Desarrollo de Software Basado en Componentes

Jonás A. Montilva C.[1], Nelson Arapé[2] y Juan Andrés Colmenares[2]

[1]Universidad de Los Andes
Facultad de Ingeniería
Escuela de Ingeniería de Sistemas
Departamento de Computación
Mérida – Venezuela
+58-274-2403811

[2]Universidad del Zulia
Facultad de Ingeniería
Instituto de Cálculo Aplicado
Maracaibo - Venezuela

jonas@ing.ula.ve, narape@ica.luz.ve, juancol@ica.luz.ve

Resumen-- La Orientación a Objetos introdujo, durante la década pasada, un cambio radical en el proceso de desarrollo de software. De un proceso caracterizado por su énfasis en la descripción algorítmica de la solución del problema, se pasó a un proceso orientado a la representación y manipulación de los objetos que caracterizan el problema. Este paradigma abrió, también, nuevas posibilidades para desarrollar software basado en la noción de *reutilización de componentes.* La Orientación a Objetos creó un rumbo diferente en el proceso de reutilización a través de la producción de componentes genéricos, fáciles de integrar, distribuidos e independientes de las plataformas de desarrollo. En este artículo, de carácter tutorial, se discuten los conceptos fundamentales de la reutilización de software, así como los modelos de procesos y los aspectos metodológicos que caracterizan esta nueva manera de producir software, denominada Desarrollo de Software basado en Componentes.

Palabras clave—desarrollo de software, reutilización de software, componentes de software.

I. INTRODUCCIÓN

La *reutilización de componentes de software* es un proceso inspirado en la manera en que se producen y ensamblan componentes en la ingeniería de sistemas físicos. La aplicación de este concepto al desarrollo de software no es nueva. Las librerías de subrutinas especializadas, comúnmente utilizadas desde la década de los setenta, representan uno de los primeros intentos por reutilizar software.

Existen variadas definiciones del término *Reutilización de Software*. Algunas de estas definiciones son las siguientes:

- Según Somerville [1], la reutilización es un enfoque de desarrollo [de software] que trata de maximizar el uso recurrente de componentes de software existentes.
- Según Sametinger [2], la reutilización de software es el proceso de crear sistemas de software a partir de software existente, en lugar de desarrollarlo desde el comienzo.
- Según Sodhi et al. [3], la reutilización de software es el proceso de implementar o actualizar sistemas de software usando activos de software existentes.

Estas tres definiciones consideran la reutilización como un enfoque o proceso de desarrollo de software. Su principal diferencia radica en el objeto de reutilización (componente, software y activo). Tal como lo plantean Sodhi et al. [3], la reutilización de software va más allá de la reutilización de piezas de software. Ella involucra el uso de otros elementos de software, tales como algoritmos, diseños, arquitecturas de software, documentación y especificaciones de requerimientos.

Con base en el análisis de estas definiciones podemos establecer nuestra propia definición en los siguientes términos:

La reutilización de software es un proceso de la Ingeniería de Software que conlleva al uso recurrente de activos de software en la especificación, análisis, diseño, implementación y pruebas de una aplicación o sistema de software.

Varios estudios han demostrado la efectividad de la reutilización del software. Sametinger [2], en particular, describe algunos de estos indicadores:

- Entre el 40 y 60% del código fuente de una aplicación es reutilizable en otra similar.

- Aproximadamente el 60% del diseño y del código de aplicaciones administrativas es reutilizable.
- Aproximadamente el 75% de las funciones son comunes a más de un programa.
- Sólo el 15% del código encontrado en muchos sistemas es único y novedoso a una aplicación específica. El rango general de uso recurrente potencial está entre el 15% y el 85%.

A partir de estos indicadores es fácil deducir el impacto y la importancia que tiene la reutilización de componentes en el proceso de desarrollo de software; particularmente, en tres de las variables más importantes de la Ingeniería de Software: el costo, tiempo y esfuerzo requerido para desarrollar un producto de software. La aplicación apropiada de la reutilización en un proyecto de software conduce, indiscutiblemente, a una reducción significativa de los valores de estas tres variables. Otros beneficios importantes son el incremento de la calidad del software producido, el aumento de la productividad de los grupos de desarrollo y la reducción del riesgo global del proyecto.

