DOCUMENTACION Y ANALISIS CRÍTICO DE ALGUNAS ARQUITECTURAS DE SOFTWARE EN APLICACIONES EMPRESARIALES

ALEJANDRA MARIA VALENCIA
MAURICIO FERRO GONZALEZ

UNIVERSIDAD TECNOLOGICA DE PEREIRA
FACULTAD DE INGENIERIAS
INGENIERIA DE SISTEMAS
PEREIRA
2011

DOCUMENTACION Y ANALISIS CRÍTICO DE ALGUNAS ARQUITECTURAS DE SOFTWARE EN APLICACIONES EMPRESARIALES

ALEJANDRA MARIA VALENCIA
MAURICIO FERRO GONZALEZ

MONOGRAFIA DE INVESTIGACION

JULIO CESAR CHAVARRO
Ingeniero en Sistemas

UNIVERSIDAD TECNOLOGICA DE PEREIRA
FACULTAD DE INGENIERIAS
INGENIERIA DE SISTEMAS
PEREIRA
2011

Nota de aceptación

_____
_____
_____
_____
_____
_____
_____
_____
_____

_____
Firma del presidente del jurado

_____
Firma del jurado

_____
Firma del jurado

DEDICATORIA

A Dios

Por habernos permitido cumplir con todos nuestros objetivos académicos.

A nuestras familias

Por acompañarnos paso a paso en este proceso de formación brindándonos todo su apoyo moral y económico.

A nuestros maestros

Por aportar todo su conocimiento para formarnos más que como profesionales como personas.

A nuestros compañeros

Por compartir con nosotros todo el proceso de formación profesional.

AGRADECIMIENTOS

Los autores de este proyecto de grado desean expresar agradecimientos a las siguientes personas que colaboraron con todo el proceso de elaboración, revisión y culminación de este trabajo:

A Dios y a nuestras familias que nos apoyaron incondicionalmente durante todo nuestro proceso de formación y en los momentos más difíciles de nuestras vidas, además de acompañarnos también en los mejores momentos.

A nuestro asesor el Ingeniero Julio César Chavarro por el apoyo, colaboración y dedicación durante el desarrollo de este proyecto.

A la Universidad Tecnológica de Pereira por alojarnos en sus instalaciones, además de proporcionarnos todos los medios y herramientas para formarnos como profesionales.

Y para finalizar a los jurados por su valioso aporte con el cual contribuyen a mejorar este trabajo.

Contenido

LISTA DE TABLAS

LISTA DE FIGURAS

1. GENERALIDADES

1.1 DEFINICIÓN DEL PROBLEMA

Las empresas están dejando de lado los sistemas separados que brindan funcionalidad aislada, para adoptar sistemas mucho más integrados en los cuales se potencian los servicios para ofrecer operaciones robustas y eficientes. Por lo tanto, los sistemas dentro de la empresa están más estrechamente integrados y los esfuerzos por modificarlos son más complejos.

El ingeniero de sistemas que trabaja en un proyecto ya no se puede focalizar exclusivamente en el sistema que se está modificando, sino que también debe comprender cómo interactúa el sistema con otros sistemas dentro de la empresa. Las empresas necesitan que sus aplicaciones reflejen la arquitectura de la empresa en sus aplicaciones para potenciar su productividad y los encargados de diseñar e implementar la arquitectura necesaria, deben tener los conocimientos suficientes de arquitectura empresarial de software y los diversos estilos que pueden utilizar para hacerlo.

1.2 JUSTIFICACIÓN

Los ingenieros de sistemas generalmente se concentran en el sistema que se está desarrollando actualmente, sin ocuparse mucho de la empresa que soporta dicho sistema. En la empresa de hoy, impulsada por los negocios, existe una relación directa entre la capacidad de negocios de la empresa y la funcionalidad implementada en los proyectos. Con un adecuado conocimiento en arquitectura empresarial de software, se puede desarrollar aplicaciones robustas y escalables para futuras implementaciones de manera que evite modificar la arquitectura de la empresa o verse limitada.

Por eso es útil tener a disposición información acerca de las arquitecturas de software más usadas para el diseño de aplicaciones empresariales para aquellos que no tienen mucha experiencia en este campo y resaltar la importancia de describir claramente la situación actual de la arquitectura empresarial antes y después de una implementación nueva.

Existen alrededor de 18 estilos arquitecturales y más de 2000 patrones arquitectónicos. Con este trabajo describiremos los estilos más utilizados de manera breve y clara para saber a qué clase de aplicaciones empresariales son aplicables cada uno.

1.3 OBJETIVOS

1.3.1 OBJETIVO GENERAL

Documentar arquitecturas de software en aplicaciones empresariales para proporcionar elementos de juicios que le permitan a los ingenieros en sistemas comprender mejor como sus esfuerzos en los proyectos que crea o modifican pueden verse limitados y a su vez modificar la arquitectura de la empresa la cual soportan.

1.3.2 OBJETIVOS ESPECÍFICOS

- Establecer un conjunto de características que permitan delimitar las aplicaciones empresariales.
- Realizar un compendio de los principales estilos arquitecturales y lenguajes de descripción arquitectural.
- Hacer un análisis crítico sobre la aplicación de la arquitectura de software en aplicaciones empresariales.

1.4 MARCO CONCEPTUAL

1.4.1 Aplicación empresarial.

A grandes rasgos podríamos decir que existen 3 grandes grupos de aplicaciones empresariales. El primer grupo está constituido por las aplicaciones de gestión general. Dentro de este grupo se encuentran los ERPs (del inglés, Enterprise Resource Managers) y los CRMs (del inglés, Customer Relationship Managers). Estas aplicaciones generalmente constituyen el núcleo de sistemas de una organización, brindando soporte a la gran mayoría de sus procesos, obligando a todos los demás sistemas a interactuar con ellos, directa o indirectamente. Ejemplos concretos son: SAP, Peoplesoft, Siebel y Financials a nivel mundial y Géminis, Tango y Calipso a nivel nacional.
El segundo grupo, está formado por aplicaciones de gestión particular que complementan a las aplicaciones del primer grupo y se encargan de alguna

problemática puntual de la organización. Generalmente, estás aplicaciones son utilizadas sólo por un grupo de usuarios de la organización.
Finalmente, en el tercer grupo se encuentran las aplicaciones de B2C (del inglés, Business to Customer). Estas aplicaciones a diferencia de las anteriores, son aplicaciones de internet, abiertas al público, por lo que la cantidad de usuarios puede ser varios órdenes de magnitud mayor que en los otros dos grupos mencionados.

1.4.2 ARQUITECTURA DE SOFTWARE

La arquitectura de software es importante como disciplina debido a que los sistemas de software crecen de forma tal que resulta muy complicado que sean diseñados, especificados y entendidos por un solo individuo. Uno de los aspectos que motivan el estudio en este campo es el factor humano, en términos de aspectos como inspecciones de diseño, comunicación a alto nivel entre los miembros del equipo de desarrollo, reutilización de componentes y comparación a alto nivel de diseños alternativos.

Bredemeyer proponen que los requerimientos arquitectónicos son necesarios para guiar las actividades de estructuración de un sistema de software. En el proceso de desarrollo se pueden aplicar diversos enfoques para garantizar el cumplimiento de los requerimientos arquitectónicos, así como la evaluación de las alternativas presentadas. La evaluación provee indicadores que permiten, en las fases tempranas, la oportunidad de resolver problemas que pueden presentarse a nivel arquitectónico.

La Arquitectura de Software es la organización fundamental de un sistema encarnada en sus componentes, las relaciones entre ellos y el ambiente y los principios que orientan su diseño y evolución.

1.4.3 DESEMPEÑO

Muchas decisiones de arquitectura son sobre el desempeño o “performance”, cualquier cambio significativo en la configuración puede invalidar el rendimiento o desempeño del sistema. Así que si se actualiza a una nueva versión de su máquina virtual, hardware, base de datos o cualquier otra cosa, se deben rediseñar las optimizaciones de desempeño y asegurarse que sigan siendo útiles.

Un problema que se tiene al hablar de desempeño es el hecho de que hay muchos términos que son usados de manera inconsistente. Nosotros utilizaremos estos términos:

Tiempo de respuesta: es la cantidad de tiempo que toma para que el sistema procese una petición externa.
Sensibilidad: es cuán rápido el sistema reconoce una petición en vez de procesarla. Esto es importante en muchos sistemas porque los usuarios se pueden sentir frustrados si el sistema tiene una sensibilidad muy baja.
Latencia: es el tiempo mínimo requerido para recibir cualquier tipo de respuesta, incluso si el trabajo a ser realizado es imperceptible. Es la razón principal por la que se debe minimizar el uso de llamadas remotas.
Capacidad de procesamiento: significa cuanto se puede hacer en una cantidad de tiempo dado. Para aplicaciones empresariales la medida típica es transacciones por segundo (tps).
Carga: Es un estado que nota cuanto estrés está manejando el sistema, el cual puede ser medido con el número de usuarios conectados al sistema.
Sensibilidad de carga: es una expresión de cómo varia el tiempo de respuesta con la carga.
Eficiencia: es el desempeño dividido por los recursos. Un sistema que recibe 30 tps en 2 CPUs es más eficiente que un sistema que recibe 40 tps en 4 CPUs idénticas.
Capacidad: es una indicación de la máxima capacidad de procesamiento o de la carga.
Escalabilidad: es una medida de cómo agregar recursos (usualmente hardware) afecta el desempeño. Un sistema escalable es el que permite agregar hardware y conseguir una mejora de desempeño conmensurable, como por ejemplo duplicar el número de servidores que se tienen para duplicar la capacidad de procesamiento. Escalabilidad vertical significa poner más poder a un solo servidor, como por ejemplo memoria. Escalabilidad horizontal significa agregar más servidores. Cuando se construyen sistemas empresariales, tiene sentido construirlo para escalabilidad de hardware en vez de capacidad o eficiencia. La escalabilidad da la opción de mejor desempeño si se necesita.

1.4.4 PATRONES

Un patrón describe un problema que se produce frecuentemente y las pautas para solucionarlo. Se basan en la práctica y aunque la base es aplicar las mismas pautas, la solución nunca es exactamente la misma. El enfoque del patrón es una solución particular, una que es común y efectiva para resolver uno o más problemas recurrentes.

Los patrones no son soluciones completas, siempre hay que editarlos y acondicionarlos al sistema propio que se esté desarrollando.

Estructura de los patrones: El primer elemento es el nombre del patrón. El nombre es crucial porque parte de los patrones es crear un vocabulario que permita a los diseñadores comunicarse efectivamente. Luego están los elementos: propósito y bosquejo. El propósito resume el patrón en una frase o 2; el bosquejo es una representación visual del patrón, a menudo es un diagrama UML. Se suele agregar la siguiente información adicional:
Cómo funciona describe la solución. Cuándo usarlo describe cuando el patrón debería ser usado. Ejemplos pequeña lista de ejemplos donde el patrón está siendo utilizado, ilustrados con código en lenguajes como C, Java, etc.

Los patrones siempre están “incompletos” y por eso los ingenieros tienen la responsabilidad de completarlos de acuerdo al contexto de su propio sistema.

1.4.5 LAS TRES CAPAS PRINCIPALES

Tabla 1. Las tres capas principales

Capas	Responsabilidades
Presentación	Prestación de servicios, presentación de la información (por ejemplo, en ventanas o HTML, manejar la petición del usuario (clics del ratón, los golpes de teclado, las solicitudes HTTP, las invocaciones de línea de comandos)
Lógica Dominio	La lógica que es el punto real del sistema
Fuente de datos	Comunicación con bases de datos, sistemas de mensajería, los administradores de transacciones, otros paquetes

Fuente FOWLER, Martin, *et al*. *Patterns of Enterprise Application Architecture.* Boston: Pearson Education Inc, 2002. 560p. ISBN 0-321-12742-0.

1.4.6 LOGICA DE DOMINIO

La lógica de dominio se refiere a todos los paquetes y clases relacionados con las reglas del negocio. Los elementos de la capa de Presentación necesitan a los de Lógica de dominio para funcionar, los de Lógica de dominio necesitan de los elementos de la capa de Datos y estos a su vez necesitan de los elementos de Lógica de Dominio. Se suele categorizar en tres patrones: Transaction Script, Domain Model, Table Module.

Transaction Script

Es el enfoque más sencillo para gestionar la lógica de dominio. Es un procedimiento que recibe los parámetros de entrada de la presentación, los procesa, almacena información en la base de datos, invoca operaciones de otros sistemas y responde con más datos a la presentación. La organización fundamental es un procedimiento único para cada acción del usuario, por ejemplo un script.

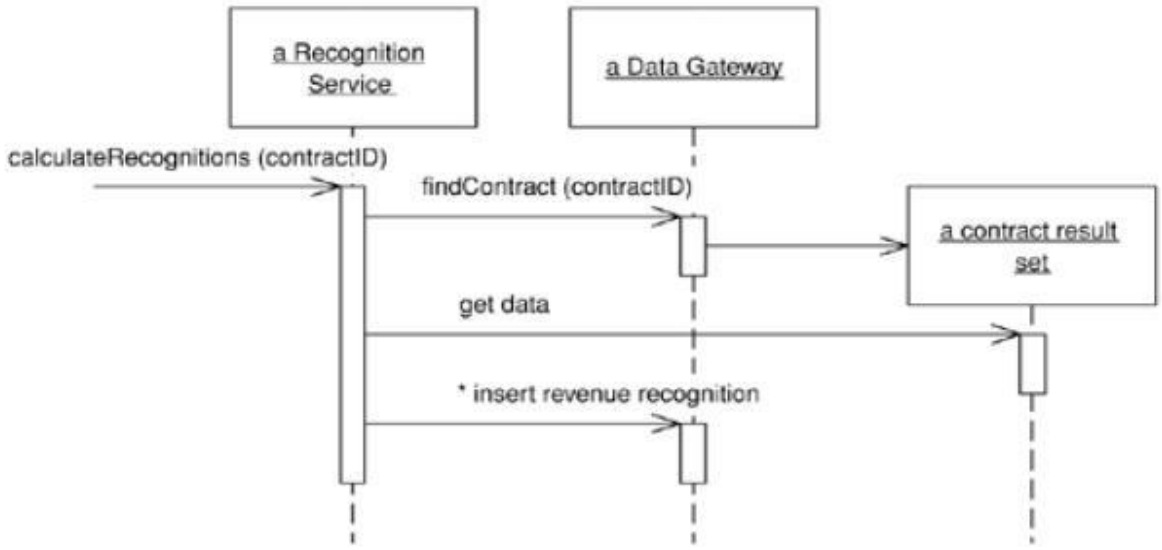
Ventajas:

- Modelo procedural simple.
- Funciona bien con una capa de datos simple.
- Hace obvios los límites de la transacción: se abre al principio del procedimiento y se cierra al acabar.

Desventajas:

- Aparecen muchas a medida que la complejidad de la lógica de dominio aumenta. A menudo habrá código duplicado, difícil de eliminar. Figura 1. Cálculo de reconocimiento de ingresos utilizando Transaction Script

Figura 1. Cálculo de reconocimiento de ingresos utilizando Transaction Script

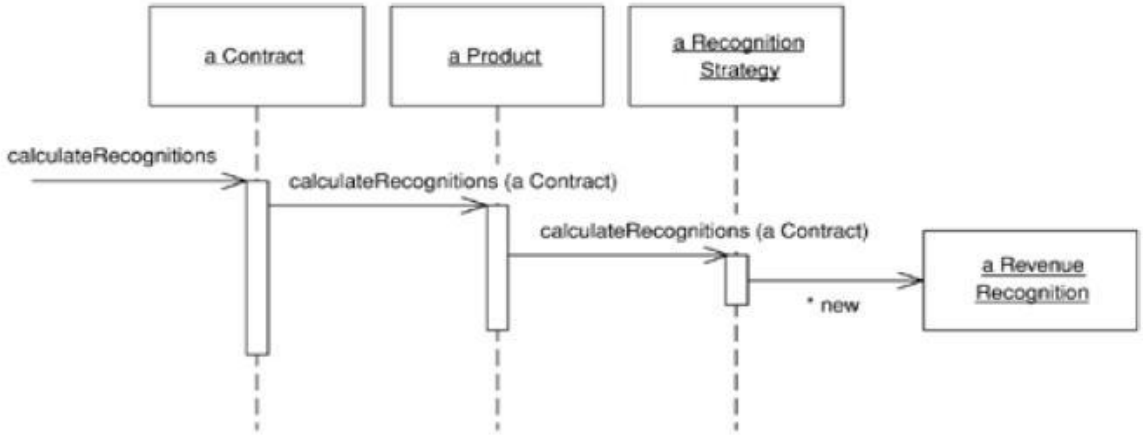


Fuente FOWLER, Martin, *et al*. *Patterns of Enterprise Application Architecture.* Boston: Pearson Education Inc, 2002. 560p. ISBN 0-321-12742-0.

Domain Model

Se construye un modelo del dominio organizado primordialmente alrededor de los nombres en el dominio. Es la manera orientada a objetos de manejar problemas donde lógica compleja es necesaria. La diferencia de utilizar un Domain Model en vez de un Transaction Script está en que en vez de utilizar una rutina que contiene toda la lógica para una acción del usuario, cada objeto toma parte de la lógica que es relevante para sí mismo. El valor fundamental del Domain Model está en que una vez el desarrollador se haya acostumbrad a él, hay muchas técnicas que permiten manejar lógica de complejidad en aumento de una forma bien organizada y con facilidad.

Figura 2. Cálculo de reconocimiento de ingresos utilizando Domain Model

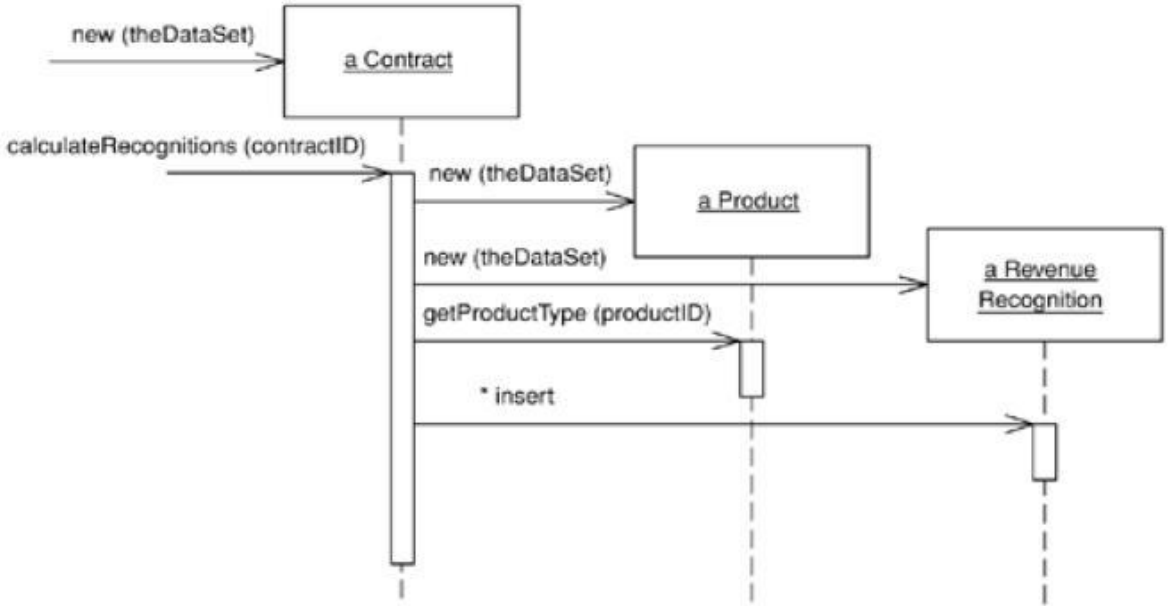


Fuente FOWLER, Martin, *et al*. *Patterns of Enterprise Application Architecture.* Boston: Pearson Education Inc, 2002. 560p. ISBN 0-321-12742-0.

Table Module

Representa el término medio entre las dos opciones anteriores. No puede manejar lógica de dominio demasiado compleja pero encaja muy bien con el modelo relacional de bases de datos. Organizar la lógica de dominio alrededor de tablas en vez de procedimientos directos provee mas estructura y hace más fácil encontrar y remover duplicación, sin embargo no permite usar varias técnicas que un Domain Model utiliza para una estructura de la lógica más finamente organizada, tales como herencia, estrategias y otros patrones orientados a objetos.

Figura 3. Cálculo de reconocimiento de ingresos utilizando Table Module



Fuente FOWLER, Martin, *et al*. *Patterns of Enterprise Application Architecture.* Boston: Pearson Education Inc, 2002. 560p. ISBN 0-321-12742-0.

1.5 MARCO TEÓRICO

Ya sea documentada o no, toda empresa tiene una arquitectura integrada por componentes y sus relaciones y colaboraciones, a menudo capturadas en dibujos, diagramas, documentos, modelos, etc. Además de la arquitectura, la empresa tiene una serie de requisitos que debe cumplir. También hay pruebas para determinar cuán bien la empresa cumple con sus requisitos.
Cuando se implementa una nueva edición de algún componente de la empresa, se realizará una cantidad de pruebas para garantizar que el componente cumpla con sus requisitos. Esto incluye que no dañe cualquier funcionalidad de mayor nivel por la forma en que interactúa con otros componentes. Si estas pruebas detectan algún problema, éste debe rastrearse como defectos de la empresa hasta tanto se resuelva.
Por ello, observamos que estos artefactos, cuando existen y se combinan, forman una descripción completa de elementos clave de la situación actual de la empresa:

- Requisitos (y sus impulsores, como motivación y objetivos)
- Arquitectura (incluyendo diseño e implementación)

- Pruebas
- Defectos

Según esto, el propósito de un programa es mover a la empresa del estado actual al estado futuro. Muchas veces esto incluye crear una serie de artefactos que describen el estado futuro. Sin embargo, si el estado actual está bien documentado, no es necesario volver a documentar las porciones de elementos (requisitos, arquitectura y pruebas) que no son modificadas por el programa. Sólo es necesario actualizar los artefactos actuales con los cambios establecidos por el programa. Estos cambios son deltas que se necesitan aplicar a los artefactos actuales para describir el estado futuro deseado.
Cada proyecto tiene un alcance específico que debe cumplir. Ese alcance está directamente relacionado con los cambios requeridos en la arquitectura para implantar la nueva capacidad. Es decir el programa define qué nueva funcionalidad se requiere de los sistemas afectados para implantar la capacidad, y cada proyecto implanta la nueva funcionalidad para su/s sistema/s.

