Introducción a la Arquitectura de Software

Contenidos:

Introducción a la Arquitectura de Software

Versión 1.0 – Marzo de 2004

Carlos Billy Reynoso
UNIVERSIDAD DE BUENOS AIRES

Introducción

Este documento constituye una introducción sumaria a la Arquitectura de Software, con el propósito puntual de brindar una visión de conjunto lo más estructurada posible para luego establecer el papel de esta disciplina emergente en relación con la estrategia arquitectónica de Microsoft, sus herramientas y sus patrones de diseño. Hay múltiples razones para desarrollar esta presentación. Por empezar, no hay todavía textos en lengua castellana que brinden aproximaciones actualizadas a la Arquitectura de Software (en adelante, AS). El proceso editorial es hoy mucho más lento que el flujo de los acontecimientos y el cambio tecnológico; casi toda la producción en papel manifiesta atraso respecto de los elementos de juicio que urge considerar, tanto en el plano conceptual como en el tecnológico. Pero aún operando en binario y en banda ancha sobre la red de redes, el flujo de información de la industria rara vez se cruza con los intercambios del circuito académico, lo que ocasiona que la empresa y la academia terminen definiendo prioridades distintas, diagnosticando la situación de maneras discrepantes, otorgando diferentes valores a los criterios y usando las mismas nomenclaturas sin compartir sus significados. Como lo ha dicho Jan Bosch, un arquitecto práctico: “Existe una considerable diferencia entre la percepción académica de la AS y la práctica industrial. … Es interesante advertir que a veces los problemas que la industria identifica como los más importantes y difíciles, no se identifican o se consideran no-problemas en la academia” [Bos00].

Por otra parte, en la estrategia de Microsoft se ha desarrollado un marco de referencia global y genérico para el desarrollo de soluciones, Microsoft Solutions Framework, hoy en día en su tercera encarnación mayor. En MSF apenas hay mención de la AS, y en la perspectiva de otros documentos que podrían tenerla más en foco (como [Platt02]) no se la trata en términos semejantes a los que son comunes en la academia, que es, después de todo, donde se originan las ideas que la constituyen. Vinculado de alguna manera (implícita) con los lineamientos de MSF y bajo el paraguas (explícito) de “arquitectura”, se encuentra además un buen número de aportes en materia de patrones de diseño y lineamientos para implementarlos en el Framework .NET, primordialmente en modelos orientados a servicios. En ese contexto, delimitado por un marco necesariamente general (más afín a IT Management que a arquitectura o ingeniería) y por una práctica sumamente concreta, las cuestiones teóricas y las metodologías específicas basadas en arquitectura han quedado sin elaborar.

Existe entonces espacio y oportunidad para comenzar a articular esas referencias pendientes, de una manera que contribuya a situar esa estrategia particular (MSF+Patrones) en el marco de las tendencias actuales de teoría y práctica arquitectónica. Sin que estos documentos expresen una visión oficial, nos parece útil llenar el vacío, tender un puente, entre la investigación básica y los aportes académicos

por un lado y las visiones y requerimientos de industria por el otro. Lo que aquí ha de hacerse es otorgar contenidos, aunque sean provisionales y contestables, al concepto de "Arquitectura de Software", dado que cada vez que aparece en la documentación de industria se da por sentado lo que esa expresión significa.

El presente estudio constituye entonces el capítulo introductorio a una visión de conjunto de la AS, articulada conforme a este temario:

1. Arquitectura de Software
    - 1.1. Antecedentes históricos
    - 1.2. Definiciones y delimitación de la disciplina
    - 1.3. Conceptos fundamentales
    - 1.4. Campos de la Arquitectura de Software
    - 1.5. Modalidades y tendencias
    - 1.6. Diferencias entre Arquitectura y Diseño
    - 1.7. Repositorios
    - 1.8. Problemas pendientes en Arquitectura de Software
    - 1.9. Relevancia de la Arquitectura de Software
    - 1.10. (Referencias bibliográficas)
2. Estilos de Arquitectura
    - 2.1. Definiciones de estilo
    - 2.2. Clasificaciones de estilos arquitectónicos
    - 2.3. Inventario y Descripción de estilos arquitectónicos
    - 2.4. Estilos y patrones de arquitectura y diseño
    - 2.5. El lugar de los estilos en los marcos de referencia y en las vistas arquitectónicas
    - 2.6. Los estilos como valor contable
    - 2.7. (Referencias bibliográficas)
3. Lenguajes de descripción arquitectónica (ADLs)
    - 3.1. Introducción a los ADLs
    - 3.2. Criterios de definición de un ADL
    - 3.3. Lenguajes: Acme / Armani ADLs: Acme /Armani – ADML – Aesop – ArTek – C2 SADL – CHAM – Darwin – Jacal – LILEANNA – MetaH / AADL – Rapide – UML – UniCon – Wright – xArch / xADL
    - 3.4. Modelos computacionales y paradigmas de modelado
    - 3.5. ADLs y Patrones
    - 3.6. (Referencias bibliográficas)
4. Modelos de proceso y diseño
    - 4.1. Métodos tradicionales y de peso completo – CMM, UPM
    - 4.2. Métodos basados en arquitectura
        - 4.2.1. Métodos de análisis y diseño en el ciclo de vida – Visión general
        - 4.2.2. Arquitectura basada en escenarios (FAAM, ALMA)
        - 4.2.3. El diseño arquitectónico en el ciclo de vida: ABD
        - 4.2.4. Active Review for Intermediate Design (ARID)
        - 4.2.5. Quality Attribute Workshops (QAW) - QASAR
        - 4.2.6. Attribute-Driven Design (ADD)
        - 4.2.7. Evaluación: Architecture Tradeoff Analysis Method (ATAM)
        - 4.2.8. Métodos de evaluación de opciones arquitectónicas (SACAM)
        - 4.2.9. Derivación de tácticas arquitectónicas
        - 4.2.10. Economía de la arquitectura: Cost-Benefits Analysis Method (CBAM)
        - 4.2.11. Documentación de la Arquitectura
    - 4.3. Métodos heterodoxos y de peso ligero: Métodos Agiles, Programación Extrema – Concepción caórdica de los sistemas complejos
    - 4.4. Relación de los métodos LW con MSF 3 y con las metodologías basadas en arquitectura.
5. Herramientas arquitectónicas
    - 5.1. El lugar de UML 1.x y 2.0 en arquitectura de software – Alcances y limitaciones
    - 5.2. Herramientas de diseño y análisis asociadas a ADLs

5.3. Herramientas de Análisis, Evaluación y Visualización (SAAMTool, AET y análogas)
5.4. Herramientas de recuperación y visualización de arquitectura
5.5. Herramientas auxiliares de integración (MBI)
5.6. Servicios, plantillas y herramientas arquitectónicas en VS .NET Architect
6. Prácticas de diseño sobre arquitecturas orientadas a servicios y modelos de integración

El presente documento cubre los puntos 1.1 a 1.9 del plan de tratamiento del tema. Ya se encuentran disponibles documentos que cubren los temas 2 (estilos de arquitectura) y 3 (lenguajes de descripción arquitectónica). Los demás elementos del itinerario están en proceso de elaboración y se irán publicando a medida que estén listos.

El propósito de la totalidad del estudio consiste en establecer las bases para una discusión teórica y los fundamentos para una puesta en práctica de un modelo de diseño y desarrollo basado en arquitectura, ya que a pesar de la buena imagen de la disciplina, sus aportes sustantivos permanecen desconocidos para una mayoría de los ingenieros y metodólogos de software. Dado que estos documentos se presentan como punto de partida para la discusión de estos temas para la comunidad de arquitectos, después de la presentación de cada tópico se formula invitación para desarrollar los mismos contenidos desde otras perspectivas, aportar elementos de juicio adicionales, suministrar referencias de casos, corregir lagunas, agregar vínculos, difundir investigaciones relacionadas, compensar los sesgos, refinar el debate o enriquecer los materiales que aquí comienzan.

Breve Historia de la Arquitectura de Software

Todavía no se ha escrito una historia aceptable de la AS. Desde que Mary Shaw o David Garlan reseñaran escuetamente la prehistoria de la especialidad a principios de los 90, los mismos párrafos han sido reutilizados una y otra vez en la literatura, sin mayor exploración de las fuentes referidas en la reseña primaria y con prisa por ir al grano, que usualmente no es de carácter histórico. En este estudio se ha optado, más bien, por inspeccionar las fuentes más de cerca, con el objeto de señalar las supervivencias y las re-semantizaciones que han experimentado las ideas fundadoras en la AS contemporánea, definir con mayor claridad el contexto, entender que muchas contribuciones que pasaron por complementarias han sido en realidad antagónicas y comprender mejor por qué algunas ideas que surgieron hace cuatro décadas demoraron un cuarto de siglo en materializarse.

Esta decisión involucra algo más que el perfeccionamiento de la lectura que pueda hacerse de un conjunto de acontecimientos curiosos. Las formas divergentes en que se han interpretado dichas ideas, después de todo, permiten distinguir corrientes de pensamiento diversas, cuyas diferencias distan de ser triviales a la hora de plasmar las ideas en una metodología. Todo lo que parece ser un simple elemento de juicio, no pocas veces implica una disyuntiva. Situar las inflexiones de la breve historia de la AS en un contexto temporal, asimismo, ayudará a comprender mejor cuáles son sus contribuciones perdurables y cuáles sus manifestaciones contingentes al espíritu de los tiempos y a las modas tecnológicas que se han ido sucediendo.

Si bien la AS acostumbra remontar sus antecedentes al menos hasta la década de 1960, su historia no ha sido tan continua como la del campo más amplio en el que se inscribe, la ingeniería de software [Pfl02] [Pre01]. Después de las tempranas inspiraciones del legendario Edsger Dijkstra, de David Parnas y de Fred Brooks, la AS quedó en estado de

vida latente durante unos cuantos años, hasta comenzar su expansión explosiva con los manifiestos de Dewayne Perry de AT&T Bell Laboratories de New Jersey y Alexander Wolf de la Universidad de Colorado [PW92]. Puede decirse que Perry y Wolf fundaron la disciplina, y su llamamiento fue respondido en primera instancia por los miembros de lo que podría llamarse la escuela estructuralista de Carnegie Mellon: David Garlan, Mary Shaw, Paul Clements, Robert Allen.

Se trata entonces de una práctica joven, de apenas unos doce años de trabajo constante, que en estos momentos experimenta una nueva ola creativa en el desarrollo cabal de sus técnicas en la obra de Rick Kazman, Mark Klein, Len Bass y otros metodólogos en el contexto del SEI, en la misma universidad. A comienzos del siglo XXI comienzan ya a discernirse tendencias, cuyas desavenencias mutuas todavía son leves: al menos una en el sur de California (Irvine y Los Angeles) con Nenad Medvidovic, David Rosenblum y Richard Taylor, otra en el SRI de Menlo Park con Mark Moriconi y sus colegas y otra más vinculada a las recomendaciones formales de la IEEE y los trabajos de Rich Hilliard. Hoy se percibe también un conjunto de posturas europeas que enfatizan mayormente cuestiones metodológicas vinculadas con escenarios y procuran inscribir la arquitectura de software en el ciclo de vida, comenzando por la elicitación de los requerimientos. Antes de Perry y Wolf, empero, se formularon ideas que serían fundamentales para la disciplina ulterior. Comencemos entonces por el principio, aunque siempre cabrá la posibilidad de discutir cuál puede haber sido el momento preciso en el que todo comenzó.

Cada vez que se narra la historia de la arquitectura de software (o de la ingeniería de software, según el caso), se reconoce que en un principio, hacia 1968, Edsger Dijkstra, de la Universidad Tecnológica de Eindhoven en Holanda y Premio Turing 1972, propuso que se establezca una estructuración correcta de los sistemas de software antes de lanzarse a programar, escribiendo código de cualquier manera [Dij68a]. Dijkstra, quien sostenía que las ciencias de la computación eran una rama aplicada de las matemáticas y sugería seguir pasos formales para descomponer problemas mayores, fue uno de los introductores de la noción de sistemas operativos organizados en capas que se comunican sólo con las capas adyacentes y que se superponen "como capas de cebolla". Inventó o ayudó a precisar además docenas de conceptos: el algoritmo del camino más corto, los *stacks*, los vectores, los semáforos, los abrazos mortales. De sus ensayos arranca la tradición de hacer referencia a "niveles de abstracción" que ha sido tan común en la arquitectura subsiguiente. Aunque Dijkstra no utiliza el término arquitectura para describir el diseño conceptual del software, sus conceptos sientan las bases para lo que luego expresarían Niklaus Wirth [Wir71] como *stepwise refinement* y DeRemer y Kron [DK76] como *programming-in-the large* (o programación en grande), ideas que poco a poco irían decantando entre los ingenieros primero y los arquitectos después.

En la conferencia de la NATO de 1969, un año después de la sesión en que se fundara la ingeniería de software, P. I. Sharp formuló estas sorprendentes apreciaciones comentando las ideas de Dijkstra:

> Pienso que tenemos algo, aparte de la ingeniería de software: algo de lo que hemos hablado muy poco pero que deberíamos poner sobre el tapete y concentrar la atención en ello. Es la cuestión de la arquitectura de software. La arquitectura es diferente de la ingeniería. Como ejemplo de lo que quiero decir, echemos una mirada a OS/360. Partes de OS/360 están extremadamente bien codificadas.

> Partes de OS, si vamos al detalle, han utilizado técnicas que hemos acordado constituyen buena práctica de programación. La razón de que OS sea un amontonamiento amorfo de programas es que no tuvo arquitecto. Su diseño fue delegado a series de grupos de ingenieros, cada uno de los cuales inventó su propia arquitectura. Y cuando esos pedazos se clavaron todos juntos no produjeron una tersa y bella pieza de software [NATO76: 150].

Sharp continúa su alegación afirmando que con el tiempo probablemente llegue a hablarse de "la escuela de arquitectura de software de Dijkstra" y se lamenta que en la industria de su tiempo se preste tan poca o ninguna atención a la arquitectura. La frase siguiente también es extremadamente visionaria:

> Lo que sucede es que las especificaciones de software se consideran especificaciones funcionales. Sólo hablamos sobre lo que queremos que haga el programa. Es mi creencia que cualquiera que sea responsable de la implementación de una pieza de software debe especificar más que esto. Debe especificar el diseño, la forma; y dentro de ese marco de referencia, los programadores e ingenieros deben crear algo. Ningún ingeniero o programador, ninguna herramienta de programación, nos ayudará, o ayudará al negocio del software, a maquillar un diseño feo. El control, la administración, la educación y todas las cosas buenas de las que hablamos son importantes; pero la gente que implementa debe entender lo que el arquitecto tiene en mente [Idem].

Nadie volvió a hablar del asunto en esa conferencia, sin embargo. Por unos años, "arquitectura" fue una metáfora de la que se echó mano cada tanto, pero sin precisión semántica ni consistencia pragmática. En 1969 Fred Brooks Jr y Ken Iverson llamaban arquitectura a la estructura conceptual de un sistema en la perspectiva del programador. En 1971, C. R. Spooner tituló uno de sus ensayos "Una arquitectura de software para los 70s" [Spo71], sin que la mayor parte de la historiografía de la AS registrara ese antecedente.

En 1975, Brooks, diseñador del sistema operativo OS/360 y Premio Turing 2000, utilizaba el concepto de arquitectura del sistema para designar "la especificación completa y detallada de la interfaz de usuario" y consideraba que el arquitecto es un agente del usuario, igual que lo es quien diseña su casa [Bro75], empleando una nomenclatura que ya nadie aplica de ese modo. En el mismo texto, identificaba y razonaba sobre las estructuras de alto nivel y reconocía la importancia de las decisiones tomadas a ese nivel de diseño. También distinguía entre arquitectura e implementación; mientras aquella decía *qué* hacer, la implementación se ocupa de *cómo*. Aunque el concepto de AS actual y el de Brooks difieren en no escasa medida, el texto de Brooks *The mythical man-month* sigue siendo, un cuarto de siglo más tarde, el más leído en ingeniería de software. Se ha señalado que Dijkstra y Brooks, el primero partidario de un formalismo matemático y el segundo de considerar las variables humanas, constituyen dos personalidades opuestas, que se sitúan en los orígenes de las metodologías fuertes y de las heterodoxias ágiles, respectivamente [Tra02]; pero eso será otra historia.

Una novedad importante en la década de 1970 fue el advenimiento del diseño estructurado y de los primeros modelos explícitos de desarrollo de software. Estos modelos comenzaron a basarse en una estrategia más orgánica, evolutiva, cíclica, dejando atrás las metáforas del desarrollo en cascada que se inspiraban más bien en la línea de

montaje de la ingeniería del hardware y la manufactura. Surgieron entonces las primeras investigaciones académicas en materia de diseño de sistemas complejos o "intensivos". Poco a poco el diseño se fue independizando de la implementación, y se forjaron herramientas, técnicas y lenguajes de modelado específicos.

En la misma época, otro precursor importante, David Parnas, demostró que los criterios seleccionados en la descomposición de un sistema impactan en la estructura de los programas y propuso diversos principios de diseño que debían seguirse a fin de obtener una estructura adecuada. Parnas desarrolló temas tales como módulos con ocultamiento de información [Par72], estructuras de software [Par74] y familias de programas [Par76], enfatizando siempre la búsqueda de calidad del software, medible en términos de economías en los procesos de desarrollo y mantenimiento. Aunque Dijkstra, con sus frases lapidarias y memorables, se ha convertido en la figura legendaria por excelencia de los mitos de origen de la AS, Parnas ha sido sin duda el introductor de algunas de sus nociones más esenciales y permanentes.

En 1972, Parnas publicó un ensayo en el que discutía la forma en que la modularidad en el diseño de sistemas podía mejorar la flexibilidad y el control conceptual del sistema, acortando los tiempos de desarrollo [Par72]. Introdujo entonces el concepto de ocultamiento de información (*information hiding*), uno de los principios de diseño fundamentales en diseño de software aún en la actualidad. La herencia de este concepto en la ingeniería y la arquitectura ulterior es inmensa, y se confunde estrechamente con la idea de abstracción. En la segunda de las descomposiciones que propone Parnas comienza a utilizarse el ocultamiento de información como criterio. "Los módulos ya no se corresponden con las etapas de procesamiento. … Cada módulo en la segunda descomposición se caracteriza por su conocimiento de una decisión de diseño oculta para todos los otros módulos. Su interfaz o definición se escoge para que revele tan poco como sea posible sobre su forma interna de trabajo" [Par72]. Cada módulo deviene entonces una caja negra para los demás módulos del sistema, los cuales podrán acceder a aquél a través de interfaces bien definidas, en gran medida invariables. Es fácil reconocer en este principio ideas ya presentadas por Dijkstra en su implementación del THE-Multiprogramming System [Dij68a]. Pero la significación del artículo de 1972 radica en la idea de Parnas de basar la técnica de modularización en decisiones de diseño, mientras que los "niveles de abstracción" de Dijkstra involucraban más bien (igual que su famosa invectiva en contra del Go-to) técnicas de programación.

El concepto de ocultamiento se fue mezclando con encapsulamiento y abstracción, tras algunos avatares de avance y retroceso. Los arquitectos más escrupulosos distinguen entre encapsulamiento y ocultamiento, considerando a aquél como una capacidad de los lenguajes de programación y a éste como un principio más general de diseño. De hecho, Parnas no hablaba en términos de programación orientada a objeto, sino de módulos y sub-rutinas, porque el momento de los objetos no había llegado todavía.

El pensamiento de Parnas sobre familias de programas, en particular, anticipa ideas que luego habrían de desarrollarse a propósito de los estilos de arquitectura:

> Una familia de programas es un conjunto de programas (no todos los cuales han sido construidos o lo serán alguna vez) a los cuales es provechoso o útil considerar como grupo. Esto evita el uso de conceptos ambiguos tales como "similitud funcional" que surgen a veces cuando se describen dominios. Por

> ejemplo, los ambientes de ingeniería de software y los juegos de video no se consideran usualmente en el mismo dominio, aunque podrían considerarse miembros de la misma familia de programas en una discusión sobre herramientas que ayuden a construir interfaces gráficas, que casualmente ambos utilizan.

Una familia de programas puede enumerarse en principio mediante la especificación del árbol de decisión que se atraviesa para llegar a cada miembro de la familia. Las hojas del árbol representan sistemas ejecutables. El concepto soporta la noción de derivar un miembro de la familia a partir de uno ya existente. El procedimiento consiste en seguir hacia atrás el árbol hasta que se alcanza un nodo (punto de decisión) genealógicamente común a ambos, y luego proceder hacia abajo hasta llegar al miembro deseado. El concepto también soporta la noción de derivar varios miembros de la familia de un punto de decisión común, aclarando la semejanza y las diferencias entre ellos.

Las decisiones iniciales son las que más probablemente permanecerán constantes entre los miembros; las decisiones más tardías (cerca de las hojas) en un árbol prudentemente estructurado deberían representar decisiones susceptibles de cambiarse trivialmente, tales como los valores de tiempo de compilación o las constantes de tiempo de carga. La significación del concepto de familia de programas para la AS es que ella corresponde a las decisiones cerca del tope del árbol de decisión de Parnas. Es importante considerar que el árbol de Parnas es *top-down* no sólo porque se construye y recorre de lo general a lo particular, sino porque sus raíces se encuentran hacia arriba (o a la izquierda si el modelo es horizontal). No menos esencial es tener en cuenta que el árbol se ofreció como alternativa a métodos de descomposición basados en diagramas de flujo, por los que la AS no ha mostrado nunca demasiada propensión.