Este artículo tiene un carácter tutorial y su objetivo es describir los fundamentos y aspectos metodológicos del desarrollo de software basado en componentes. En la sección 2 se establecen los conceptos de activo reutilizable y componente de software; de estos últimos se exponen las características mínimas y deseables que favorecen su reutilización. Las secciones 3, 4 y 5 discuten cómo los componentes se acoplan. Para ello se describe el papel de las interfaces, modelos y *frameworks* de componentes, y se analizan algunos de los mecanismos de composición de software más utilizados. Finalmente, la sección 6 describe los modelos de procesos y los aspectos metodológicos que rigen esta nueva forma de producir software.

II. ACTIVOS REUTILIZABLES Y COMPONENTES DE SOFTWARE

En el contexto de Ingeniería de Software, un *Activo Reutilizable* es un producto diseñado expresamente para ser empleado de forma recurrente en el desarrollo de muchos sistemas y aplicaciones. Ejemplos de activos reutilizables son: algoritmos, patrones de diseño, esquemas de base de datos, arquitecturas de software, especificaciones de requerimientos, de diseño y de prueba, entre otros.

En los últimos años, como resultado de presiones crecientes sobre la industria del software orientadas a reducir drásticamente el costo y tiempo de desarrollo de sistemas y aplicaciones, sin afectar los niveles de calidad del producto, ha surgido un nuevo activo reutilizable denominado *Componente de Software*. Se han propuesto numerosas definiciones a este término, entre las cuales destacan las siguientes:

- Según Philippe Krutchen de Rational Rose [4], un componente es una parte no trivial, casi independiente y reemplazable de un sistema que cumple una función dentro del contexto de una arquitectura bien definida. Un componente cumple con un conjunto de interfaces y provee la realización física de ellas.
- Según Clemens Szyperski [5], un componente de software es una unidad de composición con interfaces especificadas contractualmente y solamente dependencias explícitas de contexto. Un componente de software puede ser desplegado independientemente y está sujeto a composición por terceros.
- Según Herzum y Sims [6], un componente es un artefacto de software autocontenido y claramente identificable que describe ó ejecuta funciones específicas; que tiene, además, una interfaz claramente establecida, una documentación apropiada y un *status* de uso recurrente bien definido.
- Según el CBDi Forum [7], un componente es una pieza de software que describe y/o libera un conjunto de servicios que son usados sólo a través de interfaces bien definidas.

Más recientemente, el Instituto de Ingeniería de Software (SEI, *Software Engineering Institute*) de la Universidad Carnegie-Mellon formuló una definición con el propósito de consolidar las diferentes opiniones acerca de lo que debía ser un componente de software. Según el SEI [8], un componente es "una implementación opaca de funcionalidad, sujeta a composición por terceros y que cumple con un modelo de componentes". Con respecto al primer aspecto, un componente se considera una implementación opaca debido a que su distribución predominantemente es en formato binario y sus consumidores lo utilizan como una "caja negra" a través de su interfaz. Dicho aspecto está alineado con el principio de encapsulamiento de la programación orientada a objetos [9], [10]. Por otra parte, la composición por terceros implica que los componentes son intrínsecamente reutilizables debido a que un sistema basado en componentes puede ser ensamblado con relativa facilidad por un integrador empleando componentes suministrados por múltiples

proveedores independientes. Finalmente, la coordinación e interacción entre componentes exige un marco regulatorio estandarizado (modelo de componentes) que establece la infraestructura de software requerida (*framework*) y las convenciones y restricciones de diseño de los mismos.

Tal como lo refleja la definición anterior, un componente de software puede ser visto desde dos perspectivas distintas, como:

1. implementación: los componentes se pueden ensamblar y desplegar para crear sistemas y aplicaciones que se ejecutan en un computador
2. abstracción de arquitectura: los componentes expresan las reglas de diseño que impone el modelo de componentes.

Están disponibles diversas tecnologías de componentes típicamente asociadas con infraestructuras de procesamiento distribuido (e.g. Enterprise JavaBeans [11], CORBA Component Model [12] y .NET [13]) y aplicaciones de escritorio (e.g. Controles ActiveX [14] y JavaBeans [15]).

Una de las características más importantes de los componentes es que son reutilizables. Para ello los componentes deben satisfacer como mínimo el siguiente conjunto de características:

- **identificable:** un componente debe tener una identificación clara y consistente que facilite su catalogación y búsqueda en repositorios de componentes.
- **accesible sólo a través de su interfaz:** el componente debe exponer al público únicamente el conjunto de operaciones que lo caracteriza (interfaz) y ocultar sus detalles de implementación. Esta característica permite que un componente sea reemplazado por otro que implemente la misma interfaz.
- **sus servicios son invariantes:** las operaciones que ofrece un componente, a través de su interfaz, no deben variar. La implementación de estos servicios puede ser modificada, pero no deben afectar la interfaz.
- **documentado:** un componente debe tener una documentación adecuada que facilite su búsqueda en repositorios de componentes, evaluación, adaptación a nuevos entornos, integración con otros componentes y acceso a información de soporte.

