Un marco de trabajo de una fábrica de software para el reuso del diseño arquitectónico y de componentes de software

A software factory framework for the reuse of architectural designs and software components

Héctor A. Durán-Limón[1], María Elena Meda-Campaña[1],
Alma Lilia Sapien-Aguilar[2], Laura Cristina Piñón-Howlet[2]

Recibido: **Febrero 14, 2008** *Aceptado:* **Abril 9, 2008**

Resumen

A diferencia de otras áreas de ingeniería, el nivel de reuso en la ingeniería de software es muy bajo. Los sistemas orientados a objetos fallaron en su promesa para crear un mercado de librería de clases. La tecnología de componentes de software está emergiendo como una aproximación que promete alcanzar el nivel de reuso que los sistemas orientados a objetos no pudieron alcanzar. Las plataformas actuales de componentes EJB, COM, .NET y CCM han sido exitosas para lograr ensamblar componentes de software. Sin embargo, en la práctica actual el ensamble de componentes de software es una tarea compleja. Más aún, un diseño realizado para un modelo de componentes, por ejemplo EJB, no puede ser reusado para los otros estándares. El paradigma llamado *fábricas de software* surge como una alternativa para solucionar esta problemática. Una fábrica de software básicamente involucra el reuso sistemático de recursos como son los requerimientos, diseño arquitectónico, software, y la experiencia de la gente entre otros. Aquí se presenta un marco de trabajo de una fábrica de software, mismo que se enfoca en incrementar el nivel de reuso en dos dimensiones: diseño arquitectónico y componentes de software. Se ilustra la propuesta a través de un caso de estudio para los modelos de componentes de EJB y COM.

Palabras Clave: desarrollo basado en componentes, desarrollo dirigido por modelos, arquitecturas de software, fábrica de software

Abstract

Different from other engineering areas, the level of reuse in software engineering is very low. Object-oriented systems failed in their promise for creating a market place for class libraries. Software components are emerging as an approach that promises to reach the level of reuse that object-oriented systems could not achieve. Current component platforms i.e. EJB, COM, .NET and CCM have succeeded in providing the means for effectively composing software components. However, current practise has proved that assembling software components results in a complex task. Moreover, a component-based design for a particular standard, such as EJB, cannot be reused for other standards. A paradigm called *Software factories* is emerging as an alternative to alleviate such a situation. A software factory basically involves the systematic reuse of assets such as requirements, architecture design, software and people experience among others. We present a software factory framework which focuses on maximising the level of reuse in two dimensions: architecture design and software components. We illustrate our approach through an application case study for the EJB and COM component models.

Keywords: component-based development, model driven development, software architectures, software factory

[1] Profesor del Departamento de Sistemas de Información, CUCEA, Universidad de Guadalajara, Periferico Norte #799 Núcleo Belenes, CP 45100 Zapopan, Jalisco, México. Tel: +52 33 3770 3352 ext: 5119.
[2] Profesora de Tiempo Completo de la Facultad de Contaduría y Administración de la Universidad Autónoma de Chihuahua. Circuito Universitario #1 Nuevo Campus Universitario, C.P. 31125. Apartado postal 1552. Chihuahua, Chihuahua, México.
[4] Dirección electrónica del autor de correspondencia: *lsapien@uach.mx*.

Introducción

Es sorprendente darse cuenta que gran parte de los sistemas de software son aún desarrollados principalmente a mano; a diferencia de otras áreas de ingeniería, el nivel de reuso en la ingeniería de software es muy bajo. Los sistemas orientados a objetos fallaron en su promesa para crear un mercado de librería de clases. La razón de esto se debe a lo siguiente. Primeramente, las clases no suportan *plug-and-play*. Esto es, las clases se distribuyen en una forma en que no están listas para usarse ya que tiene que compilarse, este es el caso de C y C++ por ejemplo. Otros lenguajes como Java ya proporcionan un soporte para desarrollo basado en componentes, pues las clases pueden ser distribuidas como binarios

El problema principal es que se requiere un esfuerzo adicional para compilar y ligar la librería a la aplicación. Más aún, no es común que el proceso de compilación sea transparente, requiriendo trabajo adicional para resolver aspectos de dependencia de plataforma. Segundo, la encapsulación es violada ya que las librerías de clases son distribuidas típicamente como código fuente. De esta forma, los programadores se ven inmediatamente tentados a leer y modificar el código para adecuarlo mejor a sus aplicaciones. Es importante resaltar que los componentes de software están emergiendo como una alternativa que promete alcanzar el nivel de reuso que los sistemas orientados a objetos no pudieron alcanzar. Un componente de software es "una unidad de composición con interfaces especificadas contractualmente y con solamente dependencias de contexto explícitas. Un componente de software puede ser desplegado independientemente y es sujeto a ser ensamblado con terceras partes" (Szyperski, 2002). Un componente de software es esencialmente un binario el cual es capaz de ser desplegado en diferentes ambientes y puede ser ensamblado con componentes desarrollados por terceros. En contraste con las clases, los componentes de software están listos para ser ensamblados y su forma binaria impide que la encapsulación sea violada.

Sin embargo, existen aún varias cuestiones que tienen que ser resueltas antes de que el desarrollo basado en componentes tenga éxito. Aplicaciones grandes pueden involucrar cientos de elementos de software y miles de interacciones entre ellos. Por lo tanto desarrollar este tipo de aplicaciones resulta en una tarea ardua y propensa a errores. La industria actualmente carece de prácticas y herramientas de modelado que permitan hacer frente a esta situación. La arquitectura de software es una disciplina que está emergiendo como una aproximación capaz de alcanzar niveles de abstracción más altos para diseño de software, aminorando así la complejidad del desarrollo de software y haciendo este proceso menos propenso a errores. Dentro de este paradigma los sistemas de software son definidos como un ensamble de componentes y conectores (Bass, *et al*., 2003).

El desarrollo dirigido por modelos (MDD por sus siglas en inglés) también contribuye a disminuir la complejidad de desarrollo. El

principal objetivo de MDD es elevar el nivel de abstracción y de automatización (Selic, 2006). El nivel de abstracción es elevado al nivel de modelos en donde el reuso de modelos es promovido. El nivel de automatización es incrementado usando una técnica llamada "transformación de modelos". Así, modelos de alto nivel son transformados en modelos de más bajo nivel. Las últimas transformaciones dan como resultado código fuente y en el mejor de los casos ejecutables. Mientras que la arquitectura de software se enfoca a metodologías que guían el diseño arquitectónico así como introducir especificaciones arquitectónicas en los modelos, complementariamente a esto, MDD se enfoca en otros aspectos como reuso de modelos y transformación de modelos.