Decía Parnas que las decisiones tempranas de desarrollo serían las que probablemente permanecerían invariantes en el desarrollo ulterior de una solución. Esas “decisiones tempranas” constituyen de hecho lo que hoy se llamarían decisiones arquitectónicas. Como escriben Clements y Northrop [CN96] en todo el desenvolvimiento ulterior de la disciplina permanecería en primer plano esta misma idea: la estructura es primordial (*structure matters*), y la elección de la estructura correcta ha de ser crítica para el éxito del desarrollo de una solución. Ni duda cabe que “la elección de la estructura correcta” sintetiza, como ninguna otra expresión, el programa y la razón de ser de la AS.

En la década de 1980, los métodos de desarrollo estructurado demostraron no escalar suficientemente y fueron dejando el lugar a un nuevo paradigma, el de la programación orientada a objetos. En teoría, parecía posible modelar el dominio del problema y el de la solución en un lenguaje de implementación. La investigación que llevó a lo que después sería el diseño orientado a objetos puede remontarse incluso a la década de 1960 con Simula, un lenguaje de programación de simulaciones, el primero que proporcionaba tipos de datos abstractos y clases, y después fue refinada con el advenimiento de Smalltalk. Paralelamente, hacia fines de la década de 1980 y comienzos de la siguiente, la expresión *arquitectura de software* comienza a aparecer en la literatura para hacer referencia a la configuración morfológica de una aplicación.

Mientras se considera que la ingeniería de software se fundó en 1968, cuando F.L. Bauer usó ese sintagma por primera vez en la conferencia de la OTAN de Garmisch, Alemania, la AS, como disciplina bien delimitada, es mucho más nueva de lo que generalmente se sospecha. El primer texto que vuelve a reivindicar las abstracciones de alto nivel,

reclamando un espacio para esa reflexión y augurando que el uso de esas abstracciones en el proceso de desarrollo pueden resultar en "un nivel de arquitectura de software en el diseño" es uno de Mary Shaw [Shaw84] seguido por otro lllamado "Larger scale systems require higher level abstractions" [Shaw89]. Se hablaba entonces de un nivel de abstracción en el conjunto; todavía no estaban en su lugar los elementos de juicio que permitieran reclamar la necesidad de una disciplina y una profesión particulares.

El primer estudio en que aparece la expresión "arquitectura de software" en el sentido en que hoy lo conocemos es sin duda el de Perry y Wolf [PW92]; ocurrió tan tarde como en 1992, aunque el trabajo se fue gestando desde 1989. En él, los autores proponen concebir la AS por analogía con la arquitectura de edificios, una analogía de la que luego algunos abusaron [WWI99], otros encontraron útil y para unos pocos ha devenido inaceptable [BR01]. El artículo comienza diciendo exactamente:

> El propósito de este *paper* es construir el fundamento para la arquitectura de software. Primero desarrollaremos una intuición para la arquitectura de software recurriendo a diversas disciplinas arquitectónicas bien definidas. Sobre la base de esa intuición, presentamos un modelo para la arquitectura de software que consiste en tres componentes: elementos, forma y razón (*rationale*). Los elementos son elementos ya sea de procesamiento, datos o conexión. La forma se define en términos de las propiedades de, y las relaciones entre, los elementos, es decir, restricciones operadas sobre ellos. La razón proporciona una base subyacente para la arquitectura en términos de las restricciones del sistema, que lo más frecuente es que se deriven de los requerimientos del sistema. Discutimos los componentes del modelo en el contexto tanto de la arquitectura como de los estilos arquitectónicos ….

La declaración, como puede verse, no tiene una palabra de desperdicio: cada idea cuenta, cada intuición ha permanecido viva desde entonces. Los autores prosiguen reseñando el progreso de las técnicas de diseño en la década de 1970, en la que los investigadores pusieron en claro que el diseño es una actividad separada de la implementación y que requiere notaciones, técnicas y herramientas especiales. Los resultados de esta investigación comienzan ahora (decían en 1992) a penetrar el mercado en la forma de herramientas de ingeniería asistida por computadoras, CASE. Pero uno de los resultados del uso de estas herramientas ha sido que se produjo la absorción de las herramientas de diseño por los lenguajes de implementación. Esta integración ha tendido a esfumar, si es que no a confundir, la diferencia entre diseño e implementación. En la década de 1980 se perfeccionaron las técnicas descriptivas y las notaciones formales, permitiéndonos razonar mejor sobre los sistemas de software. Para la caracterización de lo que sucederá en la década siguiente ellos formulan esta otra frase que ha quedado inscripta en la historia mayor de la especialidad:

> 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 los arquitectos de software) y de estilo. … Es tiempo de re-examinar el papel de la arquitectura de software en el contexto más amplio del proceso de software y de su administración, así como señalar las nuevas técnicas que han sido adoptadas.

Considerada como disciplina por mérito propio, la AS ha de ser, para ellos, beneficiosa como marco de referencia para satisfacer requerimientos, una base esencial para la estimación de costos y administración del proceso y para el análisis de las dependencias y la consistencia del sistema.

Dando cumplimiento a las profecías de Perry y Wolf, la década de 1990 fue sin duda la de la consolidación y diseminación de la AS en una escala sin precedentes. Las contribuciones más importantes surgieron en torno del instituto de ingeniería de la información de la Universidad Carnegie Mellon (CMU SEI), antes que de cualesquiera organismos de industria. En la misma década, demasiado pródiga en acontecimientos, surge también la programación basada en componentes, que en su momento de mayor impacto impulsó a algunos arquitectos mayores, como Paul Clements [Cle96b], a afirmar que la AS promovía un modelo que debía ser más de integración de componentes pre-programados que de programación.

Un segundo gran tema de la época fue el surgimiento de los patrones, cristalizada en dos textos fundamentales, el de la Banda de los Cuatro en 1995 [Gof95] y la serie *POSA* desde 1996 [BMR+96]. El primero de ellos promueve una expansión de la programación orientada a objetos, mientras que el segundo desenvuelve un marco ligeramente más ligado a la AS. Este movimiento no ha hecho más que expandirse desde entonces. El originador de la idea de patrones fue Christopher Alexander, quien incidentalmente fue arquitecto *de edificios*; Alexander desarrolló en diversos estudios de la década de 1970 temas de análisis del sentido de los planos, las formas, la edificación y la construcción, en procura de un modelo constructivo y humano de arquitectura, elaborada de forma que tenga en cuenta las necesidades de los habitantes [Ale77]. El arquitecto (y puede copiarse aquí lo que decía Fred Brooks) debe ser un agente del usuario.

A lo largo de una cadena de intermediarios y pensadores originales, las ideas llegaron por fin a la informática diez años más tarde. Si bien la idea de arquitectura implícita en el trabajo actual con patrones está más cerca de la implementación y el código, y aunque la reutilización de patrones guarda estrecha relación con la tradición del diseño concreto orientado a objetos [Lar03], algunos arquitectos emanados de la escuela de Carnegie Mellon formalizaron un acercamiento con esa estrategia [Shaw96] [MKM+96] [MKM+97]. Tanto en los patrones como en la arquitectura, la idea dominante es la de re-utilización. A impulsos de otra idea mayor de la época (la crisis del software), la bibliografía sobre el impacto económico de la re-utilización alcanza hoy magnitudes de cuatro dígitos.

Como quiera que sea, la AS de este período realizó su trabajo de homogeneización de la terminología, desarrolló la tipificación de los estilos arquitectónicos y elaboró lenguajes de descripción de arquitectura (ADLs), temas que en este estudio se tratan en documentos separados. También se consolidó la concepción de las vistas arquitectónicas, reconocidas en todos y cada uno de los *frameworks* generalizadores que se han propuesto (4+1, TOGAF, RM/ODP, IEEE), como luego se verá.

Uno de los acontecimientos arquitectónicos más importantes del año 2000 fue la hoy célebre tesis de Roy Fielding que presentó el modelo REST, el cual establece definitivamente el tema de las tecnologías de Internet y los modelos orientados a servicios y recursos en el centro de las preocupaciones de la disciplina [Fie00]. En el mismo año se publica la versión definitiva de la recomendación IEEE Std 1471, que

procura homogeneizar y ordenar la nomenclatura de descripción arquitectónica y homologa los estilos como un modelo fundamental de representación conceptual.

En el siglo XXI, la AS aparece dominada por estrategias orientadas a líneas de productos y por establecer modalidades de análisis, diseño, verificación, refinamiento, recuperación, diseño basado en escenarios, estudios de casos y hasta justificación económica, redefiniendo todas las metodologías ligadas al ciclo de vida en términos arquitectónicos. Todo lo que se ha hecho en ingeniería debe formularse de nuevo, integrando la AS en el conjunto. La producción de estas nuevas metodologías ha sido masiva, y una vez más tiene como epicentro el trabajo del Software Engineering Institute en Carnegie Mellon. La aparición de las metodologías basadas en arquitectura, junto con la popularización de los métodos ágiles en general y Extreme Programming en particular, han causado un reordenamiento del campo de los métodos, hasta entonces dominados por las estrategias de diseño "de peso pesado". Después de la AS y de las tácticas radicales, las metodologías nunca volverán a ser las mismas.

La semblanza que se ha trazado no es más que una visión selectiva de las etapas recorridas por la AS. Los lineamientos de ese proceso podrían dibujarse de maneras distintas, ya sea enfatizando los hallazgos formales, las intuiciones dominantes de cada período o las diferencias que median entre la abstracción cualitativa de la arquitectura y las cuantificaciones que han sido la norma en ingeniería de software.

Definiciones

No es novedad que ninguna definición de la AS es respaldada unánimemente por la totalidad de los arquitectos. El número de definiciones circulantes alcanza un orden de tres dígitos, amenazando llegar a cuatro. De hecho, existen grandes compilaciones de definiciones alternativas o contrapuestas, como la colección que se encuentra en el SEI (http://www.sei.cmu.edu/architecture/definitions.html), a la que cada quien puede agregar la suya. En general, las definiciones entremezclan despreocupadamente (1) el trabajo dinámico de estipulación de la arquitectura dentro del proceso de ingeniería o el diseño (su lugar en el ciclo de vida), (2) la configuración o topología estática de sistemas de software contemplada desde un elevado nivel de abstracción y (3) la caracterización de la disciplina que se ocupa de uno de esos dos asuntos, o de ambos.

Una definición reconocida es la de Clements [Cle96a]: La AS es, a grandes rasgos, una vista del sistema que incluye los componentes principales del mismo, la conducta de esos componentes según se la percibe desde el resto del sistema y las formas en que los componentes interactúan y se coordinan para alcanzar la misión del sistema. La vista arquitectónica es una vista abstracta, aportando el más alto nivel de comprensión y la supresión o diferimiento del detalle inherente a la mayor parte de las abstracciones.

En una definición semejante, hay que aclararlo, la idea de "componente" no es la de la correspondiente tecnología de desarrollo (COM, CORBA Component Model, EJB), sino la de elemento propio de un estilo. Un componente es una cosa, una entidad, a la que los arquitectos prefieren llamar "componente" antes que "objeto", por razones que se verán en otros documentos de esta serie pero que ya es obvio imaginar cuáles han de ser.

A despecho de la abundancia de definiciones del campo de la AS, existe en general acuerdo de que ella se refiere a la estructura a grandes rasgos del sistema, estructura

consistente en componentes y relaciones entre ellos [BCK98]. Estas cuestiones estructurales se vinculan con el diseño, pues la AS es después de todo una forma de diseño de software que se manifiesta tempranamente en el proceso de creación de un sistema; pero este diseño ocurre a un nivel más abstracto que el de los algoritmos y las estructuras de datos. En el que muchos consideran un ensayo seminal de la disciplina, Mary Shaw y David Garlan sugieren que dichas cuestiones estructurales incluyen organización a grandes rasgos y estructura global de control; protocolos para la comunicación, la sincronización y el acceso a datos; la asignación de funcionalidad a elementos del diseño; la distribución física; la composición de los elementos de diseño; escalabilidad y performance; y selección entre alternativas de diseño [GS94].

En una definición tal vez demasiado amplia, David Garlan [Gar00] establece que la AS constituye un puente entre el requerimiento y el código, ocupando el lugar que en los gráficos antiguos se reservaba para el diseño. La definición "oficial" de AS se ha acordado que sea la que brinda el documento de IEEE Std 1471-2000, adoptada también por Microsoft, que reza así:

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

Aunque las literaturas de ambos campos rara vez convergen, entendemos que es productivo contrastar esa definición con la de *ingeniería* de software, conforme al estándar IEEE 610.12.1990:

> La aplicación de una estrategia sistemática, disciplinada y cuantificable al desarrollo, aplicación y mantenimiento del software; esto es, la aplicación de la ingeniería al software.

Obsérvese entonces que la noción clave de la arquitectura es la organización (un concepto cualitativo o estructural), mientras que la ingeniería tiene fundamentalmente que ver con una sistematicidad susceptible de cuantificarse.

Antes el número y variedad de definiciones existentes de AS, Mary Shaw y David Garlan [SG95] proporcionaron una sistematización iluminadora, explicando las diferencias entre definiciones en función de distintas clases de modelos. Destilando las definiciones y los puntos de vista implícitos o explícitos, los autores clasifican los modelos de esta forma:

1) **Modelos estructurales**: Sostienen que la AS está compuesta por componentes, conexiones entre ellos y (usualmente) otros aspectos tales como configuración, estilo, restricciones, semántica, análisis, propiedades, racionalizaciones, requerimientos, necesidades de los participantes. El trabajo en esta área está caracterizada por el desarrollo de lenguajes de descripción arquitectónica (ADLs).
2) **Modelos de framework**: Son similares a la vista estructural, pero su énfasis primario radica en la (usualmente una sola) estructura coherente del sistema completo, en vez de concentrarse en su composición. Los modelos de framework a menudo se refieren a dominios o clases de problemas específicos. El trabajo que ejemplifica esta variante incluye arquitecturas de software específicas de dominios, como CORBA, o modelos basados en CORBA, o repositorios de componentes específicos, como PRISM.

3) **Modelos dinámicos**: Enfatizan la cualidad conductual de los sistemas. “Dinámico” puede referirse a los cambios en la configuración del sistema, o a la dinámica involucrada en el progreso de la computación, tales como valores cambiantes de datos.
4) **Modelos de proceso**: Se concentran en la construcción de la arquitectura, y en los pasos o procesos involucrados en esa construcción. En esta perspectiva, la arquitectura es el resultado de seguir un argumento (*script*) de proceso. Esta vista se ejemplifica con el actual trabajo sobre programación de procesos para derivar arquitecturas.
5) **Modelos funcionales**: Una minoría considera la arquitectura como un conjunto de componentes funcionales, organizados en capas que proporcionan servicios hacia arriba. Es tal vez útil pensar en esta visión como un framework particular.

Ninguna de estas vistas excluye a las otras, ni representa un conflicto fundamental sobre lo que es o debe ser la AS. Por el contrario, representan un espectro en la comunidad de investigación sobre distintos énfasis que pueden aplicarse a la arquitectura: sobre sus partes constituyentes, su totalidad, la forma en que se comporta una vez construida, o el proceso de su construcción. Tomadas en su conjunto, destacan más bien un consenso.

Independientemente de las discrepancias entre las diversas definiciones, es común entre todos los autores el concepto de la arquitectura como un punto de vista que concierne a un alto nivel de abstracción. Revisamos las diversas definiciones del concepto de abstracción en un apartado específico más adelante en este estudio.

Es casi seguro que la percepción de la AS que prevalece entre quienes no han tenido contacto con ella, así como los estereotipos dominantes sobre su naturaleza, o los nombres que se escogerían como sus personalidades más importantes, difieren sustancialmente de lo que es el caso en el interior de la especialidad. Este sería acaso un ejercicio digno de llevarse a cabo alguna vez.

Conceptos fundamentales

Más allá de que hoy existan numerosos conceptos en el plano detallado de las técnicas y metodologías, la AS se articula alrededor de unos pocos conceptos y principios esenciales y unas pocas herramientas características.

Estilos

En el texto fundacional de la AS, Perry y Wolf establecen el razonamiento sobre estilos de arquitectura como uno de los aspectos fundamentales de la disciplina. Un estilo es un concepto descriptivo que define una forma de articulación u organización arquitectónica. El conjunto de los estilos cataloga las formas básicas posibles de estructuras de software, mientras que las formas complejas se articulan mediante composición de los estilos fundamentales.

Sucintamente descriptos, los estilos conjugan elementos (o “componentes”, como se los llama aquí), conectores, configuraciones y restricciones. Al estipular los conectores como elemento de juicio de primera clase, el concepto de estilo, incidentalmente, se sitúa en un orden de discurso y de método que el modelado orientado a objetos en general y UML en

particular no cubren satisfactoriamente. La descripción de un estilo se puede formular en lenguaje natural o en diagramas, pero lo mejor es hacerlo en un lenguaje de descripción arquitectónica o en lenguajes formales de especificación.

A diferencia de los patrones de diseños, que son centenares, los estilos se ordenan en seis o siete clases fundamentales y unos veinte ejemplares, como máximo. Es digno de señalarse el empeño por subsumir todas las formas existentes de aplicaciones en un conjunto de dimensiones tan modestas. Las arquitecturas complejas o compuestas resultan del agregado o la composición de estilos más básicos. Algunos estilos típicos son las arquitecturas basadas en flujo de datos, las *peer-to-peer*, las de invocación implícita, las jerárquicas, las centradas en datos o las de intérprete-máquina virtual.

Hemos tratado el tema de las definiciones, los catálogos de estilos, las propiedades de cada uno, el lugar de los estilos en la AS y su relación con los patrones de diseño y con la estrategia arquitectónica de Microsoft en un documento separado, en torno del cual se podrán discutir las cuestiones relacionadas con ellos.

Lenguajes de descripción arquitectónica

Los lenguajes de descripción de arquitecturas, o ADLs, ocupan una parte importante del trabajo arquitectónico desde la fundación de la disciplina. Se trata de un conjunto de propuestas de variado nivel de rigurosidad, casi todas ellas de extracción académica, que fueron surgiendo desde comienzos de la década de 1990 hasta la actualidad, más o menos en contemporaneidad con el proyecto de unificación de los lenguajes de modelado bajo la forma de UML. Los ADL difiere sustancialmente de UML, que al menos en su versión 1.x se estima inadecuado en su capacidad para expresar conectores en particular y en su modelo semántico en general para las clases de descripción y análisis que se requieren. Los ADLs permiten modelar una arquitectura mucho antes que se lleve a cabo la programación de las aplicaciones que la componen, analizar su adecuación, determinar sus puntos críticos y eventualmente simular su comportamiento.

Los ADLs son bien conocidos en los estudios universitarios de AS, pero muy pocos arquitectos de industria parecen conocerlos y son menos aún quienes los utilizan como instrumento en el diseño arquitectónico de sus proyectos. Hay unos veinte ADLs de primera magnitud y tal vez unos cuarenta o cincuenta propuestos en ponencias que no han resistido el paso del tiempo o que no han encontrado su camino en el mercado. Se han analizado esos lenguajes descriptivos en un documento aparte, en el cual se invita asimismo a su discusión.

Frameworks y Vistas

En esta sección se examinarán primero las propuestas sobre organización de vistas desarrolladas en el contexto de los *frameworks* más influyentes. En segundo lugar, se analizará algo más formalmente la historia y el significado del concepto de vistas en sí, ya que en la mayor parte de la literatura no académica la significación precisa del concepto y su función técnica se suelen dar por sentadas o se definen a la ligera y de formas cambiantes.

Existen unos cuantos organismos de estándares (ISO, CEN, IEEE, OMG) que han codificado diferentes aspectos de la AS, cuando no la AS en su totalidad, con el objetivo

de homogeneizar la terminología, los modelos y los procedimientos. Los emergentes del trabajo de sus comités son especificaciones y recomendaciones de variada naturaleza, como RM-ODP, RUP, RDS, MDA, MOF, MEMO, XMI o IEEE 1471-2000. Casi cualquier combinación de tres o cuatro letras del alfabeto corresponde al acrónimo de algún estándar a tener en cuenta. La AS hace mención eventual de esas doctrinas y los académicos ocupan sillas preferenciales en los organismos, pero su tratamiento exhaustivo en la literatura de investigación decididamente no califica como uno de los grandes temas prioritarios. Las recomendaciones de los marcos se tratan con sumo respeto pero también con alguna reticencia, tal vez porque se estima que los organismos privilegiarían más un acuerdo regido por una necesidad de equidistancia que una adecuada fundamentación formal, porque cualquier versión del estándar que se mencione en un *paper* habrá caducado cuando se lo publique, o por alguna otra razón que dejamos abierta para que se discuta.

Hay una excepción, sin embargo. Tanto los marcos arquitectónicos como las metodologías de modelado de los organismos acostumbran ordenar las diferentes perspectivas de una arquitectura en términos de vistas (*views*). La mayoría de los frameworks y estrategias reconoce entre tres y seis vistas, que son las que se incluyen en el cuadro. Una vista es, para definirla sucintamente, un subconjunto resultante de practicar una selección o abstracción sobre una realidad, desde un punto de vista determinado [Hil99].