Adicionalmente, para favorecer su reutilización es deseable que un componente sea:

- **genérico:** sus servicios pueden ser usados en una gran variedad de aplicaciones.
- **autocontenido:** es conveniente que un componente dependa lo menos posible de otros componentes para cumplir su función de forma tal que pueda ser desarrollado, probado, optimizado, utilizado, entendido y modificado individualmente.
- **mantenido:** es deseable que un componente (como toda pieza de software) esté inmerso en un proceso de mejoramiento continuo que le garantice al integrador nuevas versiones que incluyan correctivos, optimizaciones y nuevas características. Esto contribuye a que dicho componente sea seleccionado con mayor frecuencia para formar parte de sistemas de software.
- **independiente de la plataforma (hardware y sistema operativo), del lenguaje de programación y de las herramientas de desarrollo:** existen diversas plataformas de cómputo de uso frecuente (e.g. Windows/Intel, Solaris/Sparc, OSX/PPC, Linux/Intel) y es deseable que un componente pueda ejecutarse en todas ellas. Asimismo, ya que existe una amplia gama de lenguajes de programación y herramientas de desarrollo, es natural que encontremos componentes escritos empleando lenguajes y herramientas de la preferencia del programador, por lo tanto es deseable que dichas preferencias no limiten el uso de los componentes.
- **puede ser reutilizado dinámicamente:** puede ser cargado en tiempo de ejecución en una aplicación.
- **certificado:** el componente puede ser certificado por una agencia de software independiente o mediante la aplicación de modelos de auto-certificación que le permiten al comprador del componente determinar la calidad del software adquirido [16].
- **accedido uniformemente sin importar su localidad**: la forma de invocar los servicios ofrecidos por los componentes debiese ser independiente de su ubicación (local o remota). Para ello el modelo de componentes debería estar basado en tecnologías de procesamiento distribuido tales como CORBA [17], RMI [18] y .NET Remoting [13].

III. LA INTERFAZ DE UN COMPONENTE

Una interfaz define el conjunto de operaciones que un componente puede realizar; estas operaciones son llamadas también servicios o

responsabilidades. Las interfaces proveen un mecanismo para interconectar componentes y controlar las dependencias entre ellos.

La naturaleza de la interfaz varia dependiendo del lenguaje programación empleado para implementar el componente. Los lenguajes orientados a objetos (e.g. Smalltalk-80 [19], C++ [20] y Java [21]) soportan alguna forma de interfaz, que por lo general están separadas de las implementaciones. En lenguajes procedimentales (e.g. Pascal [22] y FORTRAN [23]) las interfaces se definen a través de declaraciones de funciones o procedimientos y el uso de variables globales. Algunos lenguajes neutrales de especificación de interfaces han sido desarrollados tales como IDL (*Interface Definition Language*) [17] de OMG (*Object Management Group*).

En general, una interfaz de programación de aplicaciones (API, *Application Programming Interface*) es una especificación, en un lenguaje de programación, de las propiedades de un módulo de software. Los clientes del módulo sólo deben depender exclusivamente de las propiedades definidas por el API de forma explícita.

Algunas tecnologías (e.g. Enterprise JavaBeans [11]) exigen que sus componentes implementen dos tipos de interfaces:

1. interfaz de negocio: que refleja el rol del componente en el sistema.
2. interfaz de infraestructura: es impuesta por el modelo de componentes con el fin de permitirle al *framework* interactuar con el componente.

Por otra parte, nótese que las interfaces convencionales definen la firma de las operaciones (nombre de la operación, tipo y orden de los argumentos, y la manera como se devuelven los resultados) que provee un componente. Las operaciones también se conocen como Propiedades Funcionales. Sin embargo, estas interfaces no expresan adecuadamente propiedades del componente relativas a, por ejemplo, su desempeño, precisión, disponibilidad, latencia, seguridad, entre otras. Dichas propiedades se conocen como Propiedades Extrafuncionales [24].