Los estándares de modelos de componentes actuales como EJB (Sun Microsystems, 2007), COM (Microsoft Corporation, 2007a), .NET (Microsoft Corporation, 2007b) y CCM (Object Management Group, 1999) han sido exitosas en permitir el ensamble de componentes. Sin embargo, la práctica actual ha probado que el ensamble de componentes de software es una tarea compleja. Esto es debido a que el desarrollador aún tiene que lidiar con detalles de programación de bajo-nivel para lograr el ensamble de los componentes. Más claramente, el código para ligar los componentes está entremezclado con líneas regulares del programa, haciéndolo difícil de escribir, leer y modificar. Más aún, un diseño realizado para un modelo de componentes, por ejemplo EJB, no puede ser reusado por los otros estándares por ser dependiente de una plataforma de componentes específica. Un paradigma llamado Fábricas de Software está surgiendo actualmente como una alternativa ante tal situación. Una fábrica de software básicamente involucra el reuso sistemático de recursos como requerimientos, diseño arquitectónico, herramientas, software y experiencia entre otros (Greenfield and Short, 2003).

Este artículo presenta un marco de trabajo para una fábrica de software, llamado Tulum, el cual se enfoca en incrementar el nivel de reuso en dos dimensiones: diseño arquitectónico y componentes de software. Un diseño arquitectónico es representado en términos de un ensamble de componentes UML. El marco de trabajo permite tener diferentes vistas arquitectónicas donde se obtiene una clara separación de diferentes aspectos (Bass, *et al*., 2003). La transformación de modelos permite navegar de una vista a otra. Primeramente, una arquitectura independiente de plataforma es definida mediante el ensamble conceptual de componentes expresado en un modelo de componentes independiente llamado XelComp. Este modelo genérico es después refinado con aspectos de distribución mismo que es transformado a un ensamble de componentes asociado a un estándar específico. Enseguida, el diseño arquitectónico obtenido es transformado en un ensamble de componentes de software. Un prototipo del marco de trabajo involucra un editor visual de arquitecturas de software. La herramienta utiliza el marco de trabajo de edición gráfica (GEF por su siglas en inglés) (Eclipse Graphical Editing Framework, 2007) y es implementada en Java como un plugin de Eclipse (Eclipse IDE , 2007). Se ilustra la propuesta a través de un caso de estudio para los estándares de modelos de componentes EJB y COM.

Materiales y métodos

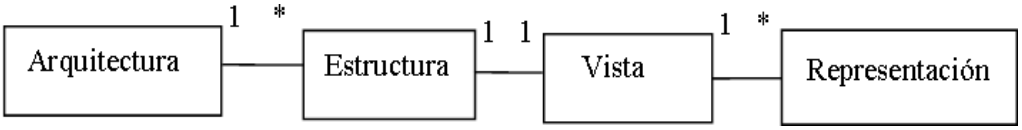
XelComp: Un modelo de componentes independiente de plataforma. El modelo de componentes que hemos creado está ampliamente influenciado por el Modelo Computacional de RM-ODP (Blair and Stefani, 1997). Los *componentes* representan funcionalidad mientras que los *conectores* se usan para capturar ya sea un protocolo de sincronización o de comunicación entre dos o más componentes. Dentro de este modelo, los componentes y conectores tienen múltiples interfaces. No solamente se soportan interfaces *operacionales* sino también interfaces de *flujo* y *señales*. Hay tres diferentes estilos de conectores los cuales están relacionados con los tres estilos de interfaces: *conectores operacionales*, *conectores de flujo*, *conectores de señales*. Los conectores operacionales están encargados de soportar los tradicionales llamados (remotos) de operación. En contraste, los conectores de flujo proporcionan soporte para interacciones de medios continuos. Un conector de flujo representa uno o varios flujos unidireccionales. Los conectores de señales son adecuados para la comunicación de eventos. Los flujos de señales también son unidireccionales. Además, en un ensamble de componentes las conexiones son explícitas, esto es, dos interfaces necesitan conectarse antes de poder comunicarse. Hay dos tipos de interfaces: proveedoras y requeridoras. La primera ofrece servicios mientras que la segunda los requiere. Por lo tanto, las interfaces requeridoras sólo pueden conectarse a interfaces proveedoras. Tanto los componentes como los conectores así como sus conexiones son visualmente representados en UML 2 (Rumbaugh *et al*., 1999). Se hace una distinción entre tres clases de componentes (esto también aplica a conectores): tipos de componentes, componentes de software e instancias de componentes. Un *tipo de componente* es un conjunto de tipos de interfaces y un conjunto de propiedades no funcionales tales como requerimientos de recursos, seguridad, transacciones, etc. Un tipo de interfaz involucra un conjunto de operaciones y de parámetros asociados. Un tipo de componente, el cual es una entidad de diseño, se puede mapear a uno o mas componentes de software dependiendo del modelo de componente usado (por ejemplo EJB, COM, etc.), la plataforma de desplegado y la implementación específica del componente, Por ejemplo, un tipo de componente que tiene asociado un comportamiento de ordenamiento puede ser implementado por diferentes algoritmos, por ejemplo método de la burbuja, método quick sort, etc. Como se ha mencionado anteriormente, los *componentes de software* son esencialmente binarios que pueden ser desplegados en diferentes ambientes y que pueden ser ensamblados con componentes desarrollados por terceros. Los componentes de software son entidades en tiempo de desplegado. Un componente de software puede mapear a una o más *instancias de componentes*. Un componente de software puede involucrar ya sea una sola abstracción instanciable o múltiples abstracciones instanciables. En el primer caso, una instancia de componente involucra una sola instancia mientras que en el segundo caso, una instancia de componente se refiere a todo un conjunto de instancias, por ejemplo un conjunto de objetos que interactúan entre ellos (Szyperski, 2002). Una instancia de componente es por lo tanto una entidad en tiempo de ejecución.

El marco de trabajo para una fábrica de software incrementa el nivel de automatización, como se muestra en la figura 1. Un modelo de un ensamble conceptual de componentes es independiente de plataforma y es construido a partir del repositorio de tipos. El *repositorio de tipos* contiene diferentes tipos de componentes, conectores e interfaces. Antes de que una arquitectura sea modelada, los tipos requeridos son definidos y después incluidos en el repositorio de tipos. Los tipos de componentes y conectores son después obtenidos de este repositorio. Tales componentes son mas tarde ensamblados dentro del contexto de Xelha (Duran-Limon y Blair., 2004), un lenguaje de arquitecturas de software (ADL por sus siglas en inglés). Los ADLs representan notaciones formales para describir arquitecturas de software en términos de componentes y conectores de granularidad gruesa (Medvidovic and Taylor, 2000). De esta forma, el arquitecto de software se concentra en aspectos arquitectónicos de mayor nivel de abstracción por lo cual aspectos de implementación de bajo nivel son ocultados. En esta etapa, la sintaxis del diseño arquitectónico puede ser verificada. Los ensambles de componentes independientes de plataforma son expresados en XelComp. El *ensamble conceptual* de componentes es refinado manualmente hasta obtener un *modelo ingenieril*. El primero representa los aspectos lógicos del sistema mientras que el segundo adicionalmente captura aspectos de distribución (ver mas abajo). Hay dos fases en el ensamble ingenieril de componentes. La primera es un modelo independiente de plataforma en donde no se tiene asociado ninguno de los estándares de modelos de componentes. La segunda fase involucra seleccionar uno de esto estándares, donde el modelo es automáticamente transformado en un modelo específico a una plataforma. Más tarde éste último es automáticamente transformado en un modelo de ensamble de componentes de software donde una plataforma de desplegado, La plataforma de desplegado involucra ambos software y hardware específico. Específica es definida. Este ensamble involucra componentes de software previamente construidos e insertados en el *repositorio de software*. Una subsecuente transformación involucra convertir el modelo del ensamble de componentes de software en *glue code* y en un conjunto de archivos de configuración para el desplegado del sistema. El *glue code* permite que un componente de software invoque los componentes a los cuales se encuentra conectado. El entorno de configuración generado es un plan para la instalación de los componentes de software requeridos, los archivos de configuración y el *glue code*.

