Estilos y Patrones en la Estrategia de Arquitectura de Microsoft

Estilos y Patrones en la Estrategia de Arquitectura de Microsoft

Versión 1.0 – Marzo de 2004

Carlos Reynoso – Nicolás Kicillof

UNIVERSIDAD DE BUENOS AIRES

Estilos arquitectónicos

El tópico más urgente y exitoso en arquitectura de software en los últimos cuatro o cinco años es, sin duda, el de los patrones (*patterns*), tanto en lo que concierne a los patrones de diseño como a los de arquitectura. Inmediatamente después, en una relación a veces de complementariedad, otras de oposición, se encuentra la sistematización de los llamados estilos arquitectónicos. Cada vez que alguien celebra la mayoría de edad de la arquitectura de software, y aunque señale otros logros como la codificación de los ADLs o las técnicas de refinamiento, esos dos temas se destacan más que cualesquiera otros [Gar00] [Shaw01]. Sin embargo, sólo en contadas ocasiones la literatura técnica existente se ocupa de analizar el vínculo entre estilos y patrones [Shaw94] [SG95] [SC96] [BMR+96] [MKM+97] [Hil01a] [Hil01b]. Se los yuxtapone cada vez que se enumeran las ideas y herramientas disponibles, se señala con frecuencia su aire de familia, pero no se articula formal y sistemáticamente su relación.

Habrá que admitir desde el vamos que ambos asuntos preocupan y tienen como destinatarios a distintas clases de profesionales, o diferentes *stakeholders*, como ahora se recomienda llamar: quienes trabajan con estilos favorecen un tratamiento estructural que concierne más bien a la teoría, la investigación académica y la arquitectura en el nivel de abstracción más elevado, mientras que quienes se ocupan de patrones se ocupan de cuestiones que están más cerca del diseño, la práctica, la implementación, el proceso, el refinamiento, el código. Los patrones coronan una práctica de diseño que se origina antes que la arquitectura de software se distinguiera como discurso en perpetuo estado de formación y proclamara su independencia de la ingeniería en general y el modelado en particular. Los estilos, en cambio, expresan la arquitectura en el sentido más formal y teórico, constituyendo un tópico esencial de lo que Goguen ha llamado el campo "seco" de la disciplina [Gog92]. Más adelante volveremos sobre esta distinción.

Conviene caracterizar el escenario que ha motivado la aparición del concepto de estilo, antes siquiera de intentar definirlo. Desde los inicios de la arquitectura de software, se observó que en la práctica del diseño y la implementación ciertas regularidades de configuración aparecían una y otra vez como respuesta a similares demandas. El número de esas formas no parecía ser muy grande. Muy pronto se las llamó estilos, por analogía con el uso del término en arquitectura de edificios. Un estilo describe entonces una clase de arquitectura, o piezas identificables de las arquitecturas empíricamente dadas. Esas piezas se encuentran repetidamente en la práctica, trasuntando la existencia de decisiones

estructurales coherentes. Una vez que se han identificado los estilos, es lógico y natural pensar en re-utilizarlos en situaciones semejantes que se presenten en el futuro [Kaz01]. Igual que los patrones de arquitectura y diseño, todos los estilos tienen un nombre: cliente-servidor, modelo-vista-controlador, tubería-filtros, arquitectura en capas…

Como conceptos, los estilos fueron formulados por primera vez cuando el escenario tecnológico era sustancialmente distinto del que se manifiesta hoy en día. Es por eso que en este estudio se analizarán las definiciones de los estilos arquitectónicos que se han propuesto, así como su posición en el campo de las vistas de arquitectura, su lugar en la metodología y su relación con los patrones, tomando en consideración las innovaciones experimentadas por el campo de los estilos desde su formulación inicial en 1992, coincidente con el instante que la IEEE identifica como el del nacimiento de la arquitectura de software en sentido estricto. Desde que surgieran tanto la disciplina como los estilos no sólo se han generalizado arquitecturas de cliente-servidor, sino que experimentaron su auge primero las configuraciones en varias capas y luego las basadas en componentes, en recursos (la Web) y en servicios (los Web services). Las metodologías de ingeniería también experimentaron evoluciones naturales y hasta revoluciones extremas, de modo que habrá que analizar en algún momento si ciertas prácticas que antes se daban por sentadas siguen o no en pie y en qué estado de salud se encuentran; y como el trabajo que se está leyendo se realiza en un momento en que son unos cuantos los que hablan de crisis, habrá que ver si las ideas en juego tienden a acentuar los problemas o constituyen una vía de solución.

En el estudio que sigue se analizará el repertorio de los estilos como un asunto de inherente interés descriptivo, taxonómico y heurístico. También se ahondará en su relación con patrones y usos como una forma de vincular la teoría con la práctica, aunque se sepa que sus mundos conceptuales difieren. Lo importante aquí es que la teoría del arquitecto derive en la práctica de la implementación de sistemas, que en ocasiones se presenta como dominada por un pragmatismo propio de *hackers* o fundamentalistas de la orientación a objetos, como si la empiria de la implementación no se coordinara con ningún orden de carácter teórico (aparte de los objetos, por supuesto), o como si el conocimiento experto que se pretende re-utilizar en el bajo nivel no pudiera dar cuenta de sus propias razones estructurales.

El marco en el que se desenvuelve el presente estudio es el de la estrategia de arquitectura de Microsoft, que en los últimos años ha hecho especial hincapié en las arquitecturas basadas en servicios y en patrones de diseño. El propósito es no sólo delimitar y tipificar un campo tan estrechamente relacionado con esa estrategia como pudiera ser el de las investigaciones académicas en arquitectura, sino establecer una nomenclatura de formas arquitectónicas que la documentación que ha comunicado la estrategia global de Microsoft no tuvo oportunidad de poner en foco, dando por sentadas la mayor parte de las cuestiones de naturaleza teórica y concentrándose en la práctica [Platt02]. Este texto servirá entonces como puente entre (1) una estrategia particular de arquitectura, (2) instancias de aplicación de estilos en productos, lenguajes y sistemas de Microsoft y (3) el dominio teórico de la arquitectura de software en general.

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. La idea dominante de esta sección del documento será proporcionar las definiciones denotativas de la clase: qué es un estilo; la de la sección siguiente, definir el valor de los ejemplares en el sistema de las posibilidades: sabiendo lo que son los estilos, definir cuántos hay y cuáles son.

La cuestión no es sólo clasificatoria. El hecho es que 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, de *refactoring*.

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 abarcativo 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. En todo caso, la definición de arquitectura a la que aquí habremos de atenernos es la misma que se establece en la documentación fundamental de la estrategia de arquitectura de Microsoft [Platt02].

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 [AG92] 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 es "estilos arquitectónicos" [RBP+91: 198], "arquitecturas comunes" [p. 199], "marcos de referencia arquitectónicos prototípicos" [p. 211] o "formas comunes" [p. 212] 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 [p. 211-216]. Algunas de estas clases, llamadas las cinco veces que se menciona su conjunto con cinco denominaciones diferentes, se pueden reconocer con los nombres siempre cambiados en los catálogos ulteriores de estilos de la arquitectura estructuralista [AG92] [Shaw94] [GS94] [BCK98] [Fie00]. 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 [PW92]. 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 [PB92].

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 }

Siguiendo con el razonamiento, encuentran tres clases de elementos: de *procesamiento* (que suministran la transformación de los datos), de *datos* (que contienen la información a procesar) y de *conexión* (por ejemplo, llamadas a procedimientos, mensajes, etc). La forma define las propiedades y las relaciones. La razón, finalmente, captura la motivación para la elección de los elementos y sus formas. Luego definen un estilo arquitectónico como una abstracción de tipos de elementos y aspectos formales a partir de diversas arquitecturas específicas. Un estilo arquitectónico encapsula decisiones esenciales sobre los elementos arquitectónicos y enfatiza restricciones importantes de los elementos y sus relaciones posibles. En énfasis en las restricciones (*constraints*) proporciona una visibilidad a ciertos aspectos de la arquitectura, de modo tal que la violación de esos aspectos y la insensibilidad hacia ellos se tornen más manifiestas. Un segundo énfasis, más circunstancial, concierne a la susceptibilidad de los estilos arquitectónicos a ser reutilizados. Llamativamente, en el estudio que inaugura la idea de los estilos arquitectónicos (y al margen de una referencia a arquitecturas secuenciales y paralelas) no aparece todavía ningún atisbo de tipología estilística. Llamo la atención respecto de que la definición minimalista propuesta por Perry y Wolf para los estilos ha sido incluso reducida a una que comprende únicamente elementos y relaciones para definir no sólo los estilos, sino la incumbencia global (el contenido y los límites) de la propia arquitectura de software [BCK98]. Señalo también que antes de Perry y Wolf ciertamente se hablaba de arquitectura, e incluso de arquitectura de software, para denotar la configuración o la organización de un sistema o de sus modelos; pero no se hablaba de arquitectura de software como un nivel de abstracción merecedor de una disciplina específica.

Algunos años más tarde Mary Shaw y Paul Clements [SC96] 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 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.

Tampoco aparece explícitamente en la caracterización de David Garlan, Andrew Kompanek, Ralph Melton y Robert Monroe [GKM+96], también de Carnegie Mellon, que definen el estilo como una entidad consistente en cuatro elementos: (1) Un vocabulario de elementos de diseño: componentes y conectores tales como tuberías, filtros, clientes, servidores, *parsers*, bases de datos, etcétera. (2) Reglas de diseño o restricciones que determinan las composiciones permitidas de esos elementos. (3) Una interpretación semántica que proporciona significados precisos a las composiciones. (4) Análisis susceptibles de practicarse sobre los sistemas construidos en un estilo, por ejemplo análisis de disponibilidad para estilos basados en procesamiento en tiempo real, o detección de abrazos mortales para modelos cliente-servidor. En un artículo conexo, Garlan, Allen y Ockerbloom establecen dos grandes categorías de estilos: (1) Idiomas y patrones, que incluye estructuras globales de organización, como ser sistemas en capas, tubería-filtros, cliente-servidor, pizarras, MVC, etcétera. (2) Modelos de referencia, que son organizaciones de sistemas que prescriben configuraciones específicas de dominio o áreas de aplicación, a menudo parametrizadas [GAO94].

En un ensayo de 1996 en el que aportan fundamentos para una caracterización formal de las conexiones arquitectónicas, Robert Allen y David Garlan [AG96] 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 [KK99]. 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 [FT02].

Con el tiempo, casi nadie propone nuevas definiciones de estilo, sino que comienzan a repetirse las definiciones consagradas. Llegado el siglo XXI, Roy Thomas Fielding [Fie00] sintetiza la definición de estilo pleonásticamente, diciendo que un estilo arquitectónico es un conjunto coordinado de restricciones arquitectónicas que restringe los roles/rasgos de los elementos arquitectónicos y las relaciones permitidas entre esos elementos dentro de la arquitectura que se conforma a ese estilo. Mientras que existen innumerables definiciones alternativas de arquitectura de software o de Web service, las definiciones de estilo ostentan muchos menos grados de libertad.

Más allá de las idiosincracias de expresión, con las definiciones vertidas hasta este punto queda claro que los estilos son entidades que ocurren en un nivel sumamente abstracto, puramente arquitectónico, que no coincide ni con la fase de *análisis* propuesta por la temprana metodología de modelado orientada a objetos (aunque sí un poco con la de diseño), ni con lo que más tarde se definirían como *paradigmas* de arquitectura, ni con los *patrones* arquitectónicos. Propongo analizar estas tres discordancias una por una:

El análisis de la metodología de objetos, tal como se enuncia en [RBP+91] está muy cerca del requerimiento y la percepción de un usuario o cliente técnicamente agnóstico, un protagonista que en el terreno de los estilos no juega ningún papel. En arquitectura de software, los estilos surgen de la experiencia que el arquitecto posee; de ningún modo vienen impuestos de manera explícita en lo que el cliente le pide.

Los paradigmas como la arquitectura orientada a objetos (p.ej. CORBA), a componentes (COM, JavaBeans, EJB, CORBA Component Model) o a servicios, tal como se los define en [WF04], se relacionan con tecnologías particulares de implementación, un elemento de juicio que en el campo de los estilos se trata a un nivel más genérico y distante, si es que se llega a tratar alguna vez. Dependiendo de la clasificación que se trate, estos paradigmas tipifican más bien como sub-estilos de estilos más englobantes (*peer-to-peer*, distribuidos, etc) o encarnan la forma de implementación de otros estilos cualesquiera.

Los patrones arquitectónicos, por su parte, se han materializado con referencia a lenguajes y paradigmas también específicos de desarrollo, mientras que ningún estilo presupone o establece preceptivas al respecto. Si hay algún código en las inmediaciones de un estilo, será código del lenguaje de descripción arquitectónica o del lenguaje de modelado; de ninguna manera será código de lenguaje de programación. Lo mismo en

cuanto a las representaciones visuales: los estilos se describen mediante simples cajas y líneas, mientras que los patrones suelen representarse en UML [Lar03].

En las definiciones consideradas, y a pesar de algunas excepciones, como las enumeraciones de Taylor y Medvidovic [TMA+95] o Mehta y Medvidovic [MM04], se percibe una unanimidad que no suele encontrarse con frecuencia en otros territorios de la arquitectura de software. La idea de estilo arquitectónico ha sido, en rigor, uno de los conceptos mejor consensuados de toda la profesión, quizá por el hecho de ser también uno de los más simples. Pero aunque posee un núcleo invariante, las discrepancias comienzan a manifestarse cuando se trata (*a*) de enumerar *todos* los estilos existentes y (*b*) de suministrar la articulación matricial de lo que (pensándolo bien) constituye, a nivel arquitectónico, una ambiciosa clasificación de todos los tipos de aplicaciones posibles. Predeciblemente, la disyuntiva (*b*) demostrará ser más aguda y rebelde que el dilema (*a*), porque, como lo estableció Georg Cantor, hay más clases de cosas que cosas, aún cuando las cosas sean infinitas. A la larga, las enumeraciones de estilos de grano más fino que se publicaron históricamente contienen todos los ejemplares de las colecciones de grano más grueso, pero nadie ordenó el mundo dos veces de la misma manera.

Para dar un ejemplo que se sitúa en el límite del escándalo, el estilo cliente-servidor ha sido clasificado ya sea como variante del estilo basado en datos [Shaw94], como estilo de *flujo* de datos [And91] [SC96], como arquitectura distribuida [GS94], como estilo jerárquico [Fie00], como miembro del estilo de máquinas virtuales [variante de BCK98], como estilo orientado a objetos [Richard Upchurch], como estilo de llamada-y-retorno [BCK98] o como estilo independiente [TMA+95]. Mientras las clasificaciones jerárquicas más complejas incluyen como mucho seis clases básicas de estilos, la comunidad de los arquitectos se las ha ingeniado para que un ejemplar pertenezca alternativamente a ocho. Los grandes *frameworks* que después revisaremos (TOGAF, RM-ODP, 4+1, IEEE) homologan los estilos como concepto, pero ninguno se atreve a brindar una taxonomía exhaustiva de referencia. Hay, entonces, más clasificaciones divergentes de estilos que estilos de arquitectura, cosa notable para números tan pequeños, pero susceptible de esperarse en razón de la diversidad de puntos de vista.

Catálogos 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 hayan descripto los estilos individuales en la sección siguiente, se dedicará otro apartado para examinar de qué forma determinadas percepciones ligadas a dominios derivaron en otras articulaciones como las que han propuesto Roy Fielding [Fie00] y Gregory Andrews [And91].

¿Cuántos y cuáles son los estilos, entonces? En un estudio comparativo de los estilos, Mary Shaw [Shaw94] considera los siguientes, mezclando referencias a las mismas entidades a veces en términos de “arquitecturas”, otras invocando “modelos de diseño”:

Arquitecturas orientadas a objeto

Arquitecturas basadas en estados

Arquitecturas de flujo de datos: Arquitecturas de control de realimentación

Arquitecturas de tiempo real

Modelo de diseño de descomposición funcional

Modelo de diseño orientado por eventos

Modelo de diseño de control de procesos

Modelo de diseño de tabla de decisión

Modelo de diseño de estructura de datos

El mismo año, Mary Shaw, junto con David Garlan [GS94], propone una taxonomía diferente, en la que se entremezclan lo que antes llamaba "arquitecturas" con los "modelos de diseño":

Tubería-filtros

Organización de abstracción de datos y orientación a objetos

Invocación implícita, basada en eventos

Sistemas en capas

Repositorios

Intérpretes orientados por tablas

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.

Organizaciones programa principal / subrutina.

Arquitecturas de software específicas de dominio

Sistemas de transición de estado

Sistemas de procesos de control

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* [BMR+96] Buschmann, Meunier, Rohnert, Sommerlad y Stal enumeran estos patrones:

Del fango a la estructura

- Capas

Tubería-filtros

Pizarra

1. Sistemas distribuidos
   - *Broker* (p. ej. CORBA, DCOM, la World Wide Web)
2. Sistemas interactivos
   - *Model-View-Controller*

Presentation-Abstraction-Control

3. Sistemas adaptables
   - *Reflection* (metanivel 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 [BCK98] se proporciona una sistematización de clases de estilo en cinco grupos:

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

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

Componentes independientes (dominado por patrones de comunicación entre procesos independientes, casi siempre concurrentes)

- Sistemas orientados por eventos

Procesos de comunicación

Centrados en datos (dominado por un almacenamiento central complejo, manipulado por computaciones independientes)

- Repositorio

Pizarra

1. Máquina virtual (caracterizado por la traducción de una instrucción en alguna otra)
    - Intérprete

Es interesante el hecho que se proporcionen sólo cinco clases abarcativas; no he tenido oportunidad de examinar si todos los estilos propuestos encajan en ellas, lo que las clasificaciones posteriores parecen desmentir. Mientras que en los inicios de la arquitectura de software se alentaba la idea de que todas las estructuras posibles en diseño de software serían susceptibles de reducirse a una media docena de estilos básicos, lo que en realidad sucedió fue que en los comienzos del siglo XXI se alcanza una fase barroca y las enumeraciones de estilos se tornan más y más detalladas y exhaustivas. Considerando solamente los estilos que contemplan alguna forma de distribución o topología de red, Roy Fielding [Fie00] establece la siguiente taxonomía:

Estilos de flujo de datos

Tubería-filtros

Tubería-filtros uniforme

Estilos de replicación

Repositorio replicado

Cache

Estilos jerárquicos

Cliente-servidor

Sistemas en capas y Cliente-servidor en capas

Cliente-Servidor sin estado

Cliente-servidor con *cache* en cliente

Cliente-servidor con *cache* en cliente y servidor sin estado

Sesión remota

Acceso a datos remoto

Estilos de código móvil

Máquina virtual

Evaluación remota

Código a demanda

Código a demanda en capas

Agente móvil

Estilos peer-to-peer

Integración basada en eventos

C2

Objetos distribuidos

Objetos distribuidos brokered

Transferencia de estado representacional (REST)

Cuando las clasificaciones de estilos se tornan copiosas, daría la impresión que algunos sub-estilos se introducen en el cuadro por ser combinatoriamente posibles, y no tanto porque existan importantes implementaciones de referencia, o porque sea técnicamente necesario. En un proyecto que se ha desarrollado en China hacia el año 2001, he podido encontrar una clasificación que si bien no pretende ser totalizadora, agrupa los estilos de manera peculiar, agregando una clase no prevista por Bass, Clements y Kazman [BCK98], llamando a las restantes de la misma manera, pero practicando enroques posicionales de algunos ejemplares:

Sistemas de flujo de datos, incluyendo

- Secuencial por lotes

Tubería y filtro

Sistemas de invocación y retorno (*call-and-return*), incluyendo

Programa principal y sub-rutina

- Sistemas orientados a objeto

Niveles jerárquicos

3. Componentes independientes, incluyendo

Procesos comunicantes

Sistemas basados en eventos

4. Máquinas virtuales, incluyendo

Intérpretes

Sistemas basados en reglas

Cliente-servidor

5. Sistemas centrados en datos, incluyendo

Bases de datos

Sistemas de hipertexto

Pizarras

6. Paradigmas de procesos de control

En un documento anónimo compilado por Richard Upchurch, del Departmento de Ciencias de la Computación y Información de la Universidad de Massachusetts en Dartmouth se proporciona una "lista de posibles estilos arquitectónicos", incluyendo:

Programa principal y subrutinas

Estilos jerárquicos

Orientado a objetos (cliente-servidor)

Procesos de comunicación

Tubería y filtros

Intérpretes

Sistemas de bases de datos transaccionales

Sistemas de pizarra

Software bus

Sistemas expertos

Paso de mensajes

Amo-esclavo

Asincrónico / sincrónico paralelo

Sistemas de tiempo real

Arquitecturas específicas de dominio

Sistemas en red, distribuidos

Arquitecturas heterogéneas

Esta lista de lavandería puede que denote la transición hacia las listas enciclopédicas, que por lo común suelen ser poco cuidadosas en cuanto a mantener en claro los criterios de articulación de la taxonomía.

En 2001, David Garlan [Gar01], uno de los fundadores del campo, proporciona una lista más articulada:

Flujo de datos

- Secuencial en lotes

Red de flujo de datos (tuberías y filtros)

Bucle de control cerrado

Llamada y Retorno

- Programa principal / subrutinas

Ocultamiento de información (ADT, objeto, cliente/servidor elemental)

1. Procesos interactivos
   - Procesos comunicantes

Sistemas de eventos (invocación implícita, eventos puros)

2. Repositorio Orientado a Datos

Bases de datos transaccionales (cliente/servidor genuino)

Pizarra

Compilador moderno

3. Datos Compartidos
   - Documentos compuestos

Hipertexto

Fortran COMMON

Procesos LW

4. Jerárquicos

- En capas (intérpretes)

También señala los inconvenientes de la vida real que embarran el terreno de las taxonomías: los estilos casi siempre se usan combinados; cada capa o componente puede ser internamente de un estilo diferente al de la totalidad; muchos estilos se encuentran ligados a dominios específicos, o a líneas de producto particulares.

Cada tanto se encuentran también presentaciones relativas a estilos arquitectónicos que no coinciden con los usos mayoritariamente aceptados del concepto. Un ejemplo sería el artículo de Taylor, Medvidovic y otros en que se formuló la presentación pública de Chiron-2 [TMA+95]. Una vez más, sin intención de elaborar un catálogo exhaustivo, los autores identifican estilos tales como tubería-filtro, arquitecturas de pizarra (propias de la Inteligencia Artificial), el estilo cliente-servidor, el modelo de *callback* (que opera bajo control de la interfaz de usuario), el modelo-vista-controlador (explotado comúnmente en aplicaciones de Smalltalk), el estilo Arch y su meta-modelo asociado, y el estilo C2. Otra taxonomía excéntrica se postula en [MMP00], donde se enumeran algunos "estilos motivados por conectores de software" como tubería-filtro, alimentación de datos en tiempo real, arquitectura definida por eventos, estilo basado en mensajes y estilo de flujo de datos.