Es útil diferenciar los tipos de propiedades de los componentes. Por ejemplo, Beugnard et al. [25] define cuatro tipos de propiedades relacionadas con:

1. **Sintaxis**: corresponden a las propiedades funcionales expresadas explícitamente a través de la interfaz del componente.
2. **Comportamiento**: definen las condiciones que deben cumplir los valores de entrada (precondiciones) y salida (postcondiciones) de las operaciones.
3. **Sincronización**: expresan aspectos de concurrencia.
4. **Calidad de Servicio**: contempla atributos tales como tiempo de respuesta, uso de memoria, precisión, confiabilidad, facilidad de mantenimiento y reutilización, entre otros.

Se han realizado algunos intentos para que las interfaces expresen mejor las propiedades extrafuncionales, tales como el lenguaje de programación Eiffel [26] y el método formal Object-Z [27] para propiedades de comportamiento, Object Calculus [28] para propiedades de sincronización y la notación NoFun [29] para las propiedades de calidad de servicio.

Los párrafos anteriores sólo describen a las interfaces como una manera de especificar el flujo unidireccional de dependencia que tiene un cliente con respecto a un componente. Sin embargo, es mejor decir que un cliente y un componente dependen el uno del otro; un cliente depende de la forma en que un componente provee sus servicios, y un componente depende de cómo los clientes utilizan los servicios que éste ofrece. Esta interdependencia ha llevado a acuñar el término Contrato de Interfaz [2], [8] en la literatura de investigación acerca sistemas basados en componentes.

IV. *FRAMEWORKS* Y MODELOS DE COMPONENTES

Existe cierta confusión en la literatura referente a la terminología de modelos y *frameworks* de componentes. Sin embargo, hay consenso acerca de que los sistemas basados en componentes confían en estándares y convenciones bien definidas (modelo de componentes) y en una infraestructura de soporte (*framework* de componentes).

Los modelos de componentes especifican las reglas de diseño que deben obedecer los componentes. Estas reglas de diseño mejoran la composición, aseguran que las calidades de servicio se logren en todo el sistema, y que los componentes se puedan desplegar fácilmente tanto en entornos de desarrollo como de producción.

Los modelos de componentes imponen estándares y convenciones sobre:

- **Tipos de Componentes:** Un tipo de componente puede ser definido en términos de las interfaces que implementa. Los tipos diferentes de componentes pueden

desempeñar diferentes roles en el sistema, y participar en distintos tipos de esquemas de interacción.

- **Esquemas de Interacción:** especifican cómo los componentes son localizados, cuáles protocolos de comunicación son utilizados, y cómo se satisfacen las calidades de servicio (e.g. seguridad, transacciones, alta disponibilidad). El modelo de componentes puede describir cómo interactúan los componentes entre sí y cómo interactúan dichos componentes con el *framework*.
- **Asociación (*bindings*) de recursos**: El proceso de composición de componentes es simplemente enlazar un componente a uno o más recursos. Un recurso es un servicio ofrecido por un *framework* o por otro componente desplegado en ese *framework*. Un modelo de componentes describe cuáles recursos están disponibles a los componentes, y cómo y cuándo se asocian estos componentes a éstos recursos.

Por otra parte, los *frameworks* de componentes proporcionan servicios que soportan y hacen cumplir el modelo componentes asociado. El *framework* maneja recursos compartidos por los componentes, y proporciona mecanismos subyacentes que permiten la comunicación (interacción) entre ellos. Los *frameworks* son activos y actúan directamente sobre componentes, por ejemplo administrando su ciclo de vida (comenzar, suspender, reasumir, o terminar la ejecución de un componente), y otros recursos.

Existen muchos ejemplos de *frameworks* de componentes, entre éstos Enterprise JavaBeans (EJB) [11] y VisualBasic Framework (VBF) de Microsoft [14] son de los más representativos. La especificación de EJB define un *framework* de *servidores* y *contenedores* para dar soporte al modelo de componentes. Los *Servidores EJB* son responsables de proporcionar servicios de sistemas tales como persistencia, transacciones y seguridad, mientras que los *Contenedores EJB* son responsables de manejar el ciclo de vida del componente. Por su parte, VBF es quizás el *framework* más popular para el desarrollo de aplicaciones de escritorio. Se concentra en la composición visual de componentes (llamados VBXs) más que en tener un entorno que garantice la calidad de servicio de éstos. VBF incluye al intérprete de VisualBasic (para ejecutar scripts y hacer composición) y el Modelo de Objetos de Componentes (COM, Component Object Model) [7] (encargado de los servicios de despliegue y comunicación).