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	Implementación	Física	Física
Tecnología	Tecnología	Desarrollo	Despliegue	Desarrollo	
Representación		Casos de uso	Casos de uso		
Funcionamiento					

Tabla 1 - Vistas en los marcos de referencia

- El marco de referencia para la arquitectura empresarial de John Zachman [Zac87] 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 AS se ha estimado que este modelo es excesivamente rígido y sobre-articulado. Parecería existir cierto consenso sobre su excesiva ambición y su posible obsolescencia. Los manuales recientes de ingeniería de software (por ejemplo [Pre02] [Pfl02]) suelen omitir toda referencia a este marco, que se estima perteneciente a una esfera de *Information Management* y estrategia empresarial, antes que inscripto en el campo de la arquitectura. El framework ha estado también bajo nutrido fuego crítico [Cib98] [Keen91]. No obstante, hay que reconocer que tres de las vistas propuestas por Zachman tan tempranamente como en 1982 (conceptual, lógica y física) se corresponden con el modelo de vistas de los marcos de referencia posteriores.
- El Modelo de Referencia para Procesamiento Distribuido Abierto (RM-ODP) es un estándar de ISO y de ITU (ex-CCITT) que define 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 que componen el modelo, los dos primeros se refieren a la motivación general del mismo 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, pero recomienda el uso de lenguajes formales de especificación como LOTOS, ESTELLE, SDL o Z. Se supone que un modelo pensado con similares propósitos, como CORBA, debería ser 100% conforme con la referencia, pero en verdad no es así; CORBA, por ejemplo, no proporciona soporte para el *viewpoint* empresarial, su modelo computacional revela desviaciones significativas del ISO/IEC o el UTI-T correspondiente, su modelo de *binding* es más restringido (carece de *multicast* o *plug and play*) y aunque hay RFPs que documentan estas divergencias, hasta donde puede saberse permanecen aún sin resolver; en los *viewpoints* de ingeniería y tecnología las discrepancias son menores, pero son muchísimas, y en ninguno hay concordancia en la nomenclatura [DIR99]. Los modelos mayores de industria como COM o J2EE son todavía más divergentes y las verificaciones de conformidad son un trámite extremadamente complejo. RM-ODP, además, no especifica nada acerca de conectores entre sistemas, integración de tecnologías *legacy* o soporte de sistemas multi-paradigmas [Bez98]. Hoy en día la interoperabilidad se garantiza más a través de tecnologías comunes de formato y protocolo ligadas a XML, SOAP, BPEL4WS o WS-I (o mediante MDA, o programando un *wrapper*) que por conformidad con esta clase de recomendaciones en todos los puntos de un proceso distribuido.

- C4ISR Architecture Framework es el marco de referencia arquitectónico promovido por el Departamento de Defensa de Estados Unidos (DoD). Algunos de los otros marcos listados en esta sección se inspiran en él, como es el caso de TOGAF. En la versión 2 de C4I, completada en diciembre de 1997, la definición de arquitectura reconocida es exactamente la misma que después se promulgaría como canónica en IEEE 1471. Las vistas arquitectónicas homologadas son la Operacional (que identifica relaciones y necesidades de información), la de Sistemas (que vincula capacidades y características a requerimientos operacionales) y la Técnica (que prescribe estándares y convenciones).
- 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. The Open Group propone un modelo de descripción arquitectónica, Architecture Description Method (ADM) que se supone independiente de las técnicas de modelado, aunque en la versión 7 se propone Metis como herramienta.
- 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]. Lo que académicamente se define como AS concierne a las dos primeras vistas. El modelo 4+1 se percibe hoy como un intento se reformular una arquitectura estructural y descriptiva en términos de objeto y de UML. 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 perpectiva, 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 arquitectónica 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 [BCK98]. Esta taxonomía de vista no coincide con ninguna otra.
- 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 riego 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 AS desde 1994 y es hoy de amplio uso.
- 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 y a la percepción 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.

Ahora que se han examinado cuáles son las vistas que proponen los diferentes frameworks, habría que precisar mejor el significado específicamente arquitectónico de las vistas. Como expresa Rich Hilliard [Hil99], a quien seguimos en este tratamiento, aunque expresiones tales como *múltiples vistas* son algo así como el Santo Grial de la ingeniería de software, de requerimientos y de sistemas, su estatus en la AS es bastante más oscuro. Las razones para ello son múltiples. En primer lugar, no existe una fundamentación coherente para su uso en la disciplina. En segundo término, muchos estudiosos las consideran problemáticas, porque la existencia de múltiples vistas introduce problemas de integración y consistencia entre las diferentes vistas. Sin embargo, los arquitectos practicantes las usan de todas maneras, porque simplifican la visualización de sistemas complejos.

La idea de contemplar un sistema complejo desde múltiples puntos de vista no es nativa de la AS contemporánea, ni es una invención de Kruchten, sino que se origina por lo menos en la década de 1970, en el trabajo de Douglas Ross sobre análisis estructurado [Ross77]. La motivación para introducir múltiples vistas radica en la separación de incumbencias (*separations of concerns*). Las vistas se introdujeron como una herramienta conceptual para poder manejar la complejidad de lo que ya por aquel entonces se llamaban artefactos, tales como especificaciones de requerimientos o modelos de diseño. En las contribuciones más tempranas, las múltiples vistas de un modelo se basaban en perspectivas fijas, puntos de vistas o *viewpoints*; casi siempre los puntos de vista eran dos, el funcional y el de datos, ninguno de los cuales aparece en arquitectura.

En la década de 1980, sin embargo, las vistas comenzaron a multiplicarse, al punto que se realizaron *surveys* interdisciplinarios y se organizaron conferencias específicas sobre la cuestión, como *Viewpoints'96*. No hay un límite necesario para el número de vistas posibles, ni un procedimiento formal para establecer lo que una vista debe o no abstraer. El estándar IEEE 1471 no delimita el número posible de vistas, ya que se estima que no puede haber acuerdo en ello, pero señala lineamientos para su constitución y considera que un *viewpoint* es a una vista como una clase es a un objeto.

Hay una "lista corta" de vistas que se usa en los textos generales de AS y una "lista larga" que gira en torno de UML, el cual especifica nueve clases de diagramas correspondientes a ocho vistas, como se indica en el cuadro. Diferentes textos de los mismos autores, por

ejemplo [BRJ99] y [RJB00], utilizan distintas terminologías no conciliadas; vistas y puntos de vista no siempre se caracterizan como conceptos distintos y el uso de la terminología, aún en el interior de cada texto, es informal e inconsistente. Es difícil creer que esto se encuentra "unificado" de alguna manera. Cuando los promotores de UML hablan de arquitectura, a instancias de Kruchten, cambian su modelo de vistas por uno que se refiere no a puntos de perspectiva o a incumbencias de los participantes, sino a niveles de abstracción; pero aún así, como se ha visto, su definición de arquitectura difiere de la definición estándar.

Área	Vista	Diagramas	Conceptos principales
Estructural	Vista estática	Diagrama de clases	Clase, asociación, generalización, dependencia, realización, interfaz
	Vista de casos de uso	Diagramas de casos de uso	Caso de uso, actor, asociación, extensión, inclusión, generalización de casos de uso
	Vista de implementación	Diagrama de componentes	Componente, interfaz, dependencia, realización
	Vista de despliegue	Diagrama de despliegue	Nodo, componente, dependencia, localización
Dinámica	Vista de máquinas de estados	Diagrama de estados	Estado, evento, transición, acción
	Vista de actividad	Diagrama de actividad	Estado, actividad, transición de terminación, división, unión
	Vista de interacción	Diagrama de secuencia	Interacción, objeto, mensaje, activación
		Diagrama de colaboración	Colaboración, interacción, rol de colaboración, mensaje
Gestión del modelo	Vista de gestión del modelo	Diagrama de clases	Paquete, subsistema, modelo

Tabla 2 - Vistas y diagramas de UML, basado en [RJB00: 22]

Como lo subraya Hilliard, en sus textos iniciales la modalidad clásica de la AS, encarnada en Garlan y Shaw, no habla de vistas en absoluto; la AS clásica se funda en una vista singular e implícita, de carácter estructural. Muchos arquitectos de la corriente principal evitan hablar de vistas, porque cuando las ellas proliferan se hace necesario o bien elaborar lenguajes formales específicos para tratar cada una de ellas, o bien multiplicar salvajemente las extensiones del lenguaje unificado. Sin duda las vistas son una simplificación conveniente, o más bien un principio de orden; pero su abundancia y sus complicadas relaciones recíprocas generan también nuevos órdenes de complejidad.

Podría discutirse el concepto y la articulación de las diferentes vistas un largo rato, y de hecho los muchos simposios que han habido sobre el particular fueron de interés pero inconcluyentes. Se acostumbra poner las vistas que denotan "niveles de abstracción" en cuadros superpuestos, por ejemplo, como si fueran análogas a las capas del modelo OSI, pero ¿son ambas representaciones estrictamente homólogas? ¿Constituye el conjunto de las vistas un sistema? ¿Por qué cada vez que se enumeran los artefactos característicos de cada vista aparece la palabra "etcétera" o la expresión "elementos principales"?

Procesos y Metodologías

En los diferentes marcos, las vistas estáticas se corresponden con las perspectivas particulares de los diferentes participantes (*stakeholders*), mientras que las vistas

dinámicas tienen que ver con etapas del proceso, ciclo de vida o metodología, caracterizadas como requerimiento, análisis, diseño (o construcción, o modelado), implementación, integración (prueba de conformidad, *testing*, evaluación). La terminología, lo mismo que la articulación temporal del proceso o el ciclo, depende de cada marco. Llegando al territorio de la metodología, hay que decir que durante varios años la AS discurrió sin elaborarlas más que circunstancialmente, como si se estimara compatible con las prácticas establecidas en ingeniería de software, cualesquiera fuesen: RUP, RAD, RDS, ARIS, PERA, CIMOSA, GRAI, GERAM, CMM. Hoy en día la metodología dominante en la industria es tal vez el Modelo de Madurez de la Capacidad (CMM), aunque el SEI no la considera formalmente como tal.

Desde 1998 y cada vez con mayor intensidad, sin embargo, el SEI y otros organismos comenzaron a elaborar métodos específicos de procesos de ingeniería que sistematizan el rol de la arquitectura en la totalidad del proceso, desde la elicitación de requerimientos hasta la terminación. Algunos de esos métodos son Architecture Based Design (ABD), Software Architecture Analysis Method (SAAM), Quality Attribute Workshops (QAW), Quality Attribute-Oriented Software Architecture Design Method (QASAR), Attribute-Driven Design (ADD), Architecture Tradeoff Analysis Method (ATAM), Active Review for Intermediate Design (ARID), Cost-Benefits Analysis Method (CBAM), Family-Architecture Analysis Method (FAAM), Architecture Level Modifiability Analysis (ALMA), y Software Architecture Comparison Analysis Method (SACAM). Al lado de esos métodos está surgiendo un nutrido conjunto de técnicas de documentación; métodos, técnicas y estudios de casos se analizarán en este estudio en documentos separados. Habiendo al menos una docena de métodos complejamente engranados entre sí, el lector puede perderse con facilidad en un laberinto bibliográfico donde no siempre se discierne cuáles de estos métodos incluyen a cuáles otros, cuáles son sus relaciones con técnicas consagradas de ingeniería, cuáles se encuentran vigentes y cuáles obsoletos. Un documento actual que describe a grandes rasgos la mayoría de esos métodos es [KNK03].

En general, la AS de la corriente principal todavía no se ha expedido en relación con los llamados métodos ágiles. No hay un solo método, sino una multiplicidad de posturas más o menos radicales y combativas: Extreme Programming, SCRUM, Crystal Methods Framework, Feature-Driven Development, DSDM, Lean Development, Adaptive Software Development, Agile Modeling, Pragmatic Programming [ASR+02] [CLC03]. De hecho, la comunidad de los metodólogos, que opera a una cierta distancia de las iniciativas formales de la AS, se encuentra dividida tras lo que ha sido y sigue siendo "el gran debate metodológico" entre los métodos pesados (o rigurosos) de tipo SEI/CMM por un lado y los métodos ágiles por el otro [Hig01]. Los teóricos de cada bando han sido, entre otros, Tom De Marco, Ed Yourdon y Tim Lister en la facción rigurosa y Ken Orr, Jim Highsmith, Martin Fowler y Michael Jackson del lado ágil-extremo, con Philippe Kruchten y RUP buscando establecerse en ambos terrenos.

Ambos grupos operan en el contexto de la crisis del software, que se da por sentada, alegando que es la mentalidad del bando opuesto lo que la ha ocasionado. Los pesados, mirados por los ágiles como formalistas fracasados, enfatizan el modelado, el control y la documentación escrupulosa; los ágiles, acusados por los pesados de *hackers* irresponsables que pretenden imponer sus juguetes en la empresa, no sólo desprecian los modelos (formales o informales) sino que incluso ocasionalmente ponen en tela de juicio

hasta a los propios patrones de diseño [Fow01]. Ese tema también será tratado en un estudio aparte.

En lo que respecta a la estrategia arquitectónica de Microsoft, el marco general de Microsoft Solutions Framework, versión 3, no se expide sobre metodologías específicas y da cabida a una gran cantidad de alternativas, desde las densas a las ligeras. Abundan en MSF 3 referencias a métodos rápidos y ágiles, a través de citas y conceptos de una de las figuras cardinales de Cutter Consortium, Martin Fowler. La documentación de MSF ratifica su adecuación tanto para el CMM de SEI o UPM como para los métodos ágiles en ambientes que requieren un alto grado de adaptabilidad [MS03].

Abstracción

El concepto de abstracción (que a veces se usa en el sentido del proceso de abstraer, otra para designar una entidad) ha sufrido también diversas acepciones, con un núcleo de significados común. Las diferencias en el uso del concepto de abstracción ayudan también a identificar las diversas corrientes en el seno de la AS. La definición que proporciona Grady Booch, por ejemplo, revela que el autor identifica la abstracción arquitectónica con el encapsulamiento propio de la tecnología de objetos: “Una abstracción –escribe Booch– denota las características esenciales de un objeto que lo distinguen de otras clases de objeto y provee de este modo delimitaciones conceptuales bien definidas, relativas a la perspectiva del observador” [Boo91].

El concepto de abstracción (que a veces se usa en el sentido del proceso de abstraer, otra para designar una entidad abstracta) ha sufrido también diversas formulaciones sintácticas, con un núcleo de significados común. En último análisis, la abstracción siempre conlleva una heurística positiva al lado de una negación. Tanto para la IEEE como para Rumbaugh, Shaw y otros autores, la abstracción consiste en extraer las propiedades esenciales, o identificar los aspectos importantes, o examinar selectivamente ciertos aspectos de un problema, posponiendo o ignorando los detalles menos sustanciales, distractivos o irrelevantes.

La idea de abstracción forma parte de lo que acaso sea la pieza conceptual más importante de la AS, el concepto de estilo; un estilo se identifica a grandes rasgos o, como se dice habitualmente, en un estilo “menos es más”. La misma idea prevalece en todos los conceptos y procedimientos que se consideran arquitectónicos. Para Len Bass, Paul Clements y Rick Kazman [BCK98], si una decisión debe posponerse hasta el momento de tratar las cosas a un bajo nivel, no se trata de una decisión de arquitectura. Clements y Northrop [CN96] sostienen que el trabajo de Garlan y Shaw sobre los estilos arquitectónicos nos enseña que aunque los programas pueden combinarse de maneras prácticamente infinitas, hay mucho que ganar si nos restringimos a un conjunto relativamente pequeño de elecciones cuando se trata de cooperación e interacción. Las ventajas incluyen mejor reutilización, mejores análisis, menor tiempo se selección y mayor interoperabilidad. “Menos es más” figura, de hecho, en el título y en los contenidos de numerosos *papers* de la profesión [MB02] [CN96]. Conceptos como el de estilo, o marcos como MSF, revelan su naturaleza arquitectónica en su abstracción y en su generalidad.

Escenarios

Esta es una noción arquitectónica importante y se encuentra en la base de muchos de los métodos de diseño y desarrollo basados en arquitectura, como ALMA, SAAM y ATAM. Hay que ser precavidos y advertir desde el comienzo que lo que habitualmente se traduce como “escenario” no es estrictamente lo que en lengua castellana se designa como tal; la traducción correcta debería ser más bien “guión” o “libreto”. La traducción literal del término no hace más que aportar confusión. Desde Kruchten en adelante, se reconoce que los escenarios se dividen en dos categorías: casos de uso (secuencias de responsabilidades) y casos de cambio (modificaciones propuestas al sistema).

Los escenarios han sido básicamente técnicas que se implementan en la elicitación de los requerimientos, particularmente en relación a los operadores de sistemas. Típicamente, la técnica comienza instrumentando sesiones de *brainstorming*. También se han utilizado escenarios como método para comparar alternativas de diseño. Los escenarios describen una utilización anticipada o deseada del sistema, y típicamente se expresan en una frase; Kazman, Abowd, Bass y Clements proponen también llamarlos viñetas [KAB+96].

Según algunas definiciones, como la de Clements y Northrop [CN96], los escenarios son algo así como libretos (en el sentido teatral o cinematográfico del término) correspondientes a las distintas piezas de funcionalidad de un sistema. Se los considera útiles para analizar una vista determinada [Cle95a] o para mostrar la forma en que los elementos de múltiples vistas se relacionan entre sí [Kru95]. Pueden concebirse también como una abstracción de los requerimientos más importantes de un sistema. Los escenarios se describen mediante texto común en prosa utilizando lo que se llama un *script* y a veces se describen mediante dibujos, como por ejemplo diagramas de interacción de objeto. Se acostumbra utilizar UML (en el contexto de 4+1) no tanto como recurso de modelado que después generará alguna clase de código, sino como instrumento de dibujo más o menos informal; pero los propios manuales de UML y los expertos mundiales en casos de uso (David Anderson, Martin Fowler, Alistair Cockburn) recomiendan desarrollar los escenarios de requerimiento en texto, no en diagramas. Algunos autores estiman que se trata de una herramienta importante para relacionar vistas arquitectónicas, porque recorriendo un escenario puede mostrar las formas en que fragmentos o escenas de esas vistas se corresponden entre sí. Los métodos propios de la arquitectura basada en escenarios se analizan también en un documento aparte.

Campos de la Arquitectura de Software

La AS es hoy en día un conjunto inmenso y heterogéneo de áreas de investigación teórica y de formulación práctica, por lo que conviene aunque más no sea enumerar algunos de sus campos y sus focos. Una semblanza semejante (de la que aquí se proporciona sólo un rudimento) dudosamente debería ser estática. La AS comenzó siendo una abstracción descriptiva puntual que en los primeros años no investigó de manera sistemática ni las relaciones que la vinculaban con los requerimientos previos, ni los pasos metodológicos a dar luego para comenzar a componer el diseño. Pero esa sincronicidad estructuralista no pudo sostenerse. Por el contrario, daría la impresión que, a medida que pasa el tiempo, la AS tiende a redefinir todos y cada uno de los aspectos de la disciplina madre, la

ingeniería de software, sólo que a un mayor nivel de abstracción y agregando una nueva dimensión reflexiva en lo que concierne a la fundamentación formal del proceso.

Hay unas pocas caracterizaciones (y mucha actividad de copiado y pegado) en torno de las áreas que componen el territorio. David Garlan y Dewayne Perry, en su introducción al volumen de abril de 1995 de IEEE Transactions on Software Engineering dedicado a la AS, en el cual se delinean las áreas de investigación más promisorias, enumeran las siguientes:

- Lenguajes de descripción de arquitecturas
- Fundamentos formales de la AS (bases matemáticas, caracterizaciones formales de propiedades extra-funcionales tales como mantenibilidad, teorías de la interconexión, etcétera).
- Técnicas de análisis arquitectónicas
- Métodos de desarrollo basados en arquitectura
- Recuperación y reutilización de arquitectura
- Codificación y guía arquitectónica
- Herramientas y ambientes de diseño arquitectónico
- Estudios de casos

Fundamental en la concepción de Clements y Northrop [CN96] es el criterio de reusabilidad como uno de los aspectos que más hacen por justificar la disciplina misma. Según estos autores, el estudio actual de la AS puede ser visto como un esfuerzo ex post facto para proporcionar un almacén estructurado de este tipo de información reutilizable de diseño de alto nivel propio de una familia (en el sentido de Parnas). De tal manera, las decisiones de alto nivel inherentes a cada miembro de una familia de programas no necesitan ser re-inventadas, re-validadas y re-descriptas. Un razonamiento arquitectónico es además un argumento sobre las cuestiones estructurales de un sistema. Se diría también que el concepto de estilo es la encarnación principal del principio de reusabilidad en el plano arquitectónico.

Paul Clements [Cle96b] define cinco temas fundamentales en torno de los cuales se agrupa la disciplina:

- **Diseño o selección de la arquitectura:** Cómo crear o seleccionar una arquitectura en base de requerimientos funcionales, de performance o de calidad.
- **Representación de la arquitectura:** Cómo comunicar una arquitectura. Este problema se ha manifestado como el problema de la representación de arquitecturas utilizando recursos lingüísticos, pero el problema también incluye la selección del conjunto de información a ser comunicada.
- **Evaluación y análisis de la arquitectura:** Cómo analizar una arquitectura para predecir cualidades del sistema en que se manifiesta. Un problema semejante es cómo comparar y escoger entre diversas arquitecturas en competencia.
- **Desarrollo y evolución basados en arquitectura:** Cómo construir y mantener un sistema dada una representación de la cual se cree que es la arquitectura que resolverá el problema correspondiente.

- **Recuperación de la arquitectura:** Cómo hacer que un sistema *legacy* evolucione cuando los cambios afectan su estructura; para los sistemas de los que se carezca de documentación confiable, esto involucra primero una "arqueología arquitectónica" que extraiga su arquitectura.