En los últimos cuatro o cinco años se han realizado esfuerzos para definir los estilos de una manera más formal, empleando ADLs como Aesop o Wright, o notaciones en lenguajes de especificación como Z o lenguajes con semántica formal como CHAM. Le Metayer [LeM98], por ejemplo, propone formalizar los estilos y otros conceptos análogos en términos de gramática de grafos, enfatizando la geometría de las arquitecturas como un objeto independiente. Bernardo, Ciancarini y Donatiello han formalizado los "tipos" arquitectónicos empleando álgebra de procesos [BCD02]. En nuestro estudio de los lenguajes de descripción de arquitectura (ADLs) hemos referido algunos lenguajes

capaces de especificar estilos, implícita o explícitamente, ejemplificando la definición de uno de ellos en la mayoría de los ADLs disponibles y en alguno que otro lenguaje formal de especificación.

Descripción de los estilos

Los estilos que habrán de describirse a continuación no aspiran a ser todos los que se han propuesto, sino apenas los más representativos y vigentes. De más está decir que, como se ha visto, la agrupación de estilos y sub-estilos es susceptible de realizarse de múltiples formas, conforme a los criterios que se apliquen en la constitución de los ejemplares. No se ha de rendir cuentas aquí por la congruencia de la clasificación (nadie ha hecho lo propio con las suyas), porque cada vez que se la ha revisado en modo *outline*, se cedió a la tentación de cambiar su estructura, la cual seguirá sufriendo metamorfosis después de despachado el artículo. Se podrá apreciar que, en consonancia con los usos de la especialidad, la caracterización de los estilos no constituye un reflejo pormenorizado de los detalles de sus estructuras, sino una visión deliberadamente sucinta y genérica que destaca sus valores esenciales y sus rasgos distintivos. Entre acumular pormenores que podrían recabarse en otras fuentes y mantener el caudal descriptivo de cada estilo en una magnitud que facilite su comparación rápida con los demás se ha escogido, saussureanamente, lo segundo.

En cuanto a las formas de representación esquemática, se ha optado por reproducir el estilo gráfico característico de la literatura principal del género, el cual es deliberadamente informal y minimalista, en lugar de aplicar una notación formal o más elaborada. La notación se establecerá entonces en términos de lo que Shaw y Clements llaman “boxology” [SC96]. Huelga decir que la descripción de los estilos puede hacerse también en términos de lenguajes descriptivos de arquitectura (ADLs) y las respectivas notaciones de las herramientas que se asocian con ellos, según puede verse en un estudio separado [Rey04b]. Resta anticipar que algunas especies de software conspicuas en la práctica no aparecen en los catálogos usuales de estilo. La ausencia más notoria concierne a los sistemas de *workflow*, que seguramente heredan el prejuicio ancestral de los ingenieros y arquitectos más refinados en contra de los diagramas de flujo y las abstracciones procesuales. Fred Brooks, por ejemplo, considera el diagrama de flujo como una abstracción muy pobre de la estructura de un sistema [Bro75] [Bro86].

Estilos de Flujo de Datos

Esta familia de estilos enfatiza la reutilización y la modificabilidad. Es apropiada para sistemas que implementan transformaciones de datos en pasos sucesivos. Ejemplares de la misma serían las arquitecturas de tubería-filtros y las de proceso secuencial en lote.

Tubería y 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 [AG92] y el que puede identificarse topológica, procesual y taxonómicamente con menor ambigüedad. Históricamente el se relaciona con las redes de proceso descriptas por Kahn hacia 1974 y con el proceso secuenciales comunicantes (CSP) ideados por Tony Hoare cuatro años más tarde. Ha prevalecido el nombre de tubería-filtros por más que se sabe muy bien que los llamados filtros no realizan forzosamente tareas de filtrado, como ser eliminación de campos o registros, sino que ejecutan formas variables de transformación, una de las cuales puede ser el filtrado. En uno de los trabajos recientes más completos sobre este estilo, Ernst-Erich Doberkat [Dob03] lo define en estos términos.

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. […] Debido a su simplicidad y su facilidad para captar una funcionalidad, es una arquitectura mascota cada vez que se trata de demostrar ideas sobre la formalización del espacio de diseño arquitectónico, igual que el tipo de datos `stack` lo fue en las especificaciones algebraicas o en los tipos de datos abstractos.

El sistema tubería-filtros se percibe como una serie de transformaciones sobre sucesivas piezas de los datos de entrada. Los datos entran al sistema y fluyen a través de los componentes. En el estilo secuencial por lotes (*batch sequential*) los componentes son programas independientes; el supuesto es que cada paso se ejecuta hasta completarse antes que se inicie el paso siguiente. Garlan y Shaw sostienen que la variante por lotes es un caso degenerado del estilo, en el cual las tuberías se han vuelto residuales [SG94].

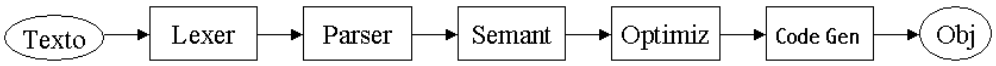

Fig. 1 - Compilador en tubería-filtro

La aplicación típica del estilo es un procesamiento clásico de datos: el cliente hace un requerimiento; el requerimiento se valida; un Web Service toma el objeto de la base de datos; se lo convierte a HTML; se efectúa la representación en pantalla. El estilo tubería-filtros propiamente dicho enfatiza la transformación incremental de los datos por sucesivos componentes. El caso típico es la familia UNIX de Sistemas operativos, pero hay otros ejemplos como la secuencia de procesos de los compiladores (sobre todo los más antiguos), el tratamiento de documentos XML como ráfaga en SAX, ciertos mecanismos de determinados motores de servidores de bases de datos, algunos procesos de *workflow* y subconjuntos de piezas encadenadas en productos tales como Microsoft Commerce Server. Puede verse, por ejemplo, una fina caracterización de conceptos de programación relativos al estilo de línea de tubería en la documentación del Software Development Kit de ese producto (http://msdn.microsoft.com/library/en-us/comsrv2k/htm/cs_sp_pipelineobj_woce.asp). Aunque la línea de tuberías no se define

allí como estilo, se la describe como un "framework de software extensible" que establece y vincula una o más etapas de un proceso de negocios, ejecutándolas en secuencia hasta completar una tarea determinada. Por ejemplo, el Order Processing Pipeline proporciona la secuencia de pasos necesaria para procesar las compras en un sitio de Web. En la primera etapa, se obtiene la información del producto de la base de datos de catálogo; en la siguiente, se procesa la dirección del comprador; otra etapa resuelve la modalidad de pago; una etapa ulterior confecciona la factura y otra más realiza el envío del pedido. Cada etapa de la tarea representa una categoría de trabajo.

En la estrategia de arquitectura de Microsoft, que se encuentra mayormente orientada a servicios, el modelo tubería-filtros tiene sin embargo su lugar de referencia. No está caracterizado literalmente como estilo, por cierto, pero figura en la documentación de patrones y prácticas como uno de los "patrones más comúnmente usados para los componentes de negocios", junto con los patrones de eventos y los de *workflow* [MS02a:45-50]. La presentación de las características del estilo en esta literatura coincide puntualmente con las ideas dominantes en el campo de los estilos arquitectónicos, lo mismo que las recomendaciones prácticas para su uso [GAO94] [Gar95] [MKM+97] [SC96] [Shaw94]. La documentación establece, en efecto, que 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).

Igualmente, se reconocen como 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.

Aunque Garlan y Shaw señalan que el estilo tiene una plétora de "devotos seguidores religiosos" que afirman que es útil en cualquier contexto [SG94], hace tiempo que se conocen sus 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.

Una desventaja adicional referida en la literatura sobre estilos concierne a que 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 (por ejemplo, el control de corrección de un objeto de fecha).

Históricamente, los primeros compiladores operaban conforme a un estilo de tubería y filtro bastante puro, en ocasiones en variantes de proceso por lotes. A medida que los compiladores se tornaron más sofisticados, se fueron añadiendo elementos tales como tablas de símbolos, generalmente compartidas por varios filtros. El añadido de formas intermedias de representación, gramáticas de atributo, árboles de *parsing* de atributos, compilaciones convergentes a formatos intermedios (como los compiladores que generan formato de lenguaje intermedio MSIL en el .NET Framework a partir de distintos lenguajes fuentes) y otras complicaciones y añadiduras, fueron haciendo que el modelo de tubo secuencial llegara a ser inadecuado para representar estos procesos, siendo preferible optar por otros estilos, como el de repositorio.

En nuestro documento sobre los lenguajes de descripción arquitectónicos (ADLs), hemos ejemplificado el estilo utilizando el lenguaje de especificación CHAM, aplicado al encadenamiento de pasos del compilador de Lisp que viene incluido como ejemplo en el Software Development Kit de Microsoft .NET Framework.

Roy Fielding considera a tubería-filtros como una de las variantes del estilo más genérico de flujo de datos. La otra variante sería tubería y filtro uniforme. En ella se agrega la restricción de que todos los filtros deben poseer la misma interfaz. El ejemplo primario se encuentra en el sistema operativo Unix, donde los procesos de filtro tienen un flujo de entrada de caracteres (`stdin`) y dos flujos de salida (`stdout` y `stderr`) [Fie00]. La literatura de estilos y la de patrones arquitectónicos y de diseño abunda en variaciones del modelo básico: líneas de tuberías, esquemas de apertura en abanico, tuberías acíclicas. En la documentación de Microsoft se ha dado tratamiento explícito a patrones tales como el filtro de intercepción (*intercepting filter*) y las cadenas de filtros componibles [MS04b] que vendrían a ser derivaciones concretas del estilo abstracto en las vistas ligadas al desarrollo de una solución.

Estilos Centrados en Datos

Esta familia de estilos enfatiza la integrabilidad de los datos. Se estima apropiada para sistemas que se fundan en acceso y actualización de datos en estructuras de almacenamiento. Sub-estilos característicos de la familia serían los repositorios, las bases de datos, las arquitecturas basadas en hipertextos y las arquitecturas de pizarra.

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 [SG96]. En base a esta distinción se han definidos dos subcategorías principales del estilo:

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).

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.

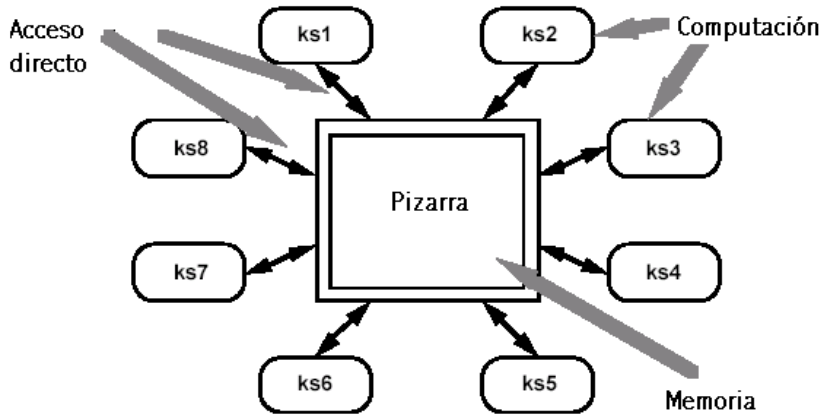

Fig. 2 - Pizarra [basado en GS94]

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. Algunas arquitecturas de compiladores que suelen presentarse como representativas del estilo tubería-filtros, se podrían representar mejor como propias del estilo de pizarra, dado que muchos compiladores contemporáneos operan en base a información compartida tal como tablas de símbolos, árboles sintácticos abstractos (AST), etcétera. Así como los estilos lineales de tubería-filtros suelen evolucionar hacia (o ser comprendidos mejor como) estilos de pizarra o repositorio, éstos suelen hacer *morphing* a estilos de máquinas virtuales o intérpretes [GS94].

El documento clásico que describe el estilo, *Blackboard Systems*, de H. Penny Nii [Nii86], bien conocido en Inteligencia Artificial, es en rigor anterior en seis años al surgimiento de la idea de estilos en arquitectura de software. Los estilos de pizarra no son sólo una curiosidad histórica; por el contrario, se los utiliza en exploraciones recientes de

inteligencia artificial distribuida o cooperativa, en robótica, en modelos multi-agentes, en programación evolutiva, en gramáticas complejas, en modelos de crecimiento afines a los L-Systems de Lindenmayer, etc. 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.

Al comienzo del proceso de resolución, se establece el problema en la pizarra. Las fuentes tratan de resolverlo cambiando el estado. La única forma en que se comunican entre sí es a través de la pizarra. Finalmente, si de la cooperación resulta una solución adecuada, ésta aparece en la pizarra como paso final.

En un desarrollo colateral, la relación entre este modelo de resolución de problemas y los lenguajes formales fue establecida hacia 1989 por los húngaros E. Csuhaj-Varjú y J. Kelemen. En su esquema, las fuentes de conocimiento corresponden a gramáticas, el cambio del estado de la pizarra a la re-escritura de formas secuenciales, y la estrategia es representada por modelos de derivación de modo que la solución corresponda a una frase terminal. Estas correspondencias han sido estudiadas primariamente en modelos de programación evolutiva, modelos ecológicos, ecogramáticas, sistemas emergentes, caos determinista, algoritmos genéticos y vida artificial, y en el desarrollo de meta-heurísticas del tipo de la simulación de templado o la búsqueda tabú, temas todos que son demasiado especializados para tratar detalladamente en este contexto [Rey04a]. Últimamente se está hablando mucho de agentes distribuidos, pero más en el sentido de entidades distribuidas en los cánones de la programación orientada a objetos, que en el de los sistemas complejos y emergentes.

A mi juicio, el estilo de pizarra tiene pleno sentido si tanto los agentes (o las fuentes de conocimiento) como la pizarra se entienden en términos virtuales y genéricos, como clases que son susceptibles de instanciarse en diversas variedades de objetos computacionales. De ser así, se podría incluir en este estilo un inmenso repertorio de aplicaciones de optimización y búsqueda en programación genética y evolutiva que de otro modo no encontraría un estilo en el cual encuadrarse. En un programa genético, efectivamente, una población (que vendría a ser homóloga a la fuente) evoluciona produciendo soluciones que se contrastan contra un criterio de adecuación (que sería la pizarra). La estrategia sería aquí el algoritmo genético propiamente dicho (mutaciones, *crossover*, reproducción, evaluación de *fitness*, selección de los más aptos, repetición del ciclo). Considero que todas las arquitecturas basadas en elementos autónomos pero globalmente orientadas a una meta de convergencia hacia valores u objetivos (como las

redes neuronales, los modelos evolutivos y meméticos, los autómatas celulares y las redes booleanas aleatorias) son susceptibles de encuadrarse en la misma variedad estilística.

La estrategia de arquitectura de Microsoft no hace demasiado hincapié en el estilo de pizarra y las referencias a repositorios son más bien convencionales o implícitas; tampoco lo hacen las estrategias mayores alternativas de la industria. Pero es de esperarse que el florecimiento de las arquitecturas basadas en agentes "inteligentes" o autónomos resulte, más temprano que tarde, en un redescubrimiento de este estilo, hasta ahora marginal, raro, heterodoxo y desganadamente descriptos por los arquitectos.

Estilos de Llamada y Retorno

Esta familia de estilos enfatiza la modificabilidad y la escalabilidad. Son los estilos más generalizados en sistemas en gran escala. Miembros de la familia son las arquitecturas de programa principal y subrutina, los sistemas basados en llamadas a procedimientos remotos, los sistemas orientados a objeto y los sistemas jerárquicos en capas.

Model-View-Controller (MVC)

Reconocido como estilo arquitectónico por Taylor y Medvidovic [TMA+95], muy rara vez mencionado en los *surveys* estilísticos usuales, considerado una micro-arquitectura por Robert Allen y David Garlan [AG97], el MVC ha sido propio de las aplicaciones en Smalltalk por lo menos desde 1992, antes que se generalizaran las arquitecturas en capas múltiples. En ocasiones se lo define más bien como un patrón de diseño o como práctica recurrente, y en estos términos es referido en el marco de la estrategia arquitectónica de Microsoft. En la documentación correspondiente es tratado a veces en términos de un estilo decididamente abstracto [MS03a] y otras como patrón de aplicación ligado a una implementación específica en Visual C++ o en ASP.NET [MS03b]. Buschmann y otros lo consideran un patrón correspondiente al estilo de los sistemas interactivos [BMR+96].

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 la performance. Sin embargo, esta idea es antagónica al hecho de que la interfaz suele cambiar, o acostumbra depender de distintas clases de dispositivos (clientes ricos, *browsers*, PDAs); la programación de interfaces de HTML, además, requiere habilidades muy distintas de la programación de lógica de negocios. Otro problema es que las aplicaciones tienden a incorporar lógica de negocios que van más allá de la transmisión de datos.

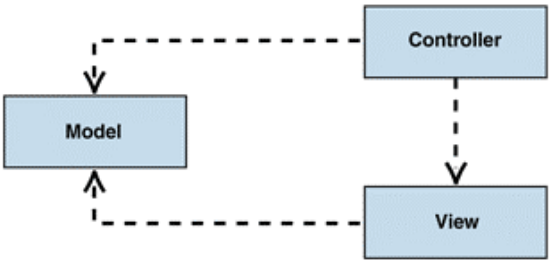

Fig. 3 - Model-View-Controller [según MS03a]

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 [Bur92]:

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).

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.

Tanto la vista como el controlador dependen del modelo, el cual no depende de las otras clases. Esta separación permite construir y probar el modelo independientemente de la representación visual. La separación entre vista y controlador puede ser secundaria en aplicaciones de clientes ricos y, de hecho, muchos *frameworks* de interfaz implementan ambos roles en un solo objeto. En aplicaciones de Web, por otra parte, la separación entre la vista (el browser) y el controlador (los componentes del lado del servidor que manejan los requerimientos de HTTP) está mucho más taxativamente definida.

Entre las ventajas del estilo señaladas en la documentación de Patterns & Practices de Microsoft están las siguientes:

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. Dado que el modelo no depende de las vistas, agregar nuevas opciones de presentación generalmente no afecta al modelo. Este patrón sentó las bases para especializaciones ulteriores, tales como `Page Controller` y `Front Controller`.

Entre las desventajas, se han señalado**:**

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. En rigor, la configuración basada en eventos de dicha interfaz corresponde a un estilo particular (arquitectura basada en eventos) que aquí se examina por separado.

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. Hace pocos años sucedía que algunas vistas, tales como las pantallas gráficas, involucraban más tiempo para plasmar el dibujo que el que demandaban los nuevos requerimientos de actualización.

Este estilo o sub-estilo ha sido ampliamente tratado en la documentación de arquitectura de Microsoft. Una visión sistemática del mismo, tratado como patrón de solución, puede consultarse en http://msdn.microsoft.com/practices/type/Patterns/Enterprise/DesMVC/. Hay documentación separada respecto de la implementación del patrón en ASP.NET en http://msdn.microsoft.com/practices/type/Patterns/Enterprise/ImpMVCinASP/, y presentaciones en detalle de otros patrones asociados. Los Mobile Web Forms que forman parte del repertorio de interfaces de Visual Studio .NET apropiadas para las llamadas Mobile Web Applications, implementan (en conjunción con las prestaciones adaptativas de ASP.NET) un ejemplo claro de un elegante modelo derivado de MVC.

Arquitecturas en Capas

Los sistemas o arquitecturas en capas constituyen uno de los estilos que aparecen con mayor frecuencia mencionados como categorías mayores del catálogo, o, por el contrario, como una de las posibles encarnaciones de algún estilo más envolvente. En [GS94] Garlan y Shaw definen el estilo en capas como una organización jerárquica tal que cada capa proporciona servicios a la capa inmediatamente superior y se sirve de las prestaciones que le brinda la inmediatamente inferior. Instrumentan así una vieja idea de organización estratigráfica que se remonta a las concepciones formuladas por el patriarca Edsger Dijkstra en la década de 1960, largamente explotada en los años subsiguientes. En algunos ejemplares, las capas internas están ocultas a todas las demás, menos para las capas externas adyacentes, y excepto para funciones puntuales de exportación; en estos sistemas, los componentes implementan máquinas virtuales en alguna de las capas de la jerarquía. En otros sistemas, las capas pueden ser sólo parcialmente opacas. En la práctica, las capas suelen ser entidades complejas, compuestas de varios paquetes o subsistemas. El uso de arquitecturas en capas, explícitas o implícitas, es frecuentísimo; solamente en *Pattern Almanac 2000* [Ris00] hay cerca de cien patrones que son variantes del patrón básico de capas. Patrones de uso común relativos al estilo son *Façade*, *Adapter*, *Bridge* y *Strategy* [Gof95] [MS04c].

En un estilo en capas, los conectores se definen mediante los protocolos que determinan las formas de la interacción. Los diagramas de sistemas clásicos en capas dibujaban las capas en adyacencia, sin conectores, flechas ni interfaces; en algunos casos se suele representar la naturaleza jerárquica del sistema en forma de círculos concéntricos [GS94: 11]. Las restricciones topológicas del estilo pueden incluir una limitación, más o menos rigurosa, que exige a cada capa operar sólo con capas adyacentes, y a los elementos de una capa entenderse sólo con otros elementos de la misma; se supone que si esta exigencia se relaja, el estilo deja de ser puro y pierde algo de su capacidad heurística [MS04c]; también se pierde, naturalmente, la posibilidad de reemplazar de cuajo una capa sin afectar a las restantes, disminuye la flexibilidad del conjunto y se complica su mantenimiento. Las formas más rígidas no admiten ni siquiera *pass-through*: cada capa debe hacer algo, siempre. En la literatura especializada hay multitud de argumentos a favor y en contra del rigor de esta clase de prescripciones. A veces se argumenta que el cruce superfluo de muchos niveles involucra eventuales degradaciones de performance; pero muchas más veces se sacrifica la pureza de la arquitectura en capas precisamente para mejorarla: colocando, por ejemplo, reglas de negocios en los procedimientos almacenados de las bases de datos, o articulando instrucciones de consulta en la capa de la interface del usuario.