Un mercado robusto de componentes de software requiere modelos y *frameworks* estandarizados. Sin embargo, la experiencia ha demostrado que distintos dominios de aplicación tienen diferentes requisitos de funcionalidad y calidad de servicio (e.g. latencia, seguridad y disponibilidad). Esto sugiere la necesidad de tener una variedad modelos y *frameworks* de componentes para cada dominio. EJB, CCM y .NET se han abocado al dominio de aplicaciones empresariales (ERP, BPM, e-commerce y sistemas financieros) el cual es lo suficientemente grande y coherente para definir estándares de modelos y *frameworks* de componentes. Sin embargo, existen iniciativas que están abordando otros dominios de aplicación. Las más notorias son las promovidas por OMG para el control de tráfico aéreo, análisis de secuencias biomoleculares, mapas genómicos, infraestructura de clave pública, entre otras. Adicionalmente, WaterBeans [30] y SIMyO [31] son iniciativas relacionadas con tratamiento de agua, y modelado y optimización, respectivamente.

V. MECANISMOS DE COMPOSICIÓN DE SOFTWARE

Bajo el modelo de desarrollo de software basado en componentes, las nuevas aplicaciones se construyen mediante la integración o composición de componentes. Sametinger [2] define la composición de software como “el proceso de construir aplicaciones mediante la interconexión de componentes de software a través de sus interfaces (de composición)". Nótese que se hace especial énfasis en las interfaces como elementos fundamentales para lograr la composición de componentes.

La composición puede concebirse como una relación cliente-servidor entre dos componentes (Figura 1). El componente cliente solicita un servicio (operación) del componente servidor, el cual ejecuta la operación solicitada y devuelve los resultados al cliente. El servidor produce un resultado que es consumido por el cliente.

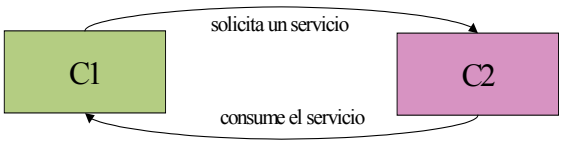

Figura 1. Interacción entre componentes.

Además de los componentes, los *frameworks* también se consideran entidades sujetas a composición. En consecuencia, existen tres clases

principales de interacción en los sistemas basados en componentes [24]:

1. Componente-Componente (C-C): permite la interacción entre componentes. De este tipo de interacción se obtiene la funcionalidad de la aplicación, de forma tal que los contratos que especifican este tipo de interacción pueden ser clasificados como Contratos a Nivel de Aplicación.
2. Componente-*Framework* (C-F): posibilita las interacciones entre el *framework* y sus componentes. Dicha interacción permite que el *framework* administre los recursos de los componentes. Los contratos que especifican estas interacciones pueden ser clasificados como Contratos a Nivel de Sistema.
3. *Framework-Framework* (F-F): posibilita las interacciones entre *framewoks* y permiten la composición de componentes desplegados en *framewoks* heterogéneos. Estos contratos puede ser clasificados como Contratos de Interoperabilidad.

La forma de materializar la composición entre componentes depende de los mecanismos especificados por su modelo de programación. Típicamente, los modelos de componentes se basan en tecnologías orientadas a objetos, por lo tanto los mecanismos de composición emplean algunas características tales como relaciones entre clases (especialización, agregación, asociación y uso) [10], polimorfismo y enlace dinámico [9]. Adicionalmente, dichos mecanismo de composición típicamente se describen mediante el uso de patrones de diseño [32], [33].

Las tecnologías de componentes no distribuidos, típicamente asociadas con aplicaciones de escritorio (e.g. Controles ActiveX [14] y JavaBeans [15]), hacen uso extensivo de características orientadas a objetos dentro de sus mecanismos de composición. Por el contrario, en la composición de componentes distribuidos (e.g. Enterprise JavaBeans [11], CORBA Component Model [12] y .NET [13]) principalmente se emplean relaciones de uso, asociación y agregación.

VI. EL PROCESO DE DESARROLLO

En las secciones anteriores, se caracterizaron los componentes de software y se describieron los mecanismos necesarios para que ellos se integren a fin de crear nuevas aplicaciones. Las preguntas que se intentan responder en esta sección son: ¿cómo se desarrolla un componente? y ¿cómo se crea una aplicación que reutilice componentes existentes?