Mary Shaw [Shaw01] considera que en el 2001 los campos más promisorios de la AS siguen teniendo que ver con el tratamiento sistemático de los estilos, el desarrollo de lenguajes de descripción arquitectónica, la formulación de metodologías y (ahora) el trabajo con patrones de diseño. Se requieren todavía modelos precisos que permitan razonar sobre las propiedades de una arquitectura y verificar su consistencia y completitud, así como la automatización del proceso de análisis, diseño y síntesis. Sugiere que debe aprenderse una lección a partir de la experiencia de la ingeniería de software, la cual no obstante haberse desenvuelto durante treinta años no ha logrado plasmar un conjunto de paradigmas de investigación comparable al de otras áreas de las ciencias de la computación. Estima que la AS se encuentra ya en su fase de desarrollo y extensión, pero que tanto las ideas como las herramientas distan de estar maduras.

Un campo que no figura en estas listas pero sobre el cual se está trabajando intensamente es en el de la coordinación de los ADLs que sobrevivan con UML 2.0 por un lado y con XML por el otro. Ningún lenguaje de descripción arquitectónica en el futuro próximo (excepto los que tengan un nicho técnico muy particular) será viable si no satisface esos dos requisitos.

Los ejercicios que pueden hacerse para precisar los campos de la AS son incontables. Ahora que la AS se ha abismado en el desarrollo de metodologías, hace falta, por ejemplo, establecer con más claridad en qué difieren sus elaboraciones en torno del diseño, del análisis de requerimientos o de justificación económica de las llevadas a cabo por la ingeniería de software. Asimismo, se está esperando todavía una lista sistemática y exhaustiva que describa los dominios de incumbencia de la disciplina, así como un examen del riesgo de duplicación de esfuerzos entre campos disciplinarios mal comunicados, una situación que a primera vista parecería contradictoria con el principio de reusabilidad.

Modalidades y tendencias

En la década de 1990 se establece definitivamente la AS como un dominio todavía hoy separado de manera confusa de ese marco global que es la ingeniería y de esa práctica puntual que es el diseño. Aunque no hay un discurso explícito y autoconsciente sobre escuelas de AS, ni se ha publicado un estudio reconocido y sistemático que analice las particularidades de cada una, en la actualidad se pueden distinguir a grandes rasgos unas seis corrientes. Algunas distinciones están implícitas por ejemplo en [MT00], pero la bibliografía sobre corrientes y alternativas prácticamente no existe y la que sigue habrá de ser por un tiempo una de las pocas propuestas contemporáneas sobre el particular.

Ahora bien, articular una taxonomía de estrategias no admite una solución simple y determinista. En distintos momentos de su trayectoria, algunos practicantes de la AS se mueven ocasionalmente de una táctica a otra, o evolucionan de un punto de vista más genérico a otro más particular, o realizan diferentes trabajos operando en marcos distintos. Además, con la excepción del "gran debate metodológico" entre métodos

pesados y ligeros [Hig01], las discusiones entre las distintas posturas rara vez se han manifestado como choques frontales entre ideologías irreconciliables, por lo que a menudo hay que leer entre líneas para darse cuenta que una afirmación cualquiera es una crítica a otra manera de ver las cosas, o trasunta una toma definida de posición. Fuera de la metodología, el único factor reconocible de discordia ha sido, hasta la fecha, la preminencia de UML y el diseño orientado a objetos. Todo lo demás parece ser más o menos negociable.

La división preliminar de escuelas de AS que proponemos es la siguiente:

1) **Arquitectura como etapa de ingeniería y diseño orientada a objetos**. Es el modelo de James Rumbaugh, Ivar Jacobson, Grady Booch, Craig Larman y otros, ligado estrechamente al mundo de UML y Rational. No cabe duda que se trata de una corriente específica: Rumbaugh, Jacobson y Booch han sido llamados "Los Tres Amigos"; de lo que sí puede dudarse es que se trate de una postura *arquitectónica*. En este postura, la arquitectura se restringe a las fases iniciales y preliminares del proceso y concierne a los niveles más elevados de abstracción, pero no está sistemáticamente ligada al requerimiento que viene antes o a la composición del diseño que viene después. Lo que sigue al momento arquitectónico es *business as usual*, y a cualquier configuración, topología o morfología de las piezas del sistema se la llama arquitectura. En esta escuela, si bien se reconoce el valor primordial de la abstracción (nadie después de Dijkstra osaría oponerse a ello) y del ocultamiento de información promovido por Parnas, estos conceptos tienen que ver más con el encapsulamiento en clases y objetos que con la visión de conjunto arquitectónica. Para este movimiento, la arquitectura se confunde también con el modelado y el diseño, los cuales constituyen los conceptos dominantes. En esta corriente se manifiesta predilección por un modelado denso y una profusión de diagramas, tendiente al modelo metodológico CMM o a UPM; no hay, empero, una precripción formal. Importa más la abundancia y el detalle de diagramas y técnicas disponibles que la simplicidad de la visión de conjunto. Cuando aquí se habla de estilos, se los confunde con patrones arquitectónicos o de diseño [Lar03]. Jamás se hace referencia a los lenguajes de descripción arquitectónica, que representan uno de los *assets* reconocidos de la AS; sucede como si la disponibilidad de un lenguaje unificado de modelado los tornara superfluos. La definición de arquitectura que se promueve en esta corriente tiene que ver con aspectos formales a la hora del desarrollo; esta arquitectura es isomorfa a la estructura de las piezas de código. Una definición típica y demostrativa sería la de Grady Booch; para él, la AS es "la estructura lógica y física de un sistema, forjada por todas las decisiones estratégicas y tácticas que se aplican durante el desarrollo" [Boo91]. Otras definiciones revelan que la AS, en esta perspectiva, concierne a decisiones sobre organización, selección de elementos estructurales, comportamiento, composición y estilo arquitectónico susceptibles de ser descriptas a través de las cinco vistas clásicas del modelo 4+1 de Kruchten [Kru95] [BRJ99: 26-28].
2) **Arquitectura estructural**, basada en un modelo estático de estilos, ADLs y vistas. Constituye la corriente fundacional y clásica de la disciplina. Los representantes de esta corriente son todos académicos, mayormente de la Universidad Carnegie Mellon en Pittsburgh: Mary Shaw, Paul Clements, David Garlan, Robert Allen, Gregory

Abowd, John Ockerbloom. Se trata también de la visión de la AS dominante en la academia, y aunque es la que ha hecho el esfuerzo más importante por el reconocimiento de la AS como disciplina, sus categorías y herramientas son todavía mal conocidas en la práctica industrial. En el interior del movimiento se pueden observar distintas divisiones. Hay una variante informal de "boxología" y una vertiente más formalista, representada por el grupo de Mark Moriconi en el SRI de Menlo Park. En principio se pueden reconocer tres modalidades en cuanto a la formalización; los más informales utilizan descripciones verbales o diagramas de cajas, los de talante intermedio se sirven de ADLs y los más exigentes usan lenguajes formales de especificación como CHAM y Z. En toda la corriente, el diseño arquitectónico no sólo es el de más alto nivel de abstracción, sino que además no tiene por qué coincidir con la configuración explícita de las aplicaciones; rara vez se encontrarán referencias a lenguajes de programación o piezas de código, y en general nadie habla de clases o de objetos. Mientras algunos participantes excluyen el modelo de datos de las incumbencias de la AS (Shaw, Garlan, etc), otros insisten en su relevancia (Medvidovic, Taylor). Todo estructuralismo es estático; hasta fines del siglo XX, no está muy claro el tema de la posición del modelado arquitectónico en el ciclo de vida.

3) **Estructuralismo arquitectónico radical**. Se trata de un desprendimiento de la corriente anterior, mayoritariamente europeo, que asume una actitud más confrontativa con el mundo UML. En el seno de este movimiento hay al menos dos tendencias, una que excluye de plano la relevancia del modelado orientado a objetos para la AS y otra que procura definir nuevos metamodelos y estereotipos de UML como correctivos de la situación. Dado que pueden existir dudas sobre la materialidad de esta corriente como tal, proporcionamos referencias bibliográficas masivas que corroboran su existencia [Abd00] [AW99] [DDT99] [Dou00] [GAD+02] [GarS/f] [Gli00] [HNS99] [KroS/f] [Sch00] [StöS/f] [Tem99] [Tem01]. Pasaremos revista a los argumentos más fuertes en contra de UML emanados de este movimiento en el capítulo correspondiente del documento sobre lenguajes de descripción arquitectónica.

4) **Arquitectura basada en patrones**.[1] Si bien reconoce la importancia de un modelo emanado históricamente del diseño orientado a objetos, esta corriente surgida hacia 1996 no se encuentra tan rígidamente vinculada a UML en el modelado, ni a CMM en la metodología. El texto sobre patrones que esta variante reconoce como referencia es la serie *POSA* de Buschmann y otros [BMR+96] y secundariamente el texto de la Banda de los Cuatro [Gof95]. La diferencia entre ambos textos sagrados de la comunidad de patrones no es menor; en el primero, la expresión "Software Architecture" figura en el mismo título; el segundo se llama *Design Patterns: Elements of reusable Object-Oriented software* y su tratamiento de la arquitectura es mínimo. En esta manifestación de la AS prevalece cierta tolerancia hacia modelos de proceso tácticos, no tan macroscópicos, y eventualmente se expresa cierta simpatía por las ideas de Martin Fowler y las premisas de la programación extrema. El diseño

[1] Se han definido los patrones arquitectónicos y de diseño en el documento sobre Estilos en Arquitectura de Software [*Ref].

consiste en identificar y articular patrones preexistentes, que se definen en forma parecida a los estilos de arquitectura [Shaw96] [MKM+96] [MKM+97].

5) **Arquitectura procesual**. Desde comienzos del siglo XXI, con centro en el SEI y con participación de algunos (no todos) los arquitectos de Carnegie Mellon de la primera generación y muchos nombres nuevos de la segunda: Rick Kazman, Len Bass, Paul Clements, Felix Bachmann, Fabio Peruzzi, Jeromy Carrière, Mario Barbacci, Charles Weinstock. Intenta establecer modelos de ciclo de vida y técnicas de elicitación de requerimientos, *brainstorming*, diseño, análisis, selección de alternativas, validación, comparación, estimación de calidad y justificación económica específicas para la arquitectura de software. Toda la documentación puede encontrarse ordenada en el SEI, pero no se mezcla jamás con la de CMM, a la que redefine de punta a punta. Otras variantes dentro de la corriente procesual caracterizan de otras maneras de etapas del proceso: extracción de arquitectura, generalización, reutilización [WL97].

6) **Arquitectura basada en escenarios**. Es la corriente más nueva. Se trata de un movimiento predominantemente europeo, con centro en Holanda. Recupera el nexo de la AS con los requerimientos y la funcionalidad del sistema, ocasionalmente borroso en la arquitectura estructural clásica. Los teóricos y practicantes de esta modalidad de arquitectura se inscriben dentro del canon delineado por la arquitectura procesual, respecto de la cual el movimiento constituye una especialización. En esta corriente suele utilizarse diagramas de casos de uso UML como herramienta informal u ocasional, dado que los casos de uso son uno de los escenarios posibles. Los casos de uso *no* están orientados a objeto. Los autores vinculados con esta modalidad han sido, aparte de los codificadores de ATAM, CBAM, QASAR y demás métodos del SEI, los arquitectos holandeses de la Universidad Técnica de Eindhoven, de la Universidad Brije, de la Universidad de Groningen y de Philips Research: Mugurel Ionita, Dieter Hammer, Henk Obbink, Hans de Bruin, Hans van Vliet, Eelke Folmer, Jilles van Gurp, Jan Bosch. La profusión de holandeses es significativa; la Universidad de Eindhoven es, incidentalmente, el lugar en el que surgió lo que P. I. Sharp propopía llamar la escuela arquitectónica de Dijkstra.

En todos los simposios han habido intentos de fundación de otras variedades de AS, tales como una arquitectura adaptativa inspirada en ideas de la programación genética o en teorías de la complejidad, la auto-organización, el caos y los fractales, una arquitectura centrada en la acción que recurre a la inteligencia artificial heideggeriana o al posmodernismo, y una arquitectura epistemológicamente reflexiva que tiene a Popper o a Kuhn entre sus referentes; consideramos preferible omitirlas, porque por el momento ni su masa crítica es notoria ni su mensaje parece sustancial [DD94] [HHr+96] [ZHH+99] [Hock00]. Pero hay al menos un movimiento cismático digno de tomarse más en serio; a esta altura de los acontecimientos es muy posible que pueda hablarse también de una anti-arquitectura, que en nombre de los métodos heterodoxos se opone tanto al modelado orientado a objetos y los métodos sobredocumentados impulsados por consultoras corporativas como a la abstracción arquitectónica que viene de la academia. El Manifiesto por la Programación Agil, en efecto, valoriza:

- Los individuos y las interacciones por encima de los procesos y las herramientas
- El software que funciona por encima de la documentación exhaustiva
- La colaboración con el cliente por encima de la negociación contractual

- La respuesta al cambio por encima del seguimiento de un plan

Habrá oportunidad de desarrollar el tratamiento de estas metodologías en otros documentos de la serie.

Hay muchas formas, por cierto, de organizar el panorama de las tendencias y las escuelas. Discernir la naturaleza de cada una puede tener efectos prácticos a la hora de comprender antagonismos, concordancias, sesgos, disyuntivas, incompatibilidades, fuentes de recursos. Nadie ha analizado mejor los límites de una herramienta como UML o de los métodos ultra-formales, por ejemplo, que los que se oponen a su uso por razones teóricas precisas, por más que éstas sean interesadas. Un buen ejercicio sería aplicar vistas y perspectivas diferentes a las que han regido esta clasificación informal, y proponer en función de otros criterios o de métodos más precisos de composición y referencias cruzadas, taxonomías alternativas de las líneas de pensamiento vigentes en la AS en la actualidad.

Diferencias entre Arquitectura y Diseño

Una vez que se reconoce la diferencia, que nunca debió ser menos que obvia, entre diseño e implementación, o entre vistas conceptuales y vistas tecnológicas ¿Es la AS solamente otra palabra para designar el diseño? Como suele suceder, no hay una sola respuesta, y las que hay no son taxativas. La comunidad de AS, en particular la de extracción académica, sostiene que ésta difiere sustancialmente del mero diseño. Pero Taylor y Medvidovic [TM00], por ejemplo, señalan que la literatura actual mantiene en un estado ambiguo la relación entre ambos campos, albergando diferentes interpretaciones y posturas:

1) Una postura afirma que arquitectura y diseño son lo mismo.
2) Otra, en cambio, alega que la arquitectura se encuentra en un nivel de abstracción por encima del diseño, o es simplemente otro paso (un artefacto) en el proceso de desarrollo de software.
3) Una tercera establece que la arquitectura es algo nuevo y en alguna medida diferente del diseño (pero de qué manera y en qué medida se dejan sin especificar).

Taylor y Medvidovic estiman que la segunda interpretación es la que se encuentra más cerca de la verdad. En alguna medida, la arquitectura y el diseño sirven al mismo propósito. Sin embargo, el foco de la AS en la estructura del sistema y en las interconexiones la distingue del diseño de software tradicional, tales como el diseño orientado a objetos, que se concentra más en el modelado de abstracciones de más bajo nivel, tales como algoritmos y tipos de datos. A medida que la arquitectura de alto nivel se refina, sus conectores pueden perder prominencia, distribuyéndose a través de los elementos arquitectónicos de más bajo nivel, resultando en la transformación de la arquitectura en diseño.

En su reciente libro sobre el arte de la AS, Stephen Albin [Alb03] se pregunta en qué difiere ella de las metodologías de diseño bien conocidas como la orientación a objetos. La AS, se contesta, es una metáfora relativamente nueva en materia de diseño de software y en realidad abarca también las metodologías de diseño, así como metodologías de análisis. El arquitecto de software contemporáneo, escribe Albin, ejecuta una

combinación de roles como los de analista de sistemas, diseñador de sistemas e ingeniero de software. Pero la arquitectura es más que una recolocación de funciones. Esas funciones pueden seguir siendo ejecutadas por otros, pero ahora caen comúnmente bajo la orquestación del *chief architect*. El concepto de arquitectura intenta subsumir las actividades de análisis y diseño en un framework de diseño más amplio y más coherente. Las organizaciones se están dando cuenta que el alto costo del desarrollo de software requiere ser sometido a algún control y que muchas de las ventajas prometidas por las metodologías aún no se han materializado. Pero la arquitectura es algo más integrado que la suma del análisis por un lado y el diseño por el otro. La integración de metodologías y modelos, concluye Albin, es lo que distingue la AS de la simple yuxtaposición de técnicas de análisis y de diseño.

Para Shaw y Garlan [SG96] la AS es el primer paso en la producción de un diseño de software, en una secuencia que distingue tres pasos:

1) Arquitectura. Asocia las capacidades del sistema especificadas en el requerimiento con los componentes del sistema que habrán de implementarla. La descripción arquitectónica incluye componentes y conectores (en términos de estilos) y la definición de operadores que crean sistemas a partir de subsistemas o, en otros términos, componen estilos complejos a partir de estilos simples.
2) Diseño del código. Comprende algoritmos y estructuras de datos; los componentes son aquí primitivas del lenguaje de programación, tales como números, caracteres, punteros e hilos de control. También hay operadores primitivos.
3) Diseño ejecutable. Remite al diseño de código a un nivel de detalle todavía más bajo y trata cuestiones tales como la asignación de memoria, los formatos de datos, etcétera.

En opinión de Clements [CBK+96] el diseño basado en arquitectura representa un paradigma de desarrollo que difiere de maneras fundamentales de las alternativas conocidas actualmente. En muchos sentidos, es diferente del diseño orientado a objetos (OOD) en la misma medida en que éste difería de sus predecesores. La AS deberá nutrir una comunidad de practicantes estableciendo una cultura en la que las ideas arquitectónicas puedan florecer. Una cuestión técnica que deberá abordarse es la creación de una articulación precisa de un paradigma de diseño basado en arquitectura (o posiblemente más de uno) y las cuestiones de proceso asociadas.

En una presentación de 1997, Dewayne Perry, uno de los fundadores de la disciplina, bosquejó la diferencia entre arquitectura y diseño. La arquitectura, una vez más (todo el mundo insiste en ello) concierne a un nivel de abstracción más elevado; se ocupa de componentes y no de procedimientos; de las interacciones entre esos componentes y no de las interfaces; de las restricciones a ejercer sobre los componentes y las interacciones y no de los algoritmos, los procedimientos y los tipos. En cuanto a la composición, la de la arquitectura es de grano grueso, la del diseño es de fina composición procedural; las interacciones entre componentes en arquitectura tienen que ver con un protocolo de alto nivel (en el sentido no técnico de la palabra), mientras que las del diseño conciernen a interacciones de tipo procedural (rpc, mensajes, llamadas a rutinas) [Per97].

En los primeros años del nuevo siglo, la AS precisó la naturaleza del proceso de diseño como metodología en diversos modelos de diseño basados en arquitectura o ABD

[BBC+00]. Esta metodología considera que el diseño arquitectónico es el de más elevado nivel de abstracción, pero debe hacer frente al hecho de un requerimiento todavía difuso y al hecho de que, en ese plano, las decisiones que se tomen serán las más críticas y las más difíciles de modificar. Fundándose en el concepto de arquitectura conceptual de Hofmeister, Nord y Soni [HNS00] y en un modelos de vistas similar al 4+1 o a las vistas del modelo arquitectónico de Microsoft, el ABD describe el sistema en función de los principales elementos y las relaciones entre ellos. El proceso se basa en tres fundamentos: (1) la descomposición de la función (usando técnicas bien establecidas de acoplamiento y cohesión), (2) la realización de los requerimientos de calidad y negocios a través de los estilos arquitectónicos, y (3) las *plantillas de software*, un concepto nuevo que incluye patrones que describen la forma en que todos los elementos de un tipo interactúan con los servicios compartidos y la infraestructura. El modelo de diseño específico de ABD se describe en los documentos correspondientes a metodologías arquitectónicas.

Seguramente el lector encontrará mucho que agregar al planteamiento de las similitudes y diferencias entre arquitectura y diseño, más allá del hecho obvio de que el diseño arquitectónico se distingue del diseño del código, que viene determinado desde mucho antes y que es mucho más crítico para el éxito o el fracaso de una solución.

Repositorios

Existen unos cuantos repositorios de información arquitectónica, cuyas direcciones son más o menos permanentes. El más importante hoy en día parece ser el del Software Engineering Institute en la Universidad Carnegie Mellon de Pittsburgh, Pennsylvania (http://www.sei.cmu.edu/ata/ata_init.html). El sitio del SEI incluye abundante literatura académica y todas las especificaciones o recomendaciones metodológicas.

Entre los organismos que definen estándares, son esenciales los servicios de información de RM-ODP en http://www.dstc.edu.au/Research/Projects/ODP/ref_model.html, TOGAF en http://www.software.org/pub/architecture/togaf.asp, DoDAF (hogar del C4ISR) en http://www.software.org/pub/architecture/dodaf.asp. El estándar IEEE 1471-2000 se puede encontrar en http://www.techstreet.com/cgi-bin/detail?product_id=879737. El marco de referencia empresarial de Zachman se puede acceder desde http://www.software.org/pub/architecture/zachman.asp.

Las páginas de cabecera de la estrategia de arquitectura de Microsoft se hallan en http://msdn.microsoft.com/architecture/. Las páginas de Patterns & Practices están en http://www.microsoft.com/resources/practices/completelist.asp. La estrategia arquitectónica misma se encuentra delineada en un documento de Michael Platt en http://msdn.microsoft.com/architecture/overview/default.aspx?pull=/library/en-us/dnea/html/eaarchover.asp.

Numerosas editoriales, universidades, consorcios y centros de investigación incluyen vínculos ligados a la AS. Muy seguramente los lectores podrán suministrar otros sitios de referencia importantes.