Figura 1. Soporte para la Automatización del Ensamble de Componentes

Vistas arquitectónicas.- Existen múltiples definiciones de arquitectura de software (Bass *et al*., 2003; Medvidovic and Taylor, 2000; Shaw and Garlan, 1996; The IEEE Standards Board, 2000) sin embargo, promovemos una de las definiciones que consideramos mas completa (Bass, *et al*., 2003): "La arquitectura de software de un programa o sistema de cómputo es la estructura o estructuras del sistema, mismas que incluyen elementos de software, las propiedades externamente visibles de esos elementos, y las relaciones entre ellos". Una estructura es un conjunto de elementos con un patrón organizacional. Similarmente a la arquitectura de un edificio que involucra múltiples estructuras como son la instalación eléctrica y el sistema de drenaje; una arquitectura de software también involucra múltiples estructuras en donde cada una de ellas presenta una vista distinta del sistema. La separación de diferentes aspectos se logra a través de esta aproximación de múltiples vistas. De esta forma se logra disminuir la complejidad de un sistema ya que diferentes aspectos del sistema tales como concurrencia y distribución pueden ser abordados de manera independiente. Se usa el término *vista* para denotar la representación de una estructura. La diferencia entre una estructura y una vista es que la primera es un conjunto de elementos (y sus relaciones) en sí mismos, mientras que la segunda es una representación de tales elementos así como de sus interrelaciones (Bass, *et al*., 2003). Adicionalmente, se considera que una vista puede tener múltiples representaciones de una misma estructura. Por ejemplo, una estructura de una base de datos puede ser representada tanto por un diagrama entidad-relación como por un *script* en SQL. Ambas representaciones nos muestran el mismo conjunto de elementos e interrelaciones, por lo tanto, estas representaciones pertenecen a una misma vista. Las relaciones entre arquitectura, estructura, vista y representaciones se resumen en la figura 2.

Figura 2. Relaciones entre Arquitectura, Estructura y Vista



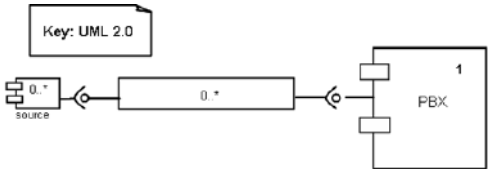
Existen múltiples vistas, por ejemplo, descomposición, concurrencia, desplegado, etc. De acuerdo con Bass, las vistas pueden ser organizadas en tres diferentes categorías: *módulo*, *componente-y-conector (C&C)*, y *asignación* (Bass, *et al*., 2003). Las vistas de módulo representan estructuras cuyos elementos involucran unidades de funcionalidad. Ejemplos de esta categoría de vista son descomposición de módulos y la vista de herencia de clases. Las estructuras que contienen elementos de tiempo de ejecución tales como procesos e hilos son representadas por la vista C&C. Las vistas de concurrencia y de procesos son ejemplos de esta categoría. Finalmente, las vistas de asignación representan estructuras que involucran relaciones de asignación. Por

ejemplo, los elementos de software son asignados a elementos de hardware en una vista de desplegado. Sin embargo, nosotros manejamos una aproximación un tanto distinta a la de Bass en la que se consideran tres clases de la vista C&C: a) C&C conceptual, b) C&C ingenieril, y c) C&C de software. La vista *C&C conceptual* es el particionamiento lógico de un sistema en entidades que representan instancias de componentes y conectores. Los aspectos de distribución, como nodos y procesos, no son tratados en esta vista. Aquí se representan modelos independientes de plataforma. La vista *C&C ingenieril* introduce aspectos de distribución. Esta vista es dividida en dos vistas más. La primera es independiente de cualquier estándar de modelo de componentes mientras que la segunda está ligada a un estándar en particular. Finalmente, la vista *C&C de software* muestra las relaciones entre los elementos de software y hardware. Esta vista es dependiente de plataforma.

Resultados y discusión

Caso de Estudio. Se utilizó un caso de estudio para ilustrar y evaluar los principios de diseño del marco de trabajo. El caso de estudio consiste en el desarrollo de un sistema middleware (Middleware es la capa que se encuentra entre la aplicación y el sistema operativo, y se encarga de tratar aspectos de heterogeneidad y distribución) de una aplicación de voz sobre IP (VoIP por sus siglas en inglés), tal como se muestra en la figura 3. El sistema permite establecer conexiones entre una PC y la red telefónica. Del lado derecho, el proveedor de servicio recibe tráfico de VoIP el cual es transferido a la red pública telefónica por un PBX (Un PBX es un conmutador telefónico). El sistema permite que se establezcan múltiples conexiones de múltiples fuentes.

Figura 3. Caso de Estudio



Componentes, conectores e interfaces. Se usó UML 2 para la representación gráfica de componentes, conectores e interfaces. Los componentes son representados como componentes en UML mientras que para los conectores se utilizaron clases en UML estereotipadas como conectores y customizadas como rectángulos. Las interfaces en UML involucran el uso de círculos vacíos llamados facetas para denotar interfaces proveedoras mientras que las interfaces requeridoras son representadas por semicírculos llamados receptáculos.

Para este caso de estudio se empezó definiendo los tipos involucrados en el sistema de middleware. Los tipos de interfaz son definidos primero y almacenados en el repositorio de tipos. Un tipo de interfaz se define en términos de un conjunto de operaciones. Cada operación incluye sus parámetros asociados. Por ejemplo, el tipo de interfaz IN del conector AudioBinding define la operación put la cual incluye un parámetro de entrada. Para definir los tipos de los parámetros usamos los tipos definidos en el lenguaje de definición de interfaces (IDL por sus siglas en inglés) de CORBA (Object Management Group, 2001).

Después de definir los tipos de interfaces se procedió a especificar los tipos de componentes y conectores. Debido a que los tipos de componentes y conectores son básicamente un conjunto de tipos de interfaces, ambos se definen a partir de los tipos de interfaces que incluirán

y que son obtenidas del repositorio de tipos. Una restricción impuesta por la gramática de Xelha es que los conectores deben tener cuando menos una interfaz requeridora y otra proveedora. Para el caso de estudio se definen los siguientes tipos de componentes: compressor, descompressor, srcStub y sinkStub. La tabla 1 define el conjunto de interfaces proveedoras y requeridas que pertenecen a cada uno de los componentes y conectores. Por ejemplo, el tipo de componente compressor tiene las interfaces proveedoras codeIN y codeCTRL así como la interfaz requeridora codeOUT. La primera y la última son interfaces estilo flujo mientras que la otra es una interfaz operacional.