Sommerville [1] clasifica los procesos de desarrollo de software basados en la reutilización de componentes en dos categorías:

- **Desarrollo de componentes:** Este proceso implica la adaptación o desarrollo de componentes con el propósito expreso de ser reutilizados en futuras aplicaciones. Su objetivo es producir repositorios de activos que puedan ser reutilizados en el desarrollo de software. Para ser reutilizables, estos componentes deben satisfacer las características descritas en la Sección 2.
- **Desarrollo de software con reutilización de componentes:** Es un proceso en el cual el desarrollo de una nueva aplicación involucra la reutilización de un conjunto de componentes existentes. Este enfoque maximiza la reutilización de componentes de software existentes y reduce el número de componentes que requieren ser desarrollados en su totalidad (desde cero). Para ser exitoso, este proceso demanda dos condiciones mínimas: i) la existencia de repositorios o bases de componentes reutilizables y ii) que los componentes sean confiables y actúen de acuerdo a sus especificaciones.

El modelo de procesos descrito por Sametinger [2] provee, sin embargo, una visión mucho más completa y amplia del desarrollo de software basado en componentes. Este modelo, denominado *ciclo de vida gemelo* (*twin life cycle*), divide el proceso de desarrollo de software en dos grandes bloques paralelos, tal como se ilustra en la Figura 2. El primer bloque, conocido como Ingeniería de Dominios, contempla los procesos necesarios para desarrollar activos de software reutilizables en un dominio particular. El segundo bloque es denominado Ingeniería de Aplicaciones. Su propósito es el desarrollo de aplicaciones basado en la reutilización de activos de software producidos a través de la Ingeniería de Dominios.

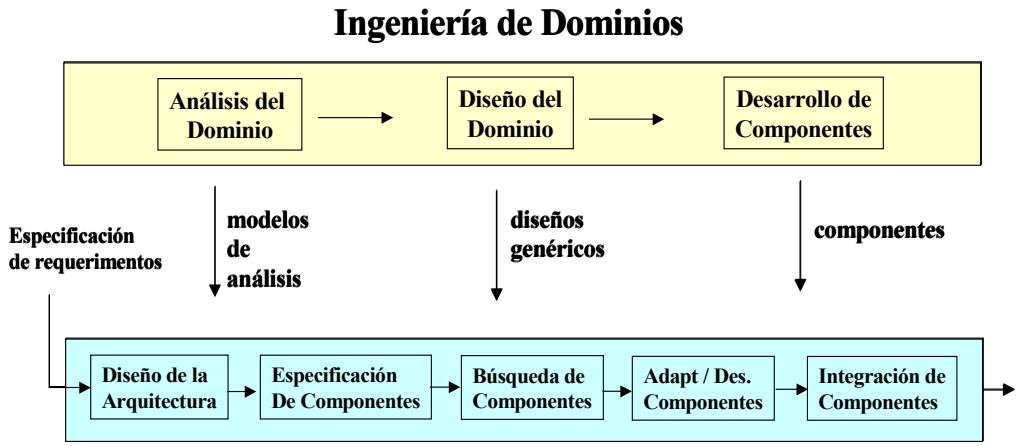

Figura 2. El modelo de procesos gemelos para el desarrollo de software basado en componentes

Un modelo alternativo al modelo de Ingeniería de Aplicaciones es el modelo WATCH [34], [35]. Este modelo combina los procesos más relevantes de la Ingeniería de Software Orientada a Objetos con el modelo de Ingeniería de Aplicaciones del ciclo de vida gemelo. Una característica importante de este modelo es la integración de los procesos gerenciales con los procesos técnicos del desarrollo de software basado en componentes. Esta integración facilita la labor del líder del proyecto, al describir cómo se debe llevar a cabo una gestión del proyecto integrada a los procesos técnicos del desarrollo de software.

La estructura del método WATCH se ilustra en la Figura 3. Esta estructura emplea la metáfora de un reloj de pulsera para describir el orden de ejecución de los procesos técnicos de desarrollo de aplicaciones, indicando además cómo los procesos gerenciales controlan o coordinan el orden de ejecución de los procesos técnicos. Los procesos gerenciales están ubicados en el centro de la estructura para indicar explícitamente que ellos programan, dirigen y controlan el proceso de desarrollo. Los procesos técnicos están ubicados en el entorno siguiendo la forma que tiene el dial de un reloj. Ello indica que el orden de ejecución de las fases técnicas se realiza en el sentido de las agujas del reloj. Los procesos gerenciales pueden, sin embargo, invertir el orden de ejecución para repetir algunas de las fases anteriores.

