además de implementar una clase strategy, de la que se derivaran las clases concrete_strategy, en esta clase definir un método virtualmente puro interfaz. Y finalmente, implementar una clase contexto en la que se manejarán las opciones del *if* o *switch*, agregar a esta clase la clase strategy. Por estructura se tendrían que encontrar clases concrete_strategy que hereden una interfaz común para la implementación de un algoritmo y una clase contexto que mantiene una referencia a un objeto strategy, se puede definir una interfaz que deja que la clase strategy acceda a los datos o pasarse la clase contexto como un argumento para operaciones strategy.

Arquitectura del sistema

La herramienta desarrollada está constituida por dos módulos (figura 1): *Módulo para la transformación del código C++ a la forma canónica y Módulo de reconocimiento de patrones de diseño*; implementados en el lenguaje de programación **Java** y en el generador de parsers javacc [MET--].

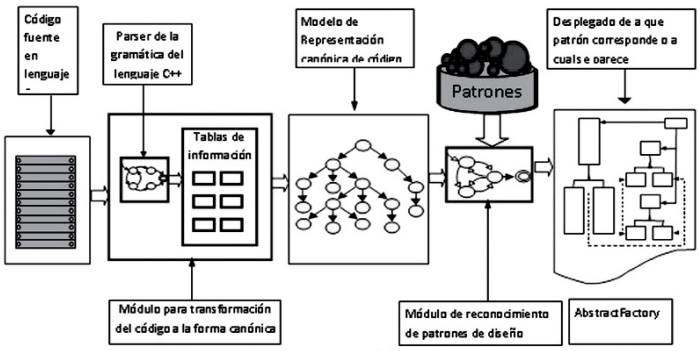


Figura *1* Estructura general de la herramienta

Fuente: Elaboración propia

Módulo para la transformación del código C++ a la forma canónica

En este modulo se convierte el código escrito en lenguaje C++ a su forma canónica. Tiene como entrada el archivo de definición de las clases que forman un programa escrito en lenguaje C++, estas definiciones pueden estar en uno o varios archivos; en el caso de que estén en varios archivos debe de incluirse en cada uno de los archivos la directiva *#include* con el nombre del archivo o archivos en los que se encuentra la definición de las clases de las que hace uso, es importante el orden en que se encuentren las definiciones de estos. Las clases principales en este modulo son CPPParser y ManTabData. CPPParser es la clase que realiza el análisis sintáctico de la gramática del lenguaje C++, ManTabData es la clase manejadora de las tablas de información necesaria para obtener la forma canónica del código analizado.

Módulo de reconocimiento de patrones de diseño

En este módulo se implementa el reconocimiento de los patrones de diseño, toma como entrada la forma canónica del código fuente y el patrón de diseño a ser localizado, y a partir de un análisis de éste código detecta si se encontró el patrón de diseño buscado. Se agregó al módelo, la implementación para el reconocimiento de los patrones de diseño *strategy* y *template method*.

Interfaz al usuario

La interfaz implementada se muestra en la figura 2, la cual consta de un menú desplegable y tres paneles de datos.

En el área de texto del primer panel se despliega el código fuente a analizar, en el área de texto del siguiente panel se muestra la forma canónica del código fuente y la forma canónica del patrón de diseño elegido para detectar en el código fuente, en el tercer panel se tienen cuatro áreas de texto la primera para mostrar las relaciones del código fuente con librerías de C++, la segunda para el resultado de la detección de contadores if's anidados con más de tres niveles o switch con más de tres casos, la tercera para mostrar el resultado de la búsqueda de métodos plantilla en las clases y en la última se muestra el resultado de la detección del patrón de diseño.

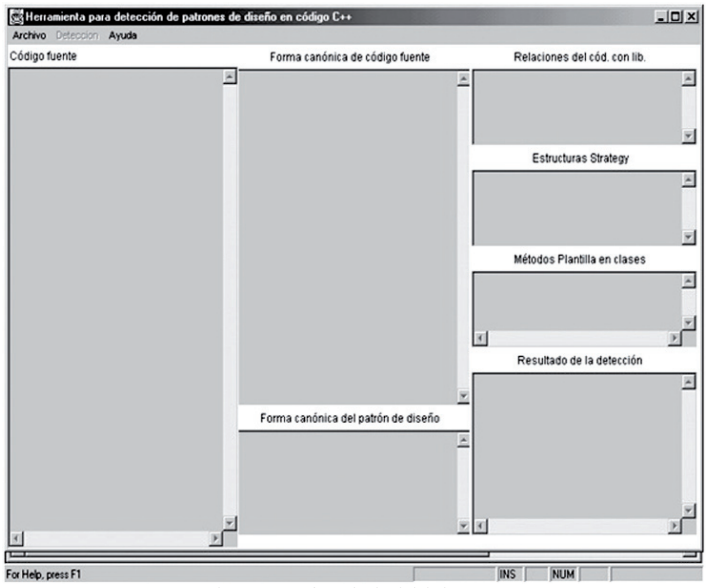


Figura 2 Interfaz principal del sistema

Evaluación del sistema

El criterio principal de prueba, es que se deben de obtener formas canónicas semejantes a las obtenidas en los casos de estudio utilizados en las pruebas realizadas al IPADIC++ antes de la implementación de la gramática, por otra parte, se prueba el reconocimiento de los nuevos patrones de diseño implementados.