Tabla 1. Tipos de Interfaces

		Tipo de interfaz proveedora	Estilo de interfaz	Tipo de interfaz requeridora	Estilo de interfaz
Tipo de componente	compressor	codeIN, codeCTRL	flujo operacional	codeOUT	flujo
	decompressor	decodeIN, decodeCTRL	flujo operacional	decodeOUT	flujo
	srcStub	srcStubIN	flujo	srcStubOUT	flujo
	sinkStub	sinkStubIN	flujo	sinkStubOUT	flujo
Tipo de conector	streamConnector	streamIN	flujo	streamOUT	flujo

Como se ha mencionado anteriormente, un repositorio de software contiene componentes de software asociados con tipos de componentes y conectores. Por ejemplo, el componente de software del compresor está asociado con el tipo de componente compressor. Este componente de software incluye un archivo de especificaciones. Éstas contienen un descriptor de desplegado con información acerca de la plataforma donde el componente puede ser desplegado así como los requerimientos de recursos del componente. Debe notarse que hay ciertas demandas de recursos que son independientes del hardware como en este caso lo es el ancho de banda de red. Algunos otros requerimientos de recursos dependen de la plataforma de hardware como es el caso de los ciclos de CPU demandados.

El descriptor de desplegado también incluye una descripción de las interfaces del componente. Esta descripción se realiza en el lenguaje de descripción de servicios Web (WSDL por sus siglas en inglés) (World Wide Web Consortium, 2007) mientras que las demandas de recursos son especificados en el lenguaje de descripción de trabajos sometidos (JSDL por sus siglas en inglés) (Global Grid Forum, 2005). Toda esta información es extraída del descriptor y almacenada en el repositorio de software. El usuario es capaz de revisar y cambiar la información o completar datos faltantes si es requerido, tal acción dará como resultado que el archivo de descripción del desplegado sea modificado. Adicionalmente, el tipo de componente es agregado al repositorio de tipos si este no fue incluido previamente.

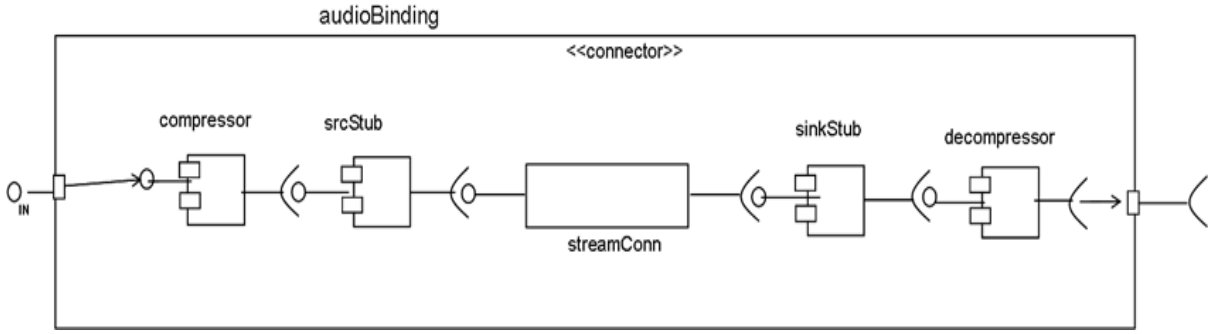
Ensamble conceptual de componentes. Los componentes pueden ser conectados entre ellos ya sea directamente o a través de un conector. Los conectores son comúnmente usados para ligar componentes cuando un protocolo de sincronización o comunicación está involucrado, como ya se mencionó antes.
Las instancias de los tipos de componentes son ensamblados en la vista C&C conceptual.

En el caso de estudio, del lado del cliente el componente `source` envía paquetes de audio mientras que del lado del servidor el componente `sink` consume los datos, como se muestra en la figura 4. El conector `AudioBinding`, el cual es un conector de flujo, permite la transmisión de datos entre el cliente y el servidor.

Figura 4. Vista C&C conceptual de la arquitectura del sistema

Tanto componentes como conectores pueden incluir otros elementos compuestos. Así, un sistema distribuido es representado como una composición jerárquica en términos de componentes y conectores compuestos. Tal composición jerárquica es similar a la ofrecida por Darwin (Magee *et al*., 1995) para la composición de componentes compuestos. Así, la granularidad de un tipo de componente puede ir desde un tipo de componente primitivo, es decir no compuesto, a un tipo que involucre un sistema complejo con múltiples niveles de composición. El componente de nivel más alto, que está en el nivel superior de una jerarquía de composición, encapsula toda la arquitectura del sistema. Para el caso de estudio se definió el ensamble de componentes internos del conector compuesto AudioBinding. Así, el compressor, decompressor, streamConn, srcStub y sinkStub son seleccionados del repositorio de tipos y ensamblados como se muestra en la figura 5.

Figura 5. Vista C&C Conceptual del Conector AudioBinding. Modelo Independiente de Plataforma



Verificación de sintaxis. En esta etapa se verifica la sintaxis del diseño arquitectónico. Para esto se realiza un chequeo de tipos de la interfaces conectadas. Primeramente, las interfaces conectadas deben coincidir con el tipo de estilo de interfaz usado. Segundo, se utilizó un subtipeo estructural mediante el cual cuando menos todas las operaciones contenidas en la interfaz requeridora deben estar contenidas en la interfaz proveedora. Por ejemplo, las interfaces codeOUT y srcStubIN pueden ser conectadas ya que ambas definen la operación put con los mismos parámetros, tal como se muestra en la tabla 2. Deberá notarse que para una operación dada perteneciente a una interfaz, un parámetro de entrada debe coincidir con el parámetro de salida del mismo tipo. Adicionalmente, el orden de los parámetros debe ser equivalente.

Tabla 2. Ejemplo de Tipos de Interfaces

Tipo de interfaz	Estilo de interfaz	Tipo de componente	Proveedora/ Requeridora	Operaciones	Parámetros
codeOUT	flujo	compressor	Requeridora	put	short [out]
codeIN	flujo	srcStub	Proveedora	put	short [in]

Adicionalmente, una jerarquía de tipos basados en nombres (*named types*) se genera a partir de un análisis de los tipos estructurales contenidos en el tipo de repositorios. La jerarquía de tipos basados en nombres se actualiza cada vez que un nuevo tipo de interfaz es agregado al repositorio. Como resultado, un chequeo de tipos basado en nombres es llevado acabo el cual es más eficiente que el chequeo de tipos estructurales (Szyperski, 2002).