Según Microsoft El diseño de la arquitectura de un sistema es el proceso por el cual se define una solución para los requisitos técnicos y operacionales del mismo. Este proceso define que componentes forman el sistema, cómo se relacionan entre ellos y como llevan a cabo la funcionalidad especificada, cumpliendo con los criterios de calidad indicados como seguridad, disponibilidad, eficiencia, etc. Durante el diseño de la arquitectura se tratan los temas que pueden determinar el éxito o fracaso de nuestro sistema Es necesario tener en cuenta:

¿En qué entorno va a ser desplegado nuestro sistema?
¿Cómo va a ser nuestro sistema puesto en producción?
¿Cómo van a utilizar nuestros usuarios nuestro sistema?
¿Qué otros requisitos debe cumplir el sistema? (seguridad, desempeño, escalabilidad...)
¿Qué cambios en la arquitectura pueden impactar al sistema ahora o una vez desplegado?
En el diseño de la arquitectura lo primero que se decide es el tipo de sistema o aplicación que vamos a construir. La selección de un tipo de aplicación determina en cierta medida el estilo arquitectural que se va usar. Los principales son Cliente/Servidor, Sistemas de componentes, Arquitectura en capas, MVC, N-Niveles, SOA, etc. Los procesos de software actuales asumen que el sistema cambiará con el paso del tiempo; el sistema tendrá que evolucionar a medida que se prueba la arquitectura contra los requisitos del mundo real.[1]

[1] DE LA TORRE LLORENTE César y ZORRILLA Unai. Guía de arquitectura de N-Capas orientada al dominio con .NET. España: Microsoft ibérica, 2010. ISBN 978-84-936696-3-8.

2. ARQUITECTURAS EMPRESARIALES DE SOFTWARE

2.1 ANTECEDENTES

Por más de una década, los patrones han influido en cómo los arquitectos de software y los desarrolladores crean sistemas de computación. Los patrones enfocados al diseño proveen un vocabulario para expresar visiones arquitecturales, diseños claros, concisos e implementaciones detalladas.

Mucho ha cambiado desde que Erich Gamma, Richard Helm, Ralph Johnson y John Vlisides (el grupo de los cuatro) publicaron "Patrones de Diseño"[2], el libro más popular sobre patrones. Este libro es aún muy influencial y popular, porque muchos desarrolladores no están al tanto de cómo ha madurado el campo o donde encontrar publicaciones sobre patrones que cubran un rango más elevado de dominios y tecnologías.

Inspirados por el éxito del libro Patrones de Diseño, mucha parte de la investigación sobre patrones en los 90 se enfocaba en patrones independientes y colecciones de patrones. Patrones independientes son 'soluciones puntuales' que se dirigen a problemas que resultan en un contexto específico. Cualquier diseño de software significativo incluye muchos patrones lo que hace que los patrones independientes sean inusuales en la práctica. Una colección de patrones es la opción adecuada. La colección de patrones más ambiciosa ha sido el libro "Handbook of Software" por Grady Booch, que relaciona aproximadamente 2000 patrones. La mayoría de colecciones de patrones son mucho más modestas y normalmente se centran en un problema o sistema en particular. Los patrones más maduros se pueden encontrar en los libros de Addison-Wesley "Pattern Languages of Program Design". A inicios de 1995 los autores líderes en la comunidad de patrones empezaron a documentar grupos de patrones para dominios específicos de desarrollo de software, cuando se publicó el primer "Pattern Languages of Program Design" se notó que los grupos de patrones eran más relacionados entre sí a diferencia de los patrones independientes anteriores.

El concepto de capas se popularizó en los 90 con la aparición de los sistemas cliente-servidor. Estos son sistemas de dos capas: el cliente con la interfaz de usuario y código de aplicación y el servidor, generalmente una base de datos.

El problema surgió con la lógica de dominio: reglas de negocio, validaciones, cálculos,... No encaja ni en la capa cliente ni en el servidor. La respuesta fue la arquitectura en tres capas: una capa presentación para la interfaz de usuario, una capa de dominio con la lógica de dominio y una capa de datos. El éxito del modelo Web ha acabado de consolidar la arquitectura de tres capas.

---

[2] GAMMA, Erich; HELM, Richard, JOHNSON, Ralph y VLISSIDES, John. *Design Patterns: Elements of reusable object oriented software.* En: *Reading, Addison-Wesley*, 1995.

La comunidad de patrones ha buscado entender las teorías, formas y metodologías de patrones y lenguajes de patrones para ayudar a codificar conocimientos sobre la aplicación efectiva de patrones de software. El trabajo más extenso en esta área ha sido “Pattern Oriented Software Architecture Volume 5” que integra varias facetas del concepto de patrones en un todo coherente.

2.2 LA IMPORTANCIA DE LA ARQUITECTURA DE SOFTWARE

La Arquitectura de software es importante como nivel de descripción para los sistemas de software. En este nivel de abstracción, se plantean temas claves del diseño que incluyen: bajo nivel de descomposición de un sistema en subsistemas que interactúan, la asignación de funciones a sistemas de cómputo, los protocolos de interacción entre los componentes, las propiedades globales del sistema (como el rendimiento y la latencia), y el ciclo de vida de los problemas (como el grado de reutilización, y la independencia de la plataforma).

Cuando los diseñadores discuten o presentan una arquitectura de software para un sistema específico, por lo general se trata el sistema como un conjunto de componentes que interactúan. Los componentes definen los cálculos primarios de la aplicación. Las interacciones o conexiones entre los componentes definen la forma como se comunican entre sí.

En la práctica una gran variedad de componentes y conectores se utilizan para representar diferentes formas de cálculo o de interacción. Como ejemplo de componentes son los filtros, objetos, bases de datos y servidores. Por el contrario, como ejemplo de tipos de conectores se incluyen las tuberías, los llamados a procedimientos, paso de mensajes y la difusión del evento.

La mayoría de las descripciones arquitectónicas son informales y esquemáticas, utilizan cajas de anotaciones para representar los componentes y líneas que representan las conexiones.[3]

La importancia de la arquitectura de software es resaltada desde diferentes puntos de vista. Por ejemplo: Según Kazman, la arquitectura de software es importante como disciplina debido a que los sistemas de software crecen de forma tal que resulta muy complicado que sean diseñados, especificados y entendidos por un solo individuo. Uno de los aspectos que motivan el estudio en este campo es el factor humano, en términos de aspectos como inspecciones de diseño,

[3] *Carnegie Mellom University* 21 de Julio de 1995. *Formalizing Style to Understand Descriptions of Software Architecture* [en linea]. Disponible desde internet en: <http://www.cs.txstate.edu/~rp31/papers/styleformalism-tosem95.pdf>

comunicación a alto nivel entre los miembros del equipo de desarrollo, reutilización de componentes y comparación a alto nivel de diseños alternativos.

De otra parte, Bredemeyer proponen que los requerimientos arquitectónicos son necesarios para guiar las actividades de estructuración de un sistema de software. En el proceso de desarrollo se pueden aplicar diversos enfoques para garantizar el cumplimiento de los requerimientos arquitectónicos, así como la evaluación de las alternativas presentadas. La evaluación provee indicadores que permiten, en las fases tempranas, la oportunidad de resolver problemas que pueden presentarse a nivel arquitectónico.[4]

La arquitectura en software es uno de los conceptos sobre los que más se habla, sin embargo el que menos se comprende y con el que luchan los desarrolladores. En conferencias, charlas y reuniones de grupos informales de debate se le presta suma atención al tema de la arquitectura, pero seguimos teniendo apenas unas vagas definiciones sobre ella. Cuando se habla sobre arquitectura, estamos refiriéndonos realmente a varios temas diferentes, pero relacionados entre sí, que generalmente se incluyen en las categorías generales de arquitectura de aplicaciones y arquitectura de negocios.

Arquitectura de aplicaciones: La arquitectura de aplicaciones describe cómo se ensamblan las piezas lógicas de la aplicación. Este es el reino de los patrones de diseño y de otras descripciones estructurales, y por lo tanto tiende a ser más abstracto y lógico que físico. Por ejemplo, podemos decir que una aplicación Web adhiere a un patrón Modelo Vista Presentador sin especificar qué marco se usa para lograr la configuración lógica.[5]

2.3 TAXONOMIA DE ARQUITECTURA Y MANUALES

El diseño arquitectónico ha sido reconocido como un aspecto crítico en la ingeniería de software de grandes sistemas. Sin embargo, es solo recientemente que la arquitectura de software ha empezado a emerger como una disciplina de estudio por derecho propio. Esto se ha logrado en parte por el reconocimiento del papel central de los patrones de diseño comunes y los estilos arquitectónicos.

[4] U.S.B Universidad Simon Bolivar. Abril del 2004. Arquitectura de software [en linea]. Disponible desde internet en: <http://prof.usb.ve/lmendoza/Documentos/PS-116/Guia%20Arquitectura%20v.2.pdf>

[5] IBM. 05-Abril-2010. Arquitectura evolutiva y diseño emergente: Investigación sobre arquitectura y diseño [en linea]. Disponible desde Internet en: <http://www.ibm.com/developerworks/ssa/java/library/j-eaed1/>

Entre los primeros esfuerzos para identificar, nombrar y analizar estas tendencias se encuentra la de Shaw, que en 1989 clasifica una serie de modismos y más tarde *Garlan* y *Shaw* amplían esta lista proporcionando varios ejemplos de su uso en la compresión de sistemas reales. Al mismo tiempo, *Perry* y *Wolf* también reconocieron la importancia de los patrones arquitectónicos, y describen el uso de estilos en las aplicaciones que se caracterizan como compiladores.[6]

Una manera diferente, pero relacionada, en el área de actividades ha surgido recientemente en la comunidad orientada a objetos a través de la articulación de los patrones de diseño. Inspirado, en parte, por el trabajo de *Christopher Alexander* sobre los lenguajes de patrones, esto ha dado lugar a los manuales de patrones comunes para la organización de software. Los patrones usualmente son un pequeño número de objetos que interactúan de forma específica.[7]

2.4 LA ARQUITECTURA DE SOFTWARE COMO ÁREA DE ESTUDIO

La necesidad del manejo de la arquitectura de un sistema de software nace con los sistemas de mediana o gran envergadura, que se proponen como solución para un problema determinado. En la medida que los sistemas de software crecen en complejidad, bien sea por número de requerimientos o por el impacto de los mismos, se hace necesario establecer medios para el manejo de esta complejidad. En general, la técnica es descomponer el sistema en piezas que agrupan aspectos específicos del mismo, producto de un proceso de abstracción[8] y que al organizarse de cierta manera constituyen la base de la solución de un problema en particular.

De aquí que la mayoría de los autores (Kazman, Hofmeister, Lane, Buschman Booch, Abowd) coinciden en que una arquitectura de software define la *estructura del sistema*. Esta estructura se constituye de *componentes* -módulos o piezas de código que nacen de la noción de abstracción, cumpliendo funciones específicas, e interactuando entre sí con un comportamiento definido.

Los componentes se organizan de acuerdo a ciertos criterios, que representan decisiones de diseño. En este sentido, hay autores que plantean que la

---

[6] *Carnegie Mellom University* 21 de Julio de 1995. *Formalizing Style to Understand Descriptions of Software Architecture* [en linea]. Disponible desde internet en: <http://www.cs.txstate.edu/~rp31/papers/styleformalism-tosem95.pdf>

[7] *Carnegie Mellom University.* 21 de Julio de 1995. *Formalizing Style to Understand Descriptions of Software Architecture* [en linea]. Disponible desde internet en: <http://www.cs.txstate.edu/~rp31/papers/styleformalism-tosem95.pdf >

[8] BASS Len, CLEMENTS, Paul y KAZMAN, Rick. *Software Architecture in Practice. Reading, Addison-Wesley*, 1998.

arquitectura de software incluye *justificaciones* referentes a la organización y el tipo de componentes, garantizando que la configuración resultante satisface los requerimientos del sistema.

De esta manera, la arquitectura de software puede ser vista como la estructura del sistema en función de la definición de los componentes y sus interacciones[9]. La práctica ha demostrado que resulta importante extender el concepto considerando los *requerimientos* y *restricciones* del sistema (Boehm), junto a un *argumento* que justifique que la estructura definida satisface los requerimientos, dándole un sentido más amplio a la definición del término.

La arquitectura de software puede considerarse entonces como el "puente" entre los *requerimientos* del sistema y la implementación (Hofmeister). Las actividades que culminan en la definición de la arquitectura pueden ubicarse en las fases tempranas del ciclo de desarrollo del sistema: luego del análisis de los requerimientos y el análisis de riesgos, y justo antes del diseño detallado.

Desde esta perspectiva, la arquitectura constituye un *artefacto* de la actividad de diseño (Hofmeister), que servirá de *medio de comunicación* entre los miembros del equipo de desarrollo, los clientes y usuarios finales, dado que contempla los aspectos que interesan a cada uno (Kazman). Además, pasa a ser la base del diseño del sistema a desarrollar, razón por la cual en la literatura, la arquitectura es considerada como plan de diseño del sistema (Hofmeister), debido a que es usada como guía para el resto de las tareas del desarrollo.

De igual manera, serán de particular importancia las *propiedades no funcionales* del sistema de software, pues influyen notoriamente en la calidad del mismo. Estas propiedades tienen un gran impacto en el desarrollo y mantenimiento del sistema, su operatividad y el uso que éste haga de los recursos[10]. Entre las propiedades no funcionales más importantes se encuentran: modificabilidad, eficiencia, mantenibilidad, interoperabilidad, confiabilidad, reusabilidad y facilidad de ejecución de pruebas (Kazman). (1998)[11] proponen que el término "requerimiento no funcional" es disfuncional, debido a que implica que tal requerimiento no existe, o que es una especie de requerimiento que puede ser especificado independientemente del comportamiento del sistema. En este sentido, Bass indica que debe hacerse referencia a atributos de calidad, en lugar de propiedades no funcionales.

---

[9] BASS Len, CLEMENTS, Paul y KAZMAN, Rick. *Software Architecture in Practice. Reading, Addison-Wesley*, 1998.

[10] BUSCHMAN, Frank. *Past, Present, and Future Trends in Software Patterns.* En: *IEEE Software Magazine.* Julio 2007.

[11] BASS Len, Op cit.

Puede observarse que al hablar de arquitectura de software, se hace alusión a la especificación de la estructura del sistema, entendida como la organización de componentes y relaciones entre ellos; los requerimientos que debe satisfacer el sistema y las restricciones a las que está sujeto, así como las propiedades no funcionales del sistema y su impacto sobre la calidad del mismo; las reglas y decisiones de diseño que gobiernan esta estructura y los argumentos que justifican las decisiones tomadas.[12]

2.5 ARQUITECTURAS POR CAPAS

Desde el punto de vista de la forma como se distribuye el software en cada nodo de la red empresarial, existen tres propuestas de arquitecturas de capas para Sistemas de Información, donde las capas a veces reciben el nombre de niveles (en inglés *tiers*):

- Arquitectura de dos capas;
- Arquitectura de tres capas;
- Arquitectura de cuatro capas.

2.5.1 Dos capas

En la actualidad muchos sistemas de información están basados en arquitecturas de dos capas, denominadas:

- Nivel de aplicación;
- Nivel de la base de datos.

Existen herramientas de amplio uso que presuponen esta estructura (p. ej. Visual Basic + Access/SQL server). Estas arquitecturas fueron las primeras en aprovecharse de la estructura cliente-servidor (aplicación en los clientes, base de datos como servidor). Las desventajas de dos niveles son bien conocidas:

- El nivel de las aplicaciones se recargan, entremezclando aspectos típicos del manejo de la interfaz con las reglas del negocio;
- Las reglas del negocio quedan dispersas entre el nivel de aplicación y los "stored procedures" de la base de datos;

[12] U.S.B Universidad Simon Bolivar. Abril del 2004. Arquitectura de software [en linea]. Disponible desde internet en: <http://prof.usb.ve/lmendoza/Documentos/PS-116/Guia%20Arquitectura%20v.2.pdf>

- La aplicación queda sobrecargada de información de bajo nivel si hay que extraer los datos de varias bases de datos, posiblemente con estructuras diferentes.
- El nivel de aplicación puede ser demasiado pesado para el cliente.

2.5.2 Tres capas

El modelo de distribución y despliegue más extendido es este y aún se puede afirmar que existe una fuerte y bien avanzada tendencia a adoptar una arquitectura de tres capas:

- Aplicación [ojo]
- Dominio de la aplicación;
- Repositorio.

La mayoría de estos sistemas buscan conservar la tecnología de BD relacional para la capa del repositorio e introducir la tecnología OO para el dominio de la aplicación. Para la capa de presentación se pueden utilizar tanto la tecnología HTML (Java-enabled) como los Generadores GUI.

En terminología de BD los tres niveles pueden equipararse, a *grosso modo,* con:

- Esquema externo;
- Esquema conceptual;
- Esquema interno o de almacenamiento.

En el caso de la capa denominada lógica de dominio, o simplemente dominio, se hacer referencia a la lógica que se debe ensamblar en la aplicación y regularmente se incorpora en una sola aplicación que puede ser almacenada en una estación cliente o en un servidor de aplicaciones. La ventaja es que ahora la aplicación puede describirse únicamente en relación a la semántica de la aplicación, sin tener que preocuparse sobre cómo está implementado ese dominio (ubicación y estructura física de la data).

Un punto interesante es dónde ubicar el nivel del dominio de la aplicación, ¿en el lado del cliente o en el lado del servidor? Hay ventajas y desventajas asociadas con cada alternativa, al estudiante interesado le recomiendo el excelente análisis del Fowler sección 12.2.1, vp. 243-245).

2.5.3 Cuatro capas

Los desarrollos más recientes empiezan a experimentar con una capa adicional (para 2000 no había visto evidencia de esto en Venezuela):

- Presentación;
- Aplicación;
- Dominio de la aplicación;
- Repositorio

La idea básica es separar todo lo que es programación GUI de la aplicación per se (y por ende tiende a usar *frameworks para GUI* como MFC). El nivel de la presentación no hace cálculos, consultas o actualizaciones sobre el dominio --de hecho ni siquiera tiene visibilidad sobre la capa del dominio. La capa de la aplicación es la encargada de acceder a la capa del dominio, simplificar la información del dominio convirtiéndolo a los tipos de datos que entiende la interfaz: enteros, reales, cadenas de caracteres, fecha y clases contenedoras (*container, collection*).[13]

2.6 ARQUITECTURA EMPRESARIAL

Hay muchas definiciones para arquitectura empresarial. Por lo general, el término arquitectura empresarial aparece en mayúsculas como sustantivo propio (por ejemplo, la disciplina de la Arquitectura Empresarial), aunque, en esta monografía no lo estamos usando de esta manera. Nos referimos a la arquitectura empresarial simplemente como la descripción de la arquitectura de la empresa en cuestión.

La disciplina de la Arquitectura Empresarial aúna negocio, estrategia, proceso, método y componentes desde una cantidad de perspectivas diferentes. Estas perspectivas están definidas y varían según los diferentes enfoques dados a la Arquitectura Empresarial. Las Arquitecturas Empresariales son realizadas por Arquitectos Empresariales. Las responsabilidades de un Arquitecto Empresarial exceden el enfoque de este documento.

Por lo tanto el propósito de un arquitecto empresarial, es describir los componentes de una empresa, sus relaciones, cómo colaboran e interactúan entre sí con el “mundo exterior”. Una arquitectura empresarial ofrece la orientación para

[13] FOWLER, Martin, *et al*. *Patterns of Enterprise Application Architecture.* Boston: Pearson Education Inc, 2002. 560p. ISBN 0-321-12742-0.

implantar los componentes de la empresa. La implantación de los componentes produce un cambio en el estado de la empresa.[14]

2.7 PROGRAMACIÓN DE CAMBIOS EN LA EMPRESA

Según lo definido anteriormente, el propósito de un programa es mover a la empresa del estado actual al estado futuro. Muchas veces esto incluye crear una serie de artefactos que describen el estado futuro. Sin embargo, si el estado actual está bien documentado, no es necesario volver a documentar las porciones de elementos (requisitos, arquitectura y pruebas) que no son modificadas por el programa. Sólo es necesario actualizar los artefactos actuales con los cambios establecidos por el programa. Estos cambios son deltas que se necesitan aplicar a los artefactos actuales para describir el estado futuro deseado.

En lugar de empezar desde cero, los artefactos del programa deberán describir los cambios del estado actual. Esto asume que se ha comprendido bien (capturado) el estado actual. Si esto no fuera así, no todo está perdido. Como de todas formas es necesario documentar el estado futuro, los artefactos creados pueden convertirse en artefactos actuales a nivel de la empresa después que se ha creado el programa.

Los programas pueden variar en cuanto a su alcance, desde modificar un aspecto de la empresa, a transformar todo el negocio de la misma. Por ello, es fácil exceder el alcance de un programa único para generar el conjunto completo de artefactos actuales para la empresa. En cambio, cada programa puede generar los artefactos para las porciones que modifica. Como muchos programas afectan varias áreas de la empresa, las brechas se eliminan.

Este enfoque evita tener que esperar un conjunto completo de artefactos actuales antes de iniciar cualquier programa de cambio. Este enfoque puede avanzar hasta que las restantes brechas en los artefactos actuales de la empresa sean relativamente pequeñas y pueden resolverse mediante tareas separadas para cerrar directamente las brechas.

Para obtener una representación completa y coherente de la empresa, todos los programas empresariales deben usar convenciones estándares para representar tanto los artefactos actuales, como los futuros (o por lo menos convertir sus artefactos de/a la convención estándar), y efectivamente deben comenzar a

[14] IBM. 05 de abril de 2010. Arquitectura empresarial para Ingenieros de Sistemas [en linea]. Disponible desde Internet en: <http://www.ibm.com/developerworks/ssa/rational/library/edge/09/jun09/enterprisearchitecture/index.html>

construir en base a los artefactos creados por los programas anteriores. De lo contrario, será muy difícil lograr la co-relatividad entre los artefactos creados por diferentes programas y lograr una representación única coherente de la empresa. Además, los artefactos actuales se deben conservar como un repositorio único homogéneo. No es importante cómo se construye el repositorio, es decir si es un archivo único o una consolidación de bases de datos. Lo importante es que se conserve y se puede acceder a él como una representación consolidada coherente.

Si (o una vez que) los artefactos actuales de la empresa están disponibles, el programa deberá comenzar con estos artefactos y capturar los cambios que se necesitan implementar al programa. Esto incluye deltas de los requisitos, actualizaciones a la arquitectura y modificaciones en las pruebas acordes a los cambios de requisitos. Los deltas de los requisitos capturan los cambios deseados en el comportamiento esperado. Estos cambios deseados, aun cuando se informan inicialmente de manera informal o se perfeccionan más adelante, impulsan los deltas de la arquitectura. Tanto los deltas de requisitos, como los deltas de la arquitectura pueden impulsar cambios en el conjunto de pruebas.

Cuando se ejecuta el programa (es decir, sus proyectos son implantados), se pueden realizar las pruebas para verificar que los requisitos hayan sido cumplidos y para detectar cualquier defecto en la implantación, los requisitos, o las pruebas propiamente dichas. Normalmente cualquier defecto detectado se resolverá en el ámbito del programa. Sin embargo, algunos defectos no pueden resolverse en el ámbito del programa, y por lo tanto se convierten en defectos adicionales a nivel de la empresa.