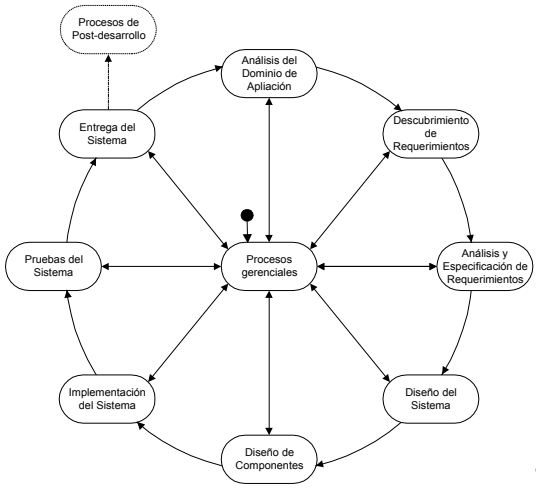

Figura 3. El modelo de procesos WATCH [34], [35]

Las tres primeras fases del modelo son similares a los modelos de procesos tradicionales. La fase de Análisis del Contexto permite que el grupo de desarrollo adquiera un conocimiento

adecuado del dominio o contexto del sistema en desarrollo. Las fases de Descubrimiento, Análisis y Especificación de Requerimientos se encargan de identificar las necesidades y requerimientos de los usuarios, así como analizarlos, especificarlos gráficamente y documentarlos.

La fase de diseño del sistema establece y describe la arquitectura del software. Describe cada uno de los componentes que requiere tal estructura y cómo esos componentes se interconectan para interactuar. El grupo de desarrollo debe, a partir de esta arquitectura, determinar cuáles componentes se pueden reutilizar y cuáles requieren ser desarrollados en su totalidad. Los componentes reutilizados deben ser adaptados, para satisfacer los requerimientos del sistema; mientras que los componentes nuevos, deben ser diseñados, codificados y probados separadamente durante la fase de Implementación. Las Pruebas del sistema permiten detectar errores en la integración de los componentes. Finalmente, la fase de Entrega se encarga de la instalación, adiestramiento de usuarios y puesta en operación del sistema.

VII. CONCLUSIONES

Los beneficios derivados de la reutilización de software están ocasionando un cambio acelerado en la manera en que la industria de software desarrolla sus productos. Los componentes de software reutilizables constituyen las unidades fundamentales para el desarrollo de nuevas aplicaciones. En este artículo, se ha hecho un intento por destacar la importancia y caracterizar el proceso de desarrollo de software basado en la reutilización de componentes. Se estableció una comparación entre los conceptos de activos reutilizables y componentes de software. Se describieron las características requeridas y deseables de un componente de software para su reutilización. Adicionalmente, se describieron los conceptos de interfaz, modelo y *framework* de componentes, así como también mecanismos de composición de software. Finalmente, se discutieron algunos de los aspectos metodológicos que rigen el desarrollo de componentes y de aplicaciones basadas en la reutilización de componentes.

VIII. AGRADECIMIENTOS

Los autores agradecen el apoyo económico suministrado por el Fondo Nacional de Ciencia, Tecnología e Innovación (FONACIT-Venezuela) a través de el proyecto de investigación titulado "Integración de Tecnologías y Sistemas de Software Heterogéneos en Aplicaciones Espacio-Temporales" (G-97000824).

IX. REFERENCIAS

[1] I. Sommerville. *Software engineering*. Addison-Wesley Pub Co, 6ta edición, Agosto 2000.

[2] J. Sametinger. *Software engineering with reusable components*. Springer Verlag, Agosto 1997.

[3] J. Sodhi, y P. Sodhi. *Software reuse: Domain analysis and design process*. McGraw-Hill. 1999.

[4] A. W. Brown and K. C.Wallnau. The current state of CBSE. *IEEE Software*, 15(5):37-46, Septiembre 1998.

[5] C. Szyperski. *Component software: Beyond object-oriented programming*. Addison-Wesley Pub Co, 2da edición, Noviembre 2002.

[6] P. Herzum and O. Sims. *Business component factory : A comprehensive overview of component-based development for the enterprise*. John Wiley & Sons, 2000.

[7] CDBI Forum. Component based development: Using componentised software. `http://www.cdbiforum.com`, Mayo 1999.

[8] F. Bachmann, L. Bass, Ch. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord y K. Wallnau. Volume II: Technical concepts of component-based software engineering, 2nd edition. Technical report, Software Engineering Institute, Carnegie Mellon University, Julio 2000.