Problemas abiertos en Arquitectura de Software

Aunque la evaluación dominante de la trayectoria de la AS es abiertamente positiva, en los primeros años del siglo comienzan a percibirse desacuerdos y frustraciones en el proyecto de la disciplina. Por empezar, está muy claro que la definición original del proyecto, formulada en círculos académicos, guarda poca relación con la imagen que se tiene de la AS en las prácticas de la industria. Aún dentro de los confines de aquel círculo, a menudo las presentaciones distan de ser contribuciones desinteresadas y representan más bien posturas teóricas particulares que se quieren imponer por vía de la argumentación. Muchas de esas posturas son más programáticas que instrumentales. Un número desmesurado de artículos y ponencias destaca asimismo la frustración que muchos sienten por no haber podido consensuar una definición de la AS universalmente aceptada.

Vale la pena revisar el inventario de aspectos negativos elaborada por Clements y Northrop [CN96]:

> **Los abogados de la distintas posturas traen sus sesgos consigo.** En efecto, mientras las definiciones propuestas para la AS coinciden en su núcleo, difieren seriamente en los bordes. Algunos autores requieren que la arquitectura incluya racionalización, otros piden más bien pasos para el proceso de construcción. Algunos exigen que se identifique la funcionalidad de los componentes, otros alegan que una simple topología es suficiente. En la lectura de los textos de AS se torna esencial entonces conocer la motivación de sus responsables.
>
> **El estudio de la AS está siguiendo a la práctica, no liderándola.** El estudio de la AS ha evolucionado de la observación de los principios de diseño y las acciones que toman los diseñadores cuando trabajan en sistemas de la vida real. La AS es un intento de abstraer los rasgos comunes inherentes al diseño de sistemas, y como tal debe dar cuenta de un amplio rango de actividades, conceptos, métodos, estrategias y resultados. Lo que en realidad sucede es que la AS se redefine constantemente en función de lo que hacen los programadores.
>
> **El estudio es sumamente nuevo.** Aunque sus raíces se prolongan en el tiempo, el campo de la AS es realmente muy nuevo, según puede juzgarse de la reciente avalancha de libros, conferencias, talleres y literatura consagrado a ella.
>
> **Las fundamentaciones han sido imprecisas.** El campo se ha destacado por la proliferación de términos inciertos, verdaderas amenazas para los no precavidos. Por ejemplo, la arquitectura definida como “la estructura global de un sistema” agrega confusión en lugar de reducirla, porque implica que un sistema posee una sola estructura.
>
> **El término se ha usado en exceso.** El significado de la palabra “arquitectura” en su relación con la ingeniería de software se ha ido diluyendo simplemente porque parece estar de moda. Es posible encontrar referencias a las siguientes “clases” de arquitectura: específica de dominio, megaprogramación, destinatario, sistemas, redes, infraestructura, aplicaciones, operaciones, técnica, framework, conceptual, de referencia, empresarial, de factoría, C4I, de manufactura, de edificio, de

máquina-herramienta, etcétera. A menudo la palabra “arquitectura” se usa de maneras inapropiadas [CN96].

Hacia el año 2000, Taylor y Medvidovic celebraban el auge de la AS, pero señalaban que por momentos daba la impresión de haberse convertido en una moda que prometía más de lo que se podía realizar. Les preocupaba también que la AS terminara confundiéndose con los ADLs y con el diseño. Si bien el foco en la arquitectura posee un tremendo potencial, argumentan, está comenzando a percibirse una cierta reacción. Las razones son numerosas. La arquitectura es aún en gran medida una noción académica, y muy poca de la tecnología resultante ha sido transferida a la industria y cuando se lo ha hecho no se lo ha comunicado bien. La concentración en la investigación como un valor en sí mismo ha sido también mal comprendida. Mientras tales errores ganen terreno, surge el peligro de que los beneficios concretos y los resultados positivos de la investigación arquitectónica sufran las consecuencias [TMA+S/f].

Jason Baragry ha cuestionado la metáfora de la arquitectura de edificios como columna vertebral de la disciplina. Ello ha ocasionado, entre otras cosas, la falta de acuerdo en cuanto a la cantidad y calidad de vistas arquitectónicas, así como la ostensible carencia de un mecanismo de coordinación formal entre las diferentes vistas [BG01], un problema técnico que también surge, incidentalmente, en la integración de las vistas y diagramas de UML y en las recomendaciones de los organismos.

Un problema que habría que tratar sistemáticamente es la discrepancia en las definiciones de AS que prevalecen en la academia y las que son comunes en la industria, incluyendo en ella tanto a los proveedores de software de base como a los equipos de desarrollos de las empresas. Jan Bosch [Bos00] ha confeccionado una tabla de constrastes que seguramente podría enriquecerse con nuevas antinomias.

Academia	Industria
La arquitectura se define explícitamente	Prevalece una comprensión conceptual de la arquitectura. Las definiciones explícitas son mínimas, eventualmente mediante notaciones
La arquitectura consiste en componentes y conectores de primera clase	No hay conectores explícitos de primera clase (a veces hay soluciones *ad hoc* de *binding* en tiempo de ejecución)
Los lenguajes de descripción de arquitectura (ADLs) describen la arquitectura explícitamente y a veces la generan	Se utilizan lenguajes de programación
Los componentes reutilizables son entidades de caja negra	Los componentes son grandes piezas de software de estructuras interna compleja, no necesariamente encapsulados
Los componentes tienen interfaces con un solo punto de acceso	Las interfaces se proporcionan mediante entidades (clases en los componentes). Las entidades de interfaz no tienen diferencias explícitas de entidades que no son de interfaz
Se otorga prioridad a la funcionalidad y la verificación formal	La funcionalidad y los atributos de calidad (performance, robustez, tamaño, reusabilidad, mantenibilidad) tienen igual importancia

Tampoco conviene sobredimensionar el contraste, imaginando que la academia está desconectada de la realidad industrial; por el contrario, la mayor parte de las herramientas académicas se elaboraron en el contexto de proyectos industriales y de gobierno, casi

todos ellos de misión crítica y de gran envergadura. El número de problemas de consistencia detectados a tiempo (y que se hubiera manifestado sin guía arquitectónica) es colosal. La cuestión radica más bien en el desconocimiento que la industria tiene de la arquitectura académica y en su obstinación por llamar arquitectura a lo que sólo es diseño.

Todavía se está por elaborar una apreciación honesta y ordenada de las dificultades enfrentadas por la disciplina. Algunas de ellas son menos del orden de la calidad conceptual que fruto de los rigores del mercado: por ejemplo, la escasa penetración de los ADLs a despecho de las limitaciones expresivas de los lenguajes de modelado que compiten con ellos, o la escasa atención que la industria concede prestarle a los métodos verdaderamente formales por poco que su utilización exija el dominio de una notación complicada. Pero a pesar que los ADLs o los métodos formales no han logrado abrirse camino, la AS en su conjunto sí lo ha hecho, aunque la imagen que el público tenga de ella pueda en algún sentido sospecharse espuria, o estar contaminada por ideas difusas referidas a metodologías o herramientas no necesariamente arquitectónicas. Con toda seguridad, hay muchos elementos para discutir sobre este punto.

Relevancia de la Arquitectura de Software

Aunque todavía no se ha constituido un repositorio uniformizado de estudios de casos en base al cual se pueda extraer una conclusión responsable, la AS ha resultado instrumental en un número respetable de escenarios reduciendo costos, evitando errores, encontrando fallas, implementando sistemas de misión crítica. Cada uno de los documentos que describen lenguajes de descripción arquitectónica, por ejemplo, subraya su utilización exitosa en proyectos de gran envergadura requeridos por organizaciones de gobierno o por grandes empresas. Aún cuando aquí y allá se señalen dificultades ocasionales, nadie duda de la necesidad de una visión arquitectónica. Escribe Barry Boehm, especialista máximo en gestión de riesgo y bien conocido creador del COCOMO y del método de desarrollo en espiral:

> Si un proyecto no ha logrado una arquitectura del sistema, incluyendo su justificación, el proyecto no debe empezar el desarrollo en gran escala. Si se especifica la arquitectura como un elemento a entregar, se la puede usar a lo largo de los procesos de desarrollo y mantenimiento [Boe95].

Antes que transcribir listas de ideas que a veces coinciden y otras señalan características heterogéneas, sintetizaremos las virtudes de la AS articulando las opiniones convergentes de los expertos [CN96] [Gar00]:

1. Comunicación mutua. La AS representa un alto nivel de abstracción común que la mayoría de los participantes, si no todos, pueden usar como base para crear entendimiento mutuo, formar consenso y comunicarse entre sí. En sus mejores expresiones, la descripción arquitectónica expone las restricciones de alto nivel sobre el diseño del sistema, así como la justificación de decisiones arquitectónicas fundamentales.
2. Decisiones tempranas de diseño. La AS representa la encarnación de las decisiones de diseño más tempranas sobre un sistema, y esos vínculos tempranos tienen un peso fuera de toda proporción en su gravedad individual con respecto al desarrollo restante

del sistema, su servicio en el despliegue y su vida de mantenimiento. La arquitectura representa lo que el método SAAM una puerta de peaje: el desarrollo no puede proseguir hasta que los participantes involucrados aprueben su diseño.

3. Restricciones constructivas. Una descripción arquitectónica proporciona *blueprints* parciales para el desarrollo, indicando los componentes y las dependencias entre ellos. Por ejemplo, una vista en capas de un arquitectura documenta típicamente los límites de abstracción entre las partes, identificando las principales interfaces y estableciendo las formas en que unas partes pueden interactuar con otras.
4. Reutilización, o abstracción transferible de un sistema. La AS encarna un modelo relativamente pequeño, intelectualmente tratable, de la forma en que un sistema se estructura y sus componentes se entienden entre sí; este modelo es transferible a través de sistemas; en particular, se puede aplicar a otros sistemas que exhiben requerimientos parecidos y puede promover reutilización en gran escala. El diseño arquitectónico soporta reutilización de grandes componentes o incluso de frameworks en el que se pueden integrar componentes.
5. Evolución. La AS puede exponer las dimensiones a lo largo de las cuales puede esperarse que evolucione un sistema. Haciendo explícitas estas "paredes" perdurables, quienes mantienen un sistema pueden comprender mejor las ramificaciones de los cambios y estimar con mayor precisión los costos de las modificaciones. Esas delimitaciones ayudan también a establecer mecanismos de conexión que permiten manejar requerimientos cambiantes de interoperabilidad, prototipado y reutilización.
6. Análisis. Las descripciones arquitectónicas aportan nuevas oportunidades para el análisis, incluyendo verificaciones de consistencia del sistema, conformidad con las restricciones impuestas por un estilo, conformidad con atributos de calidad, análisis de dependencias y análisis específicos de dominio y negocios.
7. Administración. La experiencia demuestra que los proyectos exitosos consideran una arquitectura viable como un logro clave del proceso de desarrollo industrial. La evaluación crítica de una arquitectura conduce típicamente a una comprensión más clara de los requerimientos, las estrategias de implementación y los riegos potenciales.

Una de las demostraciones más elocuentes de la capacidad de la AS como herramienta de decisión de diseño y evaluación de estilos es el *tour de force* de Mary Shaw y David Garlan [SG96] en el que comparan una misma solución de indexación de palabras claves en cuatro estilos diferentes (datos compartidos, tubería y filtro, tipos abstractos de datos e invocación implícita). El estudio articula un modelo que permite estimar relaciones de dependencia, modularidad, refinamiento, reutilización, ventajas y desventajas de cada arquitectura antes de escribir una sola línea de código; demuestra también de qué manera los diferentes estilos definen tácticas específicas de descomposición funcional y establecen la pauta que habrá de seguirse en el desarrollo. Finalmente, Shaw y Garlan aplican diversas tablas de comparación de atributos, en un ejercicio de evaluación de decisiones estilísticas que se ha convertido en un modelo en su género [Pfl02: 279-283]. Si hubiera que singularizar un trabajo simple y elegante, representativo del estado de arte

de la teoría y la práctica de la AS, sin duda escogeríamos esta demostración cabal, a treinta años de distancia, de que Dijkstra tenía razón.

La AS se encuentra, reconocidamente, en una etapa aún formativa. Sus teóricos no se encuentran todavía en condiciones de asegurar (como lo hacían los Tres Amigos) que "con este libro como guía, usted podrá producir, dentro de un plan predecible y un presupuesto más razonable, el software de la más alta calidad posible": una clase de alegación que llevó a Aron Trauring a lamentar que no hubiese algo así como una FDA que fiscalice los mensajes publicitarios de los metodólogos. Por el contrario, los mejores entre los arquitectos considera que su disciplina es tentativa y que se encuentra en estado de flujo. Pero aunque lo que resta por hacer es formidable, con lo que se lleva hecho ya hay un enorme repertorio de ideas, experiencias e instrumentos que ayudan a pensar de qué manera, aquí y ahora, se pueden mejorar las prácticas.

Referencias bibliográficas

[Abd00] Aynur Abdurazik. "Suitability of the UML as an Architecture Description Language with applications to testing". Reporte ISE-TR-00-01, George Mason University. Febrero de 2000.

[Alb03] Stephen Albin. *The Art of Software Architecture: Design methods and techniques*. Nueva York, Wiley, 2003.

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

[ASR+02] Pekka Abrahamsson, Outi Salo, Jussi Ronkainen y Juhani Warsta. "Agile Software Development Methods". *VTT Publications* 478, Universidad de Oulu, Suecia, 2002.

[AW99] David Akehurst y Gill Waters. "UML deficiencies from the perspective of automatic performance model generation". *OOSPLA'99 Workshop on Rigorous Modeling and Analysis with the UML: Challenges and Limitations*, Denver, http://www.cs.kent.ac.uk/pubs/1999/899/content.pdf, Noviembre de 1999.

[BBC+00] Felix Bachmann, Len Bass, Gary Chastek, Patrick Donohoe, Fabio Peruzzi. "The Architecture Based Design Method". *Technical Report*, CMU/SEI-2000-TR-001, ESC-TR-2000-001, Enero de 2000.

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

[Bez98] Konstantin Besnozov. "Architecture of information enterprises: Problems and perspectives". http://www.beznosov.net/konstantin/doc/cis6612-paper.pdf, Abril de 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.

[Boe95] Barry Boehm. "Engineering Context (for Software Architecture)." Invited talk, *First International Workshop on Architecture for Software Systems*. Seattle, Abril de 1995.

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

[Bos00] Jan Bosch. *Design and use of Software Architecture*. Addison-Wesley, 2000.

[BR01] Jason Baragry y Karl Reed. "Why We Need a Different View of Software Architecture". *The Working IEEE/IFIP Conference on Software Architecture (WICSA)*, Amsterdam, 2001.

[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.

[Cib98] C. U. Ciborra. "Deconstructing the concept of strategic alignment". *Scandinavian Journal of Information Systems*, vol. 9, pp. 67-82, 1998.

[CLC03] David Cohen, Mikael Lindvall y Patricia Costa. "Agile Software Development. A DACS State-of-the-Art Report", *DACS Report*, The University of Maryland, College Park, 2003.

[Cle96a] Paul Clements. "A Survey of Architecture Description Languages". *Proceedings of the International Workshop on Software Specification and Design*, Alemania, 1996.

[Cle96b] Paul Clements. "Coming attractions in Software Architecture". *Technical Report*, CMU/SEI-96-TR-008, ESC-TR-96-008, Enero de 1996.

[CBK+95] Paul Clements, Len Bass, F. Kazman y Gregory Abowd. "Predicting Software Quality by Architecture-Level Evaluation," 485- 498. *Proceedings, Fifth International Conference on Software Quality*. Austin, 23 al 26 de Octubre de 1995, pp. 485-498. Austin, American Society for Quality Control, Software Division, 1995.

[CN96] Paul Clements y Linda Northrop. "Software architecture: An executive overview". *Technical Report*, CMU/SEI-96-TR-003, ESC-TR-96-003. Febrero de 1996.

[DD94] Peter Denning y Pamela Dargan. "A discipline of software architecture". *ACM Interactions*, 1(1), pp. 55-65, Enero de 1994.

[DDT99] Serge Demeyer, Stéphane Ducasse y Sander Tichelaar. "Why FAMIX and not UML?: UML Shortcomings for coping with round-trip engineering". *UML'99 Conference Proceedings*, LNCS Series, Springer Verlag, 1999.

[Dij68a] Edsger Dijkstra. "The Structure of the THE Multiprogramming system." *Communications of the ACM,* 26(1), pp. 49-52, Enero de 1983.

[Dij68b] Edsger Dijkstra. "GO-TO statement considered harmful". *ACM Communications of the ACM*, 11(3), pp. 147-148, Marzo de 1968.

[DIR99] N. Dunlop, J. Indulska y K. A. Raymond. "CORBA and RM-ODP: Parallel or divergent?", *Distributed Systems Engineering*, 6, pp. 82-91.

[DK76] Frank DeRemer y Hans Kron, "Programming-in-the-large versus programming-in-the-small". *IEEE Transaction in Software Engineering*, 2, pp. 80-86, 1976.

[Dou00] Bruce Powel Douglas. "UML – The new language for real-timed embedded systems". http://wooddes.intranet.gr/papers/Douglass.pdf, 2000.

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

[Fow01] Martin Fowler. "Is design dead?". *XP2000 Proceedings*, http://www.martinfowler.com/articles/designDead.html, 2001.

[GAD+02] Yann-Gaël Guéhéneuc, Hervé Albin-Amiot, Rémi Douence y Pierre Cointe. "Bridging the gap between modeling and programming languages". Reporte técnico, Object Technology International, Julio de 2002.

[Gar95] David Garlan. "Research Directions in Software Architecture. " ACM Computing Surveys 27, 2, pp. 257-261, Junio de 1995.

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

[Gars/f] David Garlan. "Software architectures". Presentación en transparencias, http://www.sti.uniurb.it/events/sfm03sa/slides/garlan-B.ppt.

[Gli00] Martin Glinz. "Problems and deficiencies of UML as a requirements specification language". En *Proceedings of the 10th International Workshop of Software Specification and Design* (IWSSD-10), San Diego, noviembre de 2000, pp. 11-22.

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

[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.

[HHR+96] Pat Hall, Fiona Hovenden, Janet Rachel y Hugh Robinson. "Postmodern Software Development". *MCS Technical Reports*, 1996.

[Hig01] Jim Highsmith. "The great methodologies debate. Part I". *Cutter IT Journal*, 14(12), Diciembre de 2001.

[HNS99] Christine Hofmeister, Robert Nord y Dilip Soni. "Describing Software Architecture with UML". En *Proceedings of the First Working IFIP Conference on Software Architecture*, IEEE Computer Society Press, pp. 145-160, San Antonio, Febrero de 1999.

[HNS00] Christine Hofmeister, Robert Nord y Dilip Soni. *Applied software architecture*, Reading, Addison Wesley, 2000.

[Hock00] Dee Hock. *Birth of the chaordic age*. San Francisco, Berrett-Koehler.

[KAB+96] Rick Kazman, Gregory Abowd, Len Bass y Paul Clements. "Scenario-Based Analysis of Software Architecture". *IEEE Software*, 13(6), pp. 47-55, Noviembre de 1996.

[Keen91] P. G. W. Keen. "Relevance anr rigor in information systems research", en H.-E. Nissen, H. K. Klein y R. Hisrchheim (eds), *Information Systems Research: Contemporary approaches and emergent traditions*, Elsevier, 1991.

[KNK03] Rick Kazman, Robert Nord, Mark Klein. "A life-cycle view of architecture analysis and design methods". *Technical Report*, CMU/SEI-2002-TN026, Setiembre de 2003.

[Kros/f] John Krogstie. "UML, a good basis for the development of models of high quality?". Andersen Consulting Noruega & IDI, NTNU, http://dataforeningen.no/ostlandet/metoder/krogstie.pdf, sin fecha.

[Kru95] Philippe Kruchten. "The 4+1 View Model of Architecture." *IEEE Software* 12(6), pp. 42-50, Noviembre de 1995.

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

[MB02] Ruth Malan y Dana Bredemeyer. "Less is more with Minimalist Architecture". *IEEE IT Professional*, Setiembre-Octubre de 2002.

[McC96] Steve McConnell. "Missing in Action: Information hiding". *IEEE Software*, 13(2), Marzo de 1996.

[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.

[MS03] Geoffrey Lory, Derick Campbell, Allison Robin, Gaile Simmons y Patricia Rytkonen. "Microsoft Solutions Framework Version 3.0 overview". http://www.microsoft.com/technet/itsolutions/techguide/msf/msfovrvw.mspx, Junio de 2003.

[NATO76] I. P. Sharp. Comentario en discusión sobre teoría y práctica de la ingeniería de software en conference de NATO Science Committee, Roma, 27 al 31 de Octubre de 1969. *Software Engineering Concepts and Techniques: Proceedings of the NATO conferences*. J. N. Bruxton y B. Randall (eds.), Petrochelli/Charter, 1976.

[Par72] David Parnas. "On the Criteria for Decomposing Systems into Modules." *Communications of the ACM* 15(12), pp. 1053-1058, Diciembre de 1972.

[Par74] David Parnas. "On a 'Buzzword': Hierarchical Structure", *Programming Methodology*, pp. 335-342. Berlin, Springer-Verlag, 1978.

[Par76] David Parnas. "On the Design and Development of Program Families." *IEEE Transactions on Software Engineering* SE-2, 1, pp. 1-9, Marzo de 1976.

[Per97] Dewayne Perry. “Software Architecture and its relevance for Software Engineering”. *Coord’97*, 1997.

[Pfl02] Shari Lawrence Pfleeger. *Ingeniería de Software: Teoría y Práctica.* Madrid, Prentice-Hall.

[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.

[RJB00] Jamers Rumbaugh, Ivar Jacobson, Grady Booch. *El lenguaje unificado de modelado. Manual de Referencia*. Madrid, Addison-Wesley, 2000.