Casos representativos de este estilo son muchos de los protocolos de comunicación en capas. En ellos cada capa proporciona un sustrato para la comunicación a algún nivel de abstracción, y los niveles más bajos suelen estar asociados con conexiones de hardware. El ejemplo más característico es el modelo OSI con los siete niveles que todo el mundo recuerda haber aprendido de memoria en la escuela: nivel físico, vínculo de datos, red, transporte, sesión, presentación y aplicación. El estilo también se encuentra en forma más o menos pura en arquitecturas de bases de datos y sistemas operativos, así como en las especificaciones relacionadas con XML. Don Box, Aaron Skonnard y John Lam, por ejemplo, suministran este mapa en capas de la tecnología de XML [BSL00].

Abstracto	Clases y objetos	Específico de aplicación
	Tipos e instancias	XML Schemas (Metadata)
↑	Elementos estructurales	XML Information Set (InfoSet)
	Elementos y atributos	XML 1.0 + Namespaces
	Entidades y documentos	XML 1.0
	Archivos y paquetes	Específico de Sistema operativo o Protocolo
Concreto	Sectores y bitstreams	Específico del hardware

Tabla 1 - Capas de XML

Aunque un documento XML se percibe como un texto plano humanamente legible (tan plano que puede ser tratado como ráfaga mediante Sax), su estructura involucra un conjunto complejo de especificaciones jerárquicas estratificadas: el proceso que trate con

entidades como `external identifier` o `processing instructions`, no necesita armar caracteres de marcación componiendo bits, ni implementar instrucciones primitivas de protocolo de red, ni lidiar con el proceso básico de localizar un archivo en un directorio del disco.

La presencia del estilo en una tecnología tan envolvente como hoy en día es XML, así como en otros enclaves tecnológicos, induce a que subrayemos aquí su importancia. A nivel arquitectónico, se puede tratar con XML en diversos planos de abstracción. Si un proyecto determinado implementa de manera explícita operaciones de *parsing* o tokenización de XML crudo en una solución de negocios, ello puede significar que se está operando a un nivel indebido de abstracción, pues esas operaciones ya deberían estar resueltas a niveles inferiores ya sea por el sistema operativo, por bibliotecas de clases, por modelos de objeto, por interfaces o por entornos de programación.

El número mínimo de capas es obviamente dos, y en ese umbral la literatura arquitectónica sitúa a veces al sub-estilo cliente-servidor como el modelo arquetípico del estilo de capas y el que se encuentra con mayor frecuencia en las aplicaciones en red. Este modelo particular dudosamente necesite descripción, pero de todos modos aquí va: Un componente servidor, que ofrece ciertos servicios, escucha que algún otro componente requiera uno; un componente cliente solicita ese servicio al servidor a través de un conector. El servidor ejecuta el requerimiento (o lo rechaza) y devuelve una respuesta.

La descripción, que a fuerza de ser elemental pudo haber incomodado al lector arquitecto de software, subraya empero tres características no triviales: una, que cuando se habla de estilos todas las entidades unitarias son componentes, aunque no lo sean en términos de COM o JavaBeans; dos, que en este contexto todas las relaciones asumen la forma de conectores, aún cuando en la vida real la relación de un cliente con un servidor pueda llegar a ser no-conectada; tres, que el plano de abstracción del discurso estilístico es tal que nadie se acuerda de los *drivers* ni practica diferencias entre modelos alternativos de programación. Que la relación entre clientes y servidores sea continua o discontinua y que algunas de las partes guarden memoria de estado y cursores o dejen de hacerlo es secundario, aunque algunos autores sensibles a los matices han previsto sub-estilos distintos que corresponden a estos casos [Fie00]. Dada la popularidad de las tecnologías de bases de datos conformes a este modelo, la literatura sobre cliente-servidor ha llegado a ser inabarcable. La bibliografía sobre cliente-servidor en términos de estilo arquitectónico, sin embargo, es comparativamente modesta.

Las ventajas del estilo en capas son obvias. Primero que nada, el estilo soporta un diseño basado en niveles de abstracción crecientes, lo cual a su vez permite a los implementadores la partición de un problema complejo en una secuencia de pasos incrementales. En segundo lugar, el estilo admite muy naturalmente optimizaciones y refinamientos. En tercer lugar, proporciona amplia reutilización. Al igual que los tipos de datos abstractos, se pueden utilizar diferentes implementaciones o versiones de una misma capa en la medida que soporten las mismas interfaces de cara a las capas

adyacentes. Esto conduce a la posibilidad de definir interfaces de capa estándar, a partir de las cuales se pueden construir extensiones o prestaciones específicas.

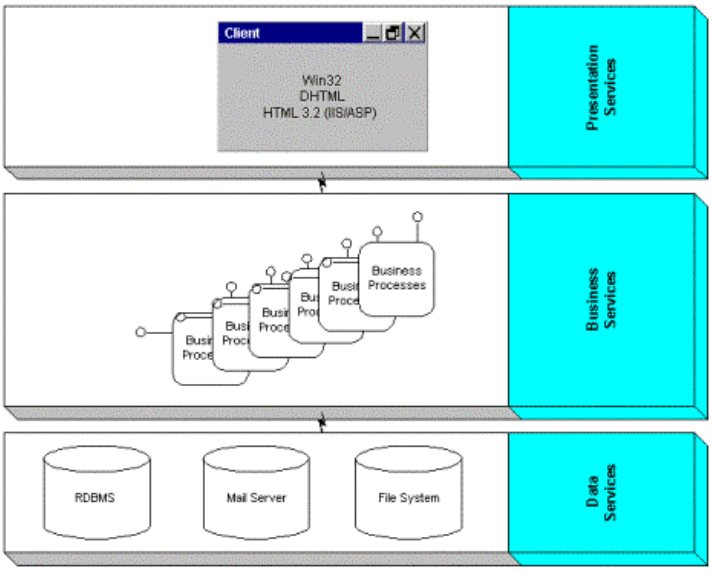

Fig. 4 - Arquitectura en 3 capas con "cliente flaco" en Windows DNA

También se han señalado algunas desventajas de este estilo [GS96]. Muchos problemas no admiten un buen mapeo en una estructura jerárquica. Incluso cuando un sistema se puede establecer lógicamente en capas, consideraciones de performance pueden requerir acoplamientos específicos entre capas de alto y bajo nivel. A veces es también extremadamente difícil encontrar el nivel de abstracción correcto; por ejemplo, la comunidad de comunicación ha encontrado complejo mapear los protocolos existentes en el framework ISO, de modo que muchos protocolos agrupan diversas capas, ocasionando que en el mercado proliferen los *drivers* o los servicios monolíticos. Además, los cambios en las capas de bajo nivel tienden a filtrarse hacia las de alto nivel, en especial si se utiliza una modalidad relajada; también se admite que la arquitectura en capas ayuda a controlar y encapsular aplicaciones complejas, pero complica no siempre razonablemente las aplicaciones simples [MS04c].

En lo que se refiere a las arquitecturas características de Microsoft, el estilo en varias capas hizo su aparición explícita con las estrategias de desarrollo que se conocieron como Windows DNA, que elocuentemente expresaron y otorgaron justificación a lo que la recomendación de la IEEE 1471 propone llamar separación de incumbencias. En aquellos tiempos, además, la prioridad esencial consistía en discutir con la aristocracia cuánto más genéricos, abstractos y simples eran los componentes en relación con los objetos, o en convencer a la muchedumbre de que era mucho más refinado programar con ellos que con funciones de APIs. En términos estilísticos, sin embargo, los documentos de arquitectura de Microsoft, que en algún momento impulsaron con fuerza el diseño en capas como idea dominante (al lado del modelo de componentes), se ha inclinado en los últimos años hacia las arquitecturas basadas en servicios y el grueso de la industria

apunta al mismo rumbo. La separación y especialización de interfaces de usuario, reglas de negocios y estructuras de datos (o sus equivalentes arquitectónicos en otros dominios), sin embargo, conservan todavía plena relevancia. La documentación correspondiente ha elaborado numerosos patrones de arquitectura y diseño derivados de este estilo, como *Layered Application* [MS04c] y *Three-layered Services Application* [MS04d].

A pesar de la versatilidad de REST o de SOAP y de que el ruido más fuerte viene de este lado, las arquitecturas en capas distan de ser un estilo fósil. La información sobre el estilo de componentes Windows DNA todavía está allí [Red99] y el modelo posee virtudes estilísticas de distribución, preservación de identidad, seguridad, performance, escalabilidad, sincronicidad, balanceo de carga, robustez y acidez transaccional que siguen siendo competitivas y que no se valoran hasta que uno se muda a un contexto que obliga a atenerse a un estilo que carece de ellas.

Arquitecturas Orientadas a Objetos

Nombres alternativos para este estilo han sido Arquitecturas Basadas en Objetos, Abstracción de Datos y Organización Orientada a Objectos. Los componentes de este estilo son los objetos, o más bien instancias de los tipos de dato abstractos. En la caracterización clásica de David Garlan y Mary Shaw [GS94], los objetos representan una clase de componentes que ellos llaman *managers*, debido a que son responsables de preservar la integridad de su propia representación. Un rasgo importante de este aspecto es que la representación interna de un objeto no es accesible desde otros objetos. En la semblanza de estos autores curiosamente no se establece como cuestión definitoria el principio de herencia. Ellos piensan que, a pesar de que la relación de herencia es un mecanismo organizador importante para definir los tipos de objeto en un sistema concreto, ella no posee una función arquitectónica directa. En particular, en dicha concepción la relación de herencia no puede concebirse como un conector, puesto que no define la interacción entre los componentes de un sistema. Además, en un escenario arquitectónico la herencia de propiedades no se restringe a los tipos de objeto, sino que puede incluir conectores e incluso estilos arquitectónicos enteros.

Si hubiera que resumir las características de las arquitecturas OO, se podría decir que:

Los componentes del estilo se basan en principios OO: encapsulamiento, herencia y polimorfismo. Son asimismo las unidades de modelado, diseño e implementación, y los objetos y sus interacciones son el centro de las incumbencias en el diseño de la arquitectura y en la estructura de la aplicación.

Las interfaces están separadas de las implementaciones. En general la distribución de objetos es transparente, y en el estado de arte de la tecnología (lo mismo que para los componentes en el sentido de CBSE) apenas importa si los objetos son locales o remotos. El mejor ejemplo de OO para sistemas distribuidos es Common Object Request Broker Architecture (CORBA), en la cual las interfaces se definen mediante Interface Description Language (IDL); un Object Request Broker media las interacciones entre objetos clientes y objetos servidores en ambientes distribuidos.

En cuanto a las restricciones, puede admitirse o no que una interfaz pueda ser implementada por múltiples clases.

En tanto componentes, los objetos interactúan a través de invocaciones de funciones y procedimientos. Hay muchas variantes del estilo; algunos sistemas, por ejemplo, admiten que los objetos sean tareas concurrentes; otros permiten que los objetos posean múltiples interfaces.

Se puede considerar el estilo como perteneciente a una familia arquitectónica más amplia, que algunos autores llaman Arquitecturas de Llamada-y-Retorno (*Call-and-Return*). Desde este punto de vista, sumando las APIs clásicas, los componentes (en el sentido COM y JavaBeans) y los objetos, C-R ha sido el tipo dominante en los últimos 20 años.

En lo que se refiere a las ventajas y desventajas de esta arquitectura, enumerarlas solamente llevaría más espacio del que se dispone en este estudio, de modo que apenas señalaré las más obvias. Entre las cualidades, la más básica concierne a que se puede modificar la implementación de un objeto sin afectar a sus clientes. Asimismo es posible descomponer problemas en colecciones de agentes en interacción. Además, por supuesto (y esa es la idea clave), un objeto es ante todo una entidad reutilizable en el entorno de desarrollo.

Entre las limitaciones, el principal problema del estilo se manifiesta en el hecho de que para poder interactuar con otro objeto a través de una invocación de procedimiento, se debe conocer su identidad. Esta situación contrasta con lo que es el caso en estilos tubería-filtros, donde los filtros no necesitan poseer información sobre los otros filtros que constituyen el sistema. La consecuencia inmediata de esta característica es que cuando se modifica un objeto (por ejemplo, se cambia el nombre de un método, o el tipo de dato de algún argumento de invocación) se deben modificar también todos los objetos que lo invocan. También se presentan problemas de efectos colaterales en cascada: si A usa B y C también lo usa, el efecto de C sobre B puede afectar a A.

En la literatura sobre estilos, las arquitecturas orientadas a objeto han sido clasificadas de formas diferentes, conforme a los diferentes puntos de vista que alternativamente enfatizan la jerarquía de componentes, su distribución topológica o las variedades de conectores. En su famosa disertación sobre REST, Roy Fielding subordina las arquitecturas de objetos distribuidos a los estilos *peer-to-peer*, al lado de la integración basada en eventos y el estilo C2 [Fie00]. Si se sitúa la idea de llamado y respuesta como organizadora del estilo, los sub-estilos serían:

Programa principal y subrutinas. Es el paradigma clásico de programación. El objetivo es descomponer un programa en pequeñas piezas susceptibles de modificarse independientemente.

Estilo orientado a objetos o tipo de dato abstracto. Esta vendría a ser la versión moderna de las arquitecturas C-R. Esta variante enfatiza el vínculo entre datos y métodos para manipular los datos, valiéndose de interfaces públicas. La abstracción de objetos forma componentes que proporcionan servicios de caja negra y componentes que requieren esos

servicios. El encapsulamiento oculta los detalles de implementación. El acceso al objeto se logra a través de operaciones, típicamente conocidas como métodos, que son formas acotadas de invocación de procedimientos.

Sistemas en capas. En este estilo los componentes se asignan a capas. Cada capa se comunica con sus vecinas inmediatas; a veces, sin embargo, razones no funcionales de practicidad, performance, escalabilidad o lo que fuere hace que se toleren excepciones a la regla. Hemos analizado este estilo por separado.

En los estudios arquitectónicos de estilos, y posiblemente por efecto de un énfasis que es más estructural que esencialista (más basado en las relaciones que en las cosas que componen un sistema), el modelo de objetos aparece como relativamente subordinado en relación con la importancia que ha tenido en otros ámbitos de la ingeniería de software, en los que la orientación a objeto ha sido y sigue siendo dominante. Pero entre los estilos, se lo sitúa al lado de las arquitecturas basadas en componentes y las orientadas a servicios, ya sea en paridad de condiciones o como precedente histórico [WF04]. El argumento de la superioridad y la masa crítica de las herramientas de modelado orientadas a objeto no se sostiene a nivel de estilos arquitectónicos, ni debería ser relevante en cuanto al estilo de implementación.

Arquitecturas Basadas en Componentes

Los sistemas de software basados en componentes se basan en principios definidos por una ingeniería de software específica (CBSE) [BW98]. En un principio, hacia 1994, se planteaba como una modalidad que extendía o superaba la tecnología de objetos, como en un famoso artículo de BYTE cuyo encabezado rezaba así: “ComponentWare – La computación Orientada a Objetos ha fracasado. Pero el software de componentes, como los controles de Visual Basic, está teniendo éxito. Aquí explicamos por qué” (Mayo de 1994, pp. 46-56). Con el paso de los años el antagonismo se fue aplacando y las herramientas (orientadas a objeto o no) fueron adaptadas para producir componentes. En la mayoría de los casos, los componentes terminan siendo formas especiales de DLLs que admiten *late binding*, que necesitan registración y que no requieren que sea expuesto el código fuente de la clase [Szy95].

Hay un buen número de definiciones de componentes, pero Clemens Alden Szyperski proporciona una que es bastante operativa: un componente de software, dice, es una unidad de composición con interfaces especificadas contractualmente y dependencias del contexto explícitas [Szy02]. Que sea una unidad de composición y no de construcción quiere decir que no es preciso confeccionarla: se puede comprar hecha, o se puede producir en casa para que otras aplicaciones de la empresa la utilicen en sus propias composiciones. Pragmáticamente se puede también definir un componente (no en el sentido estilístico, sino en el de CBSE) como un artefacto diseñado y desarrollado de acuerdo ya sea con CORBA Component Model (CCM), JavaBeans y Enterprise JavaBeans en J2EE y lo que alternativamente se llamó OLE, COM, ActiveX y COM+, y luego .NET.

En un estilo de este tipo:

Los componentes en el sentido estilístico son componentes en el sentido de CBSE y son las unidades de modelado, diseño e implementación.

Las interfaces están separadas de las implementaciones, y las interfaces y sus interacciones son el centro de incumbencias en el diseño arquitectónico. Los componentes soportan algún régimen de introspección, de modo que su funcionalidad y propiedades puedan ser descubiertas y utilizadas en tiempo de ejecución. En tecnología COM `IUnknown` es una interfaz explícita de introspección que soporta la operación `QueryInterface`.

En cuanto a las restricciones, puede admitirse que una interfaz sea implementada por múltiples componentes. Usualmente, los estados de un componente no son accesibles desde el exterior [Szy02]. Que los componentes sean locales o distribuidos es transparente en la tecnología actual.

El marco arquitectónico estándar para la tecnología de componentes está constituido por los cinco puntos de vista de RM-ODP (Empresa, Información, Computación, Ingeniería y Tecnología). La evaluación dominante del estilo de componentes subraya su mayor versatilidad respecto del modelo de objetos, pero también su menor adaptabilidad comparado con el estilo orientado a servicios. Las tecnologías de componentes del período de inmadurez, asimismo, se consideraban afectadas por problemas de incompatibilidad de versiones e inestabilidad que ya han sido largamente superados en toda la industria.

En la estrategia arquitectónica de Microsoft el estilo de componentes, en el contexto de las arquitecturas en capas de Windows DNA ha sido, como ya se ha dicho, unos de los vectores tecnológicos más importantes a fines del siglo XX; el framework de .NET permite construir componentes avanzados e interoperar componentes y servicios a nivel de la tecnología COM+ 1.5, no como prestación *legacy*, sino en el contexto mayor (estilísticamente mixto) de servicios de componentes [MS03c].

Estilos de Código Móvil

Esta familia de estilos enfatiza la portabilidad. Ejemplos de la misma son los intérpretes, los sistemas basados en reglas y los procesadores de lenguaje de comando. Fuera de las máquinas virtuales y los intérpretes, los otros miembros del conjunto han sido rara vez estudiados desde el punto de vista estilístico. Los sistemas basados en reglas, que a veces se agrupan como miembros de la familia de estilos basados en datos, han sido estudiados particularmente por Murrell, Gamble, Stiger y Plant [MPG96] [SG97] [GSP99].

Arquitectura de Máquinas Virtuales

La arquitectura de máquinas virtuales se ha llamado también intérpretes basados en tablas [GS94] [SC96]. 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.

El estilo comprende básicamente dos formas o sub-estilos, que se han llamado intérpretes y sistemas basados en reglas. Ambas variedades abarcan, sin duda, un extenso espectro que va desde los llamados lenguajes de alto nivel hasta los paradigmas declarativos no secuenciales de programación, que todo el mundo sabe que implementan un *proxy* (una especie de nivel de impostura) que encubren al usuario operaciones que en última instancia se resuelven en instrucciones de máquinas afines al paradigma secuencial imperativo de siempre.

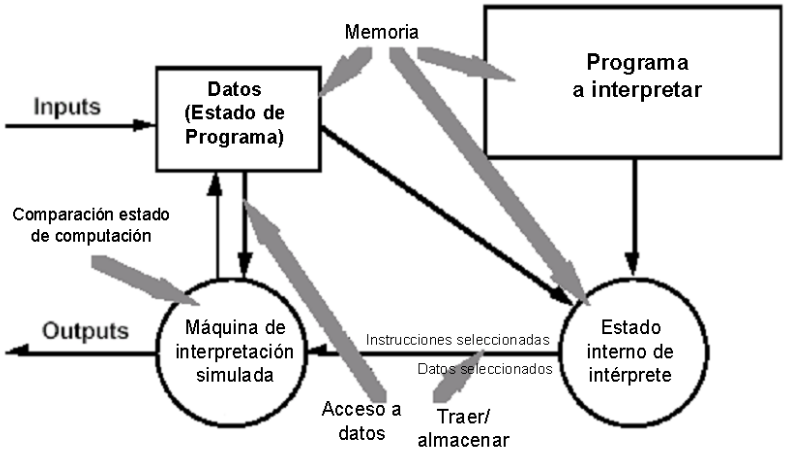

Fig. 5 - Intérprete [basado en GS94]

El estilo es su conjunto se utiliza habitualmente para construir máquinas virtuales que reducen el vacío que media entre el *engine* de computación esperado por la semántica del programa y el *engine* 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 inevtiables en ciertos contextos, nadie se ha entretenido identificando sus ventajas y deméritos.

Las máquinas virtuales no son una invención reciente ligada a Java, sino que existen desde muy antiguo. En la década de 1950, los precursores de los ingenieros de software sugirieron una máquina virtual basada en un lenguaje de máquina universal de bytecodes (un ejemplo fue UNCOL), de manera que las aplicaciones podían escribirse en las capas

más altas y ejecutarse donde fuere sin tener que recompilarse, siempre que hubiera una máquina virtual entre el programa por un lado y el sistema operativo y la máquina real por el otro [Con58]. En 1968 Alan Kay implementó una máquina virtual vinculada a un sistema orientado a objetos [Kay68] y luego participó con Dan Ingalls en el desarrollo de la MV de Smalltalk hacia 1972. Numerosos lenguajes y ambientes de scripting utilizan máquinas virtuales: Perl, Javascript, Windows Script Host (WSH), Python, PHP, Pascal. WSH, por ejemplo, tolera programación en casi cualquier lenguaje de scripting que se atenga a ciertas especificaciones simples.

En la nueva estrategia arquitectónica de Microsoft la máquina virtual por excelencia guarda relación con el Common Language Runtime, acaso unas de las dos piezas esenciales del framework .NET (la otra es la biblioteca de clases). El CLR admite, en efecto, diversos paradigmas puros y templados: programación funcional (Lisp, Scheme, F#, Haskell), programación imperativa orientada a objetos (C#, J#, C++, Python) y estructurada en bloques (Oberon), ambientes de objetos puros (Smallscript / Smalltalk), programación lógica declarativa (Prolog, P#), diseño basado en contratos (Eiffel), modelado matemático (Fortran), *scripting* interpretado (Perl), meta-programación (SML, Mondrian), programación cercana a la semántica de negocios (Cobol), programación centrada en reportes (Visual ASNA RPG), además de todos los matices y composiciones heterogéneas a que haya lugar. Si bien el lenguaje final de implementación se encuentra en un punto del proceso bastante alejado de la ideación arquitectónica en que se despliegan los estilos, el efecto de la disponibilidad de estas capacidades en el diseño inicial de un sistema no es para nada trivial. Con una máquina virtual común el proceso evita la redundancia de motores compitiendo por recursos y unifica *debuggers* y *profilers*. La congruencia entre la naturaleza semántica y sintáctica del modelo y la de los lenguajes de programación concretos ha sido, después de todo, una de las banderas del modelado orientado a objetos, desde OMT hasta UML, pasando por el modelado no sólo de las aplicaciones sino de las bases de datos [RBP+91].