Las pruebas se realizaron sobre tres conjuntos de programas de aplicación codificados en lenguaje C++: El conjunto de programas denominado "A", consta de programas bajados de Internet utilizados para realizar el plan de pruebas del IPADIC++ [CAS99], en los que de antemano se sabe que hay implementado un patrón de diseño. El conjunto de programas llamado "B", consistió en el sistema de *software*, producto de una tesis de maestría [SAN94], desarrollada en el Centro Nacional de Investigación y Desarrollo Tecnológico, en la que de antemano se sabe que la programación no fue desarrollada apegándose a algún catálogo de patrones de diseño, este conjunto es útil para comprobar si el sistema localizaría algún patrón de diseño o bien a un conjunto de clases que se aproximarán a uno, y el conjunto "C", que contiene dos programas, en el programa prueIfAn.cpp se implementaron estructuras *if's* y *switch* anidadas en más de tres niveles, para comprobar el reconocimiento del patrón de diseño *strategy*, y en el programa pruetm.cpp se implementaron dos *template method.*

Los casos de prueba aplicados a la herramienta, dieron los resultados esperados de acuerdo al análisis manual realizado a cada uno de los códigos que se probaron.

Conclusiones

•Los patrones de diseño son una solución probada de diseño en el reúso de componentes.

•La implementación de la gramática completa del lenguaje C++ permitió realizar el reconocimiento de cualquier código hecho en lenguaje C++, que obedeciera al estándar ANSI-ISO.

•De acuerdo a los resultados obtenidos en las pruebas aplicadas al sistema, el IPADIC++ realiza detección de los patrones de diseño implementados en el mismo.

Bibliografía

[BAN98] BANSIYA, Jagdis,."Automating Design-Pattern Identification", Dr. Dobb's Journal, http://www.ddj.com/ddj/1998/1998-06/lead/lead.htm, June 1998.

[BOS98] BOSH, Jan, "Design patterns as language constructs", University of Karlskrona/ Ronneby, JOOP, mayo, 1998, pp. 18-25.

[BRO97] BROWN, Kyle. "Design Reverse-engineering and Automated Design Pattern Detection in Smalltalk". clasic thesis. http://www2.ncsu.edu/eos/info/tasug/kbrown/thesis2.htm.

[BUD96] BUDINSKY, F.J., et all, "Automatic code generation from design patterns", IBM System Journal, Vol. 35, No. 2, 1996 – Object Technology.

[CAS99] CASTRO ESPINOSA, Félix Agustín, "Sistema de identificación de patrones de diseño en código C++", (Tesis de Maestría en Ciencia en Ciencias Computacionales, Cuernavaca, Morelos: Cenidet, 1999), p.87.

[CHA00] CHAMBERS, Craig, HARRISON, Bill and VLISSIDES, John, "A debate on language and tool support for design patterns", PLOPL 2000, Boston MA USA, ACM 2000.

[FOS98] FOSTER, Ted and ZHAO, Hiping, "Modelig Transport Objects with Patterns", JOOP, enero de 1998, Pp. 26-32.

[GAM95] GAMMA, Erich, et all, Design Patterns Elements of Reusable Object-Oriented Software, Addison Wesley Professional Computing Series.1995.

[KEL99] KELLER, Rudolf K., et all, "Pattern- based reverse-engineering of design components", Department IRO, Univesité de Montréal, ICSE '99 Los Angeles CA, ACM 1999, pp.226-235.

[KRA96] KRÄMER, Christian, PRECHELT, Lutz, "Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software", Working Conference on Reverse Engineering, IEEE CS Press, Monterey CA, November 8-10, 1996.

[MAR96] MARTIN, Robert, "the Open-Closed Principle", C++ report, enero de 1996.

[MET--] Metamata , "Javacc", http://www.metamata.com/javacc/

[SAN99] SANTAOLAYA SALGADO, Rene, "Sistema de administración de componentes de software basados en frameworks", Proyecto 32042, Cuernavaca, Morelos: Cenidet, 1999.

[SHU96] SHULL, Forrest, MELO, Walcélio, and BASILI, Victor, "An Inductive Method for Discovering Design Patterns from Object-Oriented Software Systems", Technical Report, University of Maryland, computer Science Department, MD, 20742 USA. 1996.

[VIS96] VISWANADHA, Sreenivasa, "C++ grammar", http://falconet.inria.fr/Indexof/java/tools/JavaCC/examples/CandCPLUSPLUS/, Sun Microsystems Inc.