[Ross77] Douglas Ross. “Structured analysus (SA): A language for communicating ideas”. *IEEE Transactions on Software Engineering*, SE-3(1), enero de 1977.

[Sch00] Klaus-Dieter Schewe. “UML: A modern dinosaur? – A critical analysis of the Unified Modelling Language”. En H. Kangassalo, H. Jaakkola y E. Kawaguchi (eds.), *Proceedings of the 10th European-Japanese Conference on Information Modelling and Knowledge Bases*, Saariselk, Finlandia, 2000.

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

[Shaw84] Mary Shaw. “Abstraction Techniques in Modern Programming Languages”. *IEEE Software*, Octubre, pp. 10-26, 1984.

[Shaw89] Mary Shaw. “Large Scale Systems Require Higher- Level Abstraction”. *Proceedings of Fifth International Workshop on Software Specification and Design*, IEEE Computer Society, pp. 143-146, 1989.

[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.

[Shaw01] Mary Shaw. “The coming-of-age of Software Architecture research”. *Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001)*, 2001.

[Spo71] C. R. Spooner. “A Software Architecture for the 70's: Part I - The General Approach.” *Software - Practice and Experience,* 1 (Enero-Marzo), pp. 5-37, 1971.

[StöS/f] Harald Störrle. “Turning UML subsystems into architectural units”. Reporte, Ludwig-Maximilians-Universität München, http://www.pst.informatik.uni-muenchen.de/personen/stoerrle/Veroeffentlichungen/PosPaperICSEFormat.pdf, sin fecha.

[Tem01] Theodor Tempelmeier. “Comments on Design Patterns for Embedded and Real-Time Systems”. En: A. Schürr (ed.): *OMER-2 (“Object-Oriented Modeling of Embedded Realtime systems”) Workshop Proceedings*. Mayo 9-12, 2001, Herrsching, Alemania. Bericht Nr. 2001-03, Universität der Bundeswehr München, Fakultät für Informatik, Mayo de 2001.

[Tem99] Theodor Tempelmeier. “UML is great for Embedded Systems – Isn’t it?” En: P. Hofmann, A. Schürr (eds.): *OMER (“Object-Oriented Modeling of Embedded Realtime systems”) Workshop Proceedings*. Mayo 28-29, 1999, Herrsching (Ammersee), Alemania. Bericht Nr. 1999-01, Universität der Bundeswehr München, Fakultät für Informatik, Mayo de 1999.

[TMA+S/f] Richard Taylor, Nenad Medvidovic, Kennet Anderson, James Whitehead Jr, Jason Robbins, Kari Nies, Peyman Oreizy y Deborah Dubrow. “A component- and message-based architectural style for GUI software”. Reporte para el proyecto F30602-94-C-0218, Advanced Research Projects Agency, sin fecha.

[Tra02] Aron Trauring. “Software methodologies: The battle of the Gurus”. *Info-Tech White Papers*, 2002.

[Wir71] Niklaus Wirth. “Program development by stepwise refinement”, C*ommunications of the ACM*, 14(4), pp. 221-227, Abril de 1971.

[WL97] Sharon White y Cuauhtémoc Lemus-Olalde. “The software architecture process”. http://nas.cl.uh.edu/whites/webpapers.dir/ETCE97pap.pdf, 1997.

[WWI99] WWISA. “Philosophy”. Worldwide Institute of Software Architects, http://www.wwisa.org, 1999.

[Zac87] John A. Zachman, “A Framework for Information Systems Architecture”, *IBM Systems Journal*, 26(3), , 1987.

[ZHH+99] Guoqiang Zhong, Babak Hodjat, Tarek Helmy y Makoto Amamiya. “Software agent evolution in adaptive agent oriented software architecture”. *IWPSE’99 Proceedings*, 1999.

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].

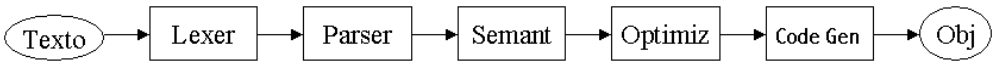


Fig. 1 - Compilador en tubería-filtro

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.

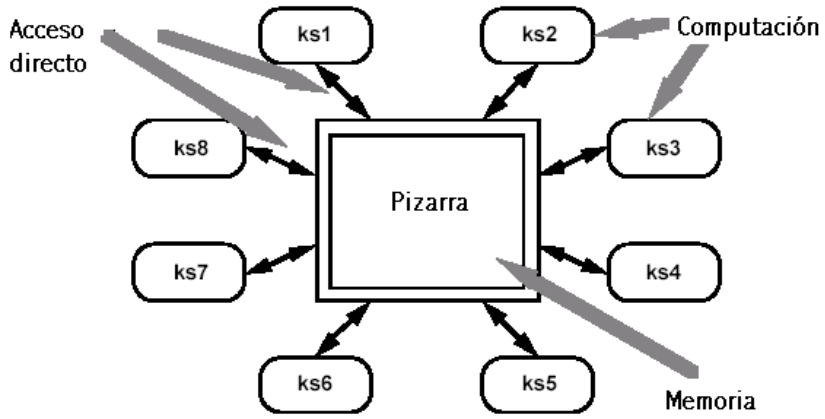


Fig. 2 - Pizarra [basado en GS94]

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.

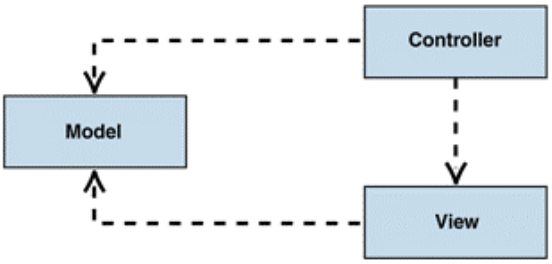


Fig. 3 - Model-View-Controller [según MS03a]

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.

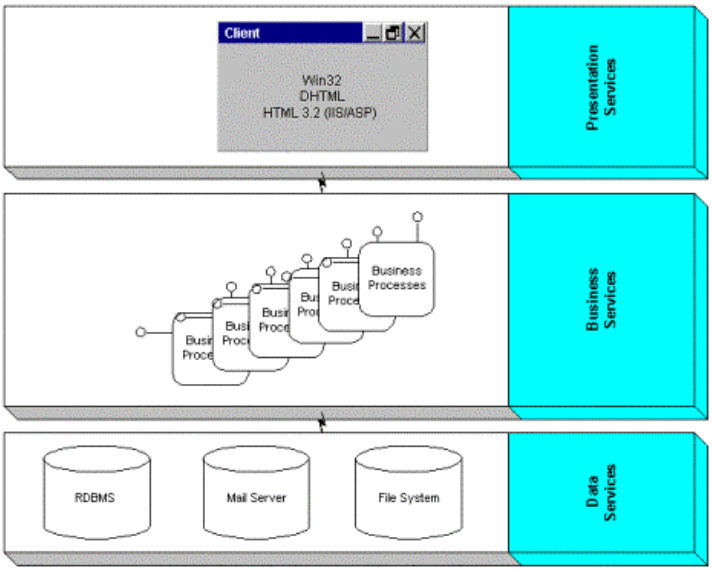


Fig. 4 - Arquitectura en 3 capas con "cliente flaco" en Windows DNA

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.

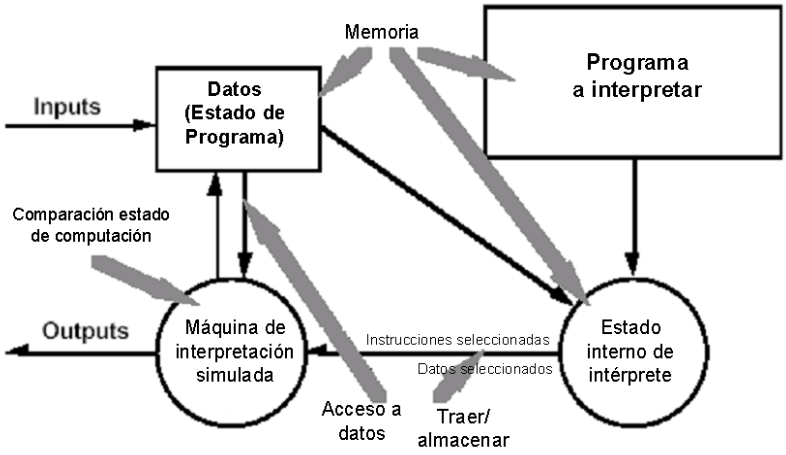


Fig. 5 - Intérprete [basado en GS94]

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.

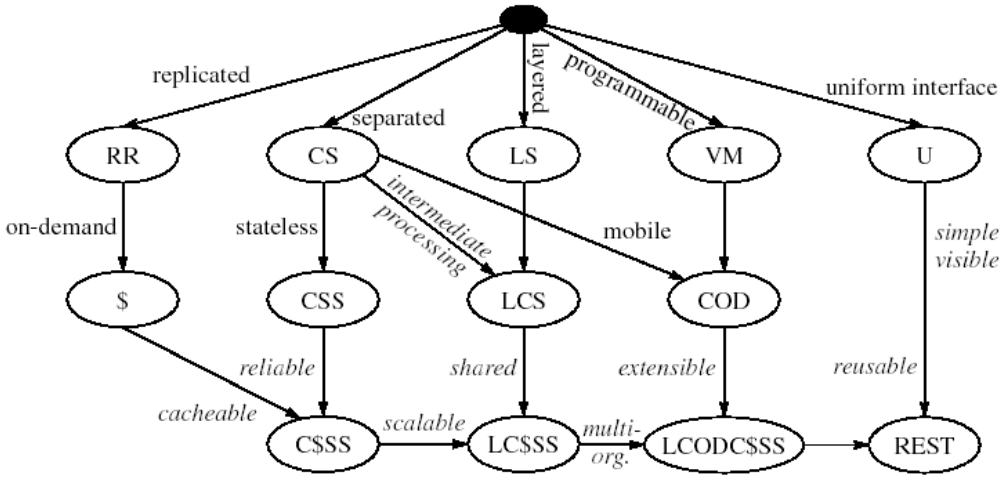


Fig. 6 - Composición de REST [FT02]

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.

Lenguajes de Descripción de Arquitectura (ADL)

Versión 1.0 – Marzo de 2004

Carlos Reynoso – Nicolás Kicillof
UNIVERSIDAD DE BUENOS AIRES

Introducción

Una vez que el arquitecto de software, tras conocer el requerimiento, se decide a delinear su estrategia y a articular los patrones que se le ofrecen hoy en profusión, se supone que debería expresar las características de su sistema, o en otras palabras, modelarlo, aplicando una convención gráfica o algún lenguaje avanzado de alto nivel de abstracción.

A esta altura del desarrollo de la arquitectura de software, podría pensarse que hay abundancia de herramientas de modelado que facilitan la especificación de desarrollos basados en principios arquitectónicos, que dichas herramientas han sido consensuadas y estandarizadas hace tiempo y que son de propósito general, adaptables a soluciones de cualquier mercado vertical y a cualquier estilo arquitectónico. La creencia generalizada sostendría que modelar arquitectónicamente un sistema se asemeja al trabajo de articular un modelo en ambientes ricos en prestaciones gráficas, como es el caso del modelado de tipo CASE o UML, y que el arquitecto puede analizar visualmente el sistema sin sufrir el aprendizaje de una sintaxis especializada. También podría pensarse que los instrumentos incluyen la posibilidad de diseñar modelos correspondientes a proyectos basados en tecnología de internet, Web services o soluciones de integración de plataformas heterogéneas, y que, una vez trazado el modelo, el siguiente paso en el ciclo de vida de la solución se produzca con naturalidad y esté servido por ténicas bien definidas. Como se verá en este documento, la situación es otra, dista de ser clara y es harto más compleja.

En primer lugar, el escenario de los web services ha forzado la definición de un estilo de arquitectura que no estaba contemplado a la escala debida en el inventario canónico de tuberías y filtros, repositorio, eventos, capas, llamada y retorno/OOP y máquinas virtuales. El contexto de situación, como lo reveló la impactante tesis de Roy Fielding sobre REST [Fie00], es hoy en día bastante distinto al de los años de surgimiento de los estilos y los lenguajes de descripción de arquitecturas (en adelante, ADLs). Los ADLs se utilizan, además, para satisfacer requerimientos descriptivos de alto nivel de abstracción que las herramientas basadas en objeto en general y UML en particular no cumplen satisfactoriamente. Entre las comunidades consagradas al modelado OO y la que patrocina o frecuenta los ADLs (así como entre las que se inclinan por el concepto de estilos arquitectónicos y las que se trabajan en función de patrones) existen relaciones complejas que algunas veces son de complementariedad y otras de antagonismo.

Aunque algunos arquitectos influyentes, como Jørgen Thelin, alegan que el período de gloria de la OOP podría estar acercándose a su fin, el hecho concreto es que el modelado orientado a objetos de sistemas basados en componentes posee ciertos número de rasgos muy convenientes a la hora de diseñar o al menos describir un sistema. En primer lugar, las notaciones de objeto resultan familiares a un gran número de ingenieros de software.

Proporcionan un mapeo directo entre un modelo y una implementación y se dispone de un repertorio nutrido de herramientas comerciales para trabajar con ellas; implementan además métodos bien definidos para desarrollar sistemas a partir de un conjunto de requerimientos.

Pero la descripción de sistemas basados en componentes presenta también limitaciones serias, que no son de detalle sino más bien estructurales. En primer lugar, sólo proporcionan una única forma de interconexión primitiva: la invocación de método. Esto hace difícil modelar formas de interacción más ricas o diferentes. Ya en los días del OMT, Rumbaugh y otros admitían cándidamente que "el *target* de implementación más natural para un diseño orientado a objetos es un lenguaje orientado a objetos" [RBP+91:296]. En segundo orden, el soporte de los modelos OO para las descripciones jerárquicas es, en el mejor de los casos, débil. Tampoco se soporta la definición de familias de sistemas o estilos; no hay recurso sintáctico alguno para caracterizar clases de sistemas en términos de las restricciones de diseño que debería observar cada miembro de la familia. En última instancia, no brindan soporte formal para caracterizar y analizar propiedades no funcionales, lo que hace difícil razonar sobre propiedades críticas del sistema, tales como performance y robustez [GMW00]. A la luz de estas observaciones y de otras que saldrán a la luz en el capítulo sobre UML de este mismo documento, está claro cuál es el nicho vacante que los ADLs vinieron a ocupar.

Según señala Mary Shaw [Shaw94] y Neno Medvidovic [Med96] en sus revisiones de los lenguajes de descripción arquitectónica, el uso que se da en la práctica y en el discurso a conceptos tales como arquitectura de software o estilos suele ser informal y ad hoc. En la práctica industrial, las configuraciones arquitectónicas se suelen describir por medio de diagramas de cajas y líneas, con algunos añadidos diacríticos; los entornos para esos diagramas suelen ser de espléndida calidad gráfica, comunican con relativa efectividad la estructura del sistema y siempre brindan algún control de consistencia respecto de qué clase de elemento se puede conectar con cuál otra; pero a la larga proporcionan escasa información sobre la computación efectiva representada por las cajas, las interfaces expuestas por los componentes o la naturaleza de sus computaciones.

En la década de 1990 y en lo que va del siglo XXI, sin embargo, se han materializado diversas propuestas para describir y razonar en términos de arquitectura de software; muchas de ellas han asumido la forma de ADLs. Estos suministran construcciones para especificar abstracciones arquitectónicas y mecanismos para descomponer un sistema en componentes y conectores, especificando de qué manera estos elementos se combinan para formar configuraciones y definiendo familias de arquitecturas o estilos. Contando con un ADL, un arquitecto puede razonar sobre las propiedades del sistema con precisión, pero a un nivel de abstracción convenientemente genérico. Algunas de esas propiedades podrían ser, por ejemplo, protocolos de interacción, anchos de banda y latencia, localización del almacenamiento, conformidad con estándares arquitectónicos y previsiones de evolución ulterior del sistema.

Hasta la publicación de las sistematizaciones de Shaw y Garlan, Kogut y Clements o Nenan Medvidovic existía relativamente poco consenso respecto a qué es un ADL, cuáles lenguajes de modelado califican como tales y cuáles no, qué aspectos de la arquitectura se deben modelar con un ADL y cuáles de éstos son más adecuados para modelar qué

clase de arquitectura o estilo. También existía (y aún subsiste) cierta ambigüedad a propósito de la diferencia entre ADLs, especificaciones formales (como CHAM, cálculo Л o Z), lenguajes de interconexión de módulos (MIL) como MIL75 o Intercol, lenguajes de modelado de diseño (UML), herramientas de CASE y hasta determinados lenguajes con reconocidas capacidades modelizadoras, como es el caso de CODE, un lenguaje de programación paralela que en opinión de Paul Clements podría tipificar como un ADL satisfactorio.

La definición de ADL que habrá de aplicarse en lo sucesivo es la de un lenguaje descriptivo de modelado que se focaliza en la estructura de alto nivel de la aplicación antes que en los detalles de implementación de sus módulos concretos [Ves93]. Los ADL que existen actualmente en la industria rondan la cifra de veinticinco; cuatro o cinco de ellos han experimentado dificultades en su desarrollo o no se han impuesto en el mercado de las herramientas de arquitectura. Todos ellos oscilan entre constituirse como ambientes altamente intuitivos con eventuales interfaces gráficas o presentarse como sistemas rigurosamente formales con profusión de notación simbólica, en la que cada entidad responde a algún teorema. Alrededor de algunos ADLs se ha establecido una constelación de herramientas de análisis, verificadores de modelos, aplicaciones de generación de código, *parsers*, soportes de *runtime*, etcétera. El campo es complejo, y de la decisión que se tome puede depender el éxito o el fracaso de un proyecto.

No existe hasta hoy una definición consensuada y unívoca de ADL, pero comúnmente se acepta que un ADL debe proporcionar un modelo explícito de componentes, conectores y sus respectivas configuraciones. Se estima deseable, además, que un ADL suministre soporte de herramientas para el desarrollo de soluciones basadas en arquitectura y su posterior evolución.

La literatura referida a los ADL es ya de magnitud respetable. Aquí hemos tomado en consideración los estudios previos de Kogut y Clements [KC94, KC95, Cle96a], Vestal [Ves93], Luckham y Vera [LV95], Shaw, DeLine y otros [SDK+95], Shaw y Garlan [SG94, SG95] y Medvidovic [Med96], así como la documentación de cada uno de los lenguajes. Los principales ADLs que se revisarán en este estudio son ACME, Armani, Aesop, ArTek, C2, Darwin, Jacal, LILEANNA, MetaH/AADL, Rapide, UniCon, Weaves, Wright y xADL. Sobre unos cuantos no incluidos en esa lista no disponemos de demasiada información ni de experiencia concreta: Adage, ASDL, Avionics ADL, FR, Gestalt, Koala, Mae, PSDL/CAPS, QAD, SAM; serán por ende dejados al margen. A pesar de las afirmaciones de Clements [Cle95], no cabe considerar que UNAS/SALE, un producto comercial para gestión del ciclo de vida de un desarrollo, constituya un ADL. No habrá de tratarse aquí tampoco, en general, de ADLs que están en proceso de formulación, como algunos estándares emergentes o los Domain Specific Languages (DSL) que Microsoft se encuentra elaborando para su inclusión en la próxima encarnación de VisualStudio ("Whidbey"). Estos serán más bien lenguajes de patrones, a los que el ADL deberá tomar en cuenta como a un elemento de juicio adicional.

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

estereotipos históricos (sobre todo en el caso de UML), no se prestan a las mismas tareas que los lenguajes descriptivos de arquitectura, diseñados específicamente para esa finalidad. Aparte de CODE, que es claramente un lenguaje de programación, se excluirá aquí entonces toda referencia a Modechart y PSDL (lenguajes de especificación o prototipado de sistemas de tiempo real), LOTOS (un lenguaje de especificación formal, asociado a entornos de diseño como CADP), ROOM (un lenguaje de modelado de objetos de tiempo real), Demeter (una estrategia de diseño y programación orientada a objetos), Resolve (una técnica matemáticamente fundada para desarrollo de componentes re-utilizables), Larch y Z (lenguajes formales de especificación) por más que todos ellos hayan sido asimilados a ADLs por algunos autores en más de una ocasión. Se hará una excepción con UML, sin embargo. A pesar de no calificar en absoluto como ADL, se ha probado que UML puede utilizarse no tanto como un ADL por derecho propio, sino como metalenguaje para simular otros ADLs, y en particular C2 y Wright [RMR98]. Otra excepción concierne a CHAM, por razones que se verán más adelante.

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

El objetivo del presente estudio es dar cuenta del estado de arte en el desarrollo de los ADLs en el momento actual (2004), varios años después de realizados los *surveys* más reputados, los cuales son a su vez anteriores a la explosión de los web services, SOAP, el modelo REST y las arquitecturas basadas en servicios. También se examinará detalladamente la disponibilidad de diversos ADLs y herramientas concomitantes en ambientes Windows, la forma en que los ADLs engranan con la estrategia general de Microsoft en materia de arquitectura y su relación con el campo de los patrones arquitectónicos.