Estilos heterogéneos

Antes de pasar a la familia más fuertemente referida en los últimos tiempos, incluyo en este grupo formas compuestas o indóciles a la clasificación en las categorías habituales. Es por cierto objetable y poco elegante que existan clases residuales de este tipo en una taxonomía, pero ninguna clasificación conocida ha podido resolver este dilema conceptual. En este apartado podrían agregarse formas que aparecen esporádicamente en los censos de estilos, como los sistemas de control de procesos industriales, sistemas de transición de estados, arquitecturas específicas de dominios [GS94] o estilos derivados de otros estilos, como GenVoca, C2 o REST.

Sistemas de control de procesos

Desde el punto de vista arquitectónico, mientras casi todos los demás estilos se pueden definir en función de componentes y conectores, los sistemas de control de procesos se

caracterizan no sólo por los tipos de componentes, sino por las relaciones que mantienen entre ellos. El objetivo de un sistema de esta clase es mantener ciertos valores dentro de ciertos rangos especificados, llamados puntos fijos o valores de calibración; el caso más clásico es el de los termostatos. Existen mecanismos tanto de retroalimentación (*feedback*) como de prealimentación (*feedforward*), y tanto reductores de oscilación como amplificadores; pero el tipo de retroalimentación negativa es el más común. En uno de los pocos tratamientos arquitectónicos de esta clase de modelos cibernéticos, Shaw y Garlan recomiendan separar los tres elementos del bucle de control (mecanismos para cambiar los valores de variables y algoritmos de control, elementos de datos; esquema del bucle). La ventaja señalada para este estilo radica en su elasticidad ante perturbaciones externas [SG96].

Arquitecturas Basadas en Atributos

Aunque algunas otras veces se ha inventado un nuevo estilo para agregarlo al inventario de las variedades existentes, como en este caso, en el de Arch, C2 o REST, la literatura estilística suele ser de carácter reactivo e historicista antes que creativa e innovadora, como si el número de estilos se quisiera mantener deliberadamente bajo. La arquitectura basada en atributos o ABAS fue propuesta por Klein y Klazman [KK99]. La intención de estos autores es asociar a la definición del estilo arquitectónico un *framework* de razonamiento (ya sea cuantitativo o cualitativo) basado en modelos de atributos específicos. Su objetivo se funda en la premisa que dicha asociación proporciona las bases para crear una disciplina de diseño arquitectónico, tornando el diseño en un proceso predecible, antes que en una metodología ad hoc. Con ello se lograría que la arquitectura de software estuviera más cerca de ser una disciplina de ingeniería, aportando el beneficio esencial de la ingeniería (predictibilidad) al diseño arquitectónico.

El modelo de Klein y Kazman en realidad no tipifica como un estilo en estado puro, sino como una asociación entre la idea de estilo con análisis arquitectónico y atributos de calidad. En este contexto, los estilos arquitectónicos definen las condiciones en que han de ser usados. Además de especificar los habituales componentes y conectores, los estilos basados en atributos incluyen atributos de calidad específicos que declaran el comportamiento de los componentes en interacción. Por ejemplo, en las arquitecturas tubería-filtros, se especifica que se considere de qué manera ha de ser administrada la performance y se presta atención a los supuestos que rigen el comportamiento de los filtros y al efecto de su re-utilización. Agregando condiciones, un estilo deviene método.

Dado el carácter peculiar de ABAS no se procederá aquí a su análisis. Llamo la atención no obstante sobre su naturaleza dinámica e instrumental. Por lo general los arquitectos del campo de los estilos, mayormente estructuralistas, no se ocupan de cuestiones procesales tales como disciplinas de desarrollo, refinamiento, evaluación o análisis de riesgo, que corresponderían más bien a las incumbencias de los ingenieros. En la estrategia de arquitectura de Microsoft, empero, hay amplias elaboraciones en este sentido bajo la forma de una metodología de verificación de patrones de software [AEA+03]; la metodología está expresamente orientada a patrones en el sentido de Christopher

Alexander [Ale77], pero es por completo aplicable a estilos arquitectónicos debido a la proximidad de las nociones de patrón y estilo.

Estilos Peer-to-Peer

Esta familia, también llamada de componentes independientes, enfatiza la modificabilidad por medio de la separación de las diversas partes que intervienen en la computación. Consiste por lo general en procesos independientes o entidades que se comunican a través de mensajes. Cada entidad puede enviar mensajes a otras entidades, pero no controlarlas directamente. Los mensajes pueden ser enviados a componentes nominados o propalados mediante *broadcast*. Miembros de la familia son los estilos basados en eventos, en mensajes (Chiron-2), en servicios y en recursos. Gregory Andrews [And91] elaboró la taxonomía más detallada a la fecha de estilos basados en transferencia de mensajes, distinguiendo ocho categorías: (1) Flujo de datos en un sentido a través de redes de filtros. (2) Requerimientos y respuestas entre clientes y servidores. (3) Interacción de ida y vuelta o pulsación entre procesos vecinos. (4) Pruebas y ecos en grafos incompletos. (5) *Broadcasts* entre procesos en grafos completos. (6) *Token passing* asincrónico. (7) Coordinación entre procesos de servidor decentralizados. (8) Operadores replicados que comparten una bolsa de tareas.

Arquitecturas Basadas en Eventos

Las arquitecturas basadas en eventos se han llamado también de invocación implícita [SG96]. 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[Lar03].

Las arquitecturas basadas en eventos se vinculan históricamente con sistemas basados en actores, *daemons* y redes de conmutación de paquetes (publicación-suscripción). Los conectores de estos sistemas incluyen procedimientos de llamada tradicionales y vínculos entre anuncios de eventos e invocación de procedimientos. La idea dominante en la invocación implícita es que, en lugar de invocar un procedimiento en forma directa (como se haría en un estilo orientado a objetos) un componente puede anunciar mediante difusión uno o más eventos. Un componente de un sistema puede anunciar su interés en un evento determinado asociando un procedimiento con la manifestación de dicho evento. Un caso clásico en ambientes Microsoft sería el Servicio de Notificación de SQL Server. Cuando el evento se anuncia, el sistema invoca todos los procedimientos que se han registrado para él. De este modo, el anuncio de un evento implícitamente ocasiona la invocación de determinados procedimientos en otros módulos. También hay elementos

de arquitectura de publicación-suscripción en el modelo de replicación de SQL Server (definido en su documentación como una "metáfora de industria"), en el Publish-Subscribe Toolkit de BizTalk Server 2002 [Chu02] o el Publish-Subscribe API de Windows CE .NET 4.2.

Desde el punto de vista arquitectónico, los componentes de un estilo de invocación implícita son módulos cuyas interfaces proporcionan tanto una colección de procedimientos (igual que en el estilo de tipos de datos abstractos) como un conjunto de eventos. Los procedimientos se pueden invocar a la manera usual en modelos orientados a objeto, o mediante el sistema de suscripción que se ha descripto.

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.

Un estilo perteneciente a esta clase es C2 o Chiron-2. Una aplicación de arquitectura C2 está constituida por componentes que se comunican a través de *buses*; la comunicación está basada en eventos. Un componente puede enviar o recibir eventos hacia o desde los *buses* a los que está conectado. Componentes y *buses* se pueden componer topológicamente de distintas maneras, siguiendo reglas y restricciones particulares. Cada componente posee dos puntos de conexión, llamados respectivamente *top* y *bottom*. El esquema no admite ciclos, de modo que un componente no puede recibir una notificación generada por él mismo [DNR99].

En la estrategia arquitectónica de Microsoft, este estilo (llamado más bien un "patrón recurrente de diseño") juega un papel de alguna importancia [MS02a]. En este estilo y en ese contexto, los eventos se disparan bajo condiciones de negocios particulares, debiéndose escribir código para responder a esos eventos. Este patrón puede utilizarse cuando se desea que sucedan varias actividades, tal que todas ellas reciben los mismos datos de iniciación y no pueden comunicarse entre sí. Diferentes implementaciones del evento pueden o no ejecutarse, dependiendo de información de filtro específica. Si las implementaciones se han configurado para que corran secuencialmente, el orden no puede garantizarse. Las prescripciones de Microsoft para el uso del modelo de eventos señalan que 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.

Entre las ventajas enumeradas en relación con el modelo se señalan:

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.

Entre las 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. Pueden surgir problemas de performance global y de manejo de recursos cuando se comparte un repositorio común para coordinar la interacción.

En esta estrategia juega un rol importante el Servicio de Eventos, el cual a su vez proporciona un buen punto de partida para la implementación del estilo o patrón, según se esté concibiendo la arquitectura o implementándola. Se puede consultar información detallada sobre Enterprise Service Events en el artículo sobre "COM+ Events" incluido en la documentación del SDK de COM+ en Microsoft Developers Network (http://msdn.microsoft.com/library/en-us/cossdk/htm/pgservices_events_2y9f.asp). La arquitectura del servicio de notificación de SQL Server (basada en XML) está descripta en http://msdn.microsoft.com/library/default.asp?url=/library/en-us/sqlntsv/htm/ns_overarch_5to4.asp.

Arquitecturas Orientadas a Servicios

Sólo recientemente estas arquitecturas que los conocedores llaman SOA han recibido tratamiento intensivo en el campo de exploración de los estilos. Al mismo tiempo se percibe una tendencia a promoverlas de un sub-estilo propio de las configuraciones distribuidas que antes eran a un estilo en plenitud. Esta promoción ocurre al compás de las predicciones convergentes de Giga o de Gartner que (después de un par de años de titubeo y consolidación) las visualizan en sus pronósticos y cuadrantes mágicos como la tendencia que habrá de ser dominante en la primera década del nuevo milenio. Ahora bien, este predominio no se funda en la idea de servicios en general, comunicados de cualquier manera, sino que más específicamente va de la mano de la expansión de los

Web services basados en XML, en los cuales los formatos de intercambio se basan en XML 1.0 Namespaces y el protocolo de elección es SOAP. SOAP significa un formato de mensajes que es XML, comunicado sobre un transporte que por defecto es HTTP, pero que puede ser también HTTPs, SMTP, FTP, IIOP, MQ o casi cualquier otro, o puede incluir prestaciones sofisticadas de última generación como WS-Routing, WS-Attachment, WS-Referral, etcétera.

Alrededor de los Web services, que dominan el campo de un estilo SOA más amplio que podría incluir otras opciones, se han generado las controversias que usualmente acompañan a toda idea exitosa. Al principio hasta resultaba difícil encontrar una definición aceptable y consensuada que no fuera una fórmula optimista de mercadotecnia. El grupo de tareas de W3C, por ejemplo, demoró un año y medio en ofrecer la primera definición canónica apta para consumo arquitectónico, que no viene mal reproducir aquí:

> Un Web service es un sistema de software diseñado para soportar interacción máquina-a-máquina sobre una red. Posee una interfaz descripta en un formato procesable por máquina (específicamente WSDL). Otros sistemas interactúan con el Web service de una manera prescripta por su descripción utilizando mensajes SOAP, típicamente transportados usando HTTP con una serialización en XML en conjunción con otros estándares relacionados a la Web [Cha03]

En la literatura clásica referida a estilos, las arquitecturas basadas en servicios podrían engranar con lo que Garlan & Shaw definen como el estilo de procesos distribuidos. Otros autores hablan de Arquitecturas de Componentes Independientes que se comunican a través de mensajes. Según esta perspectiva, no del todo congruente con la masa crítica que han ganado las arquitecturas orientadas a servicios en los últimos años, habría dos variantes del estilo:

Participantes especificados (*named*): Estilo de proceso de comunicación. El ejemplar más conocido sería el modelo cliente-servidor. Si el servidor trabaja sincrónicamente, retorna control al cliente junto con los datos; si lo hace asincrónicamente, sólo retorna los datos al cliente, el cual mantiene su propio hilo de control.

Participantes no especificados (*unnamed*): Paradigma *publish/subscribe*, o estilo de eventos.

Los estilos de la familia de la orientación a servicios, empero, se han agrupado de muchas maneras diversas, según surge del apartado que dedicamos a las taxonomías estilísticas [Mit02] [Fie00]. Existen ya unos cuantos textos consagrados a analizar los Web services basados en XML en términos de arquitectura de software en general y de estilos arquitectónicos en particular; especialmente recomendables son el volumen de Ron Schmelzer y otros [Sch+02] y *Architecting Web Services* de W. Oellermann [Oel01].

No es intención de este estudio describir en detalle las peculiaridades de este estilo, suficientemente caracterizado desde diferentes puntos de vista en la documentación primaria de la estrategia de arquitectura de Microsoft [Ms02a] [MS02b]. Lo que sí vale la pena destacar es la forma en la cual el estilo redefine los viejos modelos de ORPC

propios de las arquitecturas orientadas a objetos y componentes, y al hacerlo establece un modelo en el que es casi razonable pensar que cualquier entidad computacional (nativamente o mediando un *wrapper*) podría llegar a conversar o a integrarse con cualquier otra (ídem) una vez resueltas las inevitables coordinaciones de ontología. En el cuadro siguiente he referido las características del modelo de llamado-respuesta propio del estilo, en contraste con tecnologías clásicas bien conocidas de comunicación entre componentes y objetos en ambientes DCOM, CORBA y Java.

Lo que hace diferentes a los Web services de otros mecanismos de RPC como RMI, CORBA o DCOM es que utiliza estándares de Web para los formatos de datos y los protocolos de aplicación. Esto no sólo es un factor de corrección política, sino que permite que las aplicaciones interoperen con mayor libertad, dado que las organizaciones ya seguramente cuentan con una infraestructura activa de HTTP y pueden implementar tratamiento de XML y SOAP en casi cualquier lenguaje y plataforma, ya sea descargando un par de *kits*, adquiriendo el paquete de lenguaje o biblioteca que proporcione la funcionalidad o programándolo a mano. Esto no admite ni punto de comparación con lo que implicaría, por ejemplo, implementar CORBA en todas las plataformas participantes. Por añadidura, la descripción, publicación, descubrimiento, localización e invocación de los Web services se puede hacer en tiempo de ejecución, de modo que los servicios que interactúan pueden figurarse la forma de operar de sus contrapartes, sin haber sido diseñados específicamente caso por caso. Por primera vez, esta dinamicidad es plenamente viable.

	DCOM	CORBA	Java RMI	Web Services
Protocolo RPC	RPC	IIOP	IIOP o JRMP	SOAP
Formato de mensaje	NDR	CDR	Java Serialization Format	XML 1.0 Namespaces
Descripción	IDL	OMG IDL	Java	WSDL
Descubrimiento	Registry	Naming Service	RMI Registry o JNDI	UDDI
Denominación	GUID, OBJREF	IOR	Java.rmi.naming	URI
Marshalling	Type Library Marshaller	Dynamic Invocation/Skeleton Interface	Java.rmi Marshalling o Serialización	Serialización

Tabla 2 - Modelos de RPC

Desde el punto de vista arquitectónico, se puede hacer ahora una caracterización sucinta de las características del estilo:

Un servicio es una entidad de software que encapsula funcionalidad de negocios y proporciona dicha funcionalidad a otras entidades a través de interfaces públicas bien definidas.

Los componentes del estilo (o sea los servicios) están débilmente acoplados. El servicio puede recibir requerimientos de cualquier origen. La funcionalidad del servicio se puede ampliar o modificar sin rendir cuentas a quienes lo requieran. Los servicios son las unidades de implementación, diseño e implementación.

Los componentes que requieran un servicio pueden descubrirlo y utilizarlo dinámicamente mediante UDDI y sus estándares sucesores. En general (aunque hay alternativas) no se mantiene persistencia de estado y tampoco se pretende que un servicio recuerde nada entre un requerimiento y el siguiente.

Las especificaciones de RM-ODP son lo suficientemente amplias para servir de marco de referencia estándar tanto a objetos como a componentes y servicios, pero las herramientas usuales de diseño (por ejemplo UML) no poseen una notación primaria óptima que permita modelar servicios, a despecho de docenas de propuestas en todos los congresos de modelado. Un servicio puede incluir de hecho muchas interfaces y poseer propiedades tales como descripción de protocolos, puntos de entrada y características del servicio mismo. Algunas de estas notaciones son provistas por lenguajes declarativos basados en XML, como WSDL (Web Service Description Language).

Como todos los otros estilos, las SOA poseen ventajas y desventajas. Como se trata de una tecnología que está en su pico de expansión, virtudes y defectos están variando mientras esto se escribe. Las especificaciones fundamentales de toda la industria (y la siguiente versión de las capacidades de transporte, documentos agregados y formatos, ruteo, transacción, *workflow*, seguridad, etcétera) se definen primariamente en http://www.ws-i.org. En cuanto a una comparación entre las tres arquitecturas en paridad de complejidad y prestigio (OOA, CBA y SOA) puede consultarse la reciente evaluación de Wang y Fung [WF04] o los infaltables documentos comparativos de las empresas analistas de la industria.

En la documentación de la estrategia de arquitectura de Microsoft se encontrará abundante información y lineamientos referidos al estilo arquitectónico orientado a servicios. De hecho, la información disponible es demasiado profusa para tratarla aquí en detalle. Comenzando por *Application Architecture for .NET* [MS02a], y *Application Conceptual View* [MS02b], el lector encontrará sobrada orientación y patrones de diseño y arquitectura correspondientes a este estilo en el sitio de Microsoft Patterns & Practices, así como en la casi totalidad de las bibliotecas de MSDN de los últimos años. En el futuro próximo, el modelo de programación de Longhorn permitirá agregar a la ya extensa funcionalidad de los servicios una nueva concepción relacional del sistema de archivos (WinFS), soporte extensivo de *shell* y prácticamente toda la riqueza del API de Win32 en términos de código manejado [Rec04].

Arquitecturas Basadas en Recursos

Una de las más elocuentes presentaciones de arquitecturas peer-to-peer ha sido la disertación doctoral de Roy Fielding, elaborada con anterioridad pero expuesta con mayor impacto en el año 2000 [Fie00]. Es en ella donde se encuentra la caracterización más detallada del estilo denominado Representational State Transfer o REST. Aunque la literatura especializada tiende a considerar a REST una variante menor de las arquitecturas basadas en servicios, Fielding considera que REST resulta de la composición de varios estilos más básicos, incluyendo repositorio replicado, *cache*, cliente-servidor, sistema en capas, sistema sin estado, máquina virtual, código a demanda e interfaz uniforme [FT02]. Fielding no solamente expande más allá de lo habitual y quizá más de lo prudente el catálogo de estilos existentes, sino que su tratamiento estilístico se basa en Perry y Wolf [PW92] antes que en Garlan y Shaw [GS94], debido a que la literatura sobre estilos que se deriva de este último texto sólo considera elementos, conectores y restricciones, sin tomar en consideración los datos, que para el caso de REST al menos constituyen una dimensión esencial.

En síntesis muy apretada, podría decirse que REST define recursos identificables y métodos para acceder y manipular el estado de esos recursos. El caso de referencia es nada menos que la World Wide Web, donde los URIs identifican los recursos y HTTP es el protocolo de acceso. El argumento central de Fielding es que HTTP mismo, con su conjunto mínimo de métodos y su semántica simplísima, es suficientemente general para modelar cualquier dominio de aplicación. De esta manera, el modelado tradicional orientado a objetos deviene innecesario y es reemplazado por el modelado de entidades tales como familias jerárquicas de recursos abstractos con una interfaz común y una semántica definida por el propio HTTP. REST es en parte una reseña de una arquitectura existente y en parte un proyecto para un estilo nuevo. La caracterización de REST constituye una lectura creativa de la lógica dinámica que rige el funcionamiento de la Web (una especie de ingeniería inversa de muy alto nivel), al lado de una propuesta de nuevos rasgos y optimizaciones, o restricciones adicionales: REST, por ejemplo, no permite el paso de *cookies y* propone la eliminación de MIME debido a su tendencia estructural a la corrupción y a su discrepancia lógica con HTTP. REST se construye expresamente como una articulación compuesta a partir de estilos y sub-estilos preexistentes, con el agregado de restricciones específicas. En la figura 6, RR corresponde a Repositorio Replicado, CS a Cliente-Servidor, LS a sistema en capas, VM a Máquina Virtual y así sucesivamente en función de la nomenclatura referida en la clasificación de Fielding [Fie00], revisada oportunamente. En este sentido, REST constituye un ejemplo de referencia para derivar descripciones de arquitecturas de la vida real en base a un procedimiento de composición estilística, tal como se ilustra en la Figura 6.

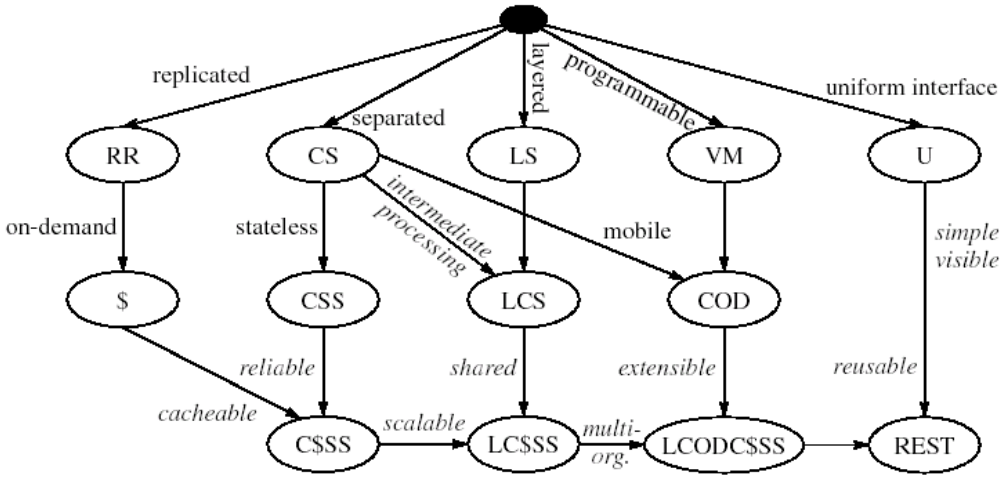

Fig. 6 - Composición de REST [FT02]

La especificación REST posee algunas peculiaridades emanadas de su lógica orientada a recursos que la hacen diferente de otras implementaciones tales como WebDAV, ebXML, BPML, XLANG, UDDI, WSCK o BPEL4WS que se analizará en un documento específico sobre arquitecturas orientadas a servicios. Sobre una comparación entre dichas variantes puede consultarse el análisis de Mitchell [Mit02].