El diseño del ensamble también se verifica con respecto a la gramática de Xelha. Se usa una aproximación basada en XML en la cual la gramática de Xelha está definida en esquemas de XML (World Wide Web Consortium, 2001). La gramática de Xelha está segmentada en diferentes esquemas: cuerpo, componente, conector, e interfaz. El esquema del cuerpo define la estructura general de la especificación de una arquitectura. Tal especificación se realiza en términos de componentes, conectores, interfaces y configuraciones. Los primeros tres tienen un esquema asociado. Las configuraciones, que son especificadas dentro del cuerpo, definen como están interconectados los componentes y conectores. Por ejemplo, parte del esquema del cuerpo se muestra en la figura 6. Este esquema establece que la especificación de un ensamble de componentes debe contener las secciones de componentes, conectores, interfaces y conexiones (líneas 18-21). Esta sección define qué instancias de tipos de componentes, conectores e interfaces están involucrados y como es que están interconectados. El esquema también establece que todos los tipos deben ser definidos como parte de la especificación (líneas 22-24).

Figura 6. Muestra del código del esquema del cuerpo

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!-- edited with XMLSpy v2006 sp2 U (http://www.altova.com) by victorio robles (cinvestav) -->
3 <!-- edited with XMLSPY v2004 rel. 3 U (http://www.xmlspy.com) by trislo (*) -->
4 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified" attributeFormDefault="unqualified">
5     <xs:element name="xelha">
6         <xs:annotation>
7             <xs:documentation>Comment describing your root element</xs:documentation>
8         </xs:annotation>
9         <xs:complexType>
10            <xs:sequence>
11                <xs:element name="name">
12                    <xs:simpleType>
13                        <xs:restriction base="xs:string">
14                            <xs:pattern value="Def component [a-zA-Z][a-zA-Z0-9]*(\s)*(\s)*:(\s)*"/>
15                        </xs:restriction>
16                    </xs:simpleType>
17                </xs:element>
18                <xs:element ref="components"/>
19                <xs:element ref="connectors" minOccurs="0" maxOccurs="unbounded"/>
20                <xs:element ref="interfaces" minOccurs="0" maxOccurs="unbounded"/>
21                <xs:element ref="graph"/>
22                <xs:element ref="defComponent"/>
23                <xs:element ref="defConnector" minOccurs="0"/>
24                <xs:element ref="defInterface"/>
25            </xs:sequence>
26        </xs:complexType>
27 </xs:element>
```

La ventaja de usar esquemas de XML es que la gramática del ADL puede ser fácilmente cambiada y extendida sin necesidad de realizar cambios mayores a la herramienta. Por ejemplo, si se requiere cambiar la gramática del componente, sólo se requiere reemplazar el esquema del componente con la nueva gramática. El prototipo actual del marco de trabajo soporta la validación de un ensamble de componentes mediante el uso del parser de XML llamado Xerces-J (XML parser, 2007). De esta forma adicional se requiere para verificar aspectos semánticos. La representación gráfica de un ensamble de instancias de tipos de componentes se transforma automáticamente a una representación textual expresada en un documento XML (detalles acerca de cómo se hace la transformación de modelos se presenta en la siguiente sección). Por ejemplo, la figura 7 ilustra una muestra del código generado de la vista C&C conceptual localizada en la figura 5. El código define los elementos involucrados, tales como los componentes source y sink (líneas 6-10), el conector AudioBinding (líneas 11-14) y las conexiones entre los elementos (líneas 15-29).

Transformación de modelos. Antes de que se lleve acabo cualquier transformación, el modelo de la figura 5 se refina introduciendo aspectos de distribución y que en este caso involucra identificar en qué nodos se ubica cada componente, como se muestra en la figura 8.

A continuación se tiene la transformación de modelos, misma que es implementada en Java. El editor de Eclipse representa los diagramas como un contenedor de figuras mediante el cual sus elementos pueden ser examinados y cambiados. Por ejemplo, las propiedades de un elemento como son tamaño y posición pueden ser cambiadas. También es posible borrar o adicionar nuevos elementos al contenedor. La transformación de un modelo C&C ingenieril

Figura 7. Muestra del código XML generado de la vista C&C conceptual