Al finalizar el programa, la empresa está en el estado futuro definido por el programa. Como este es el nuevo estado actual para la empresa, es necesario actualizar los artefactos actuales a nivel de la empresa. Esto es sencillo porque el programa ya ha producido todos los cambios necesarios para los artefactos.[15]

[15] IBM. 05 de abril de 2010. Arquitectura empresarial para Ingenieros de Sistemas [en linea]. Disponible desde Internet en: <http://www.ibm.com/developerworks/ssa/rational/library/edge/09/jun09/enterprisearchitecture/index.html>

3. LENGUAJES DE DESCRIPCION ARQUITECTURAL

3.1 Modelos de fiabilidad en lenguajes de descripción arquitectural

La ingeniería de software basada en componentes (CBSE) surgió como resultado de las funcionalidades ya existentes. La principal abstracción sobre los componentes son piezas de código reutilizable, sujetándose a la composición que ocultan su estructura y procesos internos. Hay que tener muy claro las interfaces específicas, y cuáles son los puntos de interacción con el mundo exterior.

Durante la integración de componentes, estos pueden ser adaptados pero no modificados. La descripción formal de Arquitectura de software puede ayudar a superar la complejidad que surge con los componentes existentes en el desarrollo de software. Los lenguajes de descripción arquitectural tienen como objetivo unificar la descripción de los sistemas basado en componentes de software (CBSS), arquitecturas en diferentes niveles de abstracción. Ellos estudian la arquitectura en términos de los componentes, los conectores entre éstos y sus puertos.

Muchos de los ADLS, se han desarrollado en los últimos 10 años, aunque no hay muchos aceptados. Las versiones recientes de UML (después de 2.0) también se adaptan para la descripción de arquitectura de software basada en componentes. Unos de los inconvenientes que se deben tener en cuenta en el ámbito de los ADL es que la mayoría de ellos no apoyan explícitamente la descripción formal de las características no funcionales de los sistemas basados en componentes.

El estudio de las características no funcionales (NFC) de sistemas intensivos de software es uno de los temas importantes que deben considerarse con el fin de tener éxito en los proyectos CBSE. Las características funcionales se utilizan para definir lo que el software tiene que hacer. Por otro lado las NFCs, dan los requerimientos de cómo se debe proporcionar funcionalidad, en términos de medir las cualidades y / o restricciones.[16]

Es una notación que permite una descripción y análisis preciso de las propiedades observables de una arquitectura de software, dando soporte a distintos estilos arquitectónicos en diferentes niveles de abstracción. En las etapas iniciales de desarrollo del campo, han recibido la denominación de ADL, todo tipo de lenguajes, desde sistemas puramente formales, hasta lenguajes de especificación algebraica, de programación concurrente o de definición de interfaces. Ha

[16] DIMOV, Aleksandar y ILIEVA, Sylvia. *Reliability Models in Architecture Description Languages*. En: *Proceeding CompSysTech '07 Proceedings of the 2007 international conference on Computer systems and technologies.* New York: ACM, 2007. ISBN 978-954-9641-50-9.

resultado confusa su relación con otros lenguajes específicos de dominio, como los Lenguajes de configuración y de Interconexión de Módulos (MILS), algunos de los cuales han evolucionado hasta llegar a ser auténticos ADLS.

La necesidad de utilizar una notación propia para la especificación arquitectónica, que permita separar claramente los aspectos vinculados a la estructura del resto de los detalles del desarrollo, ha sido identificada casi desde el principio. Esto no significa, tal y como hemos comentado que todo lenguaje que haga una descripción específica de la estructura sea propiamente un ADL (o Lenguaje de Descripción de Arquitectura). Por el contrario se requiere el cumplimiento de seis propiedades para que éste sea considerado un ADL:

Composición: El lenguaje debe ser tal, que el arquitecto pueda dividir un sistema complejo en partes más pequeñas de manera jerárquica, o construir un sistema a partir de los elementos que lo constituyen.

Abstracción: La arquitectura expresa una abstracción que permite identificar a los distintos elementos en una estructura de alto nivel, así como su papel en la misma. Esta abstracción es específica, y se diferencia de otras utilizadas en el desarrollo.

Reutilización: Debemos poder reutilizar componentes, conectores, estilos y arquitecturas, incluso en un contexto diferente a aquél en el que fueron definidos.

Configuración: El lenguaje debe separar con claridad la descripción de elementos individuales y de la de las estructuras en que participen. Esta característica es fundamental.

Heterogeneidad: El ADL ha de ser independiente del lenguaje en que se implemente cada uno de los componentes que manipula; se deben de poder combinar patrones arquitectónicos diferentes en un único sistema complejo.

Análisis: Se deben de poder analizar la estructura, de modo que se puedan determinar sus propiedades con independencia de una implementación concreta, así como verificarlos después de cualquier modificación. Esto puede vincularse al uso de métodos formales que permitan la definición de arquitecturas sin ambigüedad semántica.

La mayoría de ADLS existentes carecen de alguna de estas propiedades. También se identifican que aspectos se consideran esenciales para determinar que un lenguaje es, efectivamente, un ADL: concretamente, la estructuración en componentes, la especificación de conectores, la definición de sus interfaces, y sobre todo, la descripción explicita de las configuraciones compuestas por todos ellos.

A pesar de todo lo dicho, en los últimos años ha habido cierta tendencia a intentar expresar una arquitectura de software utilizando notaciones de propósito general, sobre todo UML y su combinación con ROOM. Más todavía, el Proceso Unificado de Racional se basa en un uso intensivo de arquitecturas descritas mediante UML, e incluso existe una propuesta completa para la adaptación de UML a este propósito. Este enfoque resulta, en términos generales, contra intuitivo e inadecuado. Hay un estudio muy interesante respecto a una comparativa de lenguajes de descripción de arquitecturas realizado por Medvidovic y Taylor.

3.2 COMPONENTE

El concepto fundamental de la Arquitectura de Software es el de componente. En lenguaje coloquial un componente es una caja negra, la cual tiene una serie de funcionalidades que a continuación describiremos. En términos generales, se denomina componente a cada una de las partes o unidades de composición – por definición en las que se subdivide la funcionalidad de un sistema, y cuya unión da lugar al sistema completo. En su sentido más genérico, puede hacer referencia a cualquier tipo de elemento estructural, esto es, integrado en una estructura; y es precisamente con este significado con el que habitualmente se le utiliza en la Arquitectura de Software.
El término de componente no se limita a la arquitectura, sino que es de hecho utilizado en múltiples campos de la Ingeniería de Software desde la propuesta realizada por Douglas su connotación más habitual hace referencia más bien a aspectos de implementación, vinculados a los estudios de Desarrollo Basado en Componentes (DSBC). No obstante, la siguiente definición ha alcanzado cierta popularidad y se considera comúnmente aceptada:

Un componente es una unidad de composición de aplicaciones software, que posee un conjunto de interfaces y un conjunto de requisitos, y que ha de poder ser desarrollado, adquirido, incorporado al sistema y compuesto con otros componentes de forma independiente, en tiempo y espacio, Clemens Szyperski.

Mirando la anterior definición que se sustenta en aspectos de implementación, vuelve a ser bastante general. Desde este punto de vista, un componente es una estructura fuertemente encapsulada y con una interfaz bien definida, que se concibe con independencia del resto del sistema, y que se integra con el sistema o con otros componentes mediante mecanismos de composición e interacción. Un componente ha de presentar una interfaz definida al resto del sistema, donde indica lo que proporciona y lo que necesita. Desde este punto de vista, un componente es una caja negra donde el concepto de encapsulamiento es más rígido que el aplicado en orientación a objetos. Esto se aplica por igual tanto a

componentes de implementación, de especificación y arquitectónicos. Las ventajas de esta encapsulación son innegables, pero las posibilidades de adaptación quedan limitadas, haciendo necesario el uso de envolventes (wrappers), adaptadores o mediadores. Es ahora donde entra en juego los conectores, que son desarrollados con este objetivo.

3.3 CONECTOR

La primera vez que se habla de conector fue a partir de los trabajos de Mary Shaw, de su experiencia en Unicon. En este trabajo, propuso considerar por un lado el componente y por el otro lado su interacción (el conector). De esta forma hay una separación clara, que permite ampliar el nivel de abstracción y aumentar la modularidad del sistema.

Al introducir el concepto de conector, lo que se intenta es tener dos elementos con funciones dispares. Algunos elementos en informática se relacionan con otros sin importarnos como, pero Shaw propone que los conectores son ciudadanos de primera clase, que tienen significado por sí solos. Por ello se consideran elementos autónomos, que podremos reutilizar en otros contextos, dado que no son diseñados específicamente para ese componente.

Otra forma de ver al conector es como un protocolo de comunicación, entendido en su sentido más amplio. Por regla general cualquier artefacto que se comunica con otro es un conector. Por ejemplo, la llamada de procedimiento es un tipo clásico de conector. Tenemos dos tipos de vínculos entre los distintos elementos de una descripción arquitectónica:

- El propio conector que va a expresar la interacción existente entre varios componentes.

- Aquel que establece a su vez el enlace que relaciona cada componente con un conector determinado. Este enlace recibe, normalmente, el nombre de adjunción o attachment.

El término conector se puede utilizar en dos sentidos diferentes, aunque relacionados:

- Según la propuesta de Shaw, se puede entender como otro tipo de elemento análogo a un componente, y que se describe del modo indicado.

- También podemos mencionar la palabra conector haciendo referencia a cualquier interacción explicita entre dos componentes, lo que incluye a los

bindings o las conexiones, ver Rapide. Este último concepto se aplica para especificar la interacción a nivel de sistema, cuando se especifican las relaciones entre los conectores y el sistema que los agrega.

En este trabajo nos quedaremos con el primer matiz del término conector haciendo una distinción explícita, una distinción entre conector y conexión, pero es conveniente conocer otras acepciones que proponen algunos autores. Según estos autores por el solo hecho de disponer de una noción de conector, un lenguaje ya está reforzando su capacidad para especificar configuraciones, lo que constituye la tarea principal de cualquier ADL.

3.4 PUERTO

El concepto de puerto es cercano al de conector, pero no debe confundirse bajo ningún concepto. Con este nombre describimos cada uno de los puntos por los que un componente puede realizar cualquier tipo de interacción; dicho de otro modo, es cada uno de los fragmentos en los que se segmenta el interfaz de un componente.

Si hablábamos de un componente como una caja negra, entonces el puerto hace referencia a un punto de entrada o de salida de la caja negra. Para aquellos autores que ven los componentes como procesos, el puerto sería el canal de mensajes. Ha de tenerse en cuenta, sin embargo que los puertos de un componente no sólo expresan los servicios que éste oferta, sino también los requisitos que precisa; esto es, aquellas condiciones que necesita que cumpla el entorno para funcionar correctamente. Por todo ello, la analogía con los métodos de un objeto es simple, pero peligrosa.

Los puertos se agrupan definiendo una interfaz. En algunos sistemas se permite, incluso, que definan más de una. En otros, se asume que el puerto está sub - estructurado en varios puntos de entrada, y por tanto se define todo él como una interfaz completa.
La definición de los puertos es fundamental, ya que es algo externo a los componentes y condiciona la estructura de la arquitectura.[17]

La delimitación categórica de los ADLs es problemática. Ocasionalmente, otras notaciones y formalismos se utilizan como si fueran ADLs para la descripción de arquitecturas. Algunos casos serían CHAM, UML y Z. Aunque se hará alguna referencia a ellos, cabe puntualizar que no son ADLs en sentido estricto y que, a

[17] SIMARRO, Juan Matias. APLICACIÓN DEL ANÁLISIS DE DEPENDENCIAS A LA ARQUITECTURA SOFTWARE. España: UNIVERSIDAD DE CASTILLA-LA MANCHA ESCUELA POLITÉCNICA SUPERIOR, Diciembre de 2004. 32p.

pesar de los estereotipos históricos (sobre todo en el caso de UML), no se prestan a las mismas tareas que los lenguajes descriptivos de arquitectura, diseñados específicamente para esa finalidad. Aparte de CODE, que es claramente un lenguaje de programación, se excluirá aquí entonces toda referencia a Modechart y PSDL (lenguajes de especificación o el prototipo de sistemas de tiempo real), LOTOS (un lenguaje de especificación formal, asociado a entornos de diseño como CADP), ROOM (un lenguaje de modelado de objetos de tiempo real), Demeter (una estrategia de diseño y programación orientada a objetos), Resolve (una técnica matemáticamente fundada para desarrollo de componentes re-utilizables), Larch y Z (lenguajes formales de especificación) por más que todos ellos hayan sido asimilados a ADLs por algunos autores en más de una ocasión. Se hará una excepción con UML, sin embargo. A pesar de no calificar en absoluto como ADL, se ha probado que UML puede utilizarse no tanto como un ADL por derecho propio, sino como metalenguaje para simular otros ADLs, y en particular C2 y Wright.

Tabla 2. Algunos ADLs

ADL	FECHA	ORGANISMO INVESTIGADOR	OBSERVACIONES
Acme	1995	Monroe & Garlan (CMU), Wile (USC)	Lenguaje de intercambio de ADLs
Aesop	1994	Garlan (CMU)	ADL de propósito general, énfasis en estilos
ArTec	1994	Terry, Hayes-Roth,Erman (Teknowledge, DSSA)	Lenguaje específico de dominio - No es ADL
Armani	1998	Monroe (CMU)	ADL asociado a Acme
C2 SADL	1996	Taylor / Medvidovic (UCI)	ADL específico de estilo
CHAM	1990	Berry / Boudol	Lenguaje de especificación
Darwin	1991	Magee, Dulay, Eisenbach, Kramer	ADL con énfasis en dinámica
Jacal	1997	Kicillof, Yankelevich (Universidad de Buenos Aires)	ADL - Notación de alto nivel para descripción y prototipado
LILEANNA	1993	Tracz (Loral Federal)	Lenguaje de conexión de módulos
MetaH	1993	Binns, Englehart (Honeywell)	ADL específico de dominio
Rapide	1990	Luckham (Stanford)	ADL & simulación
SADL	1995	Moriconi,Riemenschneider (SRI)	ADL con énfasis en mapeo de refinamiento
UML	1995	Rumbaugh, Jacobson, Booch (Rational)	Lenguaje genérico de modelado – No es ADL
UniCon	1995	Shaw (CMU)	ADL de propósito general, énfasis en conectores y estilos
Wright	1994	Garlan (CMU)	ADL de propósito general, énfasis en comunicación
Xadl	2000	Medvidovic, Taylor (UCI, UCLA)	ADL basado en XML

Fuente GARLAN, David y SHAW, Mary. *An introduction to software architecture. CMU Software Engineering Institute Technical Report, CMU/SEI-94-TR-21, ESC-TR-94-21,* 1994.

3.5 CRITERIOS DE DEFINICION DE UN ADL

Los ADLs se remontan a los lenguajes de interconexión de módulos (MIL) de la década de 1970, pero se han comenzado a desarrollar con su denominación actual a partir de 1992 o 1993, poco después de fundada la propia arquitectura de software como especialidad profesional. La definición más simple es la de Tracz que define un ADL como una entidad consistente en cuatro “Cs”: componentes, conectores, configuraciones y restricciones (constraints). Una de las definiciones más tempranas es la de Vestal quien sostiene que un ADL debe modelar o soportar los siguientes conceptos:

- Componentes
- Conexiones
- Composición jerárquica, en la que un componente puede contener una sub-arquitectura completa
- Paradigmas de computación, es decir, semánticas, restricciones y propiedades no funcionales
- Paradigmas de comunicación
- Modelos formales subyacentes
- Soporte de herramientas para modelado, análisis, evaluación y verificación.
- Composición automática de código aplicativo

Basándose en su experiencia sobre Rapide, Luckham y Vera [LV95] establecen como requerimientos:

- Abstracción de componentes
- Abstracción de comunicación
- Integridad de comunicación (sólo los componentes que están conectados pueden comunicarse)
- Capacidad de modelar arquitecturas dinámicas

- Composición jerárquica

- Relatividad (o sea, la capacidad de mapear o relacionar conductas entre arquitecturas)

Tomando como parámetro de referencia a UniCon, Shaw y otros alegan que un ADL debe exhibir:

- Capacidad para modelar componentes con aserciones de propiedades, interfaces e implementaciones

- Capacidad de modelar conectores con protocolos, aserción de propiedades e implementaciones

- Abstracción y encapsulamiento

- Tipos y verificación de tipos

- Capacidad para integrar herramientas de análisis

Otros autores, como Shaw y Garlan[18] estipulan que en los ADLs los conectores sean tratados explícitamente como entidades de primera clase (lo que dejaría al margen de la lista a dos de ellos al menos) y han afirmado que un ADL genuino tiene que proporcionar propiedades de composición, abstracción, reusabilidad, configuración, heterogeneidad y análisis, lo que excluiría a todos los lenguajes convencionales de programación y a los MIL.

Según Medvidovic un ADL se compone así:

Componentes

Interfaz
Tipos
Semántica
Restricciones (constraints)
Evolución
Propiedades no funcionales

[18] GARLAN, David y SHAW, Mary. *An introduction to software architecture. CMU Software Engineering Institute Technical Report, CMU/SEI-94-TR-21, ESC-TR-94-21,* 1994.

Conectores

Interfaz
Tipos
Semántica
Restricciones
Evolución
Propiedades no funcionales

Configuraciones arquitectónicas

Comprensibilidad
Composición
Heterogeneidad
Restricciones
Refinamiento y trazabilidad
Escalabilidad
Evolución
Dinamismo
Propiedades no funcionales

Soporte de herramientas

Especificación activa
Múltiples vistas
Análisis
Refinamiento
Generación de código
Dinamismo

En base a las propuestas señaladas, definiremos a continuación los elementos constitutivos primarios que, más allá de la diversidad existente, son comunes a la ontología de todos los ADLs y habrán de ser orientadores de su tratamiento en este estudio.

- Componentes: Representan los elementos computacionales primarios de un sistema. Intuitivamente, corresponden a las cajas de las descripciones de caja y línea de las arquitecturas de software. Ejemplos típicos serían clientes, servidores, filtros, objetos, pizarras y bases de datos. En la mayoría de los ADLs los componentes pueden exponer varias interfaces, las cuales definen puntos de interacción entre un componente y su entorno.

- Conectores. Representan interacciones entre componentes. Corresponden a las líneas de las descripciones de caja-y-línea. Ejemplos típicos podrían ser tuberías (pipes), llamadas a procedimientos, broadcast de eventos, protocolos cliente- servidor, o conexiones entre una aplicación y un servidor de base de datos. Los conectores también tienen una especie de interfaz que define los roles entre los componentes participantes en la interacción.

- Configuraciones o sistemas. Se constituyen como grafos de componentes y conectores. En los ADLs más avanzados la topología del sistema se define independientemente de los componentes y conectores que lo conforman. Los sistemas también pueden ser jerárquicos: componentes y conectores pueden subsumir la representación de lo que en realidad son complejos subsistemas.

- Propiedades. Representan información semántica sobre un sistema más allá de su estructura. Distintos ADLs ponen énfasis en diferentes clases de propiedades, pero todos tienen alguna forma de definir propiedades no funcionales, o pueden admitir herramientas complementarias para analizarlas y determinar, por ejemplo, el through put y la latencia probables, o cuestiones de seguridad, escalabilidad, dependencia de bibliotecas o servicios específicos, configuraciones mínimas de hardware y tolerancia a fallas.

- Restricciones. Representan condiciones de diseño que deben acatarse incluso en el caso que el sistema evolucione en el tiempo. Restricciones típicas serían restricciones en los valores posibles de propiedades o en las configuraciones topológicas admisibles. Por ejemplo, el número de clientes que se puede conectar simultáneamente a un servicio.
- Estilos. Representan familias de sistemas, un vocabulario de tipos de elementos de diseño y de reglas para componerlos. Ejemplos clásicos serían las arquitecturas de flujo de datos basados en grafos de tuberías (pipes) y filtros, las arquitecturas de pizarras basadas en un espacio de datos compartido, o los sistemas en capas. Algunos estilos prescriben un framework, un estándar de integración de componentes, patrones arquitectónicos o como se lo quiera llamar.

- Evolución. Los ADLs deberían soportar procesos de evolución permitiendo derivar subtipos a partir de los componentes e implementando refinamiento de sus rasgos. Sólo unos pocos lo hacen efectivamente, dependiendo para ello de lenguajes que ya no son los de diseño arquitectónico sino los de programación.

- Propiedades no funcionales. La especificación de estas propiedades es necesaria para simular la conducta de runtime, analizar la conducta de los componentes, imponer restricciones, mapear implementaciones sobre procesadores determinados, etcétera.

3.6 LENGUAJES

En la sección siguiente del documento, revisaremos algunos de los ADLs fundamentales de la arquitectura de software contemporánea en función de los elementos comunes de su ontología y analizando además su disponibilidad para la plataforma Windows, las herramientas gráficas concomitantes y su capacidad para generar código ejecutable, entre otras variables de relevancia.

3.6.1 Acme – Armani

Acme se define como una herramienta capaz de soportar el mapeo de especificaciones arquitectónicas entre diferentes ADLs, o en otras palabras, como un lenguaje de intercambio de arquitectura. No es entonces un ADL en sentido estricto, aunque la literatura de referencia acostumbra tratarlo como tal. De hecho, posee numerosas prestaciones que también son propias de los ADLs. En su sitio oficial se reconoce que como ADL no es necesariamente apto para cualquier clase de sistemas, al mismo tiempo que se destaca su capacidad de describir con facilidad sistemas “relativamente simples”.

Objetivo principal – La motivación fundamental de Acme es el intercambio entre arquitecturas e integración de ADLs. Garlan considera que Acme es un lenguaje de descripción arquitectónica de segunda generación; podría decirse que es de segundo orden: un metalenguaje, una *lingua franca* para el entendimiento de dos o más ADLs, incluido Acme mismo. Con el tiempo, sin embargo, la dimensión metalingüística de Acme fue perdiendo prioridad y los desarrollos actuales profundizan su capacidad intrínseca como ADL puro.

La estructura se define utilizando siete tipos de entidades: componentes, conectores, sistemas, puertos, roles, representaciones y rep-mapas (mapas de representación).

Componentes*:* Representan elementos computacionales y almacenamientos de un sistema. Un componente se define siempre dentro de una familia.

Interfaces: Todos los ADLs conocidos soportan la especificación de interfaces para sus componentes. En Acme cada componente puede tener múltiples interfaces. Igual que en Aesop y Wright los puntos de interfaz se llaman puertos (*ports*). Los puertos pueden definir interfaces tanto simples como complejas, desde una signatura de procedimiento hasta una colección de rutinas a ser invocadas en cierto orden, o un evento de multicast.

Conectores: En su ejemplar estudio de los ADLs existentes, Medvidovic (1996) llama a los lenguajes que modelan sus conectores como entidades de primera

clase lenguajes de configuración explícitos, en oposición a los lenguajes de configuración *in-line*. Acme pertenece a la primera clase, igual que Wright y UniCon. Los conectores representan interacciones entre componentes. Los conectores también tienen interfaces que están definidas por un conjunto de roles. Los conectores binarios son los más sencillos: el invocador y el invocado de un conector RPC, la lectura y la escritura de un conector de tubería, el remitente y el receptor de un conector de paso de mensajes.

Semántica: Muchos lenguajes de tipo ADL no modelan la semántica de los componentes más allá de sus interfaces. En este sentido, Acme sólo soporta cierta clase de información semántica en listas de propiedades. Estas propiedades no se interpretan, y sólo existen a efectos de documentación.