[9] T. Budd. *Introducción a la programación orientada a objetos*. Addison-Wesley Iberoamericana. 1994.

[10] L. Joyanes Aguilar. *Programación orientada a objetos*. McGraw-Hill Interamericana, Diciembre 1998.

[11] Sun Microsystems, Inc. *Enterprise javabeans specification, version 2.0*. `http://java.sun.com/products/ejb/docs.html`, Agosto 2001.

[12] Object Management Group Inc. Corba components. `http://www.omg.org/cgi-bin/doc?formal/02-06-65`, Junio 2002.

[13] T. L. Thai and H. Lam. *.NET framework essentials*. O'Reilly & Associates, 3ra edición, 2003.

[14] D. Chappell. *Understanding ActiveX and OLE*. Microsoft Press, 1ra edición, Enero 1996.

[15] Sun Microsystems, Inc. *Javabeans*. `http://java.sun.com/products/javabeans/docs/spec.html`, Agosto 1997.

[16] J. Morris, G. Lee, K. Parker, G.A. Bundell, y C.P. Lam. Software component certification. *IEEE Computer*, 34(9), Septiembre, 2001.

[17] Object Management Group, Inc. Common object request broker architecture: Core specification. `http://www.omg.org/docs/formal/02-12-06.pdf`, Diciembre 2002.

[18] Sun Microsystems, Inc. *Java remote method invocation specification*. `ftp://ftp.java.sun.com/docs/j2se1.4/rmi-spec-1.4.pdf`.

[19] A. Goldberg and D. Robson. *Smalltalk 80 : The language*. Addison-Wesley Pub Co, Enero 1989.

[20] B. Stroustrup. *The C++ programming language*. Addison-Wesley Pub Co, 3ra edición, Febrero 2000.

[21] B. Joy, G. Steele, J. Gosling y G. Bracha. *The Java(TM) languaje specification*. Addison-Wesley Pub Co, 2da edición, Junio 2000.

[22] D. W. Nance. *Pascal: Understanding programming and problem solving*. West Information Pub Group, 4ta edición, Enero 1995.

[23] D. Rev. Smorlarski, Research & Education Association y Dennis Chester Smolarski. *The essentials of FORTRAN (essentials)*. Research & Education Assn, Mayo 1994.

[24] T. Digre. Business object component architecture. *IEEE Software*, 15(5), 1998.

[25] A. Beugnard, J-M Jézéquel y N. Plouzeau. Making components contract aware. *IEEE Computer*, 32(7):38-45, Julio 1999.

[26] B. Meyer. Eiffel: The language. Prentice Hall PTR, Octubre, 1991.

[27] R. Duke, G. Rose y G. Smith. Object-Z: A specification language advocated for the description of standards. *Computer Standards and Interfaces*, 17:511-533, Septiembre 1995.

[28] K. Lano, J. Bicarregui, J. Luiz Fiadeiro y T. Maibaum. Composition of reactive system components. En *1st Workshop on Component-Based Systems*, Zurich, Switzerland, 1997.

[29] X. Franch. Systematic formulation of non-functional characteristics of software.
In *3rd International Conference on Requirements Engineering (ICRE)*, pages 174-181, Colorado Springs (USA).

[30] K. C. Wallnau and D. Plakosh. *Waterbeans: A custom component model and framework*. `http://www.sei.cmu.edu/cbs/cbse2000/papers/23/23.html`, 2000.

[31] C. Arévalo, J. Colmenares, N. Queipo, N. Arapé y J. Villalobos. A CORBA and Web technology based framework for the analysis and optimal design of complex systems in the oil industry. En *3rd Internacional Conference on Enterprise Information Systems (ICEIS 2001)*, Julio 2001.

[32] E. Gamma, R. Helm, R. Johnson y J. Vlissides. *Design patterns*. Addison-Wesley Pub Co, Enero 1995.

[33] F. Marinescu. *EJB design patterns: Advanced patterns, processes and idioms*. John Wiley & Sons, Febrero 2002.

[34] J. Montilva, K. Hazam y M. Gharawi. The Watch Model for development business software in small and midsize organization. In *IV World Multiconference on Systemics, Cybernetics and Informatics (SCI'2000)*, Orlando, Florida (USA), Julio 2000.

[35] J. Montilva and J. Barrios. A Component-Based Method for Developing Web Applications. 5th International Conference on Enterprise Information Systems (ICEIS'2003), Angers, France, 2003.