```
1 <?xml version='1.0' encoding='UTF-8'?>
2 <xelha xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
3 xsi:noNamespaceSchemaLocation='C:/users/hector/Software/Eclipse3.1.2/runtime-workspace/
4 MyProject/xelhaPluginScheme2.0.xsd'>
5               <name>Def component VoIPsystem :</name>
6               <components>
7                       <inicio>components :</inicio>
8                       <component>Component2: Sink, Capsule2</component>
9                       <component>Component1: Source, Capsule1</component>
10              </components>
11              <connectors>
12                      <inicio>connectors :</inicio>
13                      <connector>Connector1: AudioBinding(Capsule1, Capsule2)</connector>
14              </connectors>
15              <graph>
16                      <inicio>composition graph:</inicio>
17                      <interfaces>
18                              <inicio>interfaces:</inicio>
19                              <interface>Component1OUTstream: (Component1, OUTstream)</interface>
20                              <interface>Connector1OUTstream: (Connector1, OUTstream)</interface>
21                              <interface>Connector1INstream: (Connector1, INstream)</interface>
22                              <interface>Component2INstream: (Component2, INstream)</interface>
23                      </interfaces>
24                      <edges>
25                              <inicio>edges:</inicio>
26                              <edge>(Connector1INstream, Component1OUTstream)</edge>
27                              <edge>(Component2INstream, Connector1OUTstream)</edge>
28                      </edges>
29              </graph>
```

a un documento XML involucra examinar el contenedor de figuras para determinar qué componentes, conectores e interfaces están incluidos en el ensamble. Los vértices de la gráfica de composición se obtienen al determinar las facetas que tienen su área sobrepuesta con el área de un receptáculo. Esto es, un vértice está conformado por la tupla (faceta, receptáculo) cuyas áreas se sobreponen.

Figura 8. Vista C&C ingenieril del conector AudioBinding. Modelo independiente de plataforma

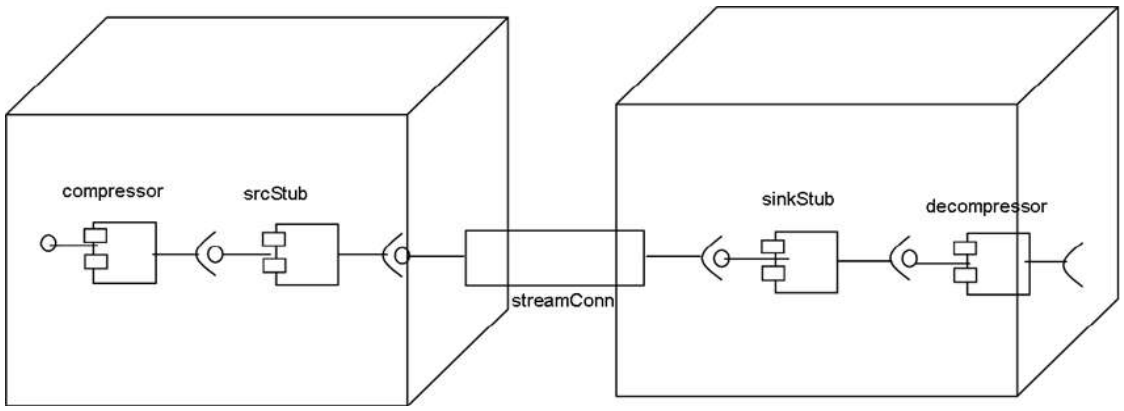
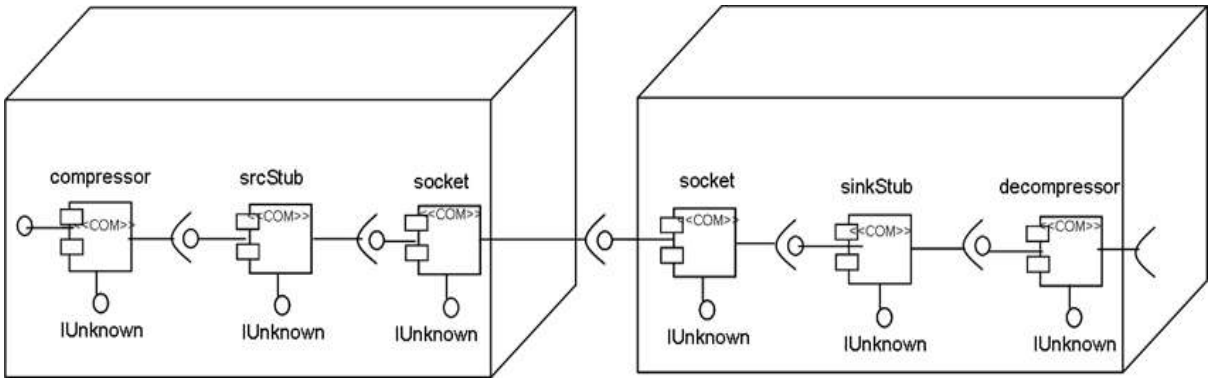
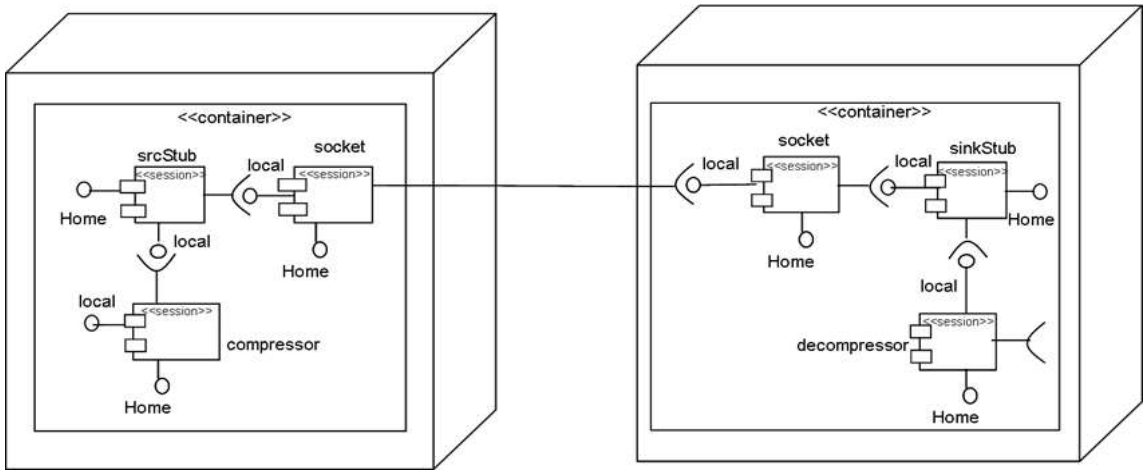


Figura 9. Vista C&C ingenieril de AudioBinding. Modelo COM



La figura 8 muestra un ensamble de componentes independiente de plataforma, el cual es transformado a un estándar específico de modelo de componentes, como se muestra en las figuras 9 y 10. En el primer caso, los componentes son customizados como componentes COM. Todos los componentes incluyen la interfaz IUnknown la cual controla el ciclo de vida del componente y también permite hacer *queries* a sus interfaces. Más aún, los conectores de flujo son transformados en conexiones UDP representadas por dos *sockets*. En contraste, el modelo EJB involucra las siguientes transformaciones. Ambos, el cliente y el servidor son situados en un contenedor distinto. Todos los componentes incluyen la interfaz Home a cargo de controlar el ciclo de vida del componente. Las interfaces locales facilitan el acceso local a las operaciones soportadas por el componente. Debido a que las interfaces remotas trabajan mediante el uso de RMI, el *socket* del lado derecho es accesado a través de una interfaz local.

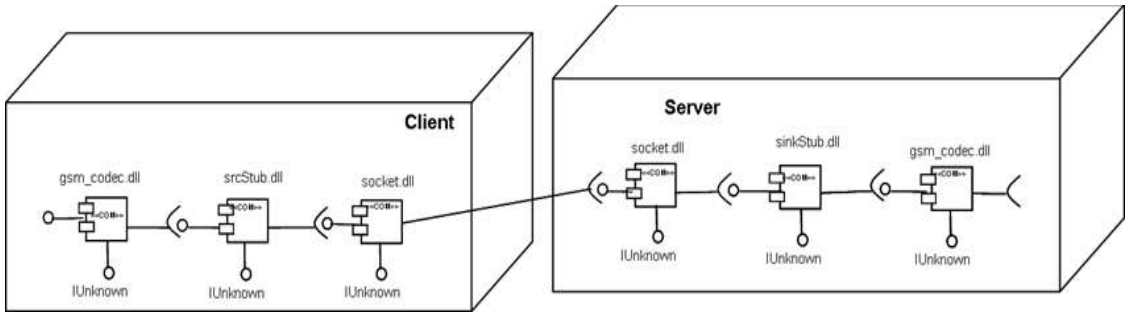
Figura 10. Vista C&C ingenieril de AudioBinding. Modelo EJB



Después, los modelos son transformados a la vista de C&C de software la cual representa el modelo del ensamble de componentes de software, como se muestra en la figura 11. Para esta transformación, cada tipo de componente es mapeado a un componente de software específico. Por ejemplo, el tipo de componente compressor mostrado en la figura 9 es transformado al elemento de software gsm_codec.dll el cual es un *codec* GSM (Spanias, 1994). La información para tal mapeo es obtenida del repositorio de software. Nótese también que los elementos de software son asignados a elementos de hardware. De tal forma que esta vista muestra como es que los archivos DLLs son asignados a los nodos del cliente y servidor.

Para lograr la reusabilidad de componentes, éstos son distribuidos como piezas de código independientes donde los componentes no saben acerca de las conexiones que establecerán con otros componentes. Por lo tanto, se requiere generar *glue code* mediante el cual se codifican las conexiones entre los componentes. Finalmente, también se generan archivos de configuración para el desplegado de los componentes.

Figura 11. Vista C&C de Software de AudioBinding. Modelo COM



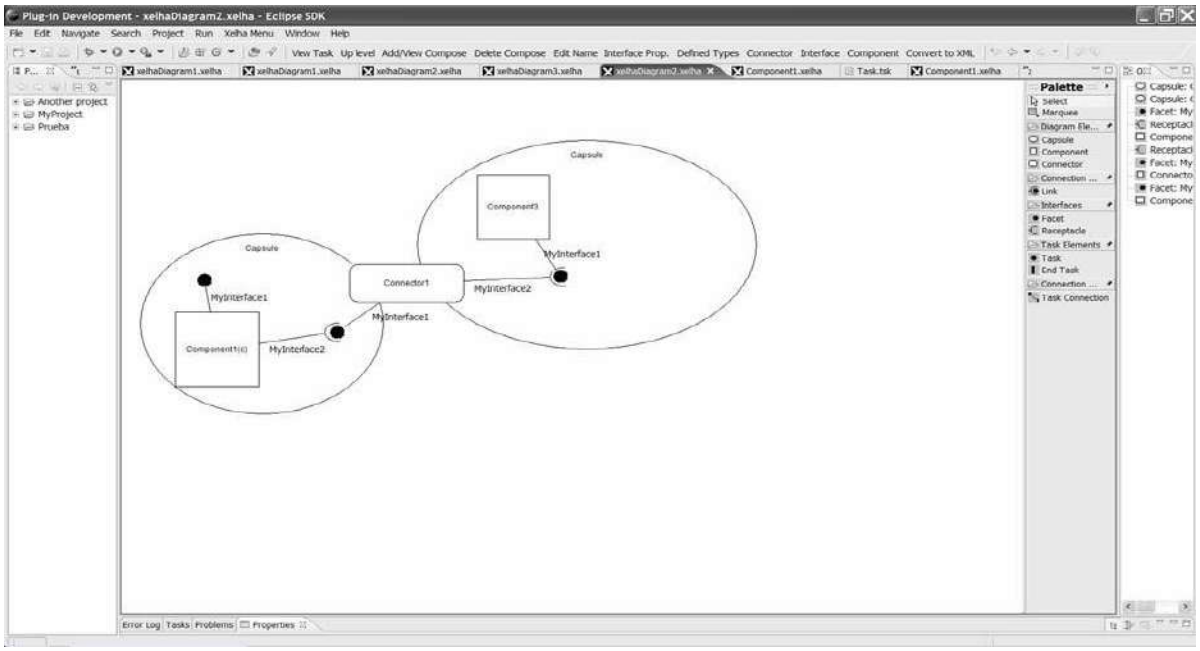
En general, creemos que el caso de estudio ha mostrado que un ensamble en el modelo independiente de componentes puede ser reusado para diferentes estándares de modelos de componentes, mediante el cual se favorece el reuso de diseños arquitectónicos. Un modelo de componentes básicamente involucra un conjunto finito de elementos (por ejemplo EJB beans, interfaces locales, etc) e interacciones (por ejemplo orientadas a conexión, sin conexión, etc.). Por lo tanto, se requiere un conjunto finito de reglas de transformación para transformar un diseño expresado en un modelo de componentes independiente a un ensamble ligado a un modelo de componentes estándar. Para el caso de estudio presentado, el modelo de componentes independiente ha mostrado ser lo suficientemente genérico para abarcar los estándares de EJB y COM. Aplicaciones enfocadas a un dominio más específico, como lo es el de sistemas de tiempo real, requieren de una especialización del modelo de componentes independiente. Dicha especialización puede implementarse en el modelo de componentes independiente mediante el uso de perfiles UML.Adicionalmente, los tres estilos del