Estilos*:* Acme posee manejo intensivo de familias o estilos. Esta capacidad está construida naturalmente como una jerarquía de propiedades correspondientes a tipos. Acme considera, en efecto, tres clase de tipos: tipos de propiedades, tipos estructurales y estilos. Así como los tipos estructurales representan conjuntos de elementos estructurales, una familia o estilo representa un conjunto de sistemas. Una familia Acme se define especificando tres elementos de juicio: un conjunto de tipos de propiedades y tipos estructurales, un conjunto de restricciones y una estructura por defecto, que prescribe el conjunto mínimo de instancias que debe aparecer en cualquier sistema de la familia. El uso del término “familia” con preferencia a “estilo” recupera una idea de uno de los precursores tempranos de la arquitectura de software, David Parnas.

3.6.2 ADML

Como hubiera sido de esperarse ante la generalización del desarrollo en la era del Web, ADML (Architecture Description Markup Language) constituye un intento de estandarizar la descripción de arquitecturas en base a XML. Está siendo promovido desde el año 2000 por The Open Group y fue desarrollado originalmente en MCC. The Open Group ha sido también promotor de The Open Group Architectural Framework (TOGAF).

ADML constituye además un tronco del que depende una cantidad de especificaciones más puntuales. Mientras ADML todavía reposaba en DTD (Document Type Definition), una sintaxis de metadata que ahora se estima obsoleta, las especificaciones más nuevas implementan esquemas extensibles de XML. La más relevante tal vez sea xADL (a pronunciar como “zaydal”), desarrollado por la Universidad de California en Irvine, que define XML Schemas para la descripción de familias arquitectónicas, o sea estilos. La especificación inicial de xADL 2.0 se encuentra en http://www.isr.uci.edu/projects/xarchuci/.

Técnicamente xADL es lo que se llama una aplicación de una especificación más abstracta y genérica, xArch, que es un lenguaje basado en XML, elaborado en Irvine y Carnegie Mellon para la descripción de arquitecturas. Cada tipo de conector, componente e interfaz de xADL incluye un placeholder de implementación que vendría a ser análogo a una clase virtual o abstracta de un lenguaje orientado a objetos. Éste es reemplazado por las variables correspondientes del modelo de programación y plataforma en cada implementación concreta. Esto permite vincular descripciones arquitectónicas y modelos directamente con cualquier binario, scripting o entidad en cualquier plataforma y en cualquier lenguaje.

3.6.3 Aesop

El nombre oficial es Aesop Software Architecture Design Environment Generator. Se ha desarrollado como parte del proyecto ABLE de la Universidad Carnegie Mellon, cuyo objetivo es la exploración de las bases formales de la arquitectura de software, el desarrollo del concepto de estilo arquitectónico y la producción de herramientas útiles a la arquitectura, de las cuales Aesop es precisamente la más relevante. La elaboración formal del proyecto ABLE, por otro lado, ha resultado en el lenguaje Wright, que en este estudio se trata separadamente. Uno de los mejores documentos sobre Aesop es el ensayo de David Garlan, Robert Allen y John Ockerbloom que explora el uso de estilos en el diseño arquitectónico.

La definición también oficial de Aesop es “una herramienta para construir ambientes de diseño de software basada en principios de arquitectura”. El ambiente de desarrollo de Aesop System se basa en el estilo de tubería y filtros propio de UNIX. Un diseño en Aesop requiere manejar toda una jerarquía de lenguajes específicos, y en particular FAM Command Language (FCL, a pronunciar como “fickle”), que a su vez es una extensión de TCL orientada a soportar modelado arquitectónico. FCL es una combinación de TCL y C densamente orientada a objetos. En lo que respecta al manejo de métodos de análisis de tiempo real, Aesop implementa EDF (Earliest Deadline First).

- Estilos - En Aesop, conforme a su naturaleza orientada a objetos, el vocabulario relativo a estilos arquitectónicos se describe mediante la definición de sub-tipos de los tipos arquitectónicos básicos: Componente, Conector, Puerto, Rol, Configuración y Binding.
- Interfaces - En Aesop (igual que en ACME y Wright) los puntos de interfaz se llaman puertos (ports).
- Modelo semántico – Aesop presupone que la semántica de una arquitectura puede ser arbitrariamente distinta para cada estilo. Por lo tanto, no incluye ningún soporte nativo para la descripción de la semántica de un estilo o

configuración, sino que apenas presenta unos cuadros vacantes para colocar esa información como comentario.

- Soporte de lenguajes - Aesop (igual que Darwin) sólo soporta nativamente desarrollos realizados en C++.
- Generación de código – Aesop genera código C++. Aunque Aesop opera primariamente desde una interfaz visual, el código de Aesop es marcadamente más procedural que el de Acme, por ejemplo, el cual posee una estructura de orden más bien declarativo.
- Disponibilidad de plataforma – Aesop no está disponible en plataforma Windows, aunque naturalmente puede utilizarse para modelar sistemas implementados en cualquier plataforma.

3.6.4 ArTek

ArTek fue desarrollado por Teknowledge. Se lo conoce también como ARDEC/Teknowledge Architecture Description Language. En opinión de Medvidovic no es un genuino ADL, por cuanto la configuración es modelada implícitamente mediante información de interconexión que se distribuye entre la definición de los componentes individuales y los conectores. En este sentido, aunque pueda no ser un ADL en sentido estricto, se le reconoce la capacidad de modelar ciertos aspectos de una arquitectura. De todas maneras, es reconocidamente un lenguaje específico de dominio y siempre fue presentado como un caso testigo de generación de un modelo a partir de una instancia particular de uso.
Disponibilidad de plataforma – Hoy en día ArTek no se encuentra disponible en ningún sitio y para ninguna plataforma.

3.6.5 C2 (C2 SADL, C2SADEL, xArch, xADL)

C2 o Chiron-2 no es estrictamente un ADL sino un estilo de arquitectura de software que se ha impuesto como estándar en el modelado de sistemas que requieren intensivamente pasaje de mensajes y que suelen poseer una interfaz gráfica dominante. C2 SADL (Simulation Architecture Description Language) es un ADL que permite describir arquitecturas en estilo C2. C2SADEL es otra variante; la herramienta de modelado canónica de este último es DRADEL (Development of Robust Architectures using a Description and Evolution Language). Llegado el momento del auge de XML, surge primero xArch y luego xADL, de los que ya se ha tratado en el apartado correspondiente a ADML y sus derivaciones, pero sin hacer referencia a su conformidad con C2, que en los hechos ha sido enfatizado

cada vez menos. Otra variante, SADL a secas, denota Structural Architecture Description Language.

Implementación de referencia – SADL se utilizó eficazmente para un sistema de control operacional de plantas de energía en Japón, implementado en Fortran 77. Se asegura que SADL permitió formalizar la arquitectura de referencia y asegurar su consistencia con la arquitectura de implementación.

Semántica – El modelo semántico de C2 es algo más primitivo que el de Rapide, por ejemplo. Los componentes semánticos se expresan en términos de relaciones causales entre mensajes de entrada y salida de una interfaz. Luego esta información se puede utilizar para rastrear linealmente una serie de eventos.

Soporte de lenguajes – C2 soporta desarrollos en C++, Ada y Java, pero en realidad no hay limitación en cuanto a los lenguajes propios de la implementación. Modelos de interoperabilidad de componentes como OLE y más recientemente COM+ no son ni perturbados ni reemplazados por C2, que los ha integrado con naturalidad.

Disponibilidad de plataforma – Existen extensiones de Microsoft Visio para C2 y xADL. Una interfaz gráfica para C2SADEL disponible para Windows es DRADEL. También se puede utilizar una herramienta llamada SAAGE (disponible en el sitio de DASADA, http://www.rl.af.mil/tech/programs/dasada/tools/saage.html) que requiere explícitamente Visual J++, COM y ya sea Rational Rose o DRADEL como front-ends.

3.6.6 CHAM

CHAM (Chemical Abstract Machine) no es estrictamente un ADL, aunque algunos autores, en particular Inverardi y Wolf aplicaron CHAM para describir la arquitectura de un compilador. Se argumenta, en efecto, que CHAM proporciona una base útil para la descripción de una arquitectura debido a su capacidad de componer especificaciones para las partes y describir explícitamente las reglas de composición. Sin embargo, la formalización mediante CHAM es idiosincrática y (por así decirlo) hecha a mano, de modo que no hay criterios claros para analizar la consistencia y la completitud de las descripciones de configuración. Convendrá contar entonces con alguna herramienta de verificación.

CHAM es una técnica de especificación basada en álgebra de procesos que utiliza como fundamento teórico los sistemas de rescritura de términos para capturar la conducta comunicativa de los componentes arquitectónicos. Disponibilidad de plataforma – CHAM es un modelo de máquina abstracta independiente de

plataforma y del lenguaje o paradigma de programación que se vaya a utilizar en el sistema que se modela.

3.6.7 Darwin

Darwin es un lenguaje de descripción arquitectónica desarrollado por Jeff Magee y Jeff Kramer. Darwin describe un tipo de componente mediante una interfaz consistente en una colección de servicios que son ya sea provistos (declarados por ese componente) o requeridos (o sea, que se espera ocurran en el entorno). Las configuraciones se desarrollan instanciando las declaraciones de componentes y estableciendo vínculos entre ambas clases de servicios.

Darwin soporta la descripción de arquitecturas que se reconfiguran dinámicamente a través de dos construcciones: instanciación tardía [lazy] y construcciones dinámicas explícitas. Utilizando instanciación laxa, se describe una configuración y se instancian componentes sólo en la medida en que los servicios que ellos provean sean utilizados por otros componentes. La estructura dinámica explícita, en cambio, se realiza mediante constructos de configuración imperativos. De este modo, la declaración de configuración deviene un programa que se ejecuta en tiempo de ejecución, antes que una declaración estática de la estructura.

Cada servicio de Darwin se modela como un nombre de canal, y cada declaración de binding es un proceso que trasmite el nombre del canal al componente que requiere el servicio. En una implementación generada en Darwin, se presupone que cada componente primitivo está implementado en algún lenguaje de programación, y que para cada tipo de servicio se necesita un ligamento (glue) que depende de cada plataforma.

- Objetivo principal – Como su nombre lo indica, Darwin está orientado más que nada al diseño de arquitecturas dinámicas y cambiantes.

- Estilos - El soporte de Darwin para estilos arquitectónicos se limita a la descripción de configuraciones que se parametriza, como la del ejemplo de la tubería que figura más arriba. Esta descripción, en particular, indica que una tubería es una secuencia lineal de filtros, en la que la salida de cada filtro se vincula a la entrada del filtro siguiente en la línea. Un estilo será entonces expresable en Darwin en la medida en que pueda ser constructivamente caracterizado; en otras palabras, para delinear un estilo hay que construir un algoritmo capaz de representar a los miembros de un estilo. Dada su especial naturaleza, es razonable suponer que Darwin se presta mejor a la descripción de sistemas que poseen características dinámicas.

- Interfaces – En Darwin las interfaces de los componentes consisten en una colección de servicios que pueden ser provistos o requeridos.

- Conectores – Al pertenecer a la clase en la que Medvidovic agrupa a los lenguajes de configuración in-line, en Darwin (al igual que en Rapide) no es posible ponerle nombre, sub-tipear o reutilizar un conector. Tampoco se pueden describir patrones de interacción independientemente de los componentes que interactúan.

- Semántica - Darwin proporciona una semántica para sus procesos estructurales mediante el cálculo. Cada servicio se modela como un nombre de canal, y cada declaración de enlace (binding) se entiende como un proceso que trasmite el nombre de ese canal a un componente que requiere el servicio. Este modelo se ha utilizado para demostrar la corrección lógica de las configuraciones de Darwin. Dado que el cálculo ha sido designado específicamente para procesos móviles, su uso como modelo semántico confiere a las configuraciones de Darwin un carácter potencialmente dinámico. En un escenario como el Web, en el que las entidades que interactúan no están ligadas por conexiones fijas ni caracterizadas por propiedades definidas de localización, esta clase de cálculo se presenta como un formalismo extremadamente útil.

- Análisis y verificación – A pesar del uso de un modelo de cálculo para las descripciones estructurales, Darwin no proporciona una base adecuada para el análisis del comportamiento de una arquitectura. Esto es debido a que el modelo no posee herramientas para describir las propiedades de un componente o de los servicios que presta. Las implementaciones de un componente vendrían a ser de este modo cajas negras no interpretadas, mientras que los tipos de servicio son una colección dependiente de la plataforma cuya semántica también se encuentra sin interpretar en el framework de Darwin.

- Interfaz gráfica – Darwin proporciona notación gráfica. Existe también una herramienta gráfica (Software Architect's Assistant) que permite trabajar visualmente con lenguaje Darwin. El desarrollo de SAA parecería estar discontinuado y ser fruto de una iniciativa poco formal, lo que sucede con alguna frecuencia en el terreno de los ADLs.

- Soporte de lenguajes – Darwin (igual que Aesop) soporta desarrollos escritos en C++, aunque no presupone que los componentes de un sistema real estén programados en algún lenguaje en particular.

- Observaciones – Darwin (lo mismo que UniCon) carece de la capacidad de definir nuevos tipos, soportando sólo una amplia variedad de tipos de servicio predefinidos. Darwin presupone que la colección de tipos de servicio es suministrada por la plataforma para la cual se desarrolla una implementación, y

confía en la existencia de nombres de tipos de servicio que se utilizan sin interpretación, sólo verificando su compatibilidad.

- Disponibilidad de plataforma – Aunque el ADL fue originalmente planeado para ambientes tan poco vinculados al modelado corporativo como hoy en día lo es Macintosh, en Windows se puede modelar en lenguaje Darwin utilizando Software Architect's Assistant. Esta aplicación requiere JRE. Se la puede descargar en forma gratuita desde http://www.doc.ic.ac.uk/~kn/java/saaj.html

3.6.8 JACAL

Es un lenguaje de descripción de arquitecturas de software de propósito general creado en la Universidad de Buenos Aires, por un grupo de investigación del Departamento de Computación de la Facultad de Ciencias Exactas y Naturales.

Objetivo principal – El objetivo principal de Jacal es lo que actualmente se denomina "animación" de arquitecturas. Esto es, poder visualizar una simulación de cómo se comportaría en la práctica un sistema basado en la arquitectura que se ha representado.

Más allá de este objetivo principal, el diseño de Jacal contempla otras características deseables en un ADL, como por ejemplo contar con una representación gráfica que permita a simple vista transmitir la arquitectura del sistema, sin necesidad de recurrir a información adicional. Para este fin, se cuenta con un conjunto predefinido (extensible) de conectores, cada uno con una representación distinta.

Sitio de referencia – http://www.dc.uba.ar/people/profesores/nicok/jacal.htm.

.

Estilos – Jacal no cuenta con una notación particular para expresar estilos, aunque por tratarse de un lenguaje de propósito general, puede ser utilizado para expresar arquitecturas de distintos estilos. No ofrece una forma de restringir una configuración a un estilo específico, ni de validar la conformidad.

Interfaces – Cada componente cuenta con puertos (*ports*) que constituyen su interfaz y a los que pueden adosarse conectores.

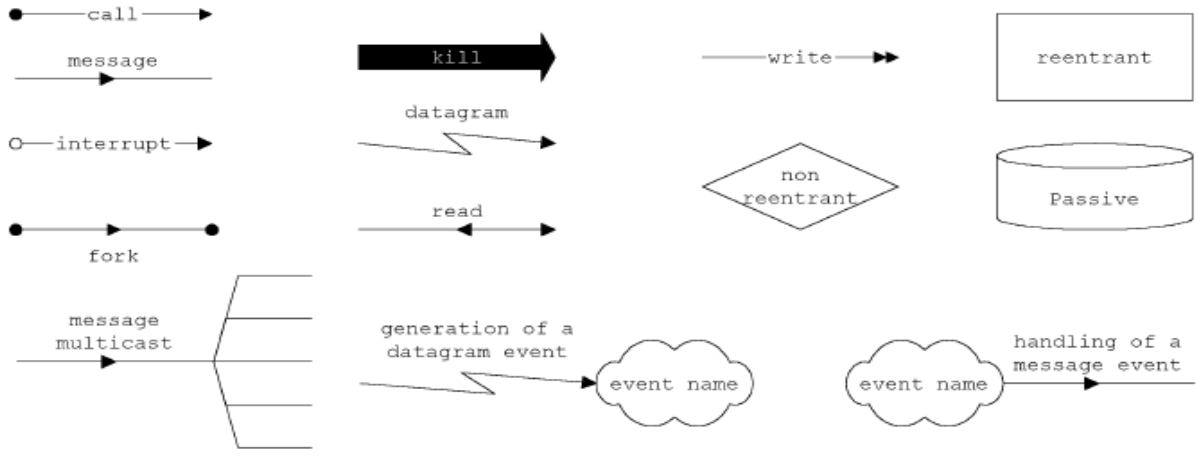
Semántica – Jacal tiene una semántica formal que está dada en función de redes de Petri. Se trata de una semántica denotación que asocia a cada arquitectura una red correspondiente. La semántica operacional estándar de las redes de Petri es la que justifica la animación de las arquitecturas.

Además del nivel de interfaz, que corresponde a la configuración de una arquitectura ya que allí se determina la conectividad entre los distintos componentes, Jacal ofrece un nivel de descripción adicional, llamado nivel de comportamiento. En este nivel, se describe la relación entre las comunicaciones recibidas y enviadas por un componente, usando diagramas de transición de estados con etiquetas en los ejes que corresponden a nombres de puertos por los que se espera o se envía un mensaje.

Análisis y verificación– Las animaciones de arquitecturas funcionan como casos de prueba. La herramienta de edición y animación disponible en el sitio del proyecto permite dibujar arquitecturas mediante un editor orientado a la sintaxis, para luego animarlas y almacenar el resultado de las ejecuciones en archivos de texto. Esta actividad se trata exclusivamente de una tarea de testing, debiendo probarse cada uno de los casos que se consideren críticos, para luego extraer conclusiones del comportamiento observado o de las trazas generadas. Si bien no se ofrecen actualmente herramientas para realizar procesos de verificación automática como modelchecking, la traducción a redes de Petri ofrece la posibilidad de aplicar al resultado otras herramientas disponibles en el mercado.

Interface gráfica – Como ya se ha dicho, la notación principal de Jacal es gráfica y hay una herramienta disponible en línea para editar y animar visualmente las arquitecturas. En el nivel de interfaz, existen símbolos predeterminados para representar cada tipo de componente y cada tipo de conector, como se muestra en la siguiente figura. A continuación, se da una explicación informal de la semántica de cada uno de los tipos de conectores mostrados en la figura.

Figura 4 tipo de conectores



Fuente REYNOSO, Carlos y KICILLOF, Nicolas. Lenguajes de Descripción de Arquitectura (ADL). [en línea]. Buenos Aires, Argentina: UNIVERSIDAD DE BUENOS AIRES, Marzo 2004 - [citado el 15 de junio de 2011] Version 1.0 disponible desde internet en: http://carlosreynoso.com.ar/archivos/arquitectura/ADL.PDF

Call: transfiere el control y espera una respuesta.

Message: coloca un mensaje en una cola y sigue ejecutando.

Interrupt: interrumpe cualquier flujo de control activo en el receptor y espera una respuesta.

Fork: agrega un flujo de control al receptor, el emisor continúa ejecutando.
Kill: detiene todos los flujos de control en el receptor, el emisor continúa ejecutando.

Datagram: si el receptor estaba esperando una comunicación por este puerto, el mensaje es recibido; de lo contrario, el mensaje se pierde; en cualquier caso el emisor sigue ejecutando.

Read: la ejecución en el emisor continúa, de acuerdo con el estado del receptor.

Write: cambia el estado del receptor, el emisor continúa su ejecución.
Generación de código En su versión actual, Jacal no genera código de ningún lenguaje de programación, ya que no fuerza ninguna implementación única para los conectores. Por ejemplo, un conector de tipo message podría implementarse mediante una cola de alguna plataforma de middleware (como MSMQ o QSeries) o directamente como código en algún lenguaje. No obstante, la herramienta de edición de Jacal permite exportar a un archivo de texto la estructura estática de una arquitectura, que luego puede ser convertida a código fuente para usar como base para la programación.

Disponibilidad de plataforma La herramienta que actualmente está disponible para editar y animar arquitecturas en Jacal es una aplicación Win32, que no requiere instalación, basta con copiar el archivo ejecutable para comenzar a usarla. El ambiente consiste en una interfaz gráfica de usuario, donde pueden dibujarse representaciones Jacal de sistemas, incluyendo tanto el nivel de interfaz como el de comportamiento. Se pueden editar múltiples sistemas simultáneamente y, abriendo distintas vistas, visualizar simultáneamente los dos niveles de un mismo sistema, para uno o más componentes.

3.6.9 LILEANNA

Al igual que en el caso de ArTek, en opinión de Medvidovic LILEANNA no es un genuino ADL, por cuanto la configuración se modela implícitamente mediante información de interconexión que se distribuye entre la definición de los componentes individuales y los conectores. En este sentido, aunque pueda no ser

un ADL en sentido estricto, se le reconoce la capacidad de modelar ciertos aspectos de una arquitectura.

LILEANNA es, visto como ADL, estructural y sintácticamente distinto a todos los demás. De hecho, es oficialmente un lenguaje de interconexión de módulos (MIL), basado en expresiones de módulo propias de la programación parametrizada. Un MIL se puede utilizar descriptivamente, para especificar y analizar un diseño determinado, o constructivamente, para generar un nuevo sistema en base a módulos preexistentes, ejecutando el diseño. Típicamente, la programación parametrizada presupone la disponibilidad de dos clases de bibliotecas, que contienen respectivamente expresiones de módulo (que describen sistemas en términos de interconexiones de módulos) y grafos de módulo (que describen módulos y relaciones entre ellos, y que pueden incluir código u otros objetos de software).

La semántica formal de LILEANNA se basa en la teoría de categorías, siguiendo ideas desarrolladas para el lenguaje de especificación Clear; posteriormente se le agregó una semántica basada en teoría de conjuntos. Las propiedades de interconexión de módulos se relacionan bastante directamente con las de los componentes efectivos a través de la semántica de las expresiones de módulo. Aunque una especificación en LILEANNA es varios órdenes de magnitud más verbosa de lo que ahora se estima deseable para visualizar una descripción, incluye un “editor de *layout*” gráfico basado en “una notación como la que usan típicamente los ingenieros, es decir cajas y flechas”. Es significativo que el documento de referencia más importante sobre el proyecto, tan puntilloso respecto de las formas sintácticas y los métodos formales concomitantes a la programación parametrizada, se refiera a su representación visual en estos términos.

3.6.10 MetaH/AADL

Así como LILEANNA es un ADL ligado a desarrollos que guardan relación específica con helicópteros, MetaH modela arquitecturas en los dominios de guía, navegación y control (GN&C) y en el diseño aeronáutico. Aunque en su origen estuvo ligado estrechamente a un dominio, los requerimientos imperantes obligaron a implementar recursos susceptibles de extrapolarse productivamente a la tecnología de ADLs de propósito general. AADL (Avionics Architecture Description Language) está basado en la estructura textual de MetaH. Aunque comparte las mismas siglas, no debe confundirse este AADL con Axiomatic Architecture Description Language, una iniciativa algo más antigua que se refiere al diseño arquitectónico físico de computadoras paralelas.