La polémica sobre si REST constituye una revolucionaria inflexión en los estilos orientados a servicios ("la nueva generación de Web services") o si es una variante colateral todavía subsiste. En opinión de Jørgen Thelin, REST es un estilo característico de las arquitecturas basadas en recursos, antes que en mensajes, y la especificación total de Web services sería un superconjunto de dicha estrategia, susceptible de visualizarse como estilo de recursos o de servicios, según convenga. Como quiera que sea, los organismos de estándares que han elaborado SOAP 1.2 y WSDL 1.2, así como WS-I, han incorporado ideas de REST, tales como atenuar el concepto originario que entendía los Web services como meros "objetos distribuidos en una red", o concebir las interacciones más en términos de intercambios asincrónicos de documentos que como una modalidad nueva de llamada remota a procedimientos [Cha03] [http://www.ws-i.org]. El lema de REST es, después de todo, "Adiós objetos, adiós RPC".

El Lugar del Estilo en Arquitectura de Software

A fin de determinar cómo se vinculan los estilos con otros conceptos y el espacio que ocupan en el marco conceptual de la arquitectura o en las secuencias de procesos de la metodología, habría que mapear buena parte, si es que no la totalidad del espacio de la arquitectura de software, a su vez complejamente vinculado con otros campos que nunca fueron demarcados de manera definitiva. Mientras algunos claman por una disciplina autónoma de diseño (Mitchell Kapor, Terry Winograd), otros estudiosos (Peter Denning, Dewayne Perry) discriminan con distinción y claridad el ámbito de la arquitectura y el del

diseño, y no admitirían jamás cruzar la línea que los separa. Como quiera que se organice la totalidad, no es intención de este texto construir un nuevo mapa que se agregue a los muchos esquemas panópticos que ya se postularon. Por lo general éstos encuentran su sentido en estructuras y paradigmas cuya vigencia, generalidad y relevancia es a veces incierta, o que están sesgadas no siempre con discreción en favor de una doctrina específica de modelado y de desarrollo (que en la mayoría de los casos está previsiblemente orientada a objetos). Está claro que casi cualquier tecnología puede de alguna forma implementar casi cualquier cosa; pero si todo fuera rigurosa y elegantemente reducible a objetos, ni falta que haría distinguir entre estilos.

Los estilos son susceptibles de asignarse a diversas posiciones en el seno de vistas y modelos mayores, aunque llamativamente la literatura sobre estilos no ha sido jamás sistemática ni explícita al respecto. Nunca nadie ha expresado que el discurso sobre estilos concierna a una coordenada específica de algún marco definido, como si el posicionamiento en dicho contexto fuera irrelevante, o resultara obvio para cualquier arquitecto bien informado. La verdad es que si ya disponíamos de un marco abarcativo (4+1, pongamos por caso, o la matriz de Zachman, o la estrategia global de Microsoft) situar en él la teoría y la práctica de los estilos dista de ser algo que pueda decidirse siempre con transparencia. Los estilos son históricamente más tardíos que esos marcos, y surgieron porque había necesidad de compensar una ausencia tanto en el canon estático de las matrices de vistas como en el paradigma dinámico de los procesos, metodologías de diseño o disciplinas de desarrollo de soluciones.

El concepto de vistas arquitectónicas (*views*), así como los *frameworks* arquitectónicos establecidos en las fases tempranas de la arquitectura de software, suministran diferentes organizaciones del espacio de conceptos, métodos, técnicas, herramientas y procesos según distintos criterios y con variados énfasis. Las terminologías globalizadoras son inestables y no siempre fáciles de comparar entre sí; se ha hablado de *frameworks*, modelos de referencia, escenarios, vistas (*views*), metodologías y paradigmas. Algunos ejemplares abarcativos han sido consistentes en el uso de la terminología, otros no tanto. Aún en el caso de que la terminología sea sintácticamente invariante (como sucede en epistemología con el célebre "paradigma" de Thomas Kuhn) la semántica que se le imprime no siempre ha sido la misma, ni en los documentos oficiales ni en su aplicación por terceras partes.

En general, los grandes marcos son agnósticos en lo que se refiere a metodologías y notaciones. Por metodologías me refiero a entidades tales como Microsoft Solutions Framework, RUP, modelos de procesos de ingeniería de software (DADP, DSSA, FODA, FORM, FAST, ODM, ROSE) o los llamados Métodos Ágiles (Lean Development, eXtreme Programming, Adaptive Software Development). Por notaciones quiero significar ya sea los lenguajes de descripción de arquitectura o ADLs (Acme/Armani, Aesop, C2SADEL, Darwin, Jacal, LILEANNA, MetaH, Rapide, Wright, xADL), los lenguajes formales de especificación (Alloy, CHAM, CSP, LARCH, Z, cálculo π, VDM), las técnicas de análisis y diseño como SADT y los lenguajes de

modelado como UML (el cual, incidentalmente, ha debido ser re-semantizado y extendido de maneras más bien tortuosas para poder integrar en su conjunto nativo modelado arquitectónico en general y de estilos en particular).

Otras propuestas que han surgido y que eventualmente se presentan como marcos de referencia envolventes, como la de MDP/EDOC/MDA (dependientes de OMG), han mapeado (o están en proceso de hacerlo) sus estructuras conceptuales contra algunos de los grandes marcos de referencia, tales como RM-ODP, o sobre marcos nomencladores como IEEE Std 1471, que se ha hecho público en el año 2000.

Algunos autores han introducido modelos de vistas y puntos de vista (*views*, *viewpoints*) insertas en marcos de referencia (*frameworks*) y numerosos organismos de estandarización han hecho lo propio. Las páginas de cabecera de la estrategia de arquitectura de Microsoft reconocen algunos de esos marcos como modelos primarios de referencia. Los que aquí mencionaríamos, señalando la posible ubicación de los estilos, son los siguientes:

El marco de referencia para la arquitectura empresarial de Zachman [Zac87] es uno de los más conocidos. Identifica 36 vistas en la arquitectura ("celdas") basadas en seis niveles (*scope*, empresa, sistema lógico, tecnología, representación detallada y funcionamiento empresarial) y seis aspectos (datos, función, red, gente, tiempo, motivación). En el uso corriente de arquitectura de software se ha estimado que este modelo es excesivamente rígido y sobre-articulado. En la literatura arquitectónica sobre estilos, en general, no se lo ha aplicado en forma explícita y al menos *off the record* parecería existir cierto acuerdo sobre su posible obsolescencia. Los manuales recientes de ingeniería de software (por ejemplo [Pre02]) suelen omitir toda referencia a este marco.

El Modelo de Referencia para Procesamiento Distribuido Abierto (RM-ODP) es un estándar ISO/ITU que brinda un marco para la especificación arquitectónica de grandes sistemas distribuidos. Define, entre otras cosas, cinco puntos de vista (*viewpoints*) para un sistema y su entorno: Empresa, Información, Computación, Ingeniería y Tecnología. Los cinco puntos de vista no corresponden a etapas de proceso de desarrollo o refinamiento. De los cuatro estándares básicos, los dos primeros se refieren a la motivación general del modelo y a sus fundamentos conceptuales y analíticos, el tercero (ISO/IEC 10746-3; UTI-T X.903) a la arquitectura, definiendo los puntos de vistas referidos; y el cuarto (ISO/IEC 10746-4; UTI-T X.904) a la formalización de la semántica arquitectónica. RM-ODP se supone neutral en relación con la metodología, las formas de modelado y la tecnología a implementarse, y es particularmente incierto situar los estilos en su contexto.

El marco de referencia arquitectónico de The Open Group (TOGAF) reconoce cuatro componentes principales, uno de los cuales es un *framework* de alto nivel que a su vez define cuatro vistas: Arquitectura de Negocios, Arquitectura de Datos/Información, Arquitectura de Aplicación y Arquitectura Tecnológica. En marzo de 2000, Rick Hilliard elaboró un preciso informe sobre el impacto de las recomendaciones de IEEE sobre el marco de TOGAF, estimando que ambos son complementarios y compatibles, pero

recomendando que TOGAF se mueva hacia el marco categorial de IEEE antes que la inversa [Hil00]. Tal como está expresado el marco de referencia, los estilos se articulan con mayor claridad en la vista de Arquitectura de Aplicación.

En 1995 Philippe Kruchten propuso su célebre modelo "4+1", vinculado al Rational Unified Process (RUP), que define cuatro vistas diferentes de la arquitectura de software: (1) La vista lógica, que comprende las abstracciones fundamentales del sistema a partir del dominio de problemas. (2) La vista de proceso: el conjunto de procesos de ejecución independiente a partir de las abstracciones anteriores. (3) La vista física: un mapeado del software sobre el hardware. (4) La vista de desarrollo: la organización estática de módulos en el entorno de desarrollo. El quinto elemento considera todos los anteriores en el contexto de casos de uso [Kru95]. Es palmario que los estilos afectan a las dos primeras vistas, mientras que los patrones tienen que ver más bien con la última. En cuanto a los estilos, Kruchten menciona la posible implementación de algunos de ellos (*pipe-filter*, cliente-servidor) en relación con la vista de proceso, mientras que recomienda adoptar un estilo en capas en la vista de desarrollo [Kru95: 3, 5]. Cuando se lee el artículo de Kruchten (un par de años posterior al surgimiento de los estilos en el discurso arquitectónico) es inevitable la sensación de que lo que él llama estilo tiene más que ver con la notación gráfica que usa para la representación (siempre ligada a ideas de objeto) que con la estructura de componentes, conectores y restricciones que según se ha consensuado definen un estilo. Ya en la vista lógica, por ejemplo, los componentes son clases y los conectores están ligados a conceptos de herencia; en la vista de desarrollo sus clases devienen "módulos" o "subsistemas" más o menos neutros, pero todo el mundo sabe qué entidad resulta cuando se instancia una clase, sobre todo cuando las incumbencias enfatizadas son "portabilidad" y "re-utilización" [Kru95: 14-15]. El modelo 4+1 se percibe hoy como un intento se reformular una arquitectura estructural y descriptiva en términos de objetos y de UML, lo cual ocasiona que ningún estilo se manifieste con alguna continuidad en las diferentes vistas. En arquitectura de software en general se admite que no hay estilos puros y que hay inflexiones en un estilo que son como encapsulamientos en miniatura de algún otro tipo; pero los estilos no trasmutan tan fácilmente en función de la vista adoptada. Con todo, las cuatro vistas de Kruchten forman parte del repertorio estándar de los practicantes de la disciplina.

En su introducción a UML (1.3), Grady Booch, James Rumbaugh e Ivar Jacobson han formulado un esquema de cinco vistas interrelacionadas que conforman la arquitectura de software, caracterizada en términos parecidos a los que uno esperaría encontrar en el discurso de la vertiente estructuralista. En esta perspectiva, la arquitectura de software (a la que se dedican muy pocas páginas) es un conjunto de decisiones significativas sobre (1) la organización de un sistema de software; (2) la selección de elementos estructurales y sus interfaces a través de los cuales se constituye el sistema; (3) su comportamiento, según resulta de las colaboraciones entre esos elementos; (4) la composición de esos elementos estructurales y de comportamiento en subsistemas progresivamente mayores; (5) el estilo arquitectónico que guía esta organización: los elementos estáticos y dinámicos y sus interfaces, sus colaboraciones y su composición. Los autores

proporcionan luego un esquema de cinco vistas posibles de la arquitectura de un sistema: (1) La vista de casos de uso, como la perciben los usuarios, analistas y encargados de las pruebas; (2) la vista de diseño que comprende las clases, interfaces y colaboraciones que forman el vocabulario del problema y su solución; (3) la vista de procesos que conforman los hilos y procesos que forman los mecanismos de sincronización y concurrencia; (4) la vista de implementación que incluye los componentes y archivos sobre el sistema físico; (5) la vista de despliegue que comprende los nodos que forma la topología de hardware sobre la que se ejecuta el sistema [BRJ99: 26-27]. Aunque las vistas no están expresadas en los mismos términos estructuralistas que campean en su caracterización de la arquitectura, y aunque la relación entre vistas y decisiones arquitectónicas es de simple yuxtaposición informal de ideas antes que de integración rigurosa, es natural inferir que las vistas que más claramente se vinculan con la semántica estilística son la de diseño y la de proceso.

En los albores de la moderna práctica de los patrones, Buschmann y otros presentan listas discrepantes de vistas en su texto popularmente conocido como *POSA* [BMR+96]. En la primera se las llama "arquitecturas", y son: (1) Arquitectura conceptual: componentes, conectores; (2) Arquitectura de módulos: subsistemas, módulos, exportaciones, importaciones; (3) Arquitectura de código: archivos, directorios, bibliotecas, inclusiones; (4) Arquitectura de ejecución: tareas, hilos, procesos. La segunda lista de vistas, por su parte, incluye: (1) Vista lógica: el modelo de objetos del diseño, o un modelo correspondiente tal como un diagrama de relación; (2) Vista de proceso: aspectos de concurrencia y sincronización; (3) Vista física: el mapeo del software en el hardware y sus aspectos distribuidos; (4) Vista de desarrollo: la organización estática del software en su entorno de desarrollo. Esta segunda lista coincide con el modelo 4+1 de Kruchten, pero sin tanto énfasis en el quinto elemento.

Bass, Clements y Kazman presentan en 1998 una taxonomía de nueve vistas, decididamente sesgadas hacia el diseño concreto y la implementación: (1) Estructura de módulo. Las unidades son asignaciones de tareas. (2) Estructura lógica o conceptual. Las unidades son abstracciones de los requerimientos funcionales del sistema. (3) Estructura de procesos o de coordinación. Las unidades son procesos o *threads.* (4) Estructura física. (5) Estructura de uso. Las unidades son procedimientos o módulos, vinculados por relaciones de presunción-de-presencia-correcta. (6) Estructura de llamados. Las unidades son usualmente (sub)procedimientos, vinculados por invocaciones o llamados. (7) Flujo de datos. Las unidades son programas o módulos, la relación es de envío de datos. (8) Flujo de control; las unidades son programas, módulos o estados del sistema. (9) Estructura de clases. Las unidades son objetos; las relaciones son `hereda-de` o `es-una-instancia-de` [BCK98]. De acuerdo con el punto de vista de la formulación de estilos, la situación de éstos en este marco puede variar sustancialmente.

La recomendación IEEE Std 1471-2000 procura establecer una base común para la descripción de arquitecturas de software, e implementa para ello tres términos básicos, que son arquitectura, vista y punto de vista. La *arquitectura* se define como la

organización fundamental de un sistema, encarnada en sus componentes, las relaciones entre ellos y con su entorno, y los principios que gobiernan su diseño y evolución. Los elementos que resultan definitorios en la utilidad, costo y riesgo de un sistema son en ocasiones físicos y otras veces lógicos. En otros casos más, son principios permanentes o patrones que generan estructuras organizacionales duraderas. Términos como *vista* o *punto de vista* son también centrales. En la recomendación se los utiliza en un sentido ligeramente distinto al del uso común. Aunque reflejan el uso establecido en los estándares y en la investigación de ingeniería, el propósito del estándar es introducir un grado de formalización homogeneizando informalmente la nomenclatura. En dicha nomenclatura, un punto de vista (*viewpoint*) define un patrón o plantilla (*template*) para representar un conjunto de incumbencias (*concerns*) relativo a una arquitectura, mientras que una vista (*view*) es la representación concreta de un sistema en particular desde una perspectiva unitaria. Un punto de vista permite la formalización de grupos de modelos. Una vista también se compone de modelos, aunque posee también atributos adicionales. Los modelos proporcionan la descripción específica, o contenido, de una arquitectura. Por ejemplo, una vista estructural consistiría de un conjunto de modelos de la estructura del sistema. Los elementos de tales modelos incluirían componentes identificables y sus interfaces, así como interconexiones entre los componentes. La concordancia entre la recomendación de IEEE y el concepto de estilo se establece con claridad en términos del llamado "punto de vista estructural". Otros puntos de vista reconocidos en la recomendación son el conductual y el de interconexión física. El punto de vista estructural ha sido motivado (afirman los redactores del estándar) por el trabajo en lenguajes de descripción arquitectónica (ADLs). El punto de vista estructural, dicen, se ha desarrollado en el campo de la arquitectura de software desde 1994 y es hoy de amplio uso. Este punto de vista es a menudo implícito en la descripción arquitectónica contemporánea y de hecho ha decantado en el concepto de estilo, plena y expresamente reconocido por la IEEE. En cuanto a los patrones, la recomendación de IEEE no ha especificado ninguna provisión respecto a principios de reutilización que constituyen la esencia del concepto de patrones. Ante tal circunstancia, Rich Hilliard, uno de sus redactores, propone extender el modelo en esa dirección [Hil01]. En un estudio sobre el particular, Hilliard ha analizado puntos de vista, estilos y patrones como tres modelos alternativos de descripción del conocimiento arquitectónico [Hil01b].

La estrategia de arquitectura de Microsoft define, en consonancia con las conceptualizaciones más generalizadas, cuatro vistas, ocasionalmente llamadas también arquitecturas: Negocios, Aplicación, Información y Tecnología [Platt02]. La vista que aquí interesa es la de la aplicación, que incluye, entre otras cosas: (1) Descripciones de servicios automatizados que dan soporte a los procesos de negocios; (2) descripciones de las interacciones e interdependencias (interfaces) de los sistemas aplicativos de la organización, y (3) planes para el desarrollo de nuevas aplicaciones y la revisión de las antiguas, basados en los objetivos de la empresa y la evolución de las plataformas tecnológicas. Cada arquitectura, a su vez, se articula en vistas también familiares desde los días de OMT que son (1) la Vista Conceptual, cercana a la semántica de negocios de

los usuarios no técnicos; (2) la Vista Lógica, que define los componentes funcionales y su relación en el interior de un sistema, en base a la cual los arquitectos construyen modelos de aplicación que representan la perspectiva lógica de la arquitectura de una aplicación; (3) la Vista Física, que es la menos abstracta y que ilustra los componentes específicos de una implementación y sus relaciones. En este marco global, los estilos pueden verse como una formulación que, en la arquitectura de una aplicación, orienta la configuración de su vista lógica. Este posicionamiento se refrenda en la relación que Michael Platt establece a propósito de los patrones de aplicación, la cual también coincide con el tratamiento que se ha dado históricamente a la relación entre estilos y patrones [Platt02] [Shaw96] [MKM97] [Land02].

Zachman (Niveles)	TOGAF (Arquitecturas)	4+1 (Vistas)	[BRJ99] (Vistas)	POSA (Vistas)	Microsoft (Vistas)
Scope	Negocios	Lógica	Diseño	Lógica	Lógica
Empresa	Datos	Proceso	Proceso	Proceso	Conceptual
Sistema lógico	Aplicación	Física	Implementa-ción	Física	Física
Tecnología	Tecnología	Desarrollo	Despliegue	Desarrollo	
Representación		Casos de uso	Casos de uso		
Funcionamiento					

Tabla 3 - Posiciones y vistas en los marcos de referencia

A excepción del modelo RM-ODP, cuya naturaleza y objetivos no son comparables a las de las otras formulaciones, los grandes marcos se avienen a situarse en una grilla que define niveles, arquitecturas o vistas. Según sea estática o dinámicamente, sus posiciones representan las perspectivas conceptuales o los momentos de materialización del proceso de desarrollo de un sistema, tal como se refleja en la tabla 3. En la misma se han sombreado las perspectivas desde las cuales la problemática de los estilos y patrones arquitectónicos se perciben con mayor claridad o se presentan con mayor pertinencia. Con la excepción de los casos de uso, que en realidad se superponen a las otras vistas, podría decirse que hacia arriba de la región sombreada se encuentra la perspectiva de usuarios, negocios y empresas de los que surge el requerimiento, y hacia abajo la región de los patrones de diseño y el código de los que resulta la implementación.

Un escenario tan complejo y una historia tan pródiga en incidentes como las de la ingeniería y la arquitectura de software podrían articularse también según otras narrativas. Hacia 1992, por ejemplo, Joseph Goguen, del Laboratorio de Computación de la Universidad de Oxford, estimaba que el desarrollo de sistemas hasta aquel entonces se había desplegado en dos campos, que él llamaba el *Húmedo* y el *Seco* [Gog92]. El campo

Húmedo está dominado fundamentalmente por la mentalidad del "*hacker*": el desarrollador de sistemas desea crear un sistema lo más rápido posible y para ello utiliza tantos principios heurísticos de diseño como pueda conseguir para alcanzar el objetivo. El sistema puede o no quedar documentado en el proceso. Si el sistema funciona, a la larga, ello es porque no está desarrollado según un proceso al azar, sino que se han utilizado patrones heurísticos de diseño, basados en experiencias previas y anécdotas que fueron pasando de un programador a otro. Salvando las distancias, la definición de Goguen de la mentalidad seca parece una descripción anticipada de lo que después sería el espíritu de al menos algunas de las facciones de la programación ágil, la misma que ha hecho que Martin Fowler se preguntara *Is design dead?* [Fow01].

El campo de desarrollo alternativo es el de la comunidad *Seca*, en el que sólo se utilizan metodologías formales. Estos principios de diseño se han propagado merced a la iniciativa de los que Ince [Ince88] llama "nuevos puritanos matemáticos". Aunque un observador externo podría juzgar que los diseños fundados matemáticamente son difíciles de expresar y de interpretar en un primer examen, un estudio más detallado revela que no son sólo piezas arbitrarias de especificación matemática, sino que de hecho se atienen a ciertos patrones de diseño. El objetivo de la investigación en estilos arquitectónicos ha sido vincular las dos metodologías en términos de sus principios de diseño, por una parte heurísticos y por la otra formales, constituyendo una estrategia que permita un diseño sólido, correcto y riguroso y que facilite mejor verificación y validación. Ejemplos tempranos de esta convergencia entre principios secos y húmedos son los principios de diseño sentados por investigadores como Jackson, Storer y otros más, que desde mediados de la década de 1980 identificaron una serie de reglas para construir "diseños estructurados" a partir de las piezas de construcción (*building blocks*) de programas, tales como secuencias de instrucción, selección e iteración. Estos y otros principios de diseño han decantado en la investigación que finalmente condujo a la arquitectura de software y a los estilos arquitectónicos tal como se los conoce hoy en día.