modelo de componentes ayudan a reducir las ambigüedades en las transformaciones. Por ejemplo, los *sockets* UDP son adecuados para los conectores de flujo a diferencia de conexiones RMI. El mapeo uno a uno entre tipos y componentes de software permite realizar las transformaciones de la vista C&C ingenieril a la vista C&C de software. Para cada nivel de transformación, los modelos pueden ser refinados para lograr una mayor precisión. En el caso del ensamble de componentes de software, el *glue code* generado permite que los componentes establezcan las conexiones especificadas entre ellos.

La verificación de sintaxis de un ensamble de componentes se logra mediante el uso de los esquemas XML y un mecanismo de chequeo de tipos. Los aspectos semánticos no pueden ser verificados y representan una línea de investigación futura para este trabajo. Mas aún, las tres diferentes vistas de un ensamble de componentes –C&C conceptual, ingenieril y software- ayudan a distinguir entre aspectos lógicos, distribución y desplegado.

El reuso de software también se favorece al automatizar el ensamble de componentes de software. Lo que básicamente se asume para que esta propuesta sea exitosa es contar con un repositorio de software, que albergue a un conjunto amplio de componentes y que éstos permitan ser ensamblados con terceras partes. Para lograr esto, es imprescindible que los componentes sean desarrollados de manera genérica de tal manera que no queden fijas conexiones con componentes predeterminados, sino que estos puedan ser conectados a cualquier componente con una interfaz compatible. Los componentes de software pueden ser hechos en casa o comprados a terceros. Aunque el mercado de componentes de software continúa madurando, la propuesta aún puede ser útil para compañías que desarrollan sus propios componentes.

Figura 12. Herramienta de la Fábrica de Software



Conclusiones

Se presenta una propuesta original para incrementar el nivel de reuso en dos dimensiones: diseño arquitectónico y componentes de software. Tulum es un marco de trabajo para una fábrica de software que permite definir modelos independientes de un estándar de modelo de componentes. El proceso de ensamble es simplificado por la capacidad de arrastrar-y-soltar componentes del repositorio de tipos.

Tales modelos pueden ser transformados automáticamente a modelos involucrando un estándar de modelo de componentes. Estos modelos alcanzan altos niveles de abstracción ya que se usa una notación de tipo ADL para describir visualmente los ensambles. Un ensamble en un modelo de componentes estándar se transforma en un ensamble de componentes de software. Se usan múltiples vistas arquitectónicas en donde la vista C&C conceptual es refinada a una vista C&C ingenieril misma que se transforma en la vista C&C de software. Esta transformación se obtiene en base a la información extraída del repositorio de software y del hecho que hay un mapeo entre tipos de componentes y componentes de software. Más aún, se genera *glue code* mismo que permite comunicar a los que de otra forma son componentes aislados. La verificación de los aspectos estructurales de los ensambles se realiza mediante el uso de esquemas de XML y un mecanismo de chequeo de tipos. El desarrollo de un caso de estudio ha ilustrado nuestra propuesta. Creemos que el marco de trabajo presentado puede ser usado para acelerar el proceso de desarrollo de software en donde el desarrollador se concentra en aspectos con un mayor nivel de abstracción.

El prototipo actual, mostrado en la figura 12, soporta el ensamble conceptual de componentes así como la validación de su sintaxis. Se está trabajando en extender la herramienta para facilitar la transformación de ensambles ingenieriles de componentes a ensambles de componentes de software. Líneas futuras de investigación incluyen proporcionar soporte para verificar la semántica de los ensambles. Lenguajes como OCL proporcionan un buen grado de flexibilidad para definir las reglas de verificación requeridas (UML 2.0 OCL Specification. OMG. 2003). Finalmente, también se esta trabajando en el tema de desarrollo basado en modelos multinivel en donde los modelos son definidos en diferentes niveles de abstracción (Duran-Limon, 2006). Un nivel de modelado se construye a partir del ensamble de elementos de software definidos en el nivel inferior inmediato. Este paradigma promete disminuir de manera efectiva la complejidad de desarrollo y facilitar el reuso a gran escala.

Literatura citada

Bass, L., P. Clements, et al. (2003). Software Architecture in Practice, Addison Wesley.

Blair, G. S. and J.-B. Stefani. “Open Distributed Processing and Multimedia.”, Addison-Wesley. 1997.

Duran-Limon, Hector. Multilevel Modeling Software Development. In Proceedings of the Tercer Congreso de Electrónica, Robótica y Mecánica Automotriz (CERMA), Morelos, México, September, 2006.