Sitio de referencia - http://www.htc.honeywell.com/metah/.

Objetivo principal – MetaH ha sido diseñado para garantizar la puesta en marcha, la confiabilidad y la seguridad de los sistemas modelados, y también considera la disponibilidad y las propiedades de los recursos de hardware.

Soporte de lenguajes – MetaH está exclusivamente ligado a desarrollos hechos en Ada en el dominio de referencia.

Disponibilidad de plataforma – Para trabajar con MetaH en ambientes Windows, Honeywell proporciona un MetaH Graphical Editor implementado en DoME, que provee un conjunto extenso de herramientas visuales y de edición de texto.

3.6.11 Rapide

Se puede caracterizar como un lenguaje de descripción de sistemas de propósito general que permite modelar interfaces de componentes y su conducta observable. Sería tanto un ADL como un lenguaje de simulación. La estructura de Rapide es sumamente compleja, y en realidad articula cinco lenguajes: el lenguaje de tipos describe las interfaces de los componentes; el lenguaje de arquitectura describe el flujo de eventos entre componentes; el lenguaje de especificación describe restricciones abstractas para la conducta de los componentes; el lenguaje ejecutable describe módulos ejecutables; y el lenguaje de patrones describe patrones de los eventos. Los diversos sub-lenguajes comparten la misma visibilidad, *scoping* y reglas de denominación, así como un único modelo de ejecución.
Sitio de referencia – Universidad de Stanford - http://pavg.stanford.edu/rapide/.

Objetivo principal – Simulación y determinación de la conformidad de una arquitectura.

Interfaces – En Rapide los puntos de interfaz de los componentes se llaman constituyentes.

Conectores – Siendo lo que Medvidovic (1996) llama un lenguaje de configuración *online*, en Rapide (al igual que en Darwin) no es posible poner nombre, sub-tipear o reutilizar un conector.

Semántica – Mientras muchos lenguajes de tipo ADL no soportan ninguna especificación semántica de sus componentes más allá de la descripción de sus interfaces, Wright y Rapide permiten modelar la conducta de sus componentes. Rapide define tipos de componentes (llamados interfaces) en términos de una colección de eventos de comunicación que pueden ser *observados* (acciones

externas) o *iniciados* (acciones públicas). Las interfaces de Rapide definen el comportamiento computacional de un componente vinculando la observación de acciones externas con la iniciación de acciones públicas.

Análisis y verificación automática – En Rapide, el monitoreo de eventos y las herramientas nativas de filtrado facilitan el análisis de arquitectura. También es posible implementar verificación de consistencia y análisis mediante simulación. En esencia, en Rapide toda la arquitectura es simulada, generando un conjunto de eventos que se supone es compatible con las especificaciones de interfaz, conducta y restricciones. La simulación es entonces útil para detectar alternativas de ejecución. En general, la arquitectura de software mantiene una actitud de reserva crítica frente a la simulación. Escribe Paul Clements: “La simulación es inherentemente una herramienta de validación débil en la medida que sólo presenta una sola ejecución del sistema cada vez; igual que el *testing*, sólo puede mostrar la presencia antes que la ausencia de fallas. Más poderosos son los verificadores o probadores de teoremas que son capaces de comparar una aserción de seguridad contra todas las posibles ejecuciones de un programa simultáneamente”.

Interfaz gráfica – Rapide soporta notación gráfica.

Soporte de lenguajes – Rapide soporta construcción de sistemas ejecutables especificados en VHDL, C. C++, Ada y Rapide mismo.

Generación de código – Rapide puede generar código C, C++ y Ada.

Observaciones – En materia de evolución y soporte de sub-tipos, Rapide soporta herencia, análoga a la de los lenguajes OOP.

Implementación de referencia – Aunque se ha señalado su falta de características de escalabilidad, Rapide se ha utilizado en diversos proyectos de gran escala. Un ejemplo representativo es el estándar de industria X/Open Distributed Transaction Processing.

Disponibilidad de plataforma – Rapide ha desarrollado un conjunto de herramientas que sólo se encontraba disponible para Solaris 2.5, SunOS 4.1.3. Y Linux. Este *toolset* no ha evolucionado desde 1997, y tampoco ha avanzado más allá de la fase de prototipo.

3.6.12 UML - De OMT al Modelado OO

UML forma parte del repertorio conocido como lenguajes semi-formales de modelado. Esta variedad de herramientas se remonta a una larga tradición que arrancó a mediados de la década de 1970 con PSL/PSA, SADT y el análisis

estructurado. Alrededor de 1990 aparecieron los primeros lenguajes de especificación orientados a objeto propuestos por Grady Booch, Peter Coad, Edward Yourdon y James Rumbaugh. A instancias de Rumbaugh, Booch e Ivar Jacobson, finalmente, estos lenguajes se orientaron hacia lo que es hoy UML (Unified Modeling Language), que superaba la incapacidad de los primeros lenguajes de especificación OO para modelar aspectos dinámicos y de comportamiento de un sistema introduciendo la noción de casos de uso. En términos de número, la masa crítica de los conocedores de UML no admite comparación con la todavía modesta población de especialistas en ADLs. En la comunidad de arquitectos existen dos facciones claramente definidas; la primera, vinculada con el mundo de Rational y UML, impulsa el uso casi irrestricto de UML como si fuera un ADL normal; la segunda ha señalado reiteradas veces las limitaciones de UML no sólo como ADL sino como lenguaje universal de modelado. La literatura crítica de UML es ya un tópico clásico de la reciente arquitectura de software.

Si bien se reconoce que con UML es posible representar virtualmente cualquier cosa, incluso fenómenos y procesos que no son software, se debe admitir que muchas veces no existen formas *estándar* de materializar esas representaciones, de modo que no pueden intercambiarse modelos entre diversas herramientas y contextos sin pérdida de información. Se ha dicho que ni las clases, ni los componentes, ni los *packages*, ni los subsistemas de UML son unidades arquitectónicas adecuadas: las clases están tecnológicamente sesgadas hacia la OO y representan entidades de granularidad demasiado pequeña; los componentes “representan piezas físicas de implementación de un sistema” y por ser unidades de implementación y no de diseño, es evidente que existen en el nivel indebido de abstracción; un *package* carece de la estructura interna necesaria; los subsistemas pueden no aparecer en los diagramas de despliegue, o mezclarse con otras entidades de diferente granularidad y propósito, carecen del concepto de puerto y su semántica y pragmática se han estimado caóticas.

3.6.13 UniCon

UniCon (Universal Connector Support) es un ADL desarrollado por Mary Shaw y otros. Proporciona una herramienta de diseño para construir configuraciones ejecutables basadas en tipos de componentes, implementaciones y “conexiones expertas” que soportan tipos particulares de conectores. UniCon se asemeja a Darwin en la medida en que proporciona herramientas para desarrollar configuraciones ejecutables de caja negra y posee un número fijo de tipos de interacción, pero el modelo de conectores de ambos ADLs es distinto.

Oficialmente se define como un ADL cuyo foco apunta a soportar la variedad de partes y estilos que se encuentra en la vida real y en la construcción de sistemas a partir de sus descripciones arquitectónicas. UniCon es el ADL propio del proyecto Vitruvius, cuyo objetivo es elucidar un nivel de abstracción de modo tal que se pueda capturar, organizar y tornar disponible la experiencia colectiva exitosa de los arquitectos de software.

Objetivo principal El propósito de UniCon es generar código ejecutable a partir de una descripción, a partir de componentes primitivos adecuados. UniCon se destaca por su capacidad de manejo de métodos de análisis de tiempo real a través de RMA (Rate Monotonic Analysis).

Sitio de referencia - http://www-2.cs.cmu.edu/People/UniCon/ - El link de distribución de UniCon en la Universidad Carnegie Mellon estaba muerto o inaccesible en el momento de redacción de este documento (enero de 2004). El sitio de Vitrivius fue modificado por última vez en diciembre de 1998.

Estilos - UniCon no proporciona medios para describir o delinear familias de sistemas o estilos.

Interfaces – En UniCon los puntos de interfaces de los componentes se llaman *players*. Estos players poseen un tipo que indica la naturaleza de la interacción esperada, y un conjunto de propiedades que detalla la interacción del componente en relación con esa interfaz. En el momento de configuración, los *players* de los componentes se asocian con los roles de los conectores.

Semántica – UniCon sólo soporta cierta clase de información semántica en listas de propiedades.

Interfaz gráfica – UniCon soporta notación gráfica.
Generación de código – UniCon genera código C mediante el procedimiento de asociar elementos arquitectónicos a construcciones de implementación, que en este caso serían archivos que contienen código fuente. Sin embargo, habría algunos problemas con esta forma de implementación, dado que presuponer que esa asociación vaya a ser siempre uno-a-uno puede resultar poco razonable.
Observaciones – UniCon (igual que Darwin) carece de la capacidad de definir nuevos tipos, soportando sólo una amplia variedad de tipos predefinidos. Los tipos de componente son definidos por enumeración, no siendo posible definir sub-tipos y careciendo por lo tanto de capacidad de evolución.

Disponibilidad de plataforma – La distribución de UniCon no se encuentra actualmente activa.

3.6.14 Weaves

Propuesto alrededor de 1991, Weaves soporta la especificación de arquitecturas de flujo de datos. En particular, se especializa en el procesamiento en tiempo real de grandes volúmenes de datos emitidos por satélites meteorológicos. Es un ADL acaso demasiado ligado a un dominio específico, sobre el cual no hay abundancia de documentación disponible.

3.6.15 Wright

Se puede caracterizar sucintamente como una herramienta de formalización de conexiones arquitectónicas. Ha sido desarrollado por la Escuela de Ciencias Informáticas de la Universidad Carnegie Mellon, como parte del proyecto mayor ABLE. Este proyecto a su vez se articula en dos iniciativas: la producción de una herramienta de diseño, que ha sido Aesop, y una especificación formal de descripción de arquitecturas, que es propiamente Wright.

Objetivo principal – Wright es probablemente la herramienta más acorde con criterios académicos de métodos formales. Su objetivo declarado es la integración de una metodología formal con una descripción arquitectónica y la aplicación de procesos formales tales como álgebras de proceso y refinamiento de procesos a una verificación automatizada de las propiedades de las arquitecturas de software.

Sitio de referencia – La página de ABLE de la Universidad Carnegie Mellon se encuentra en http://www-2.cs.cmu.edu/afs/cs/project/able/www/able.html. La del proyecto Wright está en http://www.cs.cmu.edu/afs/cs/project/able/www/wright/index.html.

Estilos - En Wright se introduce un vocabulario común declarando un conjunto de tipos de componentes y conectores y un conjunto de restricciones. Cada una de las restricciones declaradas por un estilo representa un predicado que debe ser satisfecho por cualquier configuración de la que se declare que es miembro del estilo. La notación para las restricciones se basa en el cálculo de predicados de primer orden. Las restricciones se pueden referir a los conjuntos de componentes, conectores y *attachments*, a las puertos y a las computaciones de un componente específico y a los roles y ligamentos (*glue*) de un conector particular. Verificar la conformidad de una configuración a un estilo canónico estándar.

Interfaces - En Wright (igual que en Acme y Aesop) los puntos de interfaz se llaman puertos (*ports*).

Semántica – Mientras muchos lenguajes de tipo ADL no soportan ninguna especificación semántica de sus componentes más allá de la descripción de sus interfaces, Wright y Rapide permiten modelar la conducta de sus componentes. En Wright existe una sección especial de la especificación llamada Computación que describe la funcionalidad de los componentes.

Análisis y verificación automática – Al basarse en CSP como modelo semántico, Wright permite analizar los conectores para verificar que no haya *deadlocks*. Wright define verificaciones de consistencia que se aplican estáticamente, y no mediante simulación. Esto lo hace más confiable que, por ejemplo, Rapide. También se puede especificar una forma de compatibilidad entre un puerto y un rol, de modo que se pueda cambiar el proceso de un puerto por el proceso de un rol sin que la interacción del conector detecte que algo ha sido modificado. Esto permite implementar relaciones de refinamiento entre procesos.

Interfaz gráfica – Wright no proporciona notación gráfica en su implementación nativa.

Generación de código – Wright se considera como una notación de modelado autocontenido y no genera (al menos en forma directa) ninguna clase de código ejecutable.

Implementación de referencia – Aunque se ha señalado su falta de características explícitas de escalabilidad, Wright se utilizó para modelar y analizar la estructura de *runtime* del Departamento de Defensa de Estados Unidos. Se asegura que permitió condensar la especificación original y detectar varias inconsistencias en ella.

Disponibilidad de plataforma – Wright es en principio independiente de plataforma y es un lenguaje formal, antes que una herramienta empaquetada. Debido a que se basa en CSP (Communicating Sequential Process), cualquier solución que pueda tratar código CSP en plataforma Windows es apta para obtener ya sea una visualización del modelo o verificar la ausencia de deadlocks o la consistencia del modelo. El código susceptible de ser manejado por herramientas de CSP académicas o comerciales requiere tratamiento previo por un módulo de Wright que por el momento existe sólo para Linux o SunOS; pero una vez que se ha generado el CSP (lo cual puede hacerse manualmente insertando unos pocos *tags*) se puede tratar en Windows con cualquier solución ligada a ese lenguaje, ya sea FDR o alguna otra.

3.7 Modelos computacionales y paradigmas de modelado

La descripción que se ha proporcionado hasta aquí no constituye un *scorecard* ni una evaluación sistemática de los diferentes ADLs, sino una revisión de los principales lenguajes descriptivos vigentes para que el lector arquitecto se arme de una idea más precisa de las opciones hoy en día disponibles si lo que se requiere una herramienta de descripción de arquitecturas.

Los arquitectos de software con mayor inclinación hacia las cuestiones teóricas y los métodos formales habrán podido apreciar la gran variedad de fundamentaciones semánticas y modelos computacionales que caracteriza al repertorio de los ADLs. La estructura de Darwin se basa en el cálculo _; Wright es prácticamente una aplicación rigurosa del CSP de Hoare para su semántica y de lógica de primer orden para sus restricciones; LILEANNA se respalda en programación parametrizada e hiper-programación, así como en teoría categorial; los ADLs basados en eventos se fundan en Conjuntos de Eventos Parcialmente Ordenados (*Posets*). Casi todos los ADLs se acompañan de una gramática BNF explícita y algunos (como Armani) requieren también familiaridad con la construcción de un Arbol Sintáctico Abstracto (AST). Al menos un ADL contemporáneo, SAM, implementa redes de Petri de transición de predicados y lógica temporal de primer orden y tiempo lineal.

Un concepto importante que surge de haber considerado este abanico de posibilidades, es que no todo lo que tiene que ver con diseño requiere compulsivamente notaciones y modelos de objeto. Considerando los ADLs en su conjunto, habrá podido comprobarse que la tradición de modelado OO, como la que se plasma a partir de[19], juega en este campo un papel muy modesto, si es que juega alguno; la reciente tendencia a redefinir los ADLs en términos de mensajes, servicios, integración, XML Schemas, xADL, SOAP y sus derivaciones no hará más que consolidar ese estado de cosas.

3.8 ADLs en ambientes Windows

En este estudio se ha podido comprobar que existe un amplio repertorio de implementaciones de ADLs y sus herramientas colaterales de última generación en plataforma Windows. Específicamente se puede señalar a Saage, un entorno para diseño arquitectónico que utiliza el ADL de C2SADEL y que requiere Visual J++ (o Visual J#), COM y eventualmente Rational Rose (o DRADEL); también existen extensiones de Visio para C2, xADL y naturalmente UML. Visio para xADL se utiliza como la interfaz usuario de preferencia para *data binding* de las bibliotecas de ArchEdit para xADL. En relación con xADL también se puede utilizar

[19] RUMBAUGH, James, *et al*. *Object-oriented modeling and design.* Englewood Cliffs: Prentice Hall, 1991.

en Windows ArchStudio para modelado en gran escala, o Ménage para mantenimiento y gestión de arquitecturas cambiantes en tiempo de ejecución.

Si el lenguaje de elección es Darwin se puede utilizar SAA. Si en cambio es MetaH, se dispone del MetaH Graphical Editor de Honeywell, implementado sobre DoME y nativo de Windows. En lo que se refiere a Acme, el proyecto Isis y Vanderbilt proporcionan GME (Metaprogrammable Graphical Model Editor), un ambiente de múltiples vistas que, al ser meta-programable, se puede configurar para cubrir una rica variedad de formalismos visuales de diferentes dominios. Una segunda opción para Acme podría ser VisEd, un visualizador arquitectónico y editor de propiedades de GA Tech; una tercera, Acme Studio; una cuarta, Acme Powerpoint Editor de ISI, la cual implementa COM. Los diseños de Acme y Rapide se pueden poner a prueba con Aladdin, un analizador de dependencia del Departamento de Ciencia de la Computación de la Universidad de Colorado. Cualquier ADL que se base en alguna extensión de *scripting* Tcl/Tk se habrán de implementar con facilidad en Windows. El modelado y el análisis de una arquitectura con Jacal es nativa de Windows y las próximas versiones del ADL utilizarán Visio como front-end.

Algunas herramientas son nativas de Win32, otras corren en CygWin, Interix o algún Xserver, otras necesitan JRE. Mientras los ADLs de elección estén vivos y mantengan su impulso no faltarán recursos y complementos, a menos que se escoja una plataforma muy alejada de la corriente principal para ejecutar el software de modelado. No necesariamente habrá que lidiar entonces manualmente con notaciones textuales que quizá resulten tan complejas como las de los propios lenguajes de implementación.

3.9 ADLs y Patrones

Según Microsoft, a primera vista, daría la impresión que la estrategia arquitectónica de Microsoft reposa más en la idea de patrones y prácticas que en lenguajes de descripción de diseño; pero si se examina con más cuidado el concepto de patrones se verá que dichos lenguajes engranan clara y armónicamente en un mismo contexto global. El concepto de patrón fue acuñado por Christopher Alexander entre 1977 y 1979. De allí en más fue enriqueciéndose en una multitud de sentidos, abarcando un conjunto de prácticas para capturar y comunicar la experiencia de los diseñadores expertos, documentar *best practices*, establecer formas recurrentes de solución de problemas comunes, etcétera. Las clasificaciones usuales en el campo de los patrones reconocen habitualmente que hay diferentes clases de ellos; por lo común se hace referencia a patrones de análisis, organizacionales, procedural y de diseño. Esta última categoría tiene que ver con recurrencias específicas de diseño de software y sistemas, formas relativamente fijas de interacción entre componentes, y técnicas relacionadas con

familias y estilos. En este sentido, es evidente que los ADLs denotan una clara relación con los patrones de diseño, de los que puede decirse que son una de las formas posibles de notación. Como en el caso de las heurísticas y recetas de Acme, muchas veces esos patrones son explícitos y el ADL opera como una forma regular para expresarlos en la descripción arquitectónica.[20]

[20] REYNOSO, Carlos y KICILLOF, Nicolas. Lenguajes de Descripción de Arquitectura (ADL). [en línea]. Buenos Aires, Argentina: UNIVERSIDAD DE BUENOS AIRES, Marzo 2004 - [citado el 15 de junio de 2011] Version 1.0 disponible desde internet en: <http://carlosreynoso.com.ar/archivos/arquitectura/ADL.PDF>

4. ESTILOS ARQUITECTURALES

La disciplina de arquitectura de software ha madurado sustancialmente durante la última década: hoy encontramos un aumento en el uso de estándares, métodos de desarrollo basados en arquitectura y libros para el diseño arquitectural y documentación. Y, como indicador significativo de madurez de la ingeniería, también estamos viendo una investigación en aumento sobre maneras de analizar formalmente propiedades de arquitecturas, tales como compatibilidad de componentes, desempeño, conformación de estilo y muchos otros.

Uno de los pilares importantes de la arquitectura de software moderna es el uso de estilos arquitecturales. Un estilo arquitectural define una familia de sistemas relacionados, típicamente porque proporcionan un vocabulario específico del dominio de diseño arquitectónico junto con las limitaciones sobre cómo las partes pueden encajar.

El uso de estilos como un vehículo para la caracterización de una familia de arquitecturas de software está motivado por una serie de beneficios. Los estilos proporcionan un vocabulario común para los arquitectos, permitiendo a los desarrolladores entender más fácilmente un diseño arquitectural rutinario.

Algunos beneficios de los estilos arquitecturales:[21]

- Los estilos promueven la reutilización de diseños, dado que el mismo diseño arquitectónico es utilizado por un conjunto de sistemas relacionados.
- Los estilos pueden llevar a una reutilización significante de código. Por ejemplo muchos estilos proveen middleware pre empacado para soportar implementaciones conectoras. Los sistemas basados en Unix que adopten un estilo de tubo-filtro pueden reutilizar primitivas del sistema operativo para implementar programación de tareas, sincronización, y comunicación a través de tuberías en buffer.
- Es más fácil para otros entender la organización de un sistema si se está utilizando estructuras estándar de arquitectura.
- Los estilos soportan interoperabilidad de los componentes.
- Al restringir el espacio de diseño, un estilo arquitectónico a menudo permite análisis especializados. Por ejemplo, es posible analizar ciertos sistemas construidos en un estilo tubo-filtro en cuanto a su programación, volumen de trabajo y latencia.

[21] GARLAN, David y SOO KIM, Jung. *Analyzing architectural styles.* Carnegie Mellon University, 2007.

4.1 DEFINICIONES DE ESTILO

Cuando se habla de una arquitectura en tres capas, o una arquitectura cliente-servidor, u orientada a servicios, implícitamente se está haciendo referencia a un campo de posibilidades articulatorias, a una especie de taxonomía de las configuraciones posibles, en cuyo contexto la caracterización tipológica particular adquiere un significado distintivo. No tiene caso, a fin de cuentas, hablar de tipos arquitectónicos si no se clarifica cuál es la tipología total en la que cada uno de ellos engrana. Definir una arquitectura como (por ejemplo) orientada a servicios ciertamente la tipifica, la distingue, la singulariza; pero ¿cuáles son las otras arquitecturas alternativas? ¿Es esa arquitectura específica una clase abarcadora, o es apenas una variante que está incluida en otros conjuntos clases más amplios? ¿En qué orden de magnitud se encuentra el número de las opciones con las cuales contrasta? O bien, ¿cuántas otras formas podrían adoptarse? ¿Ocho? ¿Cien? ¿A qué otras se asemeja más la estructura escogida? ¿Cuál es su modelo peculiar de tradeoff? Desde Ferdinand de Saussure, la semántica lingüística sabe que un determinado signo (“arquitectura orientada a servicios”, en este caso) adquiere un valor de significación cabal tanto por ser lo que es, como por el hecho de ser lo que otros signos no son.