Estilos y patrones

La dinámica incontenible de la producción de patrones en la práctica de la arquitectura de software, su carácter entusiasta, cuando no militante, y la profusión de sus manifestaciones han atenuado la idea de que los patrones de diseño constituyen sólo uno de los paradigmas, marcos o formas del diseño arquitectónico, cada uno de los cuales posee una historia y una fundamentación distinta, y presenta, como todas las cosas en este terreno, sus sesgos, sus beneficios y sus limitaciones. Todo el mundo acepta que existen diversas clases de patrones: de análisis, de arquitectura (divididos en progresivamente estructurales, sistemas distribuidos, sistemas interactivos, sistemas adaptables), de diseño (conductuales, creacionales, estructurales), de organización o proceso, de programación y los llamados idiomas, entre otros. Cada autor que escribe sobre el asunto agrega una clase diferente, y los estándares en vigencia no hacen ningún esfuerzo para poner un límite a la

proliferación de variedades y ejemplares. Como sea, concentrémonos inicialmente en los patrones de diseño.

Un patrón de diseño, obviamente, debe encajar por un lado con otros tipos de patrones imaginables y por el otro con la teoría, la práctica y los marcos que en general rigen el diseño. ¿Cuáles son las grandes estrategias de diseño, si puede saberse? Una vez más, cuando llega el momento de disponer en un mapa las estrategias, los marcos de referencia o los paradigmas de diseño disponibles se encuentra multitud de propuestas clasificatorias que a veces no coinciden siquiera en la identificación de la disciplina en que se formulan. De todas maneras, estimo que es razonable pensar que las estrategias mayores de diseño pueden subsumirse en un conjunto relativamente manejable de cuatro o cinco tipos, uno de los cuales es, precisamente, el diseño orientado por patrones. Estas estrategias no necesariamente son excluyentes y antagónicas, pero se encuentran bastante bien delimitadas en la literatura usual [Tek00].

Diseño arquitectónico basado en artefactos. Incluiría modalidades bien conocidas de diseño orientado a objetos, tales como el OMT de Rumbaugh [RBP+91] y el OAT de Booch [Boo91]. En OMT, que puede considerarse representativo de la clase, la metodología de diseño se divide en tres fases, que son Análisis, Diseño del Sistema y Diseño de Objetos. En la fase de análisis se aplican tres técnicas de modelado que son modelado de objetos, modelado dinámico y modelado funcional. En la fase de diseño de sistema tienen especial papel lo que Rumbaugh llama implementación de control de software y la adopción de un marco de referencia arquitectónico, punto en el que se reconoce la existencia de varios prototipos que permiten ahorrar esfuerzos o se pueden tomar como puntos de partida. Algunos de esos marcos de referencia se refieren con nombres tales como transformaciones por lotes, transformaciones continuas, interfaz interactiva, simulación dinámica, sistema en tiempo real y administrador de transacciones [RBP+91:211-216]. No cuesta mucho encontrar la analogía entre dichos marcos y los estilos arquitectónicos, concepto que en esa época todavía no había hecho su aparición. En el señalamiento de las ventajas del uso de un marco preexistente también puede verse un reflejo de la idea de patrón, una categoría que no aparece jamás en todo el marco de la OMT, aunque ya había sido propuesta por el arquitecto británico Christopher Alexander varios años antes, en 1977 [Ale77].

Diseño arquitectónico basado en casos de uso. Un caso de uso se define como una secuencia de acciones que el sistema proporciona para los actores [JBR99]. Los actores representan roles externos con los que el sistema debe interactuar. Los actores, junto con los casos de uso, forman el modelo de casos de uso. Este se define como un modelo de las funciones que deberá cumplir el sistema y de su entorno, y sirve como una especie de contrato entre el cliente y los desarrolladores. El Proceso Unificado de Jacobson, Booch y Rumbaugh [JBR99] aplica una arquitectura orientada por casos de uso. El PU consiste en *core workflows* que definen el contenido estático del proceso y describen el proceso en términos de actividades, operadores (*workers*) y artefactos. La organización del proceso en el tiempo se define en fases. El PU se compone de seis de estos *workflows*: Modelado de Negocios, Requerimientos, Análisis, Diseño, Implementación y Prueba. Estos

diagramas de flujo resultan en los siguientes modelos separados: modelo de negocio & dominio, modelo de caso de uso, modelo de análisis, modelo de diseño, modelo de implementación y modelo de prueba. De acuerdo con el punto de vista, el concepto de estilo puede caer en diversas coordenadas del modelo, en las cercanías de las fases y los modelos de análisis y diseño. La oportunidad es propicia para subrayar que mientras el concepto de patrón responde con naturalidad a un diseño orientado a objetos, el de estilo arquitectónico posee un carácter estructural diferente y es de hecho un recién llegado al mundo del modelado O-O. La estrategia de diseño basada en casos de uso, dominada ampliamente por la orientación a objetos (tantos en los modelos de alto nivel como en la implementación) y por notaciones ligadas a UML, está experimentando recién en estos días reformulaciones y extensiones a fin de acomodarse a configuraciones arquitectónicas que no están estrictamente articuladas como objetos (basadas en servicios, por ejemplo), así como a conceptos de descripción arquitectónica y estilos [Hil99] [Stö01] [GP02] [Har03].

Diseño arquitectónico basado en línea de producto. Comprende un conjunto de productos que comparten una colección de rasgos que satisfacen las necesidades de un determinado mercado o área de actividad. Esta modalidad ha sido impulsada y definida sobre todo por Clements, Northrop, Bass y Kazman [CN96] [BCK98]. En la estrategia de arquitectura de Microsoft, este modelo está soportado por un profuso conjunto de lineamientos, herramientas y patrones arquitectónicos específicos, incluyendo patrones y modelos .NET para aplicaciones de línea de negocios, modelos aplicativos en capas como arquitecturas de referencia para industrias, etcétera [http://www.microsoft.com/resources/practices/].

Diseño arquitectónico basado en dominios. Considerado una extensión del anterior, se origina en una fase de análisis de dominio que, según Prieto-Díaz y Arango, puede ser definido como la identificación, la captura y la organización del conocimiento sobre el dominio del problema, con el objeto de hacerlo reutilizable en la creación de nuevos sistemas [PA91]. El modelo del dominio se puede representar mediante distintas formas de representación bien conocidas en ingeniería del conocimiento, tales como clases, diagramas de entidad-relación, *frames*, redes semánticas y reglas. Se han publicado diversos métodos de análisis de dominio [Gom92], [KCH+90], [PA91], [SCK+96] y [Cza99]. Se han realizado diversos *surveys* de estos métodos, como [Arr94] y [WP92]. En [Cza98] se puede encontrar una referencia completa y relativamente actualizada de los métodos de ingeniería de dominios. Relacionado con este paradigma se encuentra la llamada arquitectura de software específica de dominio (DSSA*)* [HR4] [TC92]. DSSA se puede considerar como una arquitectura de múltiples *scopes*, que deriva una descripción arquitectónica para una familia de sistemas antes que para un solo sistema. Los artefactos básicos de una estrategia DSSA son el modelo de dominio, los requerimientos de referencia y la arquitectura de referencia. El método comienza con una fase de análisis del dominio sobre un conjunto de aplicaciones con problemas o funciones comunes. El análisis se basa en *escenarios*, a partir de los cuales se derivan requerimientos funcionales, información de flujo de datos y de flujo de control. El modelo de dominio

incluye escenarios, el diccionario de dominio, los diagramas de bloque de contexto, los diagramas ER, los modelos de flujo de datos, los diagramas de transición de estado y los modelos de objeto. En la próxima versión mayor de Visual Studio.NET ("Whidbey"), Microsoft incluirá, al lado de los lenguajes convencionales de propósito general, un conjunto de lenguajes específicos de dominio (DSL), tales como Business Process DSL, Web Service Interaction DSL, Business Entity DSL y Logical Systems Architecture DSL. Esos lenguajes de alto nivel, mayormente gráficos, han sido anticipados por Keith Short [Sho03] en un documento reciente.

Diseño arquitectónico basado en patrones. Las ideas del mitológico pionero Christopher Alexander [Ale77] sobre lenguajes de patrones han sido masivamente adoptadas y han conducido a la actual efervescencia por los patrones de diseño, sobre todo a partir del impulso que le confirieron las propuestas de la llamada "Banda de los Cuatro": Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides [GoF95]. Similares a los patrones de Alexander, los patrones de diseño de software propuestos por la Banda buscan codificar y hacer reutilizables un conjunto de principios a fin de diseñar aplicaciones de alta calidad. Los patrones de diseño se aplican en principio sólo en la fase de diseño, aunque a la larga la comunidad ha comenzado a definir y aplicar patrones en las otras etapas del proceso de desarrollo, desde la concepción arquitectónica inicial hasta la implementación del código. Se han definido, por ejemplo, lenguas o idiomas (*idioms*) en la fase de implementación [Cop92] que mapean diseños orientados a objeto sobre constructos de lenguaje orientado a objeto. Otros autores han aplicado patrones en la fase de análisis para derivar modelos analíticos [Fow96]. Finalmente (y esto es lo que más interesa en nuestro contexto), los estilos se han aplicado en la fase de análisis arquitectónico en términos de patrones de arquitectura [BMR+96]. Estos patrones de arquitectura son similares a los patrones de diseño pero se concentran en la estructura de alto nivel del sistema. Algunos autores sienten que estos patrones arquitectónicos son virtualmente lo mismo que los estilos [Shaw94] [SG95], aunque está claro que ocurren en diferentes momentos del ciclo corresponden a distintos niveles de abstracción.

Los patrones abundan en un orden de magnitud de tres dígitos, llegando a cuatro, y gana más puntos quien más variedades enumera; los sub-estilos se han congelado alrededor de la veintena agrupados en cinco o seis estilos mayores, y se considera mejor teórico a quien los subsume en el orden más simple. Aún cuando los foros de discusión abundan en pullas de los académicos por el desorden proliferante de los patrones y en quejas de los implementadores por la naturaleza abstracta de los estilos, desde muy temprano hay claras convergencias entre ambos conceptos, aún cuando se reconoce que los patrones se refieren más bien a prácticas de re-utilización y los estilos conciernen a teorías sobre la estructuras de los sistemas a veces más formales que concretas. Algunas formulaciones que describen patrones pueden leerse como si se refirieran a estilos, y también viceversa. Escribe Alexander, en su peculiar lenguaje aforístico:

Como un elemento en el mundo, cada patrón es una relación entre cierto contexto, cierto sistema de fuerzas que ocurre repetidas veces en ese contexto y cierta configuración espacial que permite que esas fuerzas se resuelvan. Como un elemento de lenguaje, un

patrón es una instrucción que muestra la forma en que esta configuración espacial puede usarse, una y otra vez, para resolver ese sistema de fuerzas, donde quiera que el contexto la torne relevante

El patrón es, en suma, al mismo tiempo una cosa que pasa en el mundo y la regla que nos dice cómo crear esa cosa y cuándo debemos crearla. Es tanto un proceso como una cosa; tanto una descripción de una cosa que está viva como una descripción del proceso que generará esa cosa.

Casi un cuarto de siglo más tarde, Roy Fielding [Fie00] reconoce que los patrones de Alexander tienen más en común con los estilos que con los patrones de diseño de la investigación en OOPL. Las definiciones relacionadas con los patrones y las prácticas son diversas, y no he podido encontrar (excepto en la literatura derivativa) dos caracterizaciones que reconozcan las mismas clases o que se sirva de una terminología estable.

Un protagonista de primera línea en el revuelo que se ha suscitado en torno de los patrones es el texto de Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad y Michael Stal *Pattern-oriented software architecture (POSA).* En él los patrones arquitectónicos son aproximadamente lo mismo que lo que se acostumbra definir como estilos y ambos términos se usan de manera indistinta. En *POSA* los patrones "expresan un esquema de organización estructural para los sistemas de software. Proporcionan un conjunto de subsistemas predefinidos, especifica sus responsabilidades e incluye reglas y lineamientos para organizar la relación entre ellos". Algunos patrones coinciden con los estilos hasta en el nombre con que se los designa. Escriben los autores: "El patrón arquitectónico de tubería-filtros proporciona una estructura para sistemas que procesan un flujo de datos. Cada paso del procesamiento está encapsulado en un componente de filtro. El dato pasa a través de la tubería entre los filtros adyacentes. La recombinación de filtros permite construir familias de sistemas interrelacionados" [BMR+96: 53].

Tipo de Patrón	Comentario	Problemas	Soluciones	Fase de Desarrollo
Patrones de Arquitectura	Relacionados a la interacción de objetos dentro o entre niveles arquitectónicos	Problemas arquitectónicos, adaptabilidad a requerimientos cambiantes, performance, modularidad, acoplamiento	Patrones de llamadas entre objetos (similar a los patrones de diseño), decisiones y criterios arquitectónicos, empaquetado de funcionalidad	Diseño inicial
Patrones de	Conceptos de	Claridad de diseño,	Comportamiento	Diseño

Diseño	ciencia de computación en general, independiente de aplicación	multiplicación de clases, adaptabilidad a requerimientos cambiantes, etc	de factoría, Clase-Responsabilidad-Contrato (CRC)	detallado
Patrones de Análisis	Usualmente específicos de aplicación o industria	Modelado del dominio, completitud, integración y equilibrio de objetivos múltiples, planeamiento para capacidades adicionales comunes	Modelos de dominio, conocimiento sobre lo que habrá de incluirse (p. ej. *logging* & reinicio)	Análisis
Patrones de Proceso o de Organización	Desarrollo o procesos de administración de proyectos, o técnicas, o estructuras de organización	Productividad, comunicación efectiva y eficiente	Armado de equipo, ciclo de vida del software, asignación de roles, prescripciones de comunicación	Planeamiento
Idiomas	Estándares de codificación y proyecto	Operaciones comunes bien conocidas en un nuevo ambiente, o a través de un grupo. Legibilidad, predictibilidad.	Sumamente específicos de un lenguaje, plataforma o ambiente	Implementación, Mantemimiento, Despliegue

Tabla 4 - Taxonomía (parcial) de patrones [Sar00]

Mientras en *POSA* y en la mayoría de los textos de la especialidad el vínculo que se establece con mayor frecuencia es entre estilos y patrones de arquitectura, Robert Monroe, Andrew Kompanek, Ralph Melton y David Garlan exploran más bien la estrecha relación entre estilos y patrones de diseño [MKM+97]. En primer lugar, siguiendo a Mary Shaw [Shaw96], estiman que los estilos se pueden comprender como clases de patrones, o tal vez más adecuadamente como lenguajes de patrones. Un estilo proporciona de este modo un lenguaje de diseño con un vocabulario y un *framework* a partir de los cuales los arquitectos pueden construir patrones de diseño para resolver problemas específicos. Los estilos se vinculan además con conjuntos de usos idiomáticos o patrones de arquitectura que ofician como microarquitecturas; a partir de ellas es fácil derivar patrones de diseño. Lo mismo se aplica cuando un patrón de diseño debe

coordinar diversos principios que se originan en más de un estilo de arquitectura. Sea cual fuere el caso, los estilos se comprenden mejor –afirma Monroe– como un lenguaje para construir patrones (análogo a las metodologías OO como OMT) y no como una variante peculiar de patrones de diseño, como pretenden Gamma y sus colegas. Aunque los aspectos de diseño involucrados en estilos, patrones y modelos de objeto se superponen en algunos respectos, ninguno envuelve por completo a los demás; cada uno tiene algo que ofrecer, bajo la forma de colecciones de modelos y de mecanismos de representación.

Al lado de las propuestas meta-estilísticas que hemos visto, puede apreciarse que toda vez que surge la pregunta de qué hacer con los estilos, de inmediato aparece una respuesta que apunta para el lado de las prácticas y los patrones. Podría decirse que mientras los estilos han enfatizado descriptivamente las configuraciones de una arquitectura, desarrollando incluso lenguajes y notaciones capaces de expresarlas formalmente, los patrones, aún los que se han caracterizado como arquitectónicos, se encuentran más ligados al uso y más cerca del plano físico, sin disponer todavía de un lenguaje de especificación y sin estar acomodados (según lo testimonian innumerables talleres de OOPSLA y otros similares) en una taxonomía que ordene razonablemente sus clases y en un mapa que los sitúe inequívocamente en la trama de los grandes marcos de referencia. Aunque todavía no se ha consensuado una taxonomía unificada de estilos, ellos ayudarán, sin duda, a establecer y organizar los contextos en que se implementen los patrones.

En cuanto a los patrones de arquitectura, su relación con los estilos arquitectónicos es perceptible, pero indirecta y variable incluso dentro de la obra de un mismo autor. En 1996, Mary Shaw bosquejó algunos patrones arquitectónicos que se asemejan parcialmente a algunos estilos delineados por ella misma con anterioridad, pero no explora la relación de derivación o composición entre ambos conceptos. En esta oportunidad, un patrón arquitectónico se define por (1) un modelo de sistema que captura intuitivamente la forma en que están integrados los elementos; (2) componentes; (3) conectores que establecen las reglas de la interacción entre los componentes; y (4) una estructura de control que gobierna la ejecución. Entre los patrones referidos de este modo se encuentran la línea de tubería (*pipeline*), el patrón arquitectónico de abstracción de datos, los procesos comunicantes a través de mensajes, los sistemas de invocación implícita, los repositorios, los intérpretes, programa principal/subrutinas y sistemas en capas [Shaw96]. A pesar que muchos de sus ejemplos de referencia remiten a la literatura sobre estilos, la palabra "estilo" no figura en todo el cuerpo del documento. Generalmente se reconoce que se trata de una misma idea básica reformulada con una organización ligeramente distinta [Now99].

Robert Allen [All97] sostiene que los patrones son similares a los estilos en la medida en que definen una familia de sistemas de software que comparte características comunes, pero también señala que difieren en dos importantes respectos. En primer lugar, un estilo representa específicamente una familia arquitectónica, construida a partir de bloques de construcción *arquitectónicos*, tales como los componentes y los conectores. Los patrones, en cambio, atraviesan diferentes niveles de abstracción y etapas del ciclo de vida

partiendo del análisis del dominio, pasando por la arquitectura de software y yendo hacia abajo hasta el nivel de los lenguajes de programación. En segundo lugar, la comunidad que promueve los patrones se ha concentrado, hasta la fecha, en el problema de vincular soluciones con contextos de problemas y en definir cómo seleccionar los patrones apropiados, más que en la descripción y análisis de soluciones prospectivas. Por esta razón, los patrones se basan en presentaciones informales en estilo de libro de texto de familias de sistemas, más que en representaciones de sistemas precisas y semánticamente ricas. Asimismo, los patrones de interacción típicamente se dejan implícitos en el patrón de composición del sistema, en vez de ser tratados como conceptos dignos de ser tratados en sus propios términos.

En [SC96] Mary Shaw y Paul Clements proponen una cuidadosa discriminación entre los estilos arquitectónicos existentes, seguida de una guía de diseño que oriente sobre la forma de escoger el estilo apropiado en un proyecto, dado que cada uno de ellos es apropiado para ciertas clases de problemas, pero no para otras. Para llevar a cabo su propósito, establecen una estrategia de clasificación bidimensional considerando por un lado variables de control y por el otro cuestiones relativas a datos como los dos ejes organizadores dominantes. Una vez hecho esto, sitúan los estilos mayores dentro del cuadro y luego utilizan discriminaciones de grano más fino para elaborar variaciones de los estilos. Esta operación proporciona un *framework* para organizar una guía de diseño, que parcialmente nutren con "recetas de cocina" (*rules of thumb*). Está muy claro en este discurso el intento de derivar cánones de uso conducentes a lo que hoy se conoce como patrones, lo cual es explícito a lo largo del estudio. Su objetivo es derivar orientaciones pragmáticas a partir de las particularidades estilísticas. Incluyo una versión ligeramente alterada de las recetas de Shaw y Clements no para su uso heurístico, sino para ilustrar la modalidad de razonamiento y su convergencia con la literatura de prácticas.

Si su problema puede ser descompuesto en etapas sucesivas, considere el estilo secuencial por lotes o las arquitecturas en tubería.

Si además cada etapa es incremental, de modo que las etapas posteriores pueden comenzar antes que las anteriores finalizen, considere una arquitectura en tubería.

Si su problema involucra transformaciones sobre continuos flujos de datos (o sobre flujos muy prolongados), considere una arquitectura en tuberías.

Sin embargo, si su problema involucra el traspaso de ricas representaciones de datos, evite líneas de tubería restringidas a ASCII.

Si son centrales la comprensión de los datos de la aplicación, su manejo y su representación, considere una arquitectura de repositorio o de tipo de dato abstracto. Si los datos son perdurables, concéntrese en repositorios.

Si es posible que la representación de los datos cambie a lo largo del tiempo de vida del programa, entonces los tipos de datos abstractos pueden confinar el cambio en el interior de componentes particulares.

Si está considerando repositorios y los datos de entrada son ruidosos (baja relación señal-ruido) y el orden de ejecución no puede determinarse a priori, considere una pizarra [Nii86].

Si está considerando repositorios y el orden de ejecución está determinado por un flujo de requerimientos entrantes y los datos están altamente estructurados, considere un sistema de gestión de base de datos.

Si su sistema involucra continua acción de control, está embebido en un sistema físico y está sujeto a perturbaciones externas impredecibles de modo que los algoritmos prestablecidos se tornan ineficientes, considere una arquitectura de bucle cerrado de control [Shaw95a].

Si ha diseñado una computación pero no dispone de una máquina en la que pueda ejecutarla, considere una arquitectura de intérprete.

Si su tarea requiere un alto grado de flexibilidad/configurabilidad, acoplamiento laxo entre tareas y tareas reactivas, considere procesos interactivos.

Si tiene motivos para no vincular receptores de señales con sus originadores, considere una arquitectura de eventos.

Si las tareas son de naturaleza jerárquica, considere un *worker* replicado o un estilo de pulsación (*heartbeat*).

Si las tareas están divididas entre productores y consumidores, considere cliente/servidor.

Si tiene sentido que todas las tareas se comuniquen mutuamente en un grafo totalmente conexo, considere un estilo de *token passing*.

Kazman y Klein consideran que los estilos arquitectónicos son artefactos de ingeniería importantes porque definen *clases* de diseño, junto con sus propiedades conocidas. Ofrecen evidencia basada en la experiencia sobre cómo se ha utilizado cada clase históricamente, junto con razonamiento cualitativo para explicar por qué cada clase posee propiedades específicas. “Usar el estilo tubería-filtros cuando se desea reutilización y la performance no es una prioridad cardinal” es un ejemplo del tipo de descripción que se encuentra en las definiciones de ese estilo. Los estilos son entonces una poderosa herramienta para el reutilizador porque proporcionan una sabiduría decantada por muchos diseñadores precedentes que afrontaron problemas similares. El arquitecto puede echar mano de esa experiencia de la misma forma en que los patrones de diseño orientados a objeto dan a los más novicios acceso a una vasta fuente de experiencia elaborada por la comunidad de diseño orientada a objetos [Gof95].