Eclipse Graphical Editing Framework (GEF). http://www.eclipse.org/gef/. 2007.

Eclipse IDE. http://www.eclipse.org/. 2007.

Global Grid Forum. Job Submission Description Language (JSDL), Specification, Version 1.0, 2005. http://forge.gridforum.org/projects/jsdl-wg

Hector A. Duran-Limon, Gordon S. Blair, “QoS Management Specification Support for Multimedia Middleware.” The Journal of Systems and Software, Elsevier, 72(1), pp 1-23: 2004.

Jack Greenfield and Keith Short. Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools. OOPSLA ’03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems,

languages, and applications. 2003.
Magee, J., N. Dulay, S. Eisenbach and J. Kramer. "Specifying Distributed Software Architectures." Fifth European Software Eng. Conf. (ESEC '95). September 1995.
Medvidovic, N. and R. N. Taylor, 2000. A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Trans. Software Eng. 26(1):
Microsoft. ".NET Development." Microsoft Corporation. 2007b. http://msdn.microsoft.com/net/
Microsoft. "COM: Delivering on the Promises of Component Technology." Microsoft Corporation. 2007a. http://www.microsoft.com/com/default.asp
Object Management Group. 1999. "CORBA 3.0 New Components chapters." CCM FTF Draft ptc/99-10-04.
Object Management Group. 2001."The Common Object Request Broker: Architecture and Specification - Revision 2.4.2 (CORBA v2.4.2)."
Rumbaugh, J., Jacobson, I., Booch, G. 1999.The Unified Modelling Language Reference Manual. Addison–Wesley.
Selic, B. 2006.Model-Driven Development: Its Essence and Opportunities Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC).
Shaw, M. and D. Garlan, 1996. Software Architecture: Perspectives on an Emerging Discipline., Prentice Hall.
Spanias, A. S. "Speech coding: a tutorial review." Proc. of the IEEE 82(10): 1441-1582, October 1994.
Sun. 2007. "Enterprise JavaBeans Technology." Sun Microsystems.. http://java.sun.com/products/ejb/
Szyperski, C. 2002. Component Software: Beyond Object-Oriented Programming. Second Edition. Harlow, England, Adison-Wesley.
The IEEE Standards Board. Recommended Practice for Architectural Description of Software –Intensive Systems, IEEE-std-1471 – 2000, September, 2000.
UML 2.0 OCL Specification. OMG. 2003. http://www.omg.org
World Wide Web Consortium, 2001. XML Schema: Formal Description. W3C Working Draft. http://www.w3.org/TR/xmlschema-formal/
World Wide Web Consortium. Web Services Description Language (WSDL). http://www.w3.org/TR/wsdl. 2007.
XML parser: http://xml.apache.org/xerces-j/. 2007

Este artículo es citado así:

Durán-Limón H. A. , M.E. Meda-Campaña, A.L. Sapien-Aguilar, L.C. Piñon-Howlet. 2008. Un marco de trabajo de una fábrica de software para el reuso del diseño arquitectónico y de componentes de software. *Tecnociencia Chihuahua* 2(1): 15-31.

Resúmenes curriculares de autor y coautores

Héctor Durán-Limón. Es actualmente Profesor Investigador del Departamento de Sistemas de Información del CUCEA, Universidad de Guadalajara. Terminó un PhD en la Universidad de Lancaster, Inglaterra en el año 2002. Seguido de esto, el Dr. Durán continuó con una estancia pos-doctoral en la misma universidad hasta diciembre del 2003. Sus áreas de interés incluyen arquitecturas de software, desarrollo basado en componentes, middlewareadaptable y cómputo en Grids. También son de su interés el cómputo móvil y los sistemas de tiempo real. El Dr. Durán puede ser contactado en el Departamento de Sistemas de Información, CUCEA, Universidad de Guadalajara,Guadalajara, México; hduran@cucea.udg.mx.

María Elena Meda. Es Profesor Investigador adscrita al Departamento de Sistemas de Información de la Universidad de Guadalajara, además coordina la Maestría en Tecnologías de Información en la misma Institución. La Dra. Meda obtuvo el grado de Doctor en Ciencias de la Ingeniería Eléctrica en el CINVESTAV (Centro de Investigación y de Estudios Avanzados) del IPN, en el año 2002. Sus áreas de interés se enfocan en el modelado, análisis y optimización de sistemas de eventos discretos utilizando técnicas algebraicas. Específicamente, realiza investigación en la construcción asintótica de modelos para sistemas complejos y en el diseño de sistemas tolerantes a fallas. La Dra. Meda, puede ser contactada en el Departamento de Sistemas de Información del Centro Universitario de Ciencias Económico-Administrativas CUCEA de la Universidad de Guadalajara, teléfono (33)3770-3352, email emeda@cucea.udg.mx

Alma Lilia Sapién Aguilar. Es actualmente Profesor de Tiempo Completo de la Facultad de Contaduría y Administración de la Universidad Autónoma de Chihuahua. Cursó la Licenciatura en Sistemas de Información (1988-1993), así como la Maestría en Administración (1994-1997) en la Facultad de Contaduría y Administración de la Universidad Autónoma de Chihuahua. Actualmente cursa el Doctorado en Administración en la UACh (2004-a la fecha).

Laura Cristina Piñón Howlet. Es actualmente Profesor de Tiempo Completo de la Facultad de Contaduría y Administración de la Universidad Autónoma de Chihuahua. Cursó la Licenciatura en Sistemas de Información (1987-1993), así como la Maestría en Administración (1997-2000) en la Facultad de Contaduría y Administración de la Universidad Autónoma de Chihuahua. Actualmente cursa el Doctorado en Administración en la UACh (2004-a la fecha).

DOI: https://doi.org/10.54167/tecnociencia.v2i1.66