Optar por una forma arquitectónica no solamente tiene valor distintivo, sino que define una situación pragmática. Una vez que los tipos adquieren una dimensión semántica precisa y diferencial, se verá que a su significado se asocian conceptos, herramientas, problemas, experiencias y antecedentes específicos. Después de recibir nombres variados tales como “clases de arquitectura”, “tipos arquitectónicos”, “arquetipos recurrentes”, “especies”, “paradigmas topológicos”, “marcos comunes” y varias docenas más, desde hace más de una década esas cualificaciones arquitectónicas se vienen denominando “estilos”, o alternativamente “patrones de arquitectura”. Llamarlas estilos subraya explícitamente la existencia de una taxonomía de las alternativas estilísticas; llamarlas patrones, por el contrario, establece su vinculación con otros patrones posibles pero de distinta clase: de diseño, de organización o proceso, de código, de interfaz de usuario, de prueba, de análisis.

Los estilos sólo se manifiestan en arquitectura teórica descriptiva de alto nivel de abstracción; los patrones, por todas partes. Los partidarios de los estilos se definen desde el inicio como arquitectos; los que se agrupan en torno de los patrones se confunden a veces con ingenieros y diseñadores, cuando no con programadores con conciencia sistemática o lo que alguna vez se llamó analistas de software. El primer grupo ha abundado en taxonomías internas de los estilos y en reflexión teórica; el segundo se ha mantenido, en general, refractario al impulso taxonómico, llevado por una actitud resueltamente empírica. Ambos, con mayor o menor plenitud y autoconciencia, participan del campo global de la arquitectura de

software. Los estilos se encuentran en el centro de la arquitectura y constituyen buena parte de su sustancia. Los patrones de arquitectura están claramente dentro de la disciplina arquitectónica, solapándose con los estilos, mientras que los patrones de diseño se encuentran más bien en la periferia, si es que no decididamente afuera.

En lo que sigue, y ateniéndonos a la recomendación IEEE-1471, arquitectura de software se entenderá como "la organización fundamental de un sistema encarnada en sus componentes, las relaciones de los componentes con cada uno de los otros y con el entorno, y los principios que orientan su diseño y evolución". Debe quedar claro que en esta definición el concepto de "componente" es genérico e informal, y no se refiere sólo a lo que técnicamente se concibe como tal en modelos de componentes como CORBA Component Model, J2EE (JavaBeans o EJB), ActiveX, COM, COM+ o .NET.

Según esa definición, estilos y arquitectura nacen en el mismo momento. Con una sola excepción (documentada en el párrafo siguiente) no he podido encontrar referencias a la palabra 'estilo' anteriores a 1992. Todavía en julio de ese año Robert Allen y David Garlan[22] de la Universidad de Carnegie Mellon se refieren a "paradigmas de arquitectura" y "estructuras de sistemas", mencionando entre ellos lo que luego sería el familiar estilo tubería-filtros, los modelos de pizarra y los de flujo de datos. Con nombres idénticos, esos paradigmas pasarían a llamarse estilos un mes más tarde en todos los textos de la misma escuela primero y en toda la arquitectura de software después.

Un año antes, sin embargo, aparece una mención casi accidental a los estilos de arquitectura en una escuela que no volvería a prestar atención al concepto casi hasta el siglo siguiente. En 1991 los promotores de OMT del Centro de Investigación y Desarrollo de General Electric en Shenectady liderados por James Rumbaugh habían hablado de la existencia de "estilos arquitectónicos"[23], "arquitecturas comunes", "marcos de referencia arquitectónicos prototípicos" o "formas comunes" que se aplican en la fase de diseño; aunque el espíritu es el mismo que el que animó la idea de los estilos, el inventario difiere. Los autores, en efecto, mencionan "clases de sistemas" que incluyen (1) transformaciones en lote; (2) transformaciones continuas; (3) interfaz interactiva, (4) simulación dinámica de objetos del mundo real, (5) sistemas de tiempo real, (6) administrador de transacciones con almacenamiento y actualización de datos.

Algunas de estas clases, llamadas las cinco veces que se menciona su conjunto con cinco denominaciones diferentes, se pueden reconocer con los nombres

[22] ALLEN, Robert y GARLAN, David. *A formal approach to software architectures.* En: *Proceedings of the IFIP Congress '92*, Setiembre de 1992.

[23] RUMBAUGH, James, *et al*. *Object-oriented modeling and design.* Englewood Cliffs: Prentice Hall, 1991. p. 199.

siempre cambiados en los catálogos ulteriores de estilos de la arquitectura estructuralista[24]. El equipo de Rumbaugh no volvió a mencionar la idea de estilos arquitectónicos (ni la de arquitectura) fuera de esas páginas referidas, consumando el divorcio implícito entre lo que podría llamarse la escuela de diseño orientada a objetos y la escuela de arquitectura estructuralista, mayormente ecléctica.

Las primeras definiciones explícitas y autoconscientes de estilo parecen haber sido propuestas por Dewayne Perry de AT&T Bell Laboratories de New Jersey y Alexander Wolf de la Universidad de Colorado . En octubre de 1992, estos autores profetizan que la década de 1990 habría de ser la de la arquitectura de software, y definen esa arquitectura en contraste con la idea de diseño. Leído hoy, y aunque el acontecimiento es más reciente de lo que parece, su llamamiento resulta profético y fundacional hasta lo inverosímil; en él, la arquitectura y los estilos se inauguran simbólicamente en una misma frase definitoria:

La década de 1990, creemos, será la década de la arquitectura de software. Usamos el término "arquitectura", en contraste con "diseño", para evocar nociones de codificación, de abstracción, de estándares, de entrenamiento formal (de arquitectos de software) y de estilo.

Advierten, además, que curiosamente no existen arquitecturas que tengan un nombre, como no sea en relación con estructuras de hardware. Por analogía con la arquitectura de edificios, establecen que una arquitectura se define mediante este modelo: Arquitectura de Software = {Elementos, Forma, Razón}

Algunos años más tarde Mary Shaw y Paul Clements[25] identifican los estilos arquitectónicos como un conjunto de reglas de diseño que identifica las clases de componentes y conectores que se pueden utilizar para componer en sistema o subsistema, junto con las restricciones locales o globales de la forma en que la composición se lleva a cabo.

Los componentes, incluyendo los subsistemas encapsulados, se pueden distinguir por la naturaleza de su computación: por ejemplo, si retienen estado entre una invocación y otra, y de ser así, si ese estado es público para otros componentes. Los tipos de componentes también se pueden distinguir conforme a su forma de empaquetado, o dicho de otro modo, de acuerdo con las formas en que interactúan con otros componentes. El empaquetado es usualmente implícito, lo

[24] GARLAN, David y SHAW, Mary. *An introduction to software architecture. CMU Software Engineering Institute Technical Report, CMU/SEI-94-TR-21, ESC-TR-94-21,* 1994.

[25] SHAW, Mary y CLEMENTS, Paul. *A field guide to Boxology: Preliminary classification of architectural styles forsoftware systems*. En: *Proceedings of the 21st International Computer Software and Applications Conference 1997*.

que tiende a ocultar importantes propiedades de los componentes. Para clarificar las abstracciones se aísla la definición de esas interacciones bajo la forma de conectores: por ejemplo, los procesos interactúan por medio de protocolos de transferencia de mensajes, o por flujo de datos a través de tuberías (pipes). Es en gran medida la interacción entre los componentes, mediados por conectores, lo que confiere a los distintos estilos sus características distintivas. Nótese que en esta definición de estilo, que se quiere mantener abstracta, entre las entidades de primera clase no aparece prácticamente nada que se refiera a datos o modelo de datos.

En un ensayo de 1996 en el que aportan fundamentos para una caracterización formal de las conexiones arquitectónicas, Robert Allen y David Garlan asimilan los estilos arquitectónicos a descripciones informales de arquitectura basadas en una colección de componentes computacionales, junto a una colección de conectores que describen las interacciones entre los componentes. Consideran que esta es una descripción deliberadamente abstracta, que ignora aspectos importantes de una estructura arquitectónica, tales como una descomposición jerárquica, la asignación de computación a los procesadores, la coordinación global y el plan de tareas. En esta concepción, los estilos califican como una macro-arquitectura, en tanto que los patrones de diseño (como por ejemplo el MVC) serían más bien micro arquitecturas.

En 1999 Mark Klein y Rick Kazman proponen una definición según la cual un estilo arquitectónico es una descripción del patrón de los datos y la interacción de control entre los componentes, ligada a una descripción informal de los beneficios e inconvenientes aparejados por el uso del estilo. Los estilos arquitectónicos, afirman, son artefactos de ingeniería importantes porque definen clases de diseño junto con las propiedades conocidas asociadas a ellos. Ofrecen evidencia basada en la experiencia sobre la forma en que se ha utilizado históricamente cada clase, junto con razonamiento cualitativo para explicar por qué cada clase tiene esas propiedades específicas[26]. Los datos, relegados u omitidos por Perry y Wolf, aparecen nuevamente, dando la impresión que el campo general de los estilistas se divide entre una minoría que los considera esenciales y una mayoría que no.

4.2 CATALOGOS DE ESTILOS

Aquí se considerarán solamente las enumeraciones primarias de la literatura temprana, lo que podríamos llamar taxonomías de primer orden. Una vez que se haya descrito los estilos individuales en la sección siguiente, se dedicará otro apartado para examinar de qué forma determinadas percepciones ligadas a

[26] KLEIN, Mark y KAZMAN, Rick. *Attribute-based architectural styles*. En: *Technical Report, CMU/SEI-99-TR-022, ESCTR-99-022.* Carnegie Mellon University, Octubre de 1999.

dominios derivaron en otras articulaciones como las que han propuesto Roy Fielding[27] y Gregory Andrews[28].

¿Cuántos y cuáles son los estilos, entonces? En un estudio comparativo de los estilos, Mary Shaw considera los siguientes, mezclando referencias a las mismas entidades a veces en términos de "arquitecturas", otras invocando "modelos de diseño":

1. Arquitecturas orientadas a objeto
2. Arquitecturas basadas en estados
3. Arquitecturas de flujo de datos: Arquitecturas de control de realimentación
4. Arquitecturas de tiempo real
5. Modelo de diseño de descomposición funcional
6. Modelo de diseño orientado por eventos
7. Modelo de diseño de control de procesos
8. Modelo de diseño de tabla de decisión
9. Modelo de diseño de estructura de datos

El mismo año, Mary Shaw, junto con David Garlan[29], propone una taxonomía diferente, en la que se entremezclan lo que antes llamaba "arquitecturas" con los "modelos de diseño":

1. Tubería-filtros
2. Organización de abstracción de datos y orientación a objetos
3. Invocación implícita, basada en eventos
4. Sistemas en capas
5. Repositorios
6. Intérpretes orientados por tablas
7. Procesos distribuidos, ya sea en función de la topología (anillo, estrella, etc) o de los protocolos entre procesos (p. ej. algoritmo de pulsación o heartbeat). Una forma particular de proceso distribuido es, por ejemplo, la arquitectura cliente-servidor.
8. Organizaciones programa principal / subrutina.

---

[27] FIELDING, Roy Thomas. *Architectural styles and the design of network-based software architectures*. Tesis doctoral, University of California, Irvine, 2000.

[28] ANDREWS, Gregory R. *Paradigms for Process Interaction in Distributed Programs.* En: *ACM Computing Surveys*. vol.23, Issue 1. Marzo de 1991. p. 49-90.

[29] GARLAN, David y SHAW, Mary. *An introduction to software architecture. CMU Software Engineering Institute Technical Report, CMU/SEI-94-TR-21, ESC-TR-94-21,* 1994.

9. Arquitecturas de software específicas de dominio
10. Sistemas de transición de estado
11. Sistemas de procesos de control
12. Estilos heterogéneos

De particular interés es el catálogo de "patrones arquitectónicos", que es como el influyente grupo de Buschmann denomina a entidades que, con un empaquetado un poco distinto, no son otra cosa que los estilos. Efectivamente, esos patrones "expresan esquemas de organización estructural fundamentales para los sistemas de software. Proporcionan un conjunto de subsistemas predefinidos, especifican sus responsabilidades e incluyen guías y lineamientos para organizar las relaciones entre ellos". En la hoy familiar clasificación de POSA[30] Buschmann, Meunier, Rohnert, Sommerlad y Stal enumeran estos patrones:

1. Del fango a la estructura
    - Capas
    - Tubería-filtros
    - Pizarra
2. Sistemas distribuidos
    - Broker (p. ej. CORBA, DCOM, la World Wide Web)
3. Sistemas interactivos
    - Model-View-Controller
    - Presentación-Abstraction-Control
4. Sistemas adaptables
    - Reflection (meta nivel que hace al software consciente de sí mismo)
    - Microkernel (núcleo de funcionalidad mínima)
    -

En Software Architecture in Practice, un texto fundamental de Bass, Clements y Kazman[31] se proporciona una sistematización de clases de estilo en cinco grupos:

1. Flujo de datos (movimiento de datos, sin control del receptor de lo que viene "corriente arriba")
    - Proceso secuencial por lotes
    - Red de flujo de datos
    - Tubería-filtros

[30] BUSCHMANN, Frank, MEUNIER, Regine, ROHNERT, Hans, SOMMERLAD, Peter y STAL, Michael. *Pattern oriented software architecture.* En: *A system of patterns.* John Wiley & Sons, 1996.
[31] BASS Len, CLEMENTS, Paul y KAZMAN, Rick. *Software Architecture in Practice. Reading, Addison-Wesley*, 1998.

2. Llamado y retorno (estilo dominado por orden de computación, usualmente con un solo thread de control)
    - Programa principal / Subrutinas
    - Tipos de dato abstracto
    - Objetos
    - Cliente-servidor basado en llamadas
    - Sistemas en capas

3. Componentes independientes (dominado por patrones de comunicación entre procesos independientes, casi siempre concurrentes).
    - Sistemas orientados por eventos
    - Procesos de comunicación

4. Centrados en datos (dominado por un almacenamiento central complejo, manipulado por computaciones independientes)
    - Repositorio
    - Pizarra
5. Máquina virtual (caracterizado por la traducción de una instrucción en alguna otra)
    - Intérprete

Considerando solamente los estilos que contemplan alguna forma de distribución o topología de red, Roy Fielding[32] establece la siguiente taxonomía:

1. Estilos de flujo de datos
1.1. Tubería-filtros
1.2. Tubería-filtros uniforme
2. Estilos de replicación
2.1. Repositorio replicado
2.2. Cache
3. Estilos jerárquicos
3.1. Cliente-servidor
3.2. Sistemas en capas y Cliente-servidor en capas
3.3. Cliente-Servidor sin estado
3.4. Cliente-servidor con cache en cliente
3.5. Cliente-servidor con cache en cliente y servidor sin estado
3.6. Sesión remota
3.7. Acceso a datos remoto
4. Estilos de código móvil
4.1. Máquina virtual

[32] FIELDING, Roy Thomas. *Architectural styles and the design of network-based software architectures*. Tesis doctoral, University of California, Irvine, 2000.

4.2. Evaluación remota
4.3. Código a demanda
4.4. Código a demanda en capas
4.5. Agente móvil
5. Estilos peer-to-peer
5.1. Integración basada en eventos
5.2. C2
5.3. Objetos distribuidos
5.4. Objetos distribuidos brokered
6. Transferencia de estado representacional (REST)

En 2001, David Garlan[33], uno de los fundadores del campo, proporciona una lista más articulada:

1. Flujo de datos
    - Secuencial en lotes
    - Red de flujo de datos (tuberías y filtros)
    - Bucle de control cerrado

2. Llamada y Retorno
    - Programa principal / subrutinas
    - Ocultamiento de información (ADT, objeto, cliente/servidor elemental)

3. Procesos interactivos
    - Procesos comunicantes
    - Sistemas de eventos (invocación implícita, eventos puros)

4. Repositorio Orientado a Datos
    - Bases de datos transaccionales (cliente/servidor genuino)
    - Pizarra
    - Compilador moderno
5. Datos Compartidos
    - Documentos compuestos
    - Hipertexto
    - Fortran COMMON
    - Procesos LW

6. Jerárquicos
    - En capas (intérpretes)

[33] GARLAN, David. *Next generation software architectures: Recent research and future directions.* Presentación, Columbia University, Enero de 2001.

A continuación presentamos los principales estilos arquitecturales con su descripción, características, principios, beneficios y los factores que pueden determinar su uso.[34]

4.3 Estilo Arquitectural Cliente/Servidor

Descripción
El estilo cliente/servidor define una relación entre dos aplicaciones en las cuales una de ellas (cliente) envía peticiones a la otra (servidor fuente de datos).

Características
- Es un estilo para sistemas distribuidos.
- Divide el sistema en una aplicación cliente, una aplicación servidora y una red que las conecta.
- Describe una relación entre el cliente y el servidor en la cual el primero realiza peticiones y el segundo envía respuestas.
- Puede usar un amplio rango de protocolos y formatos de datos para comunicar la información.

Principios Clave
- El cliente realiza una o más peticiones, espera por las respuestas y las procesa a su llegada.
- El cliente normalmente se conecta solo a uno o a un número reducido de servidores al mismo tiempo.
- El cliente interactúa directamente con el usuario, por ejemplo a través de una interfaz gráfica.
- El servidor no realiza ninguna petición al cliente.
- El servidor envía los datos en respuesta a las peticiones realizadas por los clientes conectados.
- El servidor normalmente autentifica y verifica primero al usuario y después procesa la petición y envía los resultados.

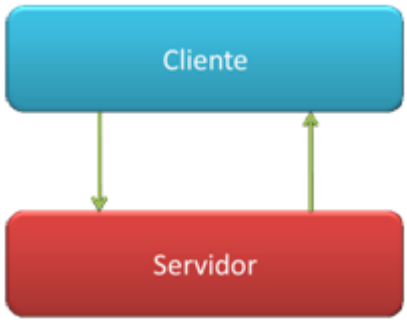
Beneficios
- Más seguridad ya que los datos se almacenan en el servidor que generalmente ofrece más control sobre la seguridad.
- Acceso centralizado a los datos que están almacenados en el servidor lo que facilita su acceso y actualización.

[34] DE LA TORRE LLORENTE César y ZORRILLA Unai. Guía de arquitectura de N-Capas orientada al dominio con .NET. España: Microsoft ibérica, 2010. ISBN 978-84-936696-3-8.

- Facilidad de mantenimiento ya que los roles y las responsabilidades se distribuyen entre los distintos servidores a través de la red lo que permite que un cliente no se vea afectado por un error en un servidor particular.

Cuando usarlo
- La aplicación se basa en un servidor y soportará múltiples clientes.
- La aplicación está normalmente limitada a un uso local y área LAN controlada.
- Estás implementando procesos de negocio que se usarán a lo largo de toda tu organización.
- Quieres centralizar el almacenamiento, backup y mantenimiento de la aplicación.
- La aplicación debe soportar distintos tipos de clientes y distintos dispositivos.

Figura 5. Estilo Cliente/Servidor



Fuente DE LA TORRE LLORENTE César y ZORRILLA Unai. Guía de arquitectura de N-Capas orientada al dominio con .NET. España: Microsoft ibérica, 2010. ISBN 978-84-936696-3-8

4.4 Estilo Arquitectural Basado en componentes

Descripción
El estilo de arquitectura basada en componentes describe un acercamiento al diseño de sistemas como un conjunto de componentes que exponen interfaces bien definidas y que colaboran entre sí para resolver el problema.

Características
- Es un estilo para diseñar aplicaciones a partir de componentes individuales.
- Enfatiza la descomposición del sistema en componentes con interfaces muy bien definidas.
- Define una aproximación al diseño a través de componentes que se comunican mediante interfaces que exponen métodos, eventos y propiedades.

Principios Clave

- Los componentes son diseñados de forma que puedan ser reutilizados en distintos escenarios en distintas aplicaciones aunque algunos componentes son diseñados para una tarea específica.
- Los componentes son diseñados para operar en diferentes entornos y contextos. Toda la información debe ser pasada al componente en lugar de incluirla en él o que este acceda a ella.
- Los componentes pueden ser extendidos a partir de otros componentes para ofrecer nuevos comportamientos.
- Los componentes exponen interfaces que permiten al código usar su funcionalidad y no revelan detalles internos de los procesos que realizan o de su estado.
- Los componentes están diseñados para ser lo más independientes posible de otros componentes, por lo que pueden ser desplegados sin afectar a otros componentes o sistemas.

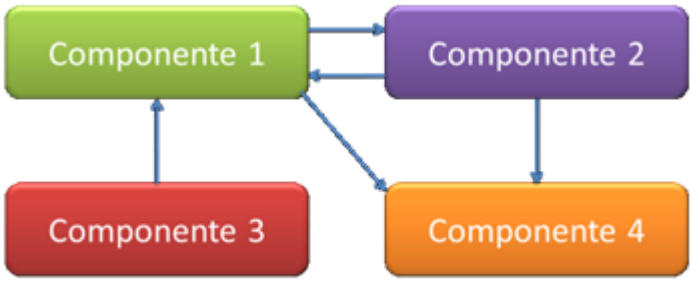
Beneficios

- Fácil despliegue ya que se puede sustituir un componente por su nueva versión sin afectar a otros componentes o al sistema.
- Reducción de costes ya que se pueden usar componentes de terceros para abaratar los costes de desarrollo y mantenimiento.
- Reusables ya que son independientes del contexto se pueden emplear en otras aplicaciones y sistemas.
- Reducción de la complejidad gracias al uso de contenedores de componentes que realizan la activación, gestión del ciclo de vida, etc.

Cuando usarlo

- Tienes componentes que sirvan a tu sistema o los puedes conseguir.
- La aplicación ejecutará generalmente procedimientos con pocos o ningún dato de entrada.
- Quieres poder combinar componentes escritos en diferentes lenguajes.
- Quieres crear una arquitectura que permita reemplazar o actualizar uno de sus componentes de forma sencilla.

Figura 6. Estilo basado en componentes



Fuente DE LA TORRE LLORENTE César y ZORRILLA Unai. Guía de arquitectura de N-Capas orientada al dominio con .NET. España: Microsoft ibérica, 2010. ISBN 978-84-936696-3-8

4.5 Estilo Arquitectural En Capas (N- Layer)

El estilo arquitectural en capas se basa en una distribución jerárquica de los roles y las responsabilidades para proporcionar una división efectiva de los problemas a resolver. Los roles indican el tipo y la forma de la interacción con otras capas y las responsabilidades la funcionalidad que implementan.

Características

- Descomposición de los servicios de forma que la mayoría de interacciones ocurre solo entre capas vecinas.
- Las capas de una aplicación pueden residir en la misma máquina o pueden estar distribuidos entre varios equipos.
- Los componentes de cada capa se comunican con los componentes de otras capas a través de interfaces bien conocidos.
- Cada nivel agrega las responsabilidades y abstracciones del nivel inferior.