Los estilos como valor contable

Un segmento importante de los estudios relativos a estilos se abocan a examinar su relación con los lenguajes de descripción de arquitecturas, que hemos examinado en un documento aparte [SC96] [Rey04b]. De hecho, los más importantes estudios comparativos de

ADLs consideran que la capacidad de expresar un estilo es definitoria entre las prestaciones de estos lenguajes [CI96]. En [AAG95] Gregory Abowd, Robert Allen y David Garlan proponen un *framework* formal para la definición de los estilos arquitectónicos que permita realizar análisis dentro de un estilo determinado y entre los diferentes estilos. Pero la pregunta que cabe formularse en un estudio como el presente es no tanto qué pueden hacer los arquitectos a propósito de los estilos, sino cuál es la utilidad de éstos, no ya para la arquitectura de software como un fin en sí misma, o como un logro sintáctico-semántico para la consolidación de una nomenclatura, sino en relación con los fines pragmáticos a los que esta disciplina emergente debe servir.

David Garlan [Gar95] [GKM+96] define en primer lugar cuatro aspectos salientes de los estilos arquitectónicos de los que todo modelo debe dar cuenta: proporcionar un vocabulario de elementos de diseño, establecer reglas de composición, definir una clara interpretación semántica y los análisis de consistencia y adecuación que pueden practicarse sobre él . Luego de ello caracteriza tres formas de comprender un estilo:

Estilo como lenguaje. En esta perspectiva, un vocabulario estilístico de diseño se modela como un conjunto de producciones gramaticales. Las reglas de configuración se definen como reglas de una gramática independientes del contexto o sensitivas a él. Se puede entonces asignar una interpretación semántica usando cualquiera de las técnicas estándar para asignar significado a los lenguajes. Los análisis serían los que se pueden ejecutar sobre "programas" arquitectónicos: verificar satisfacción de reglas gramaticales, análisis de flujo, compilación, etcétera. Característica de esta perspectiva es la elaboración de Aboud, Allen y Garlan [AAG93], donde un estilo se percibe como una semántica denotacional para los diagramas de arquitectura.

Estilo como sistema de tipos. En esta perspectiva el vocabulario arquitectónico se define como un conjunto de tipos. Por ejemplo, un estilo de tubería y filtros define tipos como filtros y tuberías. Si se especifica en un contexto orientado a objetos, son posibles las definiciones jerárquicas: "filtro" sería una subclase de un "componente" más genérico, y "tubería" una subclase de un "conector". Se pueden mantener restricciones sobre estos tipos como invariantes de los tipos de datos, realizadas operacionalmente en el código de los procedimientos que pueden modificar las instancias de los tipos. De allí en más, el análisis puede explotar el sistema de tipos, ejecutar verificación de tipo y otras manipulaciones arquitectónicas que dependen de los tipos involucrados.

Estilo como teoría. En esta perspectiva, un estilo se define como un conjunto de axiomas y reglas de inferencia. El vocabulario no se representa directamente, sino en términos de las propiedades lógicas de los elementos. Por ejemplo, el hecho de que un componente arquitectónico sea un filtro permitiría deducir que sus puertos son o bien de entrada o bien de salida. De la misma manera, el hecho de que algo sea una tubería permite deducir que posee dos extremos, uno para lectura y el otro para escritura. Las restricciones de configuración se definen como axiomas adicionales. El análisis tiene lugar mediante la prueba de un nuevo teorema que extiende la teoría de dicho estilo. Representativo de esta estrategia es el trabajo de Mark Moriconi y sus colegas [MQ94, MQR95].

Más allá de los matices que aporta cada mirada, es constante la referencia a los estilos como facilitadores del proceso de reutilización. Para David Garlan, Andrew Kompanek, Ralph Melton y Robert Monroe, los diseñadores de sistemas reconocen cada vez con más frecuencia la importancia de explotar el conocimiento de diseño en la ingeniería de un nuevo sistema. Una forma de hacerlo es definir un estilo arquitectónico. Antes que llegue el momento de pensar en patrones más cercanos a lo que ha de suceder en la máquina, el estilo determina un vocabulario coherente de elementos de diseño y reglas para su composición. Estructur el espacio de diseño para una familia relacionada en un estilo puede, en principio, simplificar drásticamente el proceso de construcción de un sistema, reducir costos de implementación a través de una infraestructura reutilizable y mejorar la integridad del sistema a través de análisis y verificaciones específicas del estilo [GKM+96]. Precisando un poco más la idea, los autores enumeran elocuentes ventajas del método:

Los estilos promueven reutilización de diseño. Las soluciones de rutina con propiedades bien entendidas se pueden aplicar otra vez a nuevos problemas con alguna confianza.

El uso de estilos puede conducir a una significativa reutilización de código. Los aspectos invariantes de un estilo conduce a replicar implementaciones. Por ejemplo, se pueden usar primitivas de un sistema tubería-filtros para manejar tareas semejantes de agendado de tareas, sincronización y comunicación a través de tuberías, o un sistema cliente-servidor puede tomar ventaja de mecanismos de RPC preexistentes y de capacidades de generación de *stub*.

Es más fácil para otros entender la organización de un sistema si se utilizan estructuras convencionales. Por ejemplo, la tipificación de un sistema como “cliente-servidor” de inmediato evoca una fuerte imagen respecto a cuales son sus piezas y cómo se vinculan recíprocamente.

El uso de estilos estandarizados sustenta la interoperabilidad. Por ejemplo, el protocolo de pila del modelo OSI para arquitecturas en red, o los estándares de XML y SOAP para intercambios de mensaje en arquitecturas orientadas a servicios.

Al acotar el espacio de diseño, un estilo arquitectónico permite de inmediato análisis especializados específicos del estilo. Por ejemplo, se pueden utilizar herramientas ya probadas para un estilo tubería-filtros para determinar la capacidad, la latencia y la ausencia de abrazos mortales. Esos análisis no estarían disponibles si el sistema fuera categorizado de una manera ad hoc, o si se implementara otro estilo.

Es usualmente posible, e incluso deseable, proporcionar visualizaciones específicas de un estilo. Esto proporciona representaciones gráficas y textuales que coinciden con intuiciones específicas de dominio sobre cómo deben representarse los diseños en él [GKM+96].

La misma idea es expresada en forma parecida por Nenan Medvidovic. En su opinión, la investigación en arquitectura de software se orienta a reducir costos de desarrollo de aplicaciones y aumentar el potencial para la comunalidad entre diferentes miembros de una familia estrechamente relacionada de productos. Un aspecto de esta investigación es el

desarrollo de estilos de arquitectura de software, formas canónicas de organizar los componentes en una familia de estilos. Típicamente, los estilos reflejan y explotan propiedades clave de uno o más dominios de aplicaciones y patrones recurrentes de diseño de aplicaciones dentro de ese dominio. Como tales, los estilos arquitectónicos poseen potencial para proporcionar una estructura pre-armada, "salida de la caja", de reutilización de componentes [MT97].

Un concepto complementario es el de discordancia arquitectónica (*architectural mismatch*), que describe conflictos de interoperabilidad susceptibles de encontrarse en el plano arquitectónico, anticipándose a su manifestación en las fases de diseño o implementación. Ha sido acuñado por David Garlan, Robert Allen y John Ockerbloom [GAO95] y refrendado por Ahmed Abd-Allah [Abd96] y Ramesh Sitaraman [Sit97] en el contexto de estudios de casos utilizando Aesop; pero si bien la idea es de manifiesta utilidad, no ha sido generalizada al conjunto de estilos ni se ha generado una teoría acabada de composición o predicción arquitectónica, señalándose que para ello se requiere ulterior investigación [KGP+99] [MM04].

Todos estos nuevos conceptos y exploraciones, por más preliminares que sean, son por cierto muy provechosos. Allí donde existan lenguajes formales para la especificación rigurosa de estilos, la cosa tendrá además cierta respetabilidad científica. Pero si bien la literatura usual exalta la conveniencia de pensar en términos teóricos de estilo como algo que tendrá incidencia en las estructuras de las prácticas ulteriores, se han hecho muy pocas evaluaciones de las relaciones entre las arquitecturas beneficiadas por la aplicación de la idea de estilo (o cualesquiera otros conceptos estructurales) con factores sistemáticos de beneficio, variables de negocios, calidad de gestión operacional, fidelidad al requerimiento, multiplicación de eficiencia, satisfacción del cliente, valor agregado, *time to market*, ahorro de mantenimiento y guarismos de TCO o retorno de inversión. La pregunta clave es: ¿Valen los estilos lo que cuestan? ¿Se ha valorizado el *tradeoff* entre la *reusabilidad* que se goza en el momento de diseño y la *inusabilidad* por saturación de ancho de banda (o por otras razones bien conocidas) en que podría llegar a traducirse? [Kru92] [Big94] [GAO95] [Shaw95b] ¿Alcanza con admitir o advertir sobre eventuales desventajas sin proporcionar métricas exhaustivas, modelos rigurosos de comparación, *benchmarks*? ¿Alcanza con evaluar la performance de distintos ADLs que sólo prueban la eficiencia del modelo arquitectónico abstracto, sin medir la resultante de aplicar un estilo en lugar de otro (o de ninguno) en una solución final?

Un problema adicional es que distintos patrones presuponen diferentes metodologías de diseño. Las arquitecturas de llamada y retorno, por ejemplo, se asocian históricamente a estructuras de programas y jerarquías de control que a su vez se vinculan a notaciones específicas, como los diagramas de Warnier-Orr o los diagramas de Jackson, elaboradas en la década de 1970. Es bien sabido que las arquitecturas orientadas a objetos requieren otras metodologías, y lo mismo se aplica a los estilos modulares orientados a componentes [Pre02: 226-233]. Hay toda una variedad de métodos de diseño, con literaturas y herramientas específicas: Hipo II, OOD (incluyendo RUP), *cleanroom*, Gist, Leonardo, KBSA. Hay otro conjunto parecido que versa sobre modelos de proceso:

lineal-secuencial, de prototipos, RAD, incremental, en espiral, *win-win*, de desarrollo concurrente, basado en componentes, PSP, CMM, CMMI. Últimamente ha surgido un conjunto de métodos autodenominados ágiles en relación confusa con todo el campo: eXtreme Programming, Scrum, Crystal Methods, Lean Development, Feature Driven Development, Agile Modeling y una formalización ágil de RAD, Dynamic System Development Method [CLC02].

Ni hablar de técnicas de prototipado, modelos de ciclo de vida e ingeniería de requerimientos, que se han constituido en ciencias aparte. Ninguno de estos métodos y procesos, complejamente relacionados entre sí, se ha estudiado en relación con factores estilísticos, como si la decisión de escoger un estilo sobre otro no tuviera consecuencias metodológicas. El estado de arte de la tipificación de métodos de diseño de DACS, por ejemplo, se remonta a una época anterior a la consolidación de la arquitectura basada en estilos y al surgimiento de los patrones. Los textos sobre estilos, con las excepciones anotadas, suelen ser silenciosos sobre cuestiones ingenieriles, sean ellas clásicas o posmodernas. Mientras los métodos ágiles suelen ser locuaces respecto del uso de patrones, son en cambio reticentes en lo que concierne a arquitectura en general y estilos en particular; y también viceversa.

En el viejo modelado con OMT el requerimiento del usuario penetraba en la semántica y la estructura del modelo hasta bien consumada la fase analítica del ciclo de diseño y el usuario debía comprender, cuando no homologar, las conclusiones del análisis. En contraste, da la impresión que los estilos aparecen demasiado pronto en el ciclo, a un nivel de abstracción que es demasiado alto para llegar a impactar en el código, pero que ya en nada se asemeja a lo que el usuario piensa. La respuesta a estas y otras inquietudes que circulan en foros y listas de interés pero que pocos transportan a libros y ponencias, es que el sentido común dicta que los estilos pueden ser beneficiosos, que *deben* serlo, aunque taxativamente no se sabe a ciencia cierta si así es. La postura del autor de este documento sostiene que es posible que los estilos constituyan una idea esencial en la eficacia de la solución, tanto o más que los patrones y con menos riesgo de catástrofe que el que supone (digamos) una metodología como XP entendida a la manera fundamentalista [BMM+98] [Beck99]; pero esta seguirá siendo una expresión de deseos en tanto el campo no impulse la búsqueda de un fuerte respaldo cuantitativo y una buena provisión de estudios de casos. Si bien se percibe por ejemplo en *Forum Risks* o en *ACM Software Engineering Notes* que las causas de riesgo reportadas se originan abrumadoramente en decisiones primarias de diseño, a diferencia de los patrones (que disponen ya de amplias colecciones de anti-patrones que ilustran los errores estructurales más comunes), los estilos no cuentan todavía con una buena teoría de los anti-estilos.

Por ahora, se percibe un movimiento emergente que procura establecer el rol de decisiones arquitectónicas como la elección de estilo en el ciclo de vida, en la correspondencia con los requerimientos y en el costo, riesgo y calidad de la solución. Después de justificar su autonomía, una arquitectura que hasta ahora ha sido cualitativa y abstracta ha descubierto que alcanzará su plena justificación cuando clarifique su relación sistemática con una ingeniería cuantitativa y concreta. En lugar de definir su

posicionamiento en el ciclo de las metodologías tradicionales, se han desarrollado métodos específicos llamados diversamente Software Architecture Analysis Method (SAAM), Architecture Tradeoff Analysis Method (ATAM) [Kaz98] [KBK99], Quality Attribute Workshop, Cost-Benefits Analysis Method (CBAM) [KAK01] y Attribute-Driven Design Method. Desde sus mismos nombres expresan con claridad y distinción que se está buscando determinar los nexos entre los requerimientos y la teoría, y las consecuencias de la teoría sobre la práctica subsiguiente. Están surgiendo también minuciosos estudios de casos de misión crítica, reportando evaluaciones de desarrollos estilísticamente orientados con presencia de contratistas, usuarios de interfaz y arquitectos participantes [BCL+03], y cuantificando en moneda el costo de las decisiones arquitectónicas [AKK01] [KAK02]. La mejor referencia unitaria por ahora es el estudio de Rick Kazman, Robert Nord y Mark Klein [KNK03]. Si bien las categorías metodológicas suenan prometedoras, detrás de esta constelación de métodos nuevos se perciben las iniciativas de unos pocos autores, en una línea de trabajo que posee menos de cinco años de sedimentación de experiencias.

Hay referencias a estas dimensiones en la Guía de Operaciones de Arquitectura de Microsoft Patterns & Practices (inscripta en Microsoft Operations Framework, MOF) [MS04a], así como una metodología formal completa basada en Microsoft Solution Framework para la verificación de patrones, en plena conformidad con la famosa "regla de tres", bien conocida para la comunidad nucleada alrededor de patrones y anti-patrones [AEA03] [http://www.antipatterns.com/whatisapattern/]. Entre los patrones testeados en esa metodología se enumeran algunos que están próximos al espíritu de los estilos: Model-View-Controller, filtros de intercepción, interfaz de servicios, aplicaciones en capas, replicación amo-esclavo, transferencia de datos.

Pero en la literatura usual de circulación abierta no hay todavía una evaluación prolija del impacto de las abstracciones de arquitectura y diseño sobre las dimensiones de consistencia con requerimientos que traje a colación, que para los técnicos de las escuelas Seca y Húmeda no serían funcionales, pero para las empresas lo son en grado sumo. Existe toda una inmensa disciplina especializada en evaluación de arquitecturas, dividida en escuelas y metodologías que han llegado a ser tantas que debieron clasificarse en tipos (evaluación basada en escenarios, en cuestionarios, en listas de verificación, en simulación, en métricas…); pero esta técnica está lejos de estar integrada a conceptos descriptivos y abstractos como los que dominan el campo estructural, y sólo de tarde en tarde se vinculan los métodos de evaluación con las variables de diseño teorizadas como estilos o puestas en práctica como patrones [KK99] [KKB+99] [Har03].

Grupos específicos, como el de Mark Moriconi y sus colegas, han insistido en vincular los estilos con procesos, utilizando los estilos para factorear patrones de refinamiento [MQR95]; Moriconi, vinculado a SRI International de Menlo Park, se ha destacado como estudioso de los efectos semánticos del cambio en las aplicaciones, la representación y el refinamiento de las especificaciones visuales, el papel de la lógica en el desarrollo y otros temas que vinculan la formalización más pura con la práctica más crítica. Pero lo que

afirmaba David Garlan en 1994 respecto de que todavía no estaba claro en qué medida y de qué manera se podían usar los estilos en relación con el refinamiento, sigue siendo verdad una década más tarde. Lo que Garlan mismo piensa a propósito de refinamiento, a fin de cuentas, se refiere a las dimensiones de optimización del software, antes que a la satisfacción de los requerimientos del cliente, que en última instancia no tienen mucho que ver con el orden de la computación [Gar94] [Gar96]. Toda la literatura que gira en torno de arquitectura de alto nivel en general y estilos en particular reconoce que las elaboraciones metodológicas son preliminares; esta afirmación, así como el resonante silencio respecto de la vigencia de las metodologías clásicas (con la excepción de OOD) llevan a pensar que en el futuro abundarán más las formulaciones *from scratch* de disciplinas de diseño basadas en arquitectura (con componentes como SAAM, ATAM, ADDM, CBAM, etc) que la aplicación de parches y extensiones a las estrategias antiguas.

El mismo retraso y preliminariedad se observa en una especialización que ha decantado como un conjunto de modelos económicos para evaluar el impacto de costo de reutilización de software. Se trata de un campo extremadamente desarrollado y nutrido; un *survey* como el de Edward Wiles (del Departamento de Ciencias de la Computación de la Universidad de Aberystwyth, Gales) ha cruzado nada menos que siete técnicas genéricas de análisis de inversión con veinticinco modelos evaluativos de reutilización, desde el famoso COCOMO de Barry Boehm y el modelo Gaffney-Durek a otros menos conocidos [Wil99]. Todos esos estudios se refieren a entidades de diseño y programación; en el campo de la arquitectura y los estilos habrá que trabajar bastante para llegar a semejante orden de magnitud. De los 5988 títulos que registra la bibliografía sobre reutilización de software de Lombard Hill (http://www.lombardhill.com) no hay uno solo que se refiera al impacto económico del uso de diseño basado en estilos. La gran expansión de la economía de software es de hecho anterior a la generalización de la arquitectura, los estilos y los patrones, cuyos procesos de rendición de cuentas recién se están estableciendo.

En fin, cabe concluir que todavía hay mucho por hacer en el posicionamiento de los estilos en un marco de vistas y metodologías, por no decir nada del contexto de negocios. Muchas veces se ha criticado el cuadro abarcativo de Zachman por su falta de especificidad técnica; pero tal vez haya que recuperar su espíritu y su visión integral para plantear cuestiones semejantes en el debido nivel de prioridad.

Conclusiones

En el desarrollo de este estudio, se han descripto en forma sucinta algunos estilos arquitectónicos representativos y señalado su posicionamiento en marcos y modelos de referencia más amplios, su lugar frente a la estrategia arquitectónica de Microsoft y sus vínculos y antagonismos con el campo emergente de los patrones de arquitectura y diseño.

A modo de síntesis, podría decirse que los estilos han llegado a la arquitectura de software para quedarse y que ya han hecho mella tanto en el desarrollo de los lenguajes específicos de descripción arquitectónica como en los lenguajes más amplios de modelado, así como en las recomendaciones envolventes de la IEEE. Tanto en los modelos de referencia estructurales como en las metodologías diacrónicas, los estilos ocupan un lugar decididamente más abstracto y un nicho temporal anterior al diseño orientado a patrones. Este es un campo por ahora amorfo, desestructurado, al cual los estilos bien podrían aportarle (en retribución por las ideas sobre reutilización) un indicador sobre cómo construir alguna vez las taxonomías, los ordenamientos y los modelos que a todas luces se están necesitando. Más allá de los logros formales de la arquitectura basada en estilos, es evidente que en el terreno de las relaciones entre teoría y práctica resta todavía mucho por hacer.

Referencias bibliográficas

[AG92] Robert Allen y David Garlan. "A formal approach to software architectures". *Proceedings of the IFIP Congress '92*, Setiembre de 1992.

[AAG93] Gregory Abowd, Robert Allen y David Garlan. "Using style to understand descriptions of software architecture". *Proceedings of SIGSOFT'93: Foundations of Software Engineering, Software Engineering Notes* 18(5), pp. 9-20. ACM Press, Diciembre de 1993.

[AAG95] Gregory Abowd, Robert Allen y David Garlan. "Formalizing style to understand descriptions of software architecture". *Technical Report*, CMU-CS-95-111, Enero de 1995.

[Abd96] Ahmed Abd-Allah. *Composing heterogeneous software architectures*. Tesis doctoral, Department of Computer Sciences, USC, Agosto de 1996.

[AEA+03] Mohammad Al-Sabt, Matthew Evans, Geethika Agastya, Dayasankar Saminathan, Vijay Srinivasan y Larry Brader. "Testing Software Patterns". Microsoft Patterns & Practices, Version 1.0.0. http://msdn.microsoft.com/architecture/patterns/Tsp/default.aspx, Octubre de 2003.

[AG96] Robert Allen y David Garlan, "The Wright Architectural Description Language", *Technical Report*, Carnegie Mellon University. Verano de 1996.

[AKK01] Jayatirtha Asundi, Rick Kazman, Mark Klein. "Using economic considerations to choose among architecture design alternatives". *Technical Report*, CMU/SEI-2001-TR-035, ESC-TR-2001-035, Diciembre de 2001.

[Ale77] Christopher Alexander. *A pattern language*. Oxford University Press, 1977.

[All97] Robert Allen. "A formal approach to Software Architecture". *Technical Report*, CMU-CS-97-144, 1997.

[And91] Gregory R. Andrews. “Paradigms for Process Interaction in Distributed Programs”. *ACM Computing Surveys*, 23(1), pp. 49-90, Marzo de 1991.

[Arr94] Guillermo Arango. “Domain Analysis Methods”. En *Software Reusability,* R. Schäfer, Prieto-Díaz y M. Matsumoto (Eds.), Ellis Horwood, Nueva York, pp. 17-49, 1994.

[BCD02] Marco Bernardo, Paolo Ciancarini, y Lorenzo Donatiello. “On The Formalization of Architectural Types With Process Algebras”. *Proceedings of the ACM Transactions on Software Engineering and Methodology,* 11(4):386-426, 2002.

[BCK98] Len Bass, Paul Clements y Rick Kazman. *Software Architecture in Practice.* Reading, Addison-Wesley, 1998.