En lugar de proceder a un análisis de rasgos, examinando la forma en que los distintos ADLs satisfacen determinados requerimientos, aquí se ha optado por describir cada ADL

en función de variables que a veces son las mismas en todos los casos y en otras oportunidades son específicas de la estructura de un ADL en particular. Nos ha parecido de interés subrayar el modelo computacional en que los ADLs se basan, a fin de matizar la presentación de un campo que hasta hace poco se estimaba coto exclusivo de los paradigmas orientados a objeto. Mientras éstos se están elaborando mayormente en organismos de estandarización, casi todos los ADLs se originan en ámbitos universitarios. Crucial en el tratamiento que aquí se dará a los ADLs es la capacidad de trabajar en términos de estilos, que se estudiarán en un documento separado [Rey04]. Un estilo define un conjunto de propiedades compartidas por las configuraciones que son miembros de él. Estas propiedades pueden incluir un vocabulario común y restricciones en la forma en que ese vocabulario puede ser utilizado en dichas configuraciones. Tengan a no los distintos ADLs soportes de estilo, patrones o lo que fuese, no interesa aquí tanto describir o asignar puntaje a productos o paquetes que por otro lado son cambiantes, sino caracterizar problemas, ideas, paradigmas y escenarios de modelado.

Criterios de definición de un ADL

Los ADLs se remontan a los lenguajes de interconexión de módulos (MIL) de la década de 1970, pero se han comenzado a desarrollar con su denominación actual a partir de 1992 o 1993, poco después de fundada la propia arquitectura de software como especialidad profesional. La definición más simple es la de Tracz [Wolf97] que define un ADL como una entidad consistente en cuatro “Cs”: componentes, conectores, configuraciones y restricciones (*constraints*). Una de las definiciones más tempranas es la de Vestal [Ves93] quien sostiene que un ADL debe modelar o soportar los siguientes conceptos:

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

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

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

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

Tomando como parámetro de referencia a UniCon, Shaw y otros [SDK+95] alegan que un ADL debe exhibir:

- Capacidad para modelar componentes con aserciones de propiedades, interfaces e implementaciones
- Capacidad de modelar conectores con protocolos, aserción de propiedades e implementaciones
- Abstracción y encapsulamiento
- Tipos y verificación de tipos
- Capacidad para integrar herramientas de análisis

Otros autores, como Shaw y Garlan [SG94] estipulan que en los ADLs los conectores sean tratados explícitamente como entidades de primera clase (lo que dejaría al margen de la lista a dos de ellos al menos) y han afirmado que un ADL genuino tiene que proporcionar propiedades de composición, abstracción, reusabilidad, configuración, heterogeneidad y análisis, lo que excluiría a todos los lenguajes convencionales de programación y a los MIL.

La especificación más completa y sutil (en tanto que se presenta como provisional, lo que es propio de un campo que recién se está definiendo) es la de Medvidovic [Med96]:

- Componentes
  Interfaz
  Tipos
  Semántica
  Restricciones (*constraints*)
  Evolución
  Propiedades no funcionales
- Conectores
  Interfaz
  Tipos
  Semántica
  Restricciones
  Evolución
  Propiedades no funcionales
- Configuraciones arquitectónicas
  Comprensibilidad
  Composicionalidad
  Heterogeneidad
  Restricciones
  Refinamiento y trazabilidad
  Escalabilidad
  Evolución
  Dinamismo

Propiedades no funcionales
- Soporte de herramientas
Especificación activa
Múltiples vistas
Análisis
Refinamiento
Generación de código
Dinamismo

Nótese, en todo caso, que no se trata tanto de aspectos definitorios del concepto de ADL, sino de criterios para la evaluación de los ADLs existentes, o sea de un marco de referencia para la clasificación y comparación de los ADLs.

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

- Componentes: Representan los elementos computacionales primarios de un sistema. Intuitivamente, corresponden a las cajas de las descripciones de caja-y-línea de las arquitecturas de software. Ejemplos típicos serían clientes, servidores, filtros, objetos, pizarras y bases de datos. En la mayoría de los ADLs los componentes pueden exponer varias interfaces, las cuales definen puntos de interacción entre un componente y su entorno.
- Conectores. Representan interacciones entre componentes. Corresponden a las líneas de las descripciones de caja-y-línea. Ejemplos típicos podrían ser tuberías (*pipes*), llamadas a procedimientos, *broadcast* de eventos, protocolos cliente-servidor, o conexiones entre una aplicación y un servidor de base de datos. Los conectores también tienen una especie de interfaz que define los roles entre los componentes participantes en la interacción.
- Configuraciones o sistemas. Se constituyen como grafos de componentes y conectores. En los ADLs más avanzados la topología del sistema se define independientemente de los componentes y conectores que lo conforman. Los sistemas también pueden ser jerárquicos: componentes y conectores pueden subsumir la representación de lo que en realidad son complejos subsistemas.
- Propiedades. Representan información semántica sobre un sistema más allá de su estructura. Distintos ADLs ponen énfasis en diferentes clases de propiedades, pero todos tienen alguna forma de definir propiedades no funcionales, o pueden admitir herramientas complementarias para analizarlas y determinar, por ejemplo, el *throughput* y la latencia probables, o cuestiones de seguridad, escalabilidad, dependencia de bibliotecas o servicios específicos, configuraciones mínimas de *hardware* y tolerancia a fallas.
- Restricciones. Representan condiciones de diseño que deben acatarse incluso en el caso que el sistema evolucione en el tiempo. Restricciones típicas serían restricciones en los valores posibles de propiedades o en las configuraciones topológicas

admisibles. Por ejemplo, el número de clientes que se puede conectar simultáneamente a un servicio.

- Estilos. Representan familias de sistemas, un vocabulario de tipos de elementos de diseño y de reglas para componerlos. Ejemplos clásicos serían las arquitecturas de flujo de datos basados en grafos de tuberías (*pipes*) y filtros, las arquitecturas de pizarras basadas en un espacio de datos compartido, o los sistemas en capas. Algunos estilos prescriben un *framework*, un estándar de integración de componentes, patrones arquitectónicos o como se lo quiera llamar.
- Evolución. Los ADLs deberían soportar procesos de evolución permitiendo derivar subtipos a partir de los componentes e implementando refinamiento de sus rasgos. Sólo unos pocos lo hacen efectivamente, dependiendo para ello de lenguajes que ya no son los de diseño arquitectónico sino los de programación.
- Propiedades no funcionales. La especificación de estas propiedades es necesaria para simular la conducta de *runtime*, analizar la conducta de los componentes, imponer restricciones, mapear implementaciones sobre procesadores determinados, etcétera.

Lenguajes

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

Acme - Armani

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

El proyecto Acme comenzó a principios de 1995 en la Escuela de Ciencias de la Computación de la Universidad Carnegie Mellon. Hoy este proyecto se organiza en dos grandes grupos, que son el lenguaje Acme propiamente dicho y el Acme Tool Developer’s Library (AcmeLib). De Acme se deriva, en gran parte, el ulterior estándar emergente ADML. Fundamental en el desarrollo de Acme ha sido el trabajado de destacados arquitectos y sistematizadores del campo, entre los cuales el más conocido es sin duda David Garlan, uno de los teóricos de arquitectura de software más activos en la década de 1990. La bibliografía relevante para profundizar en Acme es el reporte de R. T. Monroe [Mon98] y el artículo de Garlan, Monroe y Wile [GMW00].

Objetivo principal – La motivación fundamental de Acme es el intercambio entre arquitecturas e integración de ADLs. Garlan considera que Acme es un lenguaje de

descripción arquitectónica de segunda generación; podría decirse que es de segundo orden: un metalenguaje, una *lingua franca* para el entendimiento de dos o más ADLs, incluido Acme mismo. Con el tiempo, sin embargo, la dimensión metalingüística de Acme fue perdiendo prioridad y los desarrollos actuales profundizan su capacidad intrínseca como ADL puro.

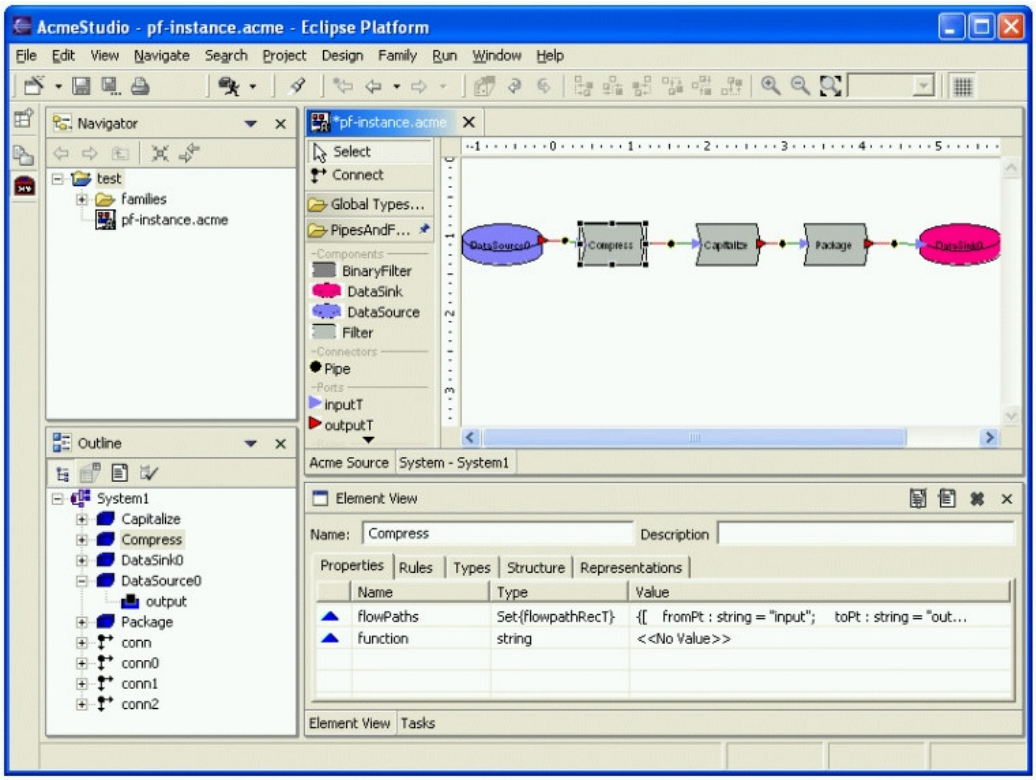


Fig. 1 - Ambiente de edición de AcmeStudio con diagrama de tubería y filtros

Sitio de referencia – El sitio de cabecera depende de la School of Computer Science de la Carnegie Mellon University en Pittsburgh. AcmeWeb se encuentra en http://www-2.cs.-cmu.edu/~acme/.

Acme soporta la definición de cuatro tipos de arquitectura: la estructura (organización de un sistema en sus partes constituyentes); las propiedades de interés (información que permite razonar sobre el comportamiento local o global, tanto funcional como no funcional); las restricciones (lineamientos sobre la posibilidad del cambio en el tiempo); los tipos y estilos. La estructura se define utilizando siete tipos de entidades: componentes, conectores, sistemas, puertos, roles, representaciones y rep-mapas (mapas de representación).

Componentes – Representan elementos computacionales y almacenamientos de un sistema. Como se verá en el ejemplo, un componente se define siempre dentro de una familia.

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

Conectores – En su ejemplar estudio de los ADLs existentes, Medvidovic (1996) llama a los lenguajes que modelan sus conectores como entidades de primera clase lenguajes de configuración explícitos, en oposición a los lenguajes de configuración *in-line*. Acme pertenece a la primera clase, igual que Wright y UniCon. Los conectores representan interacciones entre componentes. Los conectores también tienen interfaces que están definidas por un conjunto de roles. Los conectores binarios son los más sencillos: el invocador y el invocado de un conector RPC, la lectura y la escritura de un conector de tubería, el remitente y el receptor de un conector de paso de mensajes.

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

Estilos – Acme posee manejo intensivo de familias o estilos. Esta capacidad está construida naturalmente como una jerarquía de propiedades correspondientes a tipos. Acme considera, en efecto, tres clase de tipos: tipos de propiedades, tipos estructurales y estilos. Así como los tipos estructurales representan conjuntos de elementos estructurales, una familia o estilo representa un conjunto de sistemas. Una familia Acme se define especificando tres elementos de juicio: un conjunto de tipos de propiedades y tipos estructurales, un conjunto de restricciones y una estructura por defecto, que prescribe el conjunto mínimo de instancias que debe aparecer en cualquier sistema de la familia. El uso del término "familia" con preferencia a "estilo" recupera una idea de uno de los precursores tempranos de la arquitectura de software, David Parnas [Par76].

El siguiente ejemplo define una familia o estilo de la clase más simple, tubería y filtros. Este estilo está relacionado con las arquitecturas de flujo de datos. Por ser el estilo más primitivo y de definición más escueta, se ha escogido ejemplificar con él la sintaxis de Acme y de todos los ADLs que se revisarán después. El estilo es característico no sólo de arquitecturas de tipo UNIX, sino que es ampliamente usado en compiladores, flujos de datos, tratamiento de XML con SAX, procesamiento de señales y programación funcional, así como en los mecanismos de procesamiento de consultas de algunos servidores de bases de datos. Es el único estilo, por otra parte, que puede ser implementado explícita o implícitamente en todos los ADLs [Cle96]. En Acme la sintaxis para definirlo sería:

```
// Una familia Acme incluye un conjunto de tipos de
// componente,conector, puerto (port) y rol que definen el vocabulario
// propio del estilo.
Family PipeFilterFam = {
  // Declara tipos de componente.
  // Una definición de tipo de componente en Acme permite establecer
  // la estructura requerida por el tipo. Esta estructura
  // se define mediante la misma sintaxis que la instancia
  // de un componente.
  Component Type FilterT = {
        // Todos los filtros definen por lo menos dos puertos
         Ports { stdin; stdout; };
        Property throughput : int;
  };

  // Extiende el tipo básico de filtro con una subclase (herencia)
  // Las instancia de WindowsFilterT tendrán todas las propiedades y
```

```
  // puertos de las instancias de FilterT, más un puerto stderr
  // y una propiedad implementationFile.

  Component Type WindowsFilterT extends FilterT with {
        Port stderr;
        Property implementationFile : String;
  };
  // Declara el tipo de conector de tubería. Igual que los
  // tipos de componente, un tipo de conector también describe
  // la estructura requerida.
  Connector Type PipeT = {
        Roles { source; sink; };
        Property bufferSize : int;
  };
   // Declara algunos tipos de propiedad que pueden usarse en
   // sistemas del estilo PipeFilterFam
   Property Type StringMsgFormatT = Record [ size:int; msg:String; ];
   Property Type TasksT = enum {sort, transform, split, merge};
};
```

La biblioteca AcmeLib define un conjunto de clases para manipular representaciones arquitectónicas Acme en cualquier aplicación. Su código se encuentra disponible tanto en C++ como en Java, y puede ser invocada por lo tanto desde cualquier lenguaje la plataforma clásica de Microsoft o desde el framework de .NET. Es posible entonces implementar funcionalidad conforme a Acme en forma directa en cualquier programa, o llegado el caso (mediante *wrapping*) exponer Acme como web service.

Interfaz gráfica – La versión actual de Acme soporta una variedad de *front-ends* de carácter gráfico, de los cuales he experimentado con tres. El ambiente primario, llamado AcmeStudio, es un entorno gráfico basado en Windows, susceptible de ser configurado para soportar visualizaciones específicas de estilos e invocación de herramientas auxiliares. Un segundo entorno, llamado Armani, utiliza Microsoft Visio como front-end gráfico y un back-end Java, que con alguna alquimia puede ser Microsoft Visual J++ o incluso Visual J# de .NET. Armani no es un entorno redundante sino, por detrás de la fachada de Visio, un lenguaje de restricción que extiende Acme basándose en lógica de predicados de primer orden, y que es por tanto útil para definir invariantes y heurísticas de estilos. Un tercer ambiente, más experimental, diseñado en ISI, utiliza sorprendentemente el editor de PowerPoint para manipulación gráfica acoplado con analizadores que reaccionan a cambios de una representación DCOM de los elementos arquitectónicos y de sus propiedades asociadas [GB99].

Generación de código – En los últimos años se ha estimado cada vez más deseable que un ADL pueda generar un sistema ejecutable, aunque más no sea de carácter prototípico. De tener que hacerlo manualmente, se podrían suscitar problemas de consistencia y trazabilidad entre una arquitectura y su implementación. Acme, al igual que Wright, se concibe como una notación de modelado y no proporciona soporte directo de generación de código. Por el contrario, diversos ADLs actuales pueden manejar Acme, incluidos Aesop, C2, SADL, UniCon y Wright.

Disponibilidad de plataforma – Ya me he referido a AcmeStudio (2.1), un front-end gráfico programado en Visual C++ y Java que corre en plataforma Windows y que proporciona un ambiente completo para diseñar modelos de arquitectura. La sección de Java requiere JRE, pero también se puede trabajar en términos de COM y Win32

ejecutando AcmeStudio.exe. Los otros dos ambientes gráficos (Armani y el entorno de ISI) son nativos de Windows e implementan intensivamente tecnología COM.

Paulatinamente, Armani se constituyó en un lenguaje de tipo ADL, especializado en la descripción de la estructura de un sistema y su evolución en el tiempo [Mon98]. Es un lenguaje puramente declarativo que describe la estructura del sistema y las restricciones a respetar, pero no hace referencia alguna a la generación del sistema o a la verificación de sus propiedades no funcionales o de consistencia. De alguna manera, la naturaleza de Armani captura también el *expertise* de los diseñadores, señalando su vinculación con la práctica de los patrones arquitectónicos, en este caso patrones de diseño. Armani se basa en siete entidades para describir las instancias del diseño: componentes, conectores, puertos, roles, sistemas, representaciones y propiedades. Para capturar las experiencias y las recetas o "reglas de pulgar", Armani implementa además otras seis entidades que son: tipos de elementos de diseño, tipos de propiedades, invariantes de diseño, heurísticas, análisis y estilos arquitectónicos. La sección denotacional de Armani tiene un aire de familia con la sintaxis de cláusulas de Horn en lenguaje Prolog.

El siguiente ejemplo ilustra la descripción de un modelo de tubería y filtros en Armani:

```
Style Pipe-and-Filter = {
	// Definir el vocabulario de diseño
	// Definir el tipo de flujo
	Property Type flowpathRecT = Record [ fromPt : string; toPt : string; ];
	// Definir tipos de puerto y rol
	Port Type inputT = { Property protocol : string = "char input"; };
	Port Type outputT = { Property protocol : string = "char output"; };
	Role Type sourceT = { Property protocol : string = "char source"; };
	Role Type sinkT = { Property protocol : string = "char sink"; };
	// Definir tipos de componentes
	Component Type Filter = {
		Port input : inputT;
		Port output : outputT;
		Property function : string;
		Property flowPaths : set{flowpathRecT}
			<< default : set{flowpathRecT} =
			[ fromPt : string = "input"; toPt : string = "output"]; >>;
			// restricción para limitar añadido de otras puertos
			Invariant forall p : port in self.Ports |
			satisfiesType(p, inputT) or satisfiesType(p,outputT);
	};
	// Definir tipos de componentes
	Connector Type Pipe = {
		Role source : sourceT;
		Role sink : sinkT;
		Property bufferSize : int;
		Property flowPaths : set{flowpathRecT} =
		[ from : string = "source"; to : string = "sink" ];
		// los invariantes requieren que un Pipe tenga
		// exactamente dos roles y un buffer con capacidad positiva.
		Invariant size(self.Roles) == 2;
		Invariant bufferSize >= 0;
```

```
};
// Definir diseño abstracto de análisis para todo el estilo
// y verificar ciclos en el grafo del sistema
Design Analysis hasCycle(sys :System) : boolean =
       forall c1 : Component in sys.Components | reachable(c1,c1);
       // definir análisis de diseño externo que computa la performance
       // de un componente
External Analysis throughputRate(comp :Component) : int =
armani.tools.rateAnalyses.throughputRate(comp);
// Especificar invariantes de diseño y heurística
// para sistemas construidos en este estilo.
// Vincular inputs a sinks y outputs a fuentes
Invariant forall comp : Component in self.Components |
Forall conn : Connector in self.Connectors |
Forall p : Port in comp.Ports |
Forall r : Role in conn.Roles |
       attached(p,r) ->
       ((satisfiesType(p,inputT) and satisfiesType(r,sinkT)) or
       (satisfiesType(p,outputT) and satisfiesType(r,sourceT)));
// no hay roles no asignados
Invariant forall conn : Connector in self.Connectors | forall r : Role in
conn.Roles |
exists comp : Component in self.Components | exists p : Port in
comp.Ports |
attached(p,r);
// flag de puertos no vinculados
Heuristic forall comp : Component in self.Components |
       forall p : Port in comp.Ports | exists conn : Connector in
       self.Connectors |
       exists r : Role in conn.Roles |
       attached(p,r);
// Restricción: En un estilo de tubería y filtros no puede haber ciclos
Invariant !hasCycle(self);
// Propiedad no funcional: Todos los componentes deben tener un
// throughput de 100 (unidades) como mínimo.
Heuristic forall comp : Component in self.Components |
       throughputRate(comp) >= 100;
}; // fin de la definición del estilo-familia.
```

Como puede verse, en tanto lenguaje Armani es transparentemente claro, aunque un tanto verboso. Una vez que se ha realizado la declaración del estilo, aún restaría agregar código para definir una instancia y especificar un conjunto de invariantes para limitar la evolución de la misma.

ADML

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

(TOGAF). La página de cabecera de la iniciativa se encuentra en http://www.enterprise-architecture.info/Images/ADML/WEB%20ADML.htm.