Principios Clave

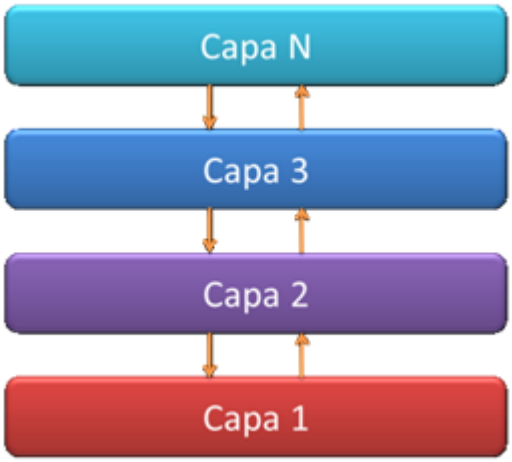
- Muestra una vista completa del modelo y a la vez proporciona suficientes detalles para entender las relaciones entre capas.
- No realiza ninguna suposición sobre los tipos de datos, métodos, propiedades y sus implementaciones.
- Separa de forma clara la funcionalidad de cada capa.
- Cada capa contiene la funcionalidad relacionada solo con las tareas de esa capa.
- Las capas inferiores no tienen dependencias de las capas superiores.
- La comunicación entre capas está basada en una abstracción que proporciona un bajo acoplamiento entre capas.

Beneficios

- Abstracción ya que los cambios se realizan a alto nivel y se puede incrementar o reducir el nivel de abstracción que se usa en cada capa del modelo.
- Aislamiento ya que se pueden realizar actualizaciones en el interior de las capas sin que esto afecte al resto del sistema.
- Rendimiento ya que distribuyendo las capas en distintos niveles físicos se puede mejorar la escalabilidad, la tolerancia a fallos y el rendimiento.
- Fácil de probar ya que cada capa tiene una interfaz bien definida sobre la que realizar las pruebas y la habilidad de cambiar entre diferentes implementaciones de una capa.
- Independencia ya que elimina la necesidad de considerar el hardware y el despliegue así como las dependencias con interfaces externas.

Cuando usarlo
- Ya tienes construidas capas de una aplicación anterior, que pueden reutilizarse o integrarse.
- Ya tienes aplicaciones que exponen su lógica de negocio a través de interfaces de servicios.
- La aplicación es compleja y el alto nivel de diseño requiere la separación para que los distintos equipos puedan concentrarse en distintas áreas de funcionalidad.
- La aplicación debe soportar distintos tipos de clientes y distintos dispositivos.
- Quieres implementar reglas y procesos de negocio complejos o configurables.

Figura 7. Estilo en capas (N-Layer)



Fuente DE LA TORRE LLORENTE César y ZORRILLA Unai. Guía de arquitectura de N-Capas orientada al dominio con .NET. España: Microsoft ibérica, 2010. ISBN 978-84-936696-3-8

4.6 Estilo Arquitectural Presentación Desacoplada

Descripción
El estilo de presentación separada indica cómo debe realizarse el manejo de las acciones del usuario, la manipulación de la interfaz y los datos de la aplicación. Este estilo separa los componentes de la interfaz del flujo de datos y de la manipulación.

Características
- Es un estilo, para diseñar aplicaciones, basado en patrones de diseño conocidos.
- Separa la lógica para el manejo de la interacción de la representación de los datos con que trabaja el usuario.
- Permite a los diseñadores crear una interfaz gráfica mientras los desarrolladores escriben el código para su funcionamiento.

- Ofrece un mejor soporte para el testeo ya que se pueden testear los comportamientos individuales.

Principios Clave
- El estilo de presentación desacoplada separa el procesamiento de la interfaz en distintos roles.
- Permite construir *mocks* que replican el comportamiento de otros objetos durante el testeo.
- Usa eventos para notificar a la vista cuando hay datos del modelo que han sido modificados.
- El controlador maneja los eventos disparados desde los controles de usuario en la vista.

Beneficios
- Fácil de probar ya que en las implementaciones comunes los roles son simplemente clases que pueden ser testeadas y reemplazadas por *mocks* que simulen su comportamiento.
- Reusabilidad ya que los controladores pueden ser aprovechados en otras vistas compatibles y las vistas pueden ser aprovechadas en otros controladores compatibles.

Cuando usarlo

- Quieres mejorar el testeo y el mantenimiento de la funcionalidad de la interfaz.
- Quieres separar la tarea de crear la interfaz de la lógica que la maneja.
- No quieres que la Interfaz contenga ningún código de procesamiento de eventos.
- El código de procesamiento de la interfaz no implementa ninguna lógica de negocio.

Figura 8. Estilo presentación desacoplada

Fuente DE LA TORRE LLORENTE César y ZORRILLA Unai. Guía de arquitectura de N-Capas orientada al dominio con .NET. España: Microsoft ibérica, 2010. ISBN 978-84-936696-3-8

4.7 Estilo Arquitectural N- Niveles (N- Tier)

Descripción

El estilo arquitectural de N- Niveles define la separación de la funcionalidad en segmentos/niveles físicos separados, similar que el estilo en N- Capas pero sitúa cada segmento en una máquina distinta. En este caso hablamos de Niveles físicos (Tiers)

Características

- Separación de niveles físicos (Servidores normalmente) por razones de escalabilidad, seguridad, o simplemente necesidad (p.e. si la aplicación cliente se ejecuta en máquinas remotas, la Capa de Presentación necesariamente se ejecutará en un Nivel físico separado).

Principios Clave

- Es un estilo para definir el despliegue de las capas de la aplicación.
- Se caracteriza por la descomposición funcional de las aplicaciones, componentes de servicio y su despliegue distribuido, que ofrece mejor escalabilidad, disponibilidad, rendimiento, manejabilidad y uso de recursos.
- Cada nivel es completamente independiente de los otros niveles excepto del inmediatamente inferior.
- Este estilo tiene al menos 3 niveles lógicos o capas separados. Cada capa implementa funcionalidad específica y está físicamente separada en distintos servidores.
- Una capa es desplegada en un nivel si uno o más servicios o aplicaciones dependen de la funcionalidad expuesta por dicha capa.

Beneficios

- Mantenibilidad ya que cada nivel es independiente de los otros las actualizaciones y los cambios pueden ser llevados a cabo sin afectar a la aplicación como un todo.
- Escalabilidad porque los niveles están basados en el despliegue de capas realizar el escalado de la aplicación es bastante directo.
- Disponibilidad ya que las aplicaciones pueden redundar cualquiera de los niveles y ofrecer así tolerancia a fallos.

Cuando usarlo

- Los requisitos de procesamiento de las capas de la aplicación difieren.
- Los requisitos de seguridad de las capas de la aplicación difieren.
- Quieres compartir la lógica de negocio entre varias aplicaciones.
- Tienes el suficiente hardware para desplegar el número necesario de servidores en cada nivel.

Figura 9. Estilo Arquitectural N- Niveles (N- Tier)

Fuente DE LA TORRE LLORENTE César y ZORRILLA Unai. Guía de arquitectura de N-Capas orientada al dominio con .NET. España: Microsoft ibérica, 2010. ISBN 978-84-936696-3-8

4.8 Estilo Arquitectural Arquitectura Orientada al Dominio (DDD)

Descripción

DDD (*Domain Driven Design*) no es solo un estilo arquitectural, es también una forma de afrontar los proyectos a nivel de trabajo del equipo de desarrollo, Ciclo de vida del proyecto, identificación del „Lenguaje Ubicuo" a utilizar con los Expertos
En el negocio, etc. Sin embargo, DDD también identifica una serie de patrones de diseño y estilo de Arquitectura concreto.
DDD es también, por lo tanto, una aproximación concreta para diseñar software basándonos sobre todo en la importancia del
Dominio del Negocio, sus elementos y comportamientos y las relaciones entre ellos. Está muy indicado para diseñar e implementar aplicaciones empresariales complejas donde es fundamental definir un Modelo de Dominio expresado en el propio lenguaje de los expertos del Dominio de negocio real (el llamado Lenguaje Ubicuo). El modelo de Dominio puede verse como un marco dentro del cual se debe diseñar la aplicación.
DDD identifica una serie de patrones de Arquitectura importantes a tener en cuenta en el Diseño de una aplicación, como son:
- Arquitectura N- Capas (Capas específicas tendencias arquitectura DDD)
- Patrones de Diseño:
    Reporitory
    Entity
    Aggregate

Value- Object
Unit of Work
Services

- En DDD también es fundamental el desacoplamiento entre componentes.

Principios Clave

Para aplicar DDD se debe de tener un buen entendimiento del Dominio de Negocio que se quiere modelar, o conseguir ese conocimiento adquiriéndolo a partir de los expertos del Dominio real. Todo el equipo de desarrollo debe de tener contacto con los expertos del dominio (expertos funcionales) para modelar correctamente el Dominio. En DDD, los Arquitectos, desarrolladores, Jefe de proyecto y Testers (todo el equipo) deben acordar hacer uso de un único lenguaje sobre el Dominio del Negocio que esté centrado en cómo los expertos de dicho dominio articulan su trabajo. No debemos implementar nuestro propio lenguaje/términos internos de aplicación. El corazón del software es el Modelo del Dominio el cual es una proyección directa de dicho lenguaje acordado (Lenguaje Ubicuo) y permite al equipo de desarrollo el encontrar rápidamente áreas incorrectas en el software al analizar el lenguaje que lo rodea. La creación de dicho lenguaje común no es simplemente un ejercicio de aceptar información de los expertos del dominio y aplicarlo. A menudo, los problemas en la comunicación de requerimientos funcionales no son solo por mal entendidos del lenguaje del Dominio, también deriva del hecho de que dicho lenguaje sea ambiguo. Es importante destacar que aunque DDD proporciona muchos beneficios técnicos, como mantenibilidad, debe aplicarse solamente en dominios complejos donde el modelo y los procesos linguisticos proporcionan claros beneficios en la comunicación de información compleja y en la formulación de un entendimiento común del Dominio. Así mismo, la complejidad Arquitectural es también mayor que una aplicación orientada a datos, si bien ofrece una mantenibilidad y desacoplamiento entre componentes mucho mayor.

Beneficios

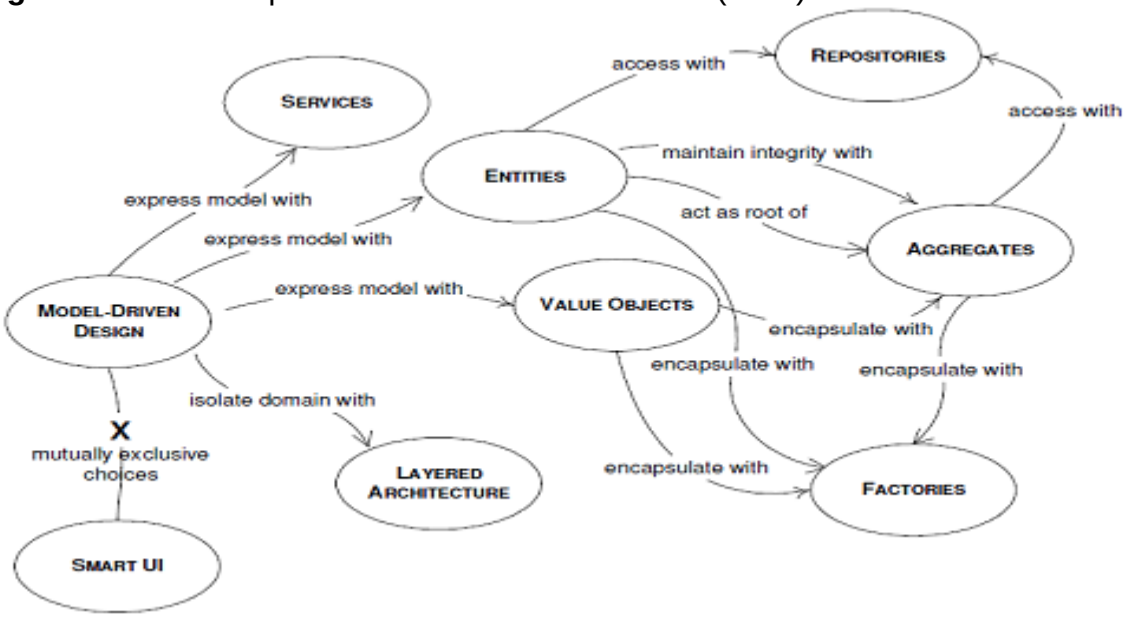
DDD ofrece los siguientes beneficios:

- **Comunicación:** Todas las partes de un equipo de desarrollo pueden usar el modelo de dominio y las entidades que define para comunicar conocimiento del negocio y requerimientos, haciendo uso de un lenguaje común.
- **Extensibilidad:** La Capa del Dominio es el corazón del software y por lo tanto estará completamente desacoplada de las capas de infraestructura, siendo más fácil así extender/evolucionar la tecnología del software
- **Mejor Testing:** La Arquitectura DDD también facilita el Testing y Mocking, debido a que la tendencia de diseño es a desacoplar los objetos de las diferentes capas de la Arquitectura, lo cual facilita el Mocking y Testing correctos.

Cuando usarlo

- Considerar DDD en aplicaciones complejas con mucha lógica de negocio, con Dominios complejos y donde se desea mejorar la comunicación y minimizar los malos entendidos en la comunicación del equipo de desarrollo.
- DDD es también una aproximación ideal en escenarios empresariales grandes y complejos que son difíciles de manejar con otras técnicas.

Figura 10. Estilo Arquitectura Orientada al Dominio (DDD)



Fuente DE LA TORRE LLORENTE César y ZORRILLA Unai. Guía de arquitectura de N-Capas orientada al dominio con .NET. España: Microsoft ibérica, 2010. ISBN 978-84-936696-3-8

4.9 Estilo Arquitectural Orientado a Objetos

Descripción

El estilo orientado a objetos es un estilo que define el sistema como un conjunto de objetos que cooperan entre sí en lugar de como un conjunto de procedimientos. Los objetos son discretos, independientes y poco acoplados, se comunican mediante interfaces y permiten enviar y recibir mensajes.

Características

- Es un estilo para diseñar aplicación basada en un número de unidades lógicas y código reusable.

- Describe el uso de objetos que contienen los datos y el comportamiento para trabajar con esos datos y además tienen un rol o responsabilidad distinta.
- Hace hincapié en la reutilización a través de la encapsulación, la modularidad, el polimorfismo y la herencia.
- Contrasta con el acercamiento procedimental donde hay una secuencia predefinida de tareas y acciones. El enfoque orientado a objetos emplea el concepto de objetos interactuando unos con otros para realizar las tareas.

Principios Clave
- Permite reducir una operación compleja mediante generalizaciones que mantienen las características de la operación.
- Los objetos se componen de otros objetos y eligen esconder estos objetos internos de otras clases o exponerlos como simples interfaces.
- Los objetos heredan de otros objetos y usan la funcionalidad de estos objetos base o redefinen dicha funcionalidad para implementar un nuevo comportamiento, la herencia facilita el mantenimiento y la actualización ya que los cambios se propagan del objeto base a los herederos automáticamente.
- Los objetos exponen la funcionalidad solo a través de métodos, propiedades y eventos y ocultan los detalles internos como el estado o las referencias a otros objetos. Esto facilita la actualización y el reemplazo de objetos asumiendo que sus interfaces son compatibles sin afectar al resto de objetos.
- Los objetos son polimórficos, es decir, en cualquier punto donde se espere un objeto base se puede poner un objeto heredero.
- Los objetos se desacoplan de los objetos que los usan implementando una interfaz que los objetos que los usan conocen. Esto permite proporcionar otras implementaciones sin afectar a los objetos consumidores de la interfaz.

Beneficios
- Comprensión ya que el diseño orientado a objetos define una serie de componentes mucho más cercanos a los objetos del mundo real.
- Reusabilidad ya que el polimorfismo y la abstracción permiten definir contratos en interfaces y cambiar las implementaciones de forma transparente.
- Fácil de probar gracias a la encapsulación de los objetos.
- Extensibilidad gracias a la encapsulación, el polimorfismo y la abstracción.

Cuando usarlo
- Quieres modelar la aplicación basándote en objetos reales y sus acciones.
- Ya tienes objetos que encajan en el diseño y con los requisitos operacionales.
- Necesitas encapsular la lógica y los datos juntos de forma transparente.

Figura 11. Estilo Arquitectural Orientado a Objetos

Fuente DE LA TORRE LLORENTE César y ZORRILLA Unai. Guía de arquitectura de N-Capas orientada al dominio con .NET. España: Microsoft ibérica, 2010. ISBN 978-84-936696-3-8

4.10 Estilo Arquitectural Orientación a Servicios (SOA)

Descripción
El estilo orientado a servicios permite a una aplicación ofrecer su funcionalidad como un conjunto de servicios para que sean consumidos. Los servicios usan interfaces estándares que pueden ser invocadas, publicadas y descubiertas. Se centran en proporcionar un esquema basado en mensajes con operaciones de nivel de aplicación y no de componente o de objeto.

Características
- La interacción con el servicio está muy desacoplada.
- Puede empaquetar procesos de negocio como servicios.
- Los clientes y otros servicios pueden acceder a servicios locales corriendo en el mismo nivel.
- Los clientes y otros servicios acceden a los servicios remotos a través de la red.
- Puede usar un amplio rango de protocolos y formatos de datos.

Principios Clave
- Los servicios son autónomos, es decir cada servicio se mantiene, se desarrolla, se despliega y se versiona independientemente.
- Los servicios pueden estar situados en cualquier nodo de una red local o remota mientras esta soporte los protocolos de comunicación necesarios.
- Cada servicio es independiente del resto de servicios y puede ser reemplazado o actualizado sin afectar a las aplicaciones que lo usan mientras la interfaz sea compatible.
- Los servicios comparten esquemas y contratos para comunicarse, no clases.
- La compatibilidad se basa en políticas que definen funcionalidades como el mecanismo de transporte, el protocolo y la seguridad.

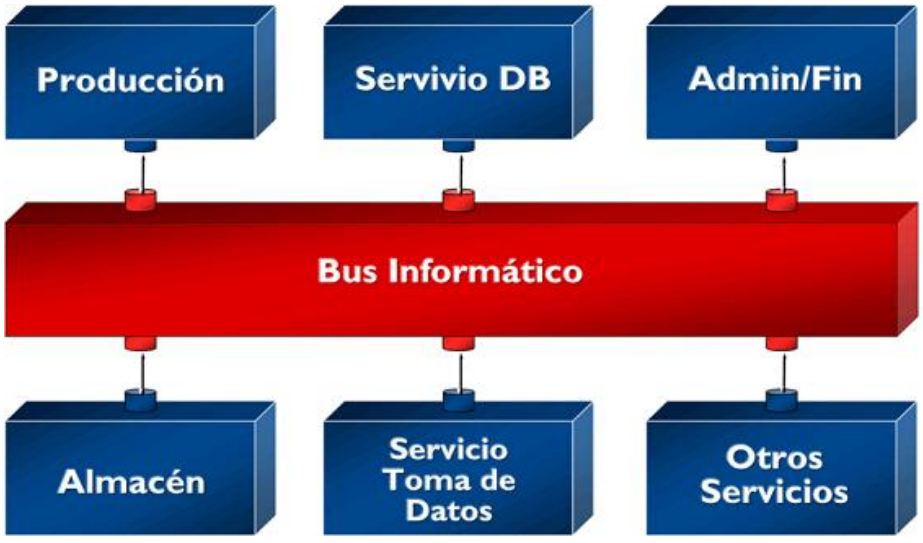
Beneficios

- Alineamiento con el dominio ya que la reutilización de servicios comunes con interfaces estándar aumenta las oportunidades tecnológicas y de negocio y reduce costes.
- Abstracción ya que los servicios son autónomos y se accede a ellos a través de un contrato formal que proporciona bajo acoplamiento y abstracción.
- Descubrimiento ya que los servicios exponen descripciones que permiten a otras aplicaciones y servicios encontrarlos y determinar su interfaz automáticamente.

Cuando usarlo

- Tienes acceso a servicios adecuados o puedes comprarlos a una empresa.
- Quieres construir aplicaciones que compongan múltiples servicios en una interfaz única.
- Estás creando S+S, SaaS o una aplicación en la nube.
- Necesitas soportar comunicación basada en mensajes para segmentos de la aplicación.
- Necesitas exponer funcionalidad de forma independiente de la plataforma.
- Necesitas utilizar servicios federados como servicios de autentificación.
- Quieres exponer servicios que puedan ser descubiertos y usados por clientes que no tuviesen conocimiento previo de sus interfaces.
- Quieres soportar escenarios de interoperabilidad e integración.

Figura 12. Estilo Orientación a Servicios (SOA)



Fuente DE LA TORRE LLORENTE César y ZORRILLA Unai. Guía de arquitectura de N-Capas orientada al dominio con .NET. España: Microsoft ibérica, 2010. ISBN 978-84-936696-3-8

4.11 Estilo Arquitectural Bus de Servicios (Mensajes)

Descripción

El estilo de arquitectural de bus de mensajes define un sistema software que puede enviar y recibir mensajes usando uno o más canales de forma que las aplicaciones pueden interactuar sin conocer detalles específicos la una de la otra.

Características

- Es un estilo para diseñar aplicaciones donde la interacción entre las mismas se realiza a través del paso de mensajes por un canal de comunicación común.
- Las comunicaciones entre aplicaciones suelen ser asíncronas.
- Se implementa a menudo usando un sistema de mensajes como MSMQ.
- Muchas implementaciones consisten en aplicaciones individuales que se comunican usando esquemas comunes y una infraestructura compartida para el envío y recepción de mensajes.

Principios Clave

- Toda la comunicación entre aplicaciones se basa en mensajes que usan esquemas comunes.
- Las operaciones complejas pueden ser creadas combinando un conjunto de operaciones más simples que realizan determinadas tareas.
- Se pueden añadir o eliminar aplicaciones del bus para cambiar la lógica usada para procesar los mensajes.
- Al usar un modelo de comunicación con mensajes basados en estándares se puede interactuar con aplicaciones desarrolladas para distintas plataformas.

Beneficios

- Expansión ya que las aplicaciones pueden ser añadidas o eliminadas del bus sin afectar al resto de aplicaciones existentes.
- Baja complejidad, la complejidad de la aplicación se reduce dado que cada aplicación solo necesita conocer cómo comunicarse con el bus.
- Mejor rendimiento ya que no hay intermediarios en la comunicación entre dos aplicaciones, solo la limitación de lo rápido que entregue el bus los mensajes.
- Escalable ya que muchas instancias de la misma aplicación pueden ser asociadas al bus para dar servicio a varias peticiones al mismo tiempo.
- Simplicidad ya que cada aplicación solo tiene que soportar una conexión con el bus en lugar de varias conexiones con el resto de aplicaciones.

Cuando usarlo

- Tienes aplicaciones que interactúan unas con otras para realizar tareas.
- Estás implementando una tarea que requiere interacción con aplicaciones externas.

- Estás implementando una tarea que requiere interacción con otras aplicaciones desplegadas en entornos distintos.
- Tienes aplicaciones que realizan tareas separadas y quieres combinar esas tareas en una sola operación.

Figura 13. Estilo Arquitectural Bus de Servicios (Mensajes)

Fuente DE LA TORRE LLORENTE César y ZORRILLA Unai. Guía de arquitectura de N-Capas orientada al dominio con .NET. España: Microsoft ibérica, 2010. ISBN 978-84-936696-3-8

4.12 Estilo Tubería-filtros

Siempre se encuadra este estilo dentro de las llamadas arquitecturas de flujo de datos. Es sin duda alguna el estilo que se definió más temprano. Una tubería (pipeline) es una popular arquitectura que conecta componentes computacionales (filtros) a través de conectores (pipes), de modo que las computaciones se ejecutan a la manera de un flujo. Los datos se transportan a través de las tuberías entre los filtros, transformando gradualmente las entradas en salidas.