[BCL+03] Mario Barbacci, Paul Clements, Anthony Lattanze, Linda Northrop, William Wood. “Using the Architecture Tradeoff Analysis Method^{SM} (ATAM^{SM}) to evaluate the software architecture for a product line of avionics systems: A case study”. *Technical Note*, CMU/SEI-2003-TN-012, Julio de 2003.

[Beck99] Kent Beck. *Extreme Programming Explained: Embrace change*. Reading, Addison-Wesley, 1999.

[Big94] T. J. Biggerstaff. “The Library Scaling Problem and the Limits of Concrete Component Reuse”. *Proceedings of the Third International Conference on Software Reuse*, pp. 102-109, Rio de Janeiro, Noviembre de 1994.

[BMM+98] William Brown, Raphael Malveau, Hays “Skip” McCormick, Thomas Mawbray. *Anti-Patterns: Refactoring software, architectures, and projects in crisis*. John Wiley & Sons, 1998.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad y Michael Stal. *Pattern-oriented software architecture – A system of patterns*. John Wiley & Sons, 1996.

[Boo91] Grady Booch. *Object-Oriented Design with Applications.* The Benjamin/Cummings Publishing Company, Inc, 1991.

[BRJ99] Grady Booch, James Rumbaugh e Ivar Jacobson. *El Lenguaje Unificado de Modelado*. Madrid, Addison-Wesley, 1999.

[Bro75] Frederick Brooks Jr. *The mythical man-month*. Reading, Addison-Wesley. 1975.

[Bro86] Frederick Brooks Jr. “No silver bullet: Essence and accidents of software engineering”. En H. G. Kluger (ed.), *Information Processing*, North Holland, Elsevier Science Publications, 1986.

[BSL00] Don Box, Aaron Skonnard y John Lam. *Essential XML. Beyond markup*. Reading, Addison-Wesley, 2000.

[Bur92] Steve Burbeck. “Application programming in Smalltalk-80: How to use Model-View-Controller (MVC)”. University of Illinois in Urbana-Champaign, Smalltalk Archive, http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html.

[BW98] Alan Brown y Kurt Wallnau. “The current state of CBSE”. *IEEE Software*, pp.37-46, Octubre de 1998.

[Cha03] Michael Champion. “Towards a reference architecture for Web services”. *XML Conference and Exposition 2003*, Filadelfia, 2003.

[Chu02] Martin Chung. *Publish-Subscribe Toolkit documentation for Microsoft BizTalk Server 2002*. MSDN Library, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbiz2k2/html/bts_PubSubEAI.asp, Mayo de 2002.

[Cle96] Paul Clements. “A Survey of Architecture Description Languages”. *Proceedings of the International Workshop on Software Specification and Design*, Alemania, 1996.

[CLC02] David Cohen, Mikael Lindvall y Patricia Costa. *Agile Software Development: A DACS State-of-the-art Report*. Nueva York, DACS, http://www.dacs.dtic.mil/techs/agile/agile.pdf, Enero de 2002.

[CN96] Paul Clements y Linda Northrop. “Software Architecture: An Executive Overview”. *Technical Report*, CMU/SEI-96-TR-003, Pittsburg, Carnegie Mellon University, 1996.

[Con58] M. Conway. “Proposal for a universal computer-oriented language”. *Communications of the ACM*, 1(10), pp. 5-8, Octubre de 1958.

[Cop92] James O. Coplien. *Advanced C++ – Programming Styles and Idioms.* Reading, Addison-Wesley, 1992.

[Cza98] Krzysztof Czarnecki. *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models.* Tesis de doctorado, Technical University of Ilmenau, 1998.

[DNR99] Elisabetta Di Nitto y David Rosenblum. “Exploiting ADLs to specify architectural styles induced by middleware infrastructures”. *Proceedings of the 21st International Conference on Software Engineering (ICSE’99)*, Los Angeles, 16 al 22 de Mayo de 1999.

[Dob02] Ernst-Erich Doberkat. “Pipes and filters: Modelling a software architecture through relations”. *Internes Memorandum des Fachbereich Informatik Lehrstuhl für Software Technologie*, Universidad de Dortmund, Memo N° 123, Junio de 2002.

[Fie00] Roy Thomas Fielding. “Architectural styles and the design of network-based software architectures”. Tesis doctoral, University of California, Irvine, 2000.

[FT02] Roy Thomas Fielding y Richard Taylor. “Principled design of the modern Web architecture”. *ACM Transactions on Internet Technologies*, 2(2), pp. 115-150, Mayo de 2002.

[Fow96] Martin Fowler. *Analysis Patterns: Reusable Object Models*. Reading, Addison-Wesley, 1996.

[Fow01] Martin Fowler. “Is design dead?”. *Proceedings of the XP 2000 Conference*, 2001.

[GAO94] David Garlan, Robert Allen y John Ockerbloom. “Exploiting style in architectural design environments”. *Proceedings of SIGSOFT'94: Foundations of Software Engineering*. ACM Press, Diciembre de 1994.

[GAO95] David Garlan, Robert Allen y John Ockerbloom. "Architectural Mismatch, or, Why It's Hard to Build Systems out of Existing Parts". *Proceedings of the 17th InternationalConference on Software Engineering*, pp. 179-185, Seattle, Abril de 1995.

[Gar95] David Garlan. "What is style". *Proceedings of Dagshtul Workshop on Software Architecture*. Febrero de 1995.

[Gar96] David Garlan. "Style-based refinement". En A. L. Wolf, ed., *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, pp. 72-75, San Francisco, Octubre de 1996.

[Gar00] David Garlan. "Software Architecture: A Roadmap". En *The future of software architecture*, A. Finkelstein (ed.), ACM Press, 2000.

[Gar01] David Garlan. "Next generation software architectures: Recent research and future directions". Presentación, Columbia University, Enero de 2001.

[GB98] Håkan Grahn y Jan Bosch. "Some initial performance characteristics of three architectural styles". *WOSP*, http://citeseer.nj.nec.com/50862.html, 1998.

[GD90] David Garlan y Norman Delisle. "Formal specifications as reusable frameworks". En *VDM'90: VDM and Z - Formal Methods in Software Development*, pp. 150-163, Kiel, Abril de 1990. Springer-Verlag, LNCS 428.

[GKM+96] David Garlan, Andrew Kompanek, Ralph Melton y Robert Monroe. "Architectural Style: An Object-Oriented Approach". http://www.cs.cmu.edu/afs/cs/project/able/www/able/papers_bib.html, Febrero de 1996.

[GoF95] Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides. *Design Patterns: Elements of reusable object-oriented software*. Reading, Addison-Wesley, 1995.

[Gog92] Joseph Goguen. "The dry and the wet". *Monograph PR-10*0, Programming Research Group, Oxford University Computer Laboratory, 1992.

[Gom92] Hassan Gomaa. "An Object-Oriented Domain Analysis and Modeling Method for Software Reuse". *Proceedings of the Hawaii International Conference on System Sciences*, Hawaii, Enero de 1992.

[GP92] Nicolas Guelfi, Gilles Perrouin. "Rigorous engineering of software architectures: Integrating ADLs, UML and development methodologies". Luxembourg Ministry of Higher Education and Research, Proyecto MEN/IST/01/001, 2002.

[GS94] David Garlan y Mary Shaw. "An introduction to software architecture". *CMU Software Engineering Institute Technical Report*, CMU/SEI-94-TR-21, ESC-TR-94-21, 1994.

[GSP99] R. F. Gamble, P. R. Stiger y R. T. Plant. "Rule-based systems formalized within a software architectural style". *Knowledge-Based Systems*, v. 12, pp. 13-26, 1999.

[Har03] Maarit Harsu. "From architectural requirements to architectural design". *Report 34*, Institute of Software Systems, Tampere University of Technology, Mayo de 2003

[Hil99] Rich Hilliard. "Using the UML for architectural descriptions". *Lecture Notes in Computer Science*, vol. 1723, 1999,

[Hil00] Rich Hilliard. “Impact assessment of the IEEE 1471 on The Open Group Architecture Framework”, http://www.opengroup.org/togaf/procs/p1471-togaf-impact.pdf.

[Hil01a] Rich Hilliard. “IEEE Std 1471 and beyond”. Position paper, SEI First Architecture Representation Workshop, 16 y 17 de enero de 2001.

[Hil01b] Rich Hilliard. “Three models for the description of architectural knowledge: Viewpoints, styles and patterns”. Presentado a WICSA-2, enero de 2001.

[HR94] Frederick Hayes-Roth. *Architecture-Based Acquisition and Development of Software: Guidelines and Recommendations from the ARPA Domain-Specific Software Architecture (DSSA) Program*, 1994.

[IEEE00] IEEE Std 1471-2000. “IEEE Recommended Practice for Architectural Description of Software Intensive Systems”, IEEE, 2000.

[Ince88] Darrell Ince. *Software development: Fashioning the Baroque*, Oxford Science Publications, Oxford University Press, New York, 1988.

[JBR99] Ivar Jacobson, Grady Booch y James Rumbaugh. *The Unified Software Development Process.* Reading, Addison-Wesley, 1999.

[KAK01] Rick Kazman, Jai Asundi y Mark Klein. “Quantifying the Costs and Benefits of Architectural Decisions,”. *Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001)*. Toronto, 12 al 19 de Mayo de 2001, pp. 297-306. Los Alamitos, IEEE Computer Society, 2001.

[KAK02] Rick Kazman, Jai Asundi y Mark Klein. “Making architecture design decisions: An economic approach”. *Technical Report*, CMU/SEI-2002-TR-035, ESC-TR-2002-035, Setiembre de 2002.

[Kay68] A. Kay. *FLEX, a flexible extensible language*. Tesis doctoral, Universidad de Utah, 1968.

[Kaz98] Rick Kazman. “The Architectural Tradeoff Analysis Method”. *Software Engineering Institute, CMU/SEI-98-TR-008*, Julio de 1998.

[Kaz01] Rick Kazman. “Software Architecture”. En *Handbook of Software Engineering and Knowledge Engineering*, S-K Chang (ed.), World Scientific Publishing, 2001.

[KBK99] Rick Kazman, Mario Barbacci, Mark Klein, Jeromy Carrière y Steven Woods, “Experience with Performing Architecture Tradeoff Analysis,”. *Proceedings of the 21st International Conference on Software Engineering (ICSE 99),* Los Angeles, 16 al 22 de Mayo de 1999, pp. 54-63. Nueva York, Association for Computing Machinery, 1999.

[KCH+90] Kyo Kang, Sholom Cohen, James Hess, William Nowak y A. Spencer Peterson. “Feature-Oriented Domain Analysis (FODA) Feasibility Study”. *Technical Report*, CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Noviembee de 1990.

[KGP+99] R. Keshav, R. Gamble, J. Payton y K. Frasier. “Architecture integration elements”. *Technical Report UTULSA-MCS-16-99*, Mayo de 1999.

[KK99] Mark Klein y Rick Kazman. “Attribute-based architectural styles”. *Technical Report*, CMU/SEI-99-TR-022, ESC-TR-99-022, Carnegie Mellon University, Octubre de 1999.

[KKB+99] Mark Klein, Rick Kazman, Len Bass, Jeromy Carriere, Mario Barbacci y Howard Lipson. “Attribute-based architecture styles”. En Patric Donohoe (ed.), *Software Architecture*, pp. 225-243. Kluwer Academic Publishers, 1999.

[KNK03] Rick Kazman, Robert Nord y Mark Klein. “A life-cycle view of architecture analysis and design methods”. *Technical Note*, CMU/SEI-2003-TN-026, Setiembre de 2003.

[Kru92] Charles W. Krueger. “Software Reuse”. *ACM Computing Surveys*, vol. 24(2), pp. 131-183, Junio de 1992.

[Kru95] Philippe B. Kruchten. “Architectural blueprints: The 4+1 view model of architecture”. *IEEE Software*, 12(6):42–50, 1995.

[Land02] Rikard Land. “A brief survey of software architecture”. *Mälardalen Real-Time Research Center (MRTC) Report*. Västerås, Suecia, Febrero de 2002.

[Lar03] Craig Larman. *UML y Patrones*. 2ª edición, Madrid, Prentice Hall.

[LeM98] Daniel Le Métayer. “Describind architectural styles using graph grammars”. *IEEE Transactions of Software Engineering*, 24(7), pp. 521-533, 1998.

[Mit02] Kevin Mitchell. “A matter of style: Web Services architectural patterns”. *XML Conference & Exposition 2002*, Baltimore, 8 al 13 de diciembre de 2002.

[MKM+96] Robert Monroe, Andrew Kompanek, Ralph Melton y David Garlan. “Stylized architecture, design patterns, and objects”. http://citeseer.nj.nec.com/monroe96stylized.html.

[MKM+97] Robert Monroe, Andrew Kompanek, Ralph Melton y David Garlan. “Architectural Styles, design patterns, and objects”. *IEEE Software*, pp. 43-52, Enero de 1997.

[MM04] Nikunj Mehta y Nenad Medvidovic. “Toward composition of style-conformant software architectures”. *Technical Report*, USC-CSE-2004-500, University of Southern California Center for Software Engineering, Enero de 2004.

[MMP00] Nikunj Mehta, Nenad Medvidovic y Sandeep Phadke. “Towards a taxonomy of software connectors”. *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, pp. 178-187, Limerick, Irlanda, 4 al 11 de junio de 2000.

[MPG96] S. Murrell, R. Plant y R. Gamble. “Defining architectural styles for knowledge-based systems”. *AAAI-95, Workshop on Verification and Validation of Knowledge-Based Systems and Subsystem*s, pp. 51-58. Agosto de 1996.

[MQ94] Mark Moriconi y Xiaoliei Qian. “Correctness and composition of software architectures”. *Proceedings of ACM SIGSOFT’94: Symposium on Foundations of Software Engineering*. Nueva Orleans, Diciembre de 1994.

[MQR95] Mark Moriconi, Xiaolei Qian y Robert Riemenschneider. “Corrects architecture refinement”. *IEEE Transactions of Software Engineering*, 1995.

[MS02a] *Application Architecture for .NET: Designing applications and services*. Microsoft Patterns & Practices. http://msdn.microsoft.com/architecture/application/default.aspx?pull=/library/en-us/dnbda/html/distapp.asp, 2002.

[MS02b] *Application Conceptual View*. Microsoft Patterns & Practices. http://msdn.microsoft.com/architecture/application/default.aspx?pull=/library/en-us/dnea/html/eaappconland.asp, 2002.

[MS03a] *Enterprise Solution Patterns: Model-View-Controller*. Microsoft Patterns & Practices, http://msdn.microsoft.com/practices/type/Patterns/Enterprise/DesMVC/, 2003.

[MS03b] *Enterprise Solution Patterns: Implementing Model-View-Controller in ASP.NET*. Microsoft Patterns & Practices, http://msdn.microsoft.com/practices/type/Patterns/Enterprise/ImpMVCinASP/, 2003.

[MS03c] *COM+ (Component Services), Version 1.5*. Platform SDK. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncomg/html/msdn_comppr.asp, 2003.

[MS04a] *Operations Architecture Guide*. Microsoft Patterns & Practices, http://www.microsoft.com/technet/treeview/default.asp?url=/technet/itsolutions/idc/oag/oagc18.asp, 2004.

[MS04b] *Intercepting filter. Version 1.0.1.* Microsoft Patterns & Practices, http://msdn.microsoft.com/architecture/patterns/default.aspx?pull=/library/en-us/dnpatterns/html/DesInterceptingFilter.asp, 2004.

[MS04c] *Layered application. Version 1.0.1*. Microsoft Patterns & Practices, http://msdn.microsoft.com/architecture/patterns/default.aspx?pull=/library/en-us/dnpatterns/html/ArcLayeredApplication.asp, 2004.

[MS04d] *Three-layered services application*. Microsoft Patterns & Practices, http://msdn.microsoft.com/architecture/patterns/default.aspx?pull=/library/en-us/dnpatterns/html/ArcThreeLayeredSvcsApp.asp, 2004.

[MT97] Nenad Medvidovic y Richard Taylor. “Exploiting architectural style to develop a family of applications”. *IEEE Proceedings of Software Engineering*, 144(5-6), pp. 237-248, 1997.

[Nii86] H. Penny Nii. “Blackboard Systems, parts 1 & 2”. *AI Magazine* (7)3:38-53 & 7(4):62-69, 1986.

[Now99] Palle Nowak. *Structures and interactions*. Tesis de doctorado, Universidad del Sur de Dinamarca, 1999.

[Oel02] William Oellermann, Jr. *Architecting Web Services*. Apress, 2001.

[PA91] Rubén Prieto-Díaz y Guillermo Arango. *Domain Analysis and Software Systems Modeling*. IEEE Computer Society Press, Los Alamitos, 1991.

[Platt02] Michael Platt. "Microsoft Architecture Overview: Executive summary", http://msdn.microsoft.com/architecture/default.aspx?pull=/library/en-us/dnea/html/eaarchover.asp, 2002.

[Pre02] Roger Pressman. *Ingeniería del Software: Un enfoque práctico*. Madrid, McGraw Hill, 2001.

[PW92] Dewayne E. Perry y Alexander L. Wolf. "Foundations for the study of software architecture". *ACM SIGSOFT Software Engineering Notes*, 17(4), pp. 40–52, Octubre de 1992.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy y William Lorensen. *Object-oriented modeling and design*. Englewood Cliffs, Prentice Hall, 1991.

[Rec04] Brent Rector. *Introducing Longhorn for developers*. Microsoft Press, 2004.

[Red99] Frank Redmond III. *Introduction to designing and building Windows DNA applications*. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndna/html/windnadesign_intro.asp, Junio de 1999.

[Rey04a] Carlos Reynoso. *Teorías y métodos de la complejidad y el caos*. En proceso de edición en Gedisa, Barcelona.

[Rey04b] Carlos Reynoso. *Lenguajes de descripción de arquitecturas (ADLs)*. 2004.

[Ris00] Linda Rising. *Pattern Almanac 2000*. Reading, Addison-Wesley.

[Sar00] Toby Sarver. "Pattern refactoring workshop". *Position paper*, OOSPLA 2000, http://www.laputan.org/patterns/positions/Sarver.html, 2000.

[SC96] Mary Shaw y Paul Clements. "A field guide to Boxology: Preliminary classification of architectural styles for software systems". Documento de Computer Science Department and Software Engineering Institute, Carnegie Mellon University. Abril de 1996. Publicado en *Proceedings of the 21st International Computer Software and Applications Conference*, 1997.

[Sch+02] Ron Schmelzer, Travis Vandersypen, Jason Bloomberg, Maddhu Siddalingaiah, Sam Hunting y Michael Qualls. *XML and Web Services unleashed*. SAMS, 2002.

[SCK+96] Mark Simos, Dick Creps, Carol Klinger, L. Levine y Dean Allemang. "Organization Domain Modeling (ODM) Guidebook, Version 2.0". *Informal Technical Report for STARS*, STARS-VC-A025/001/00, 14 de junio de 1996, http://www.synquiry.com.

[SG96] Mary Shaw y David Garlan. *Software Architecture: Perspectives on an emerging discipline*. Upper Saddle River, Prentice Hall, 1996.

[SG97] P. R. Stiger y R. F. Gamble. "Blackboard systems formalized within a software architectural style". *International Conference on Systems, Man, Cybernetics*, Octubre de 1997.

[Shaw01] Mary Shaw. "The coming-of-age of software architecture research". *International Conference of Software Engineering*, 2001, pp. 656-664.

[Shaw95a] Mary Shaw. “Comparing architectural design styles”. IEEE Software 0740-7459, 1995.

[Shaw95b] Mary Shaw. “Architectural Issues in Software Reuse: It’s Not Just the Functionality, It’s the Packaging. *Proceedings of IEEE Symposium on Software Reusability*, pp. 3-6, Seattle, Abril de 1995.

[Shaw96] Mary Shaw. “Some Patterns for Software Architecture,” en *Pattern Languages of Program Design, Vol. 2*, J. Vlissides, J. Coplien, y N. Kerth (eds.), Reading, Addison-Wesley, pp. 255-269, 1996.

[Sho03] Keith Short. “Modeling languages for distributed application”. Microsoft, Octubre de 2003. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnvsent/html/vsent_ModelingLangs.asp.

[Sit97] Ramesh Sitaraman. *Integration of software systems at an abstract architectural level*. Tesis doctoral, Dept. MCS, Universidad de Tulsa, 1997.

[Stö01] Harald Störrle. “Towards architectural modeling with UML subsystems”. Ludwig-Maximilians-Universität, Munich, http://www.pst.informatik.uni-muenchen.de/lehre/WS0102/architektur/UML01AM.pdf, 2001.

[Szy95] Clemens Alden Szyperski. “Component Oriented Programming: A refined variation of Object-Oriented Programming”. *The Oberon Tribune*, 1(2), Diciembre de 1995.

[Szy02] Clemens Alden Szyperski. *Component software: Beyond Object-Oriented programming*. Reading, Addison-Wesley, 2002.

[TC92] Will Tracz y Lou Coglianese. “DSSA Engineering Process Guidelines”. *Technical Report*, ADAGE-IBM-9202, IBM Federal Systems Company, Diciembre de 1992.

[Tek00] Bedir Tekinerdoğan. *Synthesis-based architecture design*. Disertación doctoral, Departamento de Ciencias de la Computación, Universidad de Twente, Enschede, Holanda, 2000.

[TMA+95] Richard Taylor, Nenad Medvidovic, Kenneth Anderson, James Whitehead, Jason Robbins, Kari Nies, Peyman Oreizy y Deborah Dubrow. “A Component- and Message-Based Architectural Style for GUI Software”. *Proceedings of the 17th International Conference on Software Engineering*, 23 al 30 de Abril de 1995, Seattle, ACM Press, pp. 295-304, 1995.

[WF04] Guijun Wang y Casey Fung. “Architecture paradigms and their influences and impacts on componente-based software systems”. *Proceedings of the 37th Hawaii Intrnational Conference on System Sciences*, 2004.

[Wil99] Edward Wiles. “Economic models of software reuse: A survey, comparison and partial validation”. *Technical Report UWA-DCS-99-032,* Department of Computer Sciences, University of Wales, Aberystwyth, 1999.

[WP92] Steven Wartik y Rubén Prieto-Díaz. “Criteria for Comparing Domain Analysis Approaches”. *International Journal of Software Engineering and Knowledge Engineering*, 2(3), pp. 403-431, Setiembre de 1992.

[Zac87] John A. Zachman. “A Framework for information systems architecture”. *IBM Systems Journal*, 26(3), pp. 276-292, 1987.