Como quiera que sea, ADML agrega al mundo de los ADLs una forma de representación basada en estándares de la industria, de modo que ésta pueda ser leída por cualquier parser de XML. En ambientes Windows el *parser* primario y el serializador de XML se instala con Microsoft Internet Explorer de la versión 4 en adelante, y todas las aplicaciones de Office, así como SQL Server, poseen soporte nativo de XML y por lo tanto del lenguaje arquitectónico de *markup*. El Framework .NET de Microsoft incluye además clases (`xmlreader`, `xmlwriter`) que hacen que implementar tratamiento de documentos ADML, xADL, xArch y sus variantes resulte relativamente trivial. En consonancia con la expansión de Internet, ADML permite también definir vínculos con objetos externos a la arquitectura (fundamentación racional, diseños, componentes, etcétera), así como interactuar con diversos repositorios de industria, tales como las especificaciones de OASIS relativas a esquemas para SWIFT, IFX, OFX/OFE, BIPS, OTP, OMF, HL7, RosettaNet o similares.

ADML constituye además un tronco del que depende una cantidad de especificaciones más puntuales. Mientras ADML todavía reposaba en DTD (Document Type Definition), una sintaxis de metadata que ahora se estima obsoleta, las especificaciones más nuevas implementan esquemas extensibles de XML. La más relevante tal vez sea xADL (a pronunciar como "zaydal"), desarrollado por la Universidad de California en Irvine, que define XML Schemas para la descripción de familias arquitectónicas, o sea estilos. La especificación inicial de xADL 2.0 se encuentra en http://www.isr.uci.edu/projects/xarchuci/. Técnicamente xADL es lo que se llama una aplicación de una especificación más abstracta y genérica, xArch, que es un lenguaje basado en XML, elaborado en Irvine y Carnegie Mellon para la descripción de arquitecturas [DH01]. Cada tipo de conector, componente e interfaz de xADL incluye un *placeholder* de implementación que vendría a ser análogo a una clase virtual o abstracta de un lenguaje orientado a objetos. Éste es reemplazado por las variables correspondientes del modelo de programación y plataforma en cada implementación concreta. Esto permite vincular descripciones arquitectónicas y modelos directamente con cualquier binario, *scripting* o entidad en cualquier plataforma y en cualquier lenguaje.

Aesop

El nombre oficial es Aesop Software Architecture Design Environment Generator. Se ha desarrollado como parte del proyecto ABLE de la Universidad Carnegie Mellon, cuyo objetivo es la exploración de las bases formales de la arquitectura de software, el desarrollo del concepto de estilo arquitectónico y la producción de herramientas útiles a la arquitectura, de las cuales Aesop es precisamente la más relevante. La elaboración formal del proyecto ABLE, por otro lado, ha resultado en el lenguaje Wright, que en este estudio se trata separadamente. Uno de los mejores documentos sobre Aesop es el ensayo de David Garlan, Robert Allen y John Ockerbloom que explora el uso de estilos en el diseño arquitectónico [GAO94].

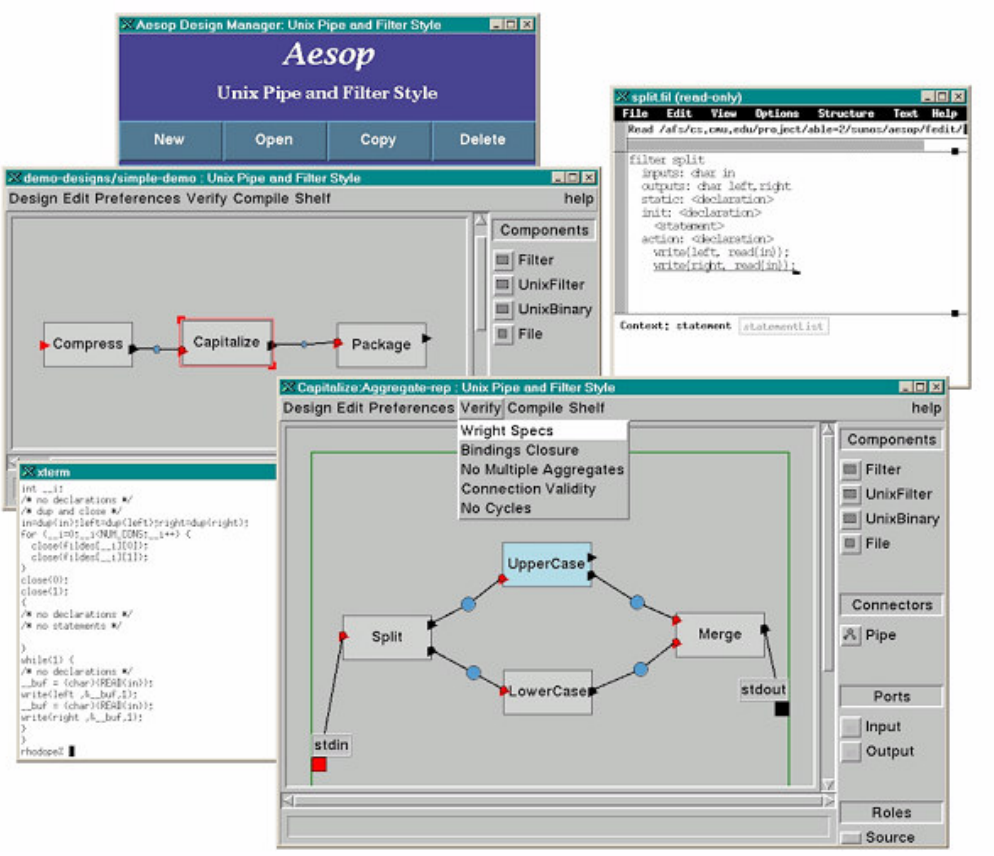


Fig. 2 - Ambiente gráfico de Aesop con diagrama de tubería y filtro

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

Sitio de referencia – Razonablemente, se lo encuentra siguiendo el rastro de la Universidad Carnegie Mellon, la escuela de computación científica y el proyecto ABLE, en la página http://www-2.cs.cmu.edu/afs/cs/project/able/www/aesop/aesop_home.html. Aunque hay bastante información en línea, el vínculo de distribución de Aesop estaba muerto o inaccesible en el momento de redacción de este documento (enero de 2004).

Estilos - En Aesop, conforme a su naturaleza orientada a objetos, el vocabulario relativo a estilos arquitectónicos se describe mediante la definición de sub-tipos de los tipos arquitectónicos básicos: Componente, Conector, Puerto, Rol, Configuración y Binding.

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

Modelo semántico – Aesop presupone que la semántica de una arquitectura puede ser arbitrariamente distinta para cada estilo. Por lo tanto, no incluye ningún soporte nativo

para la descripción de la semántica de un estilo o configuración, sino que apenas presenta unos cuadros vacantes para colocar esa información como comentario.

Soporte de lenguajes - Aesop (igual que Darwin) sólo soporta nativamente desarrollos realizados en C++.

Generación de código – Aesop genera código C++. Aunque Aesop opera primariamente desde una interfaz visual, el código de Aesop es marcadamente más procedural que el de Acme, por ejemplo, el cual posee una estructura de orden más bien declarativo. El siguiente ejemplo muestra un estilo de tubería y filtros instanciado al tratamiento de un texto. Omito el código de encabezamiento básico del filtro:

```
// Primero corresponde definir el filtro
#include "filter_header.h"
void split(in,left,right)
{
char __buf;
int __i;
/* no hay declaraciones */
/* dup y cerrar */
in = dup(in);left = dup(left);
right = dup(right);
for (__i=0;__i<NUM_CONS;__i++) {
close(fildes[__i][0]);
close(fildes[__i][1]);
}
close(0);
close(1);
{
/*no hay declaraciones*/
/*no hacer nada*/
}
while(1)
{
/*no hay declaraciones*/
__buf = (char)((char) READ(in));
write(left,&__buf,1);
__buf = (char)((char) READ(in));
write(right,&__buf,1);
}
}
// Genera código para un sistema de estilo tubería y filtros
int main(int argc,char **argv) {
fable_init_event_system(&argc,argv,BUILD_PF); // inicializa eventos locales
fam_initialize(argc,argv); // inicializa la base de datos
arch_db = fam_arch_db::fetch(); // obtiene el puntero de la base de datos
t = arch_db.open(READ_TRANSACTION); // comienza transacción de lectura
fam_object o = get_object_parameter(argc,argv); // obtiene objeto raíz
if (!o.valid() || !o.type().is_type(pf_filter_type)) { // filtro no válido?
cerr << argv[0] << ": invalid parameter\n"; // si no es válido, cerrar
t.close();
fam_terminate();
```

```
exit(1);
}
pf_filter root = pf_filter::typed(o);
pf_aggregate ag = find_pf_aggregate(root); // encontrar agregado de la raiz
// (si no lo hay, imprimir diagnóstico; omito el código)
start_main(); // escribir el comienzo del main() generado
outer_io(root); // vincular puertos externas a stdin/out
if (ag.valid()) {
pipe_names(ag); // escribir código para conectar tuberías
bindings(root); // definir alias de bindings
spawn_filters(ag); // y hacer "fork off" de los filtros
}
finish_main(); // escribir terminación del main() generado
make_filter_header(num_pipes); // escribir header para los nombres de tuberías
t.close(); // cerrar transacción
fam_terminate(); // terminar fam
fable_main_event_loop(); // esperar el evento de terminación
fable_finish(); // y finalizar
return 0;
} // main
```

Disponibilidad de plataforma – Aesop no está disponible en plataforma Windows, aunque naturalmente puede utilizarse para modelar sistemas implementados en cualquier plataforma.

ArTek

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

Disponibilidad de plataforma – Hoy en día ArTek no se encuentra disponible en ningún sitio y para ninguna plataforma.

C2 (C2 SADL, C2SADEL, xArch, xADL)

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

derivaciones, pero sin hacer referencia a su conformidad con C2, que en los hechos ha sido enfatizado cada vez menos. Otra variante, SADL a secas, denota Structural Architecture Description Language; fue promovido alguna vez por SRI, pero no parece gozar hoy de buena salud.

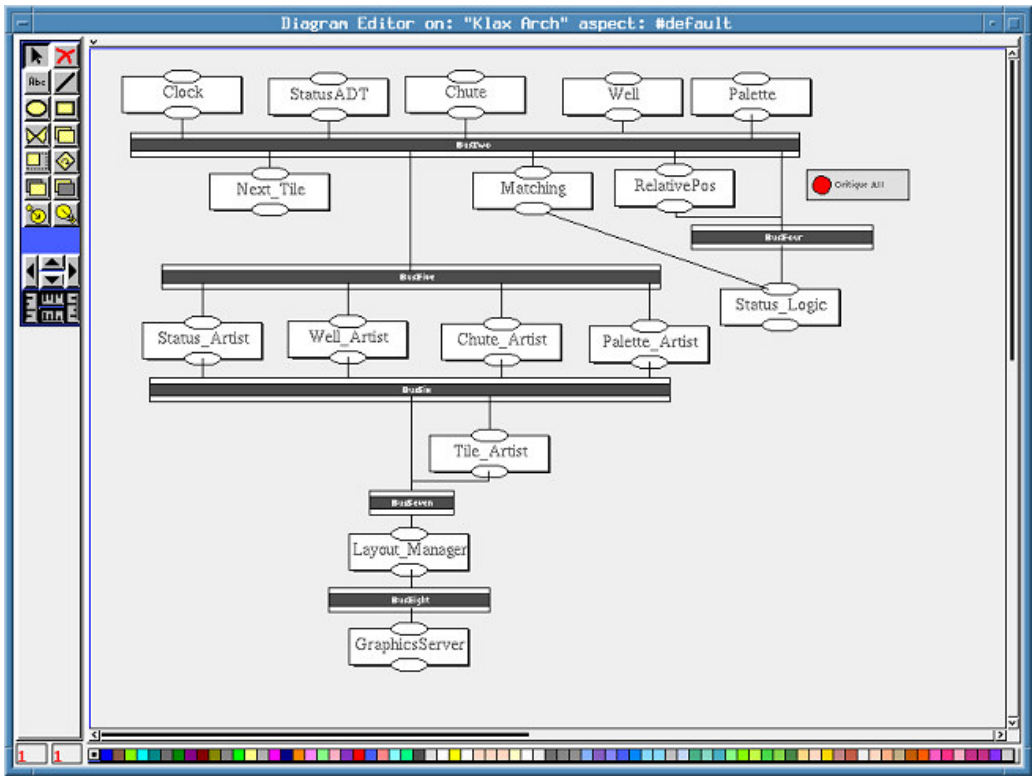


Fig. 3 - Ambiente de modelado C2 con diagrama de estilo

En una arquitectura de estilo C2, los conectores trasmiten mensajes entre componentes, los cuales mantienen el estado, ejecutan operaciones e intercambian mensajes con otros componentes a través de dos interfaces (llamadas *top* y *bottom*). Los componentes no intercambian mensajes directamente, sino a través de conectores. Cada interfaz de un componente puede vincularse con un solo conector, el cual a su vez se puede vincular a cualquier número de otros conectores o componentes. Los mensajes de requerimiento sólo se pueden enviar "hacia arriba" en la arquitectura, y los de notificación sólo "hacia abajo". La única forma de comunicación es a través de pasaje de mensajes, de modo que nunca se utiliza memoria compartida. Esta limitación se impone para permitir la independencia de sustratos, que en la jerga de C2 se refiere a la capacidad de reutilizar un componente en otro contexto. C2 no presupone nada respecto del lenguaje de programación en que se implementan los componentes o conectores (que pueden ser tranquilamente Visual Basic .NET o C#), ni sobre el manejo de *threads* de control de los componentes, el deployment de los mismos o el protocolo utilizado por los conectores. Por ello ha sido una de las bases de las modernas arquitecturas basadas en servicios.

Sitio de referencia – El sitio de ISR en la Universidad de California en Irvine está en http://www.isr.uci.edu/architecture/adl/SADL.html. xARch se encuentra en http://www.-isr.uci.edu/architecture/xarch/. XADL 2.0 se localiza en http://www.isr.uci.edu/projects-/xarchuci/.

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

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

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

En un ADL conforme a C2 de la generación SADL, el ejemplo de referencia de tubería y filtros se convertiría en un paso de mensajes entre componentes a través de una interfaz. Lo primero a definir sería un componente en función de una IDN (Interface Description Notation). La sintaxis sería como sigue:

```
component StackADT is
    interface
        top_domain
            in
                null;
            out
                null;
        bottom_domain
            in
                PushElement (value : stack_type);
                PopElement ();
                GetTopElement ();
            out
                ElementPushed (value : stack_type);
                ElementPopped (value : stack_type);
                TopStackElement (value : stack_type);
                StackEmpty ();
    parameters
            null;
    methods
            procedure Push (value : stack_type);
            function Pop () return stack_type;
            function Top () return stack_type;
            function IsEmpty () return boolean;
    behavior
        received_messages PushElement;
            invoke_methods Push;
            always_generate ElementPushed;
        received_messages PopElement;
            invoke_methods IsEmpty, Pop;
            always_generate StackEmpty xor ElementPopped;
        received_messages GetTopElement;
```

```
            invoke_methods IsEmpty, Top;
            always_generate StackEmpty xor TopStackElement;
    context
        top_most ADT
end StackADT;
```

Nótese que en este lenguaje, suficientemente legible como para que aquí prescindamos de comentarios, se trata un *stream* como un *stack*. Hasta aquí se han definido componentes, interfaces, métodos y conducta. En segundo término vendría la especificación declarativa de la arquitectura a través de ADN (Architecture Description Notation):

```
architecture StackVisualizationArchitecture is
    components
        top_most
            StackADT;
        internal
            StackVisualization1;
            StackVisualization2;
        bottom_most
            GraphicsServer;
    connectors
        connector TopConnector is
            message_filter no_filtering
        end TopConnector;
        connector BottomConnector is
            message_filter no_filtering
        end BottomConnector;
    architectural_topology
        connector TopConnector connections
            top_ports
                StackADT;
            bottom_ports
                StackVisualization1;
                StackVisualization2;
        connector BottomConnector connections
            top_ports
                StackVisualization1;
                StackVisualization2;
            bottom_ports
                GraphicsServer;
end StackVisualizationArchitecture;
system StackVisualizationSystem is
    architecture StackVisualizationArchitecture with
        StackADT is_bound_to IntegerStack;
        StackVisualization1 is_bound_to StackArtist1;
        StackVisualization2 is_bound_to StackArtist2;
        GraphicsServer is_bound_to C2GraphicsBinding;
end StackVisualizationSystem;
```

Por último, se podrían especificar imperativamente cambios en la arquitectura por medio de un tercer lenguaje, que sería un ACN (Architecture Construction Notation). Omitiremos esta ejemplificación. En algún momento C2 SADL (que nunca pasó de la

fase de prototipo alfa) fue reemplazado por C2SADEL, cuya sintaxis es diferente y ya no se articula en la misma clase de módulos declarativos e imperativos.

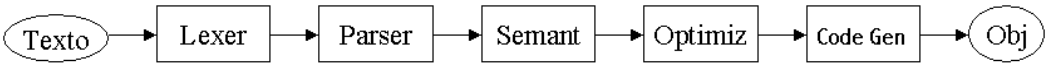
Disponibilidad de plataforma – Existen extensiones de Microsoft Visio para C2 y xADL. Una interfaz gráfica para C2SADEL disponible para Windows es DRADEL. También se puede utilizar una herramienta llamada SAAGE (disponible en el sitio de DASADA, http://www.rl.af.mil/tech/programs/dasada/tools/saage.html) que requiere explícitamente Visual J++, COM y ya sea Rational Rose o DRADEL como *front-ends*. Hay multitud de herramientas que generan lenguaje conforme al estándar C2, ya sea en estado crudo o, como es el caso en los proyectos más recientes, en formato xADL. Una de ellas, independientes de plataforma, es el gigantesco entorno gráfico ArchStudio, él mismo implementado en estilo C2. En una época ArchStudio incluía Argo, otro ambiente de diseño gráfico para C2 actualmente discontinuado.

CHAM

CHAM (Chemical Abstract Machine) no es estrictamente un ADL, aunque algunos autores, en particular Inverardi y Wolf [BB92] aplicaron CHAM para describir la arquitectura de un compilador. Se argumenta, en efecto, que CHAM proporciona una base útil para la descripción de una arquitectura debido a su capacidad de componer especificaciones para las partes y describir explícitamente las reglas de composición. Sin embargo, la formalización mediante CHAM es idiosincrática y (por así decirlo) hecha a mano, de modo que no hay criterios claros para analizar la consistencia y la completitud de las descripciones de configuración. Convendrá contar entonces con alguna herramienta de verificación.

CHAM es una técnica de especificación basada en álgebra de procesos que utiliza como fundamento teórico los sistemas de rescritura de términos para capturar la conducta comunicativa de los componentes arquitectónicos. El modelo de CHAM reposa en una metáfora química en la cual la conducta de una arquitectura se especifica definiendo moléculas y soluciones de moléculas. Las moléculas constituyen los componentes básicos, mientras que las soluciones son multiconjuntos de moléculas que definen los estados de una CHAM. Una especificación CHAM también contiene reglas de transformación que dictan las formas en que pueden evolucionar las soluciones (o sea, en que pueden cambiar los estados). En un momento dado, se puede aplicar un número arbitrario de reglas a una solución, siempre que no entren en conflicto. De esta forma es posible modelar conductas paralelas, ejecutando transformaciones en paralelo. Cuando se aplica más de una regla a una molécula o conjunto, CHAM efectúa una decisión no determinista, escogiendo alguna de las reglas definidas.

A fin de ilustrar un modelo en CHAM relacionado con nuestro ejemplo canónico de tubería y filtros, supongamos que se desea compilar una pieza de código en Lisp semejante a la que se encuentra en el Software Development Kit que acompaña a Visual Studio.NET, el cual incluye un programa en C# para generar un compilador Lisp y sus respectivas piezas. Estas piezas están articuladas igual que en el ejemplo, incluyendo un Lexer, un Parser, un Semantizador y un Generador de código. A través de las sucesivas transformaciones operadas en los filtros, el programa fuente se convierte en código objeto en formato MSIL.



La especificación del modelo CHAM correspondiente es altamente intuitiva. Se comienza con un conjunto de constantes *P* que representan los elementos de procesamiento, un conjunto de constantes *D* que denotan los elementos de dato y un operador infijo *P* que expresa el estado de un elemento. Los elementos de conexión están dados por un tercer conjunto *C* consistente en dos operaciones, *i* y *o*, que actúan sobre los elementos de dato y que obviamente denotan entrada y salida. Lo que se llama sintaxis *sigma* de moléculas en esta arquitectura secuencial es entonces:

- `M ::= P | C | M <> M`
- `P ::= text | lexer | parser | semantor | optimizer | generator`
- `D ::= char | tok | phr | cophr | obj`
- `C ::= i(D) | o(D)`

El siguiente paso en CHAM es siempre definir una solución inicial S_1. Esta solución es un subconjunto de todas las moléculas posibles que se pueden construir bajo *sigma* y corresponde a la configuración inicial conforme a la arquitectura.

```
S1 = text <> o(char), i(char) <> o(tok) <> lexer
       i(tok) <> o(phr) <> parser, i(phr) <> o(cophr) <> semantor,
       i(cophr) <> o(cophr) <> optimizer, i(cophr) <> o(obj) <> generator
```

Esto quiere decir que se ingresa inicialmente texto en forma de caracteres, el Lexer lo transforma en *tokens*, el Parser en frases, el Semantor en co-frases que luego se optimizan, y el Generador de código lo convierte en código objeto. El paso final es especificar tres reglas de transformación:

```
T1 = text <> o(char) --> o(char) <> text
T2 = i(d) <> m1, o(d) <> m2 --> m1 <> i(d), m2 <> o(d)
T3 = o(obj) <> generator <> i(cophr) --> i(char) <> o(tok) <> lexer,
       i(tok) <> o(phr) <> parser, i(phr) <> o(cophr) <> semantor,
       i(cophr) <> o(cophr) <> optimizer, i