Figura 14. Estilo Tubería-filtros

Fuente REYNOSO, Carlos y KICILLOF, Nicolas. Estilos y Patrones en la Estrategia de Arquitectura de Microsoft. UNIVERSIDAD DE BUENOS AIRES, Marzo 2004.

El estilo se debe usar cuando:

- Se puede especificar la secuencia de un número conocido de pasos.
- No se requiere esperar la respuesta asincrónica de cada paso.

- Se busca que todos los componentes situados corriente abajo sean capaces de inspeccionar y actuar sobre los datos que vienen de corriente arriba (pero no viceversa).

Ventajas del estilo tubería-filtros:

- Es simple de entender e implementar. Es posible implementar procesos complejos con editores gráficos de líneas de tuberías o con comandos de línea.
- Fuerza un procesamiento secuencial.
- Es fácil de envolver (wrap) en una transacción atómica.
- Los filtros se pueden empaquetar, y hacer paralelos o distribuidos.

Desventajas:

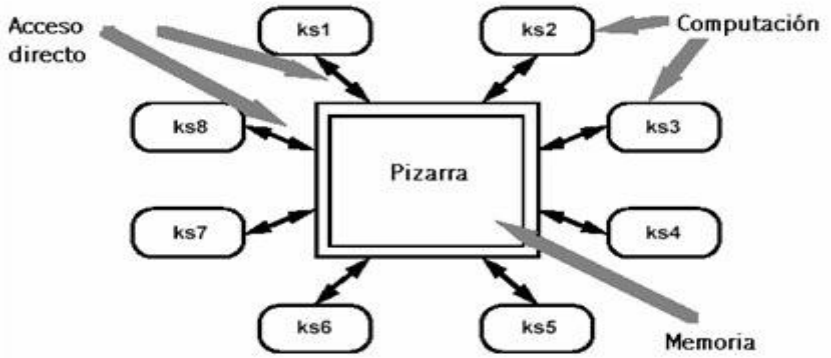
- El patrón puede resultar demasiado simplista, especialmente para orquestación de servicios que podrían ramificar la ejecución de la lógica de negocios de formas complicadas.
- No maneja con demasiada eficiencia construcciones condicionales, bucles y otras lógicas de control de flujo. Agregar un paso suplementario afecta la performance de cada ejecución de la tubería.
- Eventualmente pueden llegar a requerirse buffers de tamaño indefinido, por ejemplo en las tuberías de clasificación de datos.
- El estilo no es apto para manejar situaciones interactivas, sobre todo cuando se requieren actualizaciones incrementales de la representación en pantalla.
- La independencia de los filtros implica que es muy posible la duplicación de funciones de preparación que son efectuadas por otros filtros.

4.13 Arquitecturas de Pizarra o Repositorio

En esta arquitectura hay dos componentes principales: una estructura de datos que representa el estado actual y una colección de componentes independientes que operan sobre él. En base a esta distinción se han definidos dos subcategorías principales del estilo:

1. Si los tipos de transacciones en el flujo de entrada definen los procesos a ejecutar, el repositorio puede ser una base de datos tradicional (implícitamente no cliente-servidor).
2. Si el estado actual de la estructura de datos dispara los procesos a ejecutar, el repositorio es lo que se llama una pizarra pura o un tablero de control.

Figura 15. Arquitecturas de Pizarra o Repositorio



Fuente REYNOSO, Carlos y KICILLOF, Nicolas. Estilos y Patrones en la Estrategia de Arquitectura de Microsoft. UNIVERSIDAD DE BUENOS AIRES, Marzo 2004.

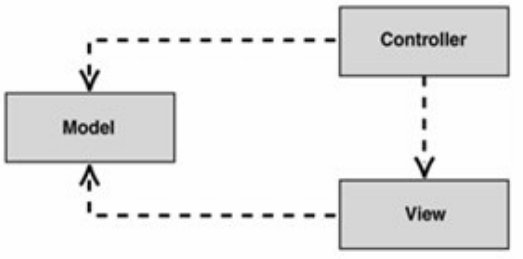
Estos sistemas se han usado en aplicaciones que requieren complejas interpretaciones de proceso de señales (reconocimiento de patrones, reconocimiento de habla, etc), o en sistemas que involucran acceso compartido a datos con agentes débilmente acoplados. También se han implementado estilos de este tipo en procesos en lotes de base de datos y ambientes de programación organizados como colecciones de herramientas en torno a un repositorio común. Muchos más sistemas de los que se cree están organizados como repositorios: bibliotecas de componentes reutilizables, grandes bases de datos y motores de búsqueda.
Un sistema de pizarra se implementa para resolver problemas en los cuales las entidades individuales se manifiestan incapaces de aproximarse a una solución, o para los que no existe una solución analítica, o para los que sí existe, pero es inviable por la dimensión del espacio de búsqueda.
Todo modelo de este tipo consiste en las siguientes tres partes:

- Fuentes de conocimiento, necesarias para resolver el problema.
- Una pizarra que representa el estado actual de la resolución del problema.
- Una estrategia, que regula el orden en que operan las fuentes.

4.14 Model-View-Controller (MVC)

Reconocido como estilo arquitectónico por Taylor y Medvidovic[35], muy rara vez mencionado en los surveys estilísticos usuales, el MVC ha sido propio de las aplicaciones en Smalltalk por lo menos desde 1992, antes que se generalizaran las arquitecturas en capas múltiples.
Un propósito común en numerosos sistemas es el de tomar datos de un almacenamiento y mostrarlos al usuario. Luego que el usuario introduce modificaciones, las mismas se reflejan en el almacenamiento. Dado que el flujo de información ocurre entre el almacenamiento y la interfaz, una tentación común, un impulso espontáneo (hoy se llamaría un anti-patrón) es unir ambas piezas para reducir la cantidad de código y optimizar el desempeño.

Figura 16. Model-View-Controller (MVC)



Fuente REYNOSO, Carlos y KICILLOF, Nicolas. Estilos y Patrones en la Estrategia de Arquitectura de Microsoft. UNIVERSIDAD DE BUENOS AIRES, Marzo 2004.

El patrón conocido como Modelo-Vista-Controlador (MVC) separa el modelado del dominio, la presentación y las acciones basadas en datos ingresados por el usuario en tres clases diferentes:

- Modelo. El modelo administra el comportamiento y los datos del dominio de aplicación, responde a requerimientos de información sobre su estado (usualmente formulados desde la vista) y responde a instrucciones de cambiar el estado (habitualmente desde el controlador).

[35] TAYLOR, Richard, *et al*. *A Component and Message Based Architectural Style for GUI Software.* En: *Proceedings of the 17th International Conference on Software Engineering*, 23 al 30 de Abril de 1995. Seattle: *ACM Press*, 1995. p. 295-304.

- Vista. Maneja la visualización de la información.
- Controlador. Interpreta las acciones del ratón y el teclado, informando al modelo y/o a la vista para que cambien según resulte apropiado.

Ventajas:

- Soporte de vistas múltiples. Dado que la vista se halla separada del modelo y no hay dependencia directa del modelo con respecto a la vista, la interfaz de usuario puede mostrar múltiples vistas de los mismos datos simultáneamente. Por ejemplo, múltiples páginas de una aplicación de Web pueden utilizar el mismo modelo de objetos, mostrado de maneras diferentes.
- Adaptación al cambio. Los requerimientos de interfaz de usuario tienden a cambiar con mayor rapidez que las reglas de negocios. Los usuarios pueden preferir distintas opciones de representación, o requerir soporte para nuevos dispositivos como teléfonos celulares o PDAs.

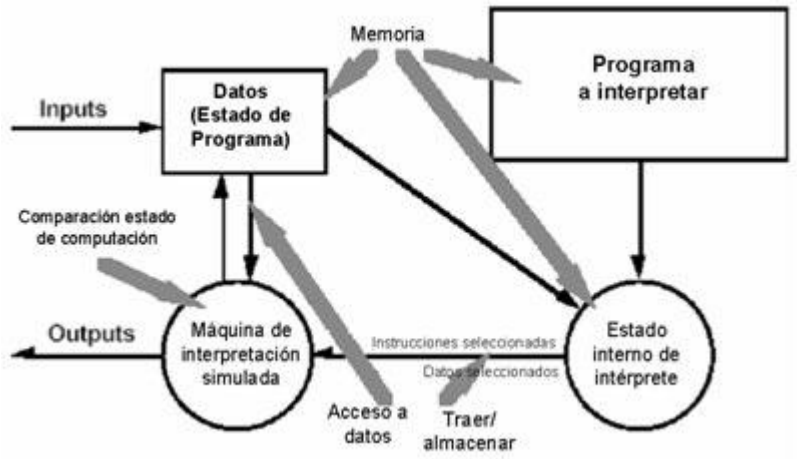
Desventajas:

- Complejidad. El patrón introduce nuevos niveles de indirección y por lo tanto aumenta ligeramente la complejidad de la solución. También se profundiza la orientación a eventos del código de la interfaz de usuario, que puede llegar a ser difícil de depurar.
- Costo de actualizaciones frecuentes. Desacoplar el modelo de la vista no significa que los desarrolladores del modelo puedan ignorar la naturaleza de las vistas. Si el modelo experimenta cambios frecuentes, por ejemplo, podría desbordar las vistas con una lluvia de requerimientos de actualización.

4.15 Arquitectura de Máquinas Virtuales

La arquitectura de máquinas virtuales se ha llamado también intérpretes basados en tablas. De hecho, todo intérprete involucra una máquina virtual implementada en software. Se puede decir que un intérprete incluye un seudo-programa a interpretar y una máquina de interpretación. El seudo-programa a su vez incluye el programa mismo y el análogo que hace el intérprete de su estado de ejecución (o registro de activación). La máquina de interpretación incluye tanto la definición del intérprete como el estado actual de su ejecución. De este modo, un intérprete posee por lo general cuatro componentes: (1) una máquina de interpretación que lleva a cabo la tarea, (2) una memoria que contiene el seudo-código a interpretar, (3) una representación del estado de control de la máquina de interpretación, y (4) una representación del estado actual del programa que se simula.

Figura 17. Arquitectura de Máquinas Virtuales



Fuente REYNOSO, Carlos y KICILLOF, Nicolas. Estilos y Patrones en la Estrategia de Arquitectura de Microsoft. UNIVERSIDAD DE BUENOS AIRES, Marzo 2004.

El estilo en su conjunto se utiliza habitualmente para construir máquinas virtuales que reducen el vacío que media entre el motor de computación esperado por la semántica del programa y el motor físicamente disponible. Las aplicaciones inscriptas en este estilo simulan funcionalidades no nativas al hardware y software en que se implementan, o capacidades que exceden a (o que no coinciden con) las capacidades del paradigma de programación que se está implementando.
Dado que hasta cierto punto las máquinas virtuales no son una opción sino que devienen inevitables en ciertos contextos, nadie se ha entretenido identificando sus ventajas y deméritos.

4.16 Arquitecturas Basadas en Eventos

Las arquitecturas basadas en eventos se han llamado también de invocación implícita. Otros nombres propuestos para el mismo estilo han sido integración reactiva o difusión (broadcast) selectiva. Por supuesto que existen estrategias de programación basadas en eventos, sobre todo referidas a interfaces de usuario, y hay además eventos en los modelos de objetos y componentes, pero no es a eso a lo que se refiere primariamente el estilo, aunque esa variedad no está del todo excluida. En términos de patrones de diseño, el patrón que corresponde más estrechamente a este estilo es el que se conoce como Observer, un término que se hizo popular en Smalltalk a principios de los ochenta; en el mundo de Java se le conoce como modelo de delegación de eventos.

Los ejemplos de sistemas que utilizan esta arquitectura son numerosos. El estilo se utiliza en ambientes de integración de herramientas, en sistemas de gestión de base de datos para asegurar las restricciones de consistencia (bajo la forma de disparadores, por ejemplo), en interfaces de usuario para separar la presentación de los datos de los procedimientos que gestionan datos, y en editores sintácticamente orientados para proporcionar verificación semántica incremental.

Este estilo puede utilizarse cuando:

- Se desea manejar independientemente y de forma aislada diversas implementaciones de una "función" específica.
- Las respuestas de una implementación no afectan la forma en que trabajan otras implementaciones.
- Todas las implementaciones son de escritura solamente o de dispararse-y-olvidar, tal que la salida del proceso de negocios no está definida por ninguna implementación, o es definida sólo por una implementación de negocios específica.

Ventajas

- Se optimiza el mantenimiento haciendo que procesos de negocios que no están relacionados sean independientes.
- Se alienta el desarrollo en paralelo, lo que puede resultar en mejoras de performance.
- Es fácil de empaquetar en una transacción atómica.
- Es agnóstica en lo que respecta a si las implementaciones corren sincrónica o asincrónicamente porque no se espera una respuesta.
- Se puede agregar un componente registrándolo para los eventos del sistema; se pueden reemplazar componentes.

Desventajas:

- El estilo no permite construir respuestas complejas a funciones de negocios.
- Un componente no puede utilizar los datos o el estado de otro componente para efectuar su tarea.
- Cuando un componente anuncia un evento, no tiene idea sobre qué otros componentes están interesados en él, ni el orden en que serán invocados, ni el momento en que finalizan lo que tienen que hacer.

5. CONCLUSIONES

La arquitectura de software es uno de los temas de los que más se habla entre desarrolladores, pero es quizá el que menos se comprende y del cual se tienen definiciones poco precisas, al buscar fuentes encontramos que existe mucha ambigüedad en cuanto a definición de términos y estandarización.

Creemos que la arquitectura de software puede considerarse como el “puente” entre los requerimientos del sistema y la implementación en las aplicaciones empresariales. No consideramos posible que un desarrollador se siente a codificar inmediatamente después de tener los requerimientos cuando se trata de aplicaciones robustas y de gran complejidad. Ante estos proyectos es absolutamente necesario elaborar una detallada planeación y estudio de los componentes de la empresa.

Consideramos necesario al intentar modificar un sistema de dominio empresarial conocer a fondo la arquitectura del software existente para así lograr que cada uno de sus componentes se comunique armónicamente entre sí.

Diseñando un sistema con una arquitectura bien definida se previenen problemas y se evita limitar o modificar la arquitectura propia de la empresa, evitar fallos, problemas y restricciones en fases tempranas del desarrollo.

En un ámbito empresarial es común que trabajen varias personas en el mismo proyecto a través del tiempo, los sistemas de software crecen de tal forma que resulta muy complicado que sean diseñados, especificados y entendidos por una sola persona y por esto creemos que se hace necesario que exista una forma de expresar los sistemas en términos comunes, para esto existen los estilos arquitecturales y los lenguajes de descripción arquitectural. Son herramientas para hablar en términos comunes, identificar similitudes y así asimilar más fácilmente un proyecto entre varios desarrolladores.

No todo lo que tiene que ver con diseño requiere compulsivamente notaciones y modelos de objeto. Considerando los ADLs en su conjunto, habrá podido comprobarse que la tradición de modelado OO, juega en este campo un papel muy modesto, si es que juega alguno; la reciente tendencia a redefinir los ADLs en términos de mensajes, servicios, integración, XML Schemas, xADL, SOAP y sus derivaciones no hará más que consolidar ese estado de cosas.

Los ADLs son convenientes pero no han demostrado aún ser imprescindibles. Numerosas piezas de software, desde sistemas operativos a aplicaciones de misión crítica, se realizaron echando mano de recursos de diseño, o de formalismos incapaces de soportar evolución, expresar un estilo, implementar patrones o generar código; y es por esto que en los últimos años se ha generado

una demanda de ADLs como un tópico permanente de la arquitectura de software, y por eso nos ha parecido útil elaborar este estudio.

El objetivo final de la arquitectura es identificar los requisitos que producen un impacto en la estructura del sistema y reducir los riesgos asociados con la construcción del sistema. La arquitectura debe soportar los cambios futuros del software, del hardware y de funcionalidad demandada por los clientes. Del mismo modo, es responsabilidad del arquitecto analizar el impacto de sus decisiones de diseño y establecer un compromiso entre los diferentes requisitos de calidad así como entre los compromisos necesarios para satisfacer a los usuarios, al sistema y los objetivos del negocio.

Lo normal en una arquitectura es que no se base en un solo estilo arquitectural, sino que combine varios de dichos estilos arquitecturales para obtener las ventajas de cada uno.

Es importante entender que los estilos arquitecturales son indicaciones abstractas de cómo dividir en partes el sistema y de cómo estas partes deben interactuar. En las aplicaciones reales los estilos arquitecturales se “instancian” en una serie de componentes e interacciones concretas. Esta es la principal diferencia existente entre un estilo arquitectural y un patrón de diseño. Los patrones de diseño son descripciones estructurales y funcionales de cómo resolver de forma concreta un determinado problema mediante orientación a objetos, mientras que los estilos arquitecturales son descripciones abstractas y no están atados a ningún paradigma de programación específico.

En el diseño de la arquitectura lo primero que se decide es el tipo de sistema o aplicación que vamos a construir. La selección de un tipo de aplicación determina en cierta medida el estilo arquitectural que se va usar. Los principales son Cliente/Servidor, Sistemas de componentes, Arquitectura en capas, MVC, N-Niveles, SOA, etc. Los procesos de software actuales asumen que el sistema cambiará con el paso del tiempo; el sistema tendrá que evolucionar a medida que se prueba el sistema contra los requisitos del mercado y por eso, al finalizar el documento queremos dejar claro que al abordar el desarrollo de una aplicación empresarial se debe tener en cuenta que es **necesaria** una planeación detallada de la arquitectura a utilizar. Para esto hemos recompilado un listado de Estilos arquitecturales y Lenguajes de descripción arquitectural que ilustran de manera breve en qué consiste cada uno para el que no esté familiarizado con ellos y decida profundizar más sobre alguno en específico.

6. REFERENCIAS BIBLIOGRÁFICAS

ALLEN, Robert y GARLAN, David. *A formal approach to software architectures.* En: *Proceedings of the IFIP Congress '92*, Setiembre de 1992.

ANDREWS, Gregory R. *Paradigms for Process Interaction in Distributed Programs.* En: *ACM Computing Surveys*. vol.23, Issue 1. Marzo de 1991. p. 49-90.

ARLOW, Jim y NEUSTADT, iLA. *Patterns and MDA: Building Better Software with Archetype Patterns and UML.* Boston: *Pearson Education, Inc*, 2003. 528p. ISBN 0-321-11230-X.

BASS Len, CLEMENTS, Paul y KAZMAN, Rick. *Software Architecture in Practice. Reading, Addison-Wesley*, 1998.

BUSCHMANN, Frank, MEUNIER, Regine, ROHNERT, Hans, SOMMERLAD, Peter y STAL, Michael. *Pattern oriented software architecture.* En: *A system of patterns.* John Wiley & Sons, 1996.

BUSCHMAN, Frank. *Past, Present, and Future Trends in Software Patterns.* En: *IEEE Software Magazine.* Julio 2007.

Carnegie Mellom University 21 de Julio de 1995. *Formalizing Style to Understand Descriptions of Software Architecture* [en linea]. Disponible desde internet en: <http://www.cs.txstate.edu/~rp31/papers/styleformalism-tosem95.pdf>

DE LA TORRE LLORENTE César y ZORRILLA Unai. Guía de arquitectura de N-Capas orientada al dominio con .NET. España: Microsoft ibérica, 2010. ISBN 978-84-936696-3-8.

DIMOV, Aleksandar y ILIEVA, Sylvia. *Reliability Models in Architecture Description Languages*. En: *Proceeding CompSysTech '07 Proceedings of the 2007 international conference on Computer systems and technologies.* New York: ACM, 2007. ISBN 978-954-9641-50-9.

FIELDING, Roy Thomas. *Architectural styles and the design of network-based software architectures*. Tesis doctoral, University of California, Irvine, 2000.

FOWLER, Martin, *et al*. *Patterns of Enterprise Application Architecture.* Boston: Pearson Education Inc, 2002. 560p. ISBN 0-321-12742-0.

GAMMA, Erich; HELM, Richard, JOHNSON, Ralph y VLISSIDES, John. *Design Patterns: Elements of reusable object oriented software.* En: *Reading, Addison-Wesley*, 1995.

GARLAN, David. *Next generation software architectures: Recent research and future directions.* [Presentación], Columbia University, Enero de 2001.

GARLAN, David y SHAW, Mary. *An introduction to software architecture. CMU Software Engineering Institute Technical Report, CMU/SEI-94-TR-21, ESC-TR-94-21,* 1994.

GARLAN, David y SOO KIM, Jung. *Analyzing architectural styles.* Carnegie Mellon University, 2007.

IBM. 05 de abril de 2010. Arquitectura evolutiva y diseño emergente: Investigación sobre arquitectura y diseño [en linea]. Disponible desde Internet en: <http://www.ibm.com/developerworks/ssa/java/library/j-eaed1/>

IBM. 05 de abril de 2010. Arquitectura empresarial para Ingenieros de Sistemas [en linea]. Disponible desde Internet en: <http://www.ibm.com/developerworks/ssa/rational/library/edge/09/jun09/enterprise architecture/index.html>

KLEIN, Mark y KAZMAN, Rick. *Attribute-based architectural styles*. En: *Technical Report, CMU/SEI-99-TR-022, ESCTR-99-022.* Carnegie Mellon University, Octubre de 1999.

REYNOSO, Carlos y KICILLOF, Nicolas. Lenguajes de Descripción de Arquitectura (ADL). [en línea]. Buenos Aires, Argentina: UNIVERSIDAD DE BUENOS AIRES, Marzo 2004 - [citado el 15 de junio de 2011] Version 1.0 disponible desde internet en: <http://carlosreynoso.com.ar/archivos/arquitectura/ADL.PDF>

REYNOSO, Carlos y KICILLOF, Nicolas. Estilos y Patrones en la Estrategia de Arquitectura de Microsoft. UNIVERSIDAD DE BUENOS AIRES, Marzo 2004.

RUMBAUGH, James, *et al*. *Object-oriented modeling and design.* Englewood Cliffs: Prentice Hall, 1991.

SIMARRO, Juan Matias. APLICACIÓN DEL ANÁLISIS DE DEPENDENCIAS A LA ARQUITECTURA SOFTWARE. España: UNIVERSIDAD DE CASTILLA-LA MANCHA ESCUELA POLITÉCNICA SUPERIOR, Diciembre de 2004. 32p.

SHAW, Mary y CLEMENTS, Paul*. A field guide to Boxology: Preliminary classification of architectural styles forsoftware systems*. En: *Proceedings of the 21st International Computer Software and Applications Conference 1997*.

TAYLOR, Richard, *et al*. *A Component and Message Based Architectural Style for GUI Software.* En: *Proceedings of the 17th International Conference on Software Engineering*, 23 al 30 de Abril de 1995. Seattle: *ACM Press*, 1995. p. 295-304.

U.S.B Universidad Simon Bolivar. Abril del 2004. Arquitectura de software [en linea]. Disponible desde internet en: <http://prof.usb.ve/lmendoza/Documentos/PS-116/Guia%20Arquitectura%20v.2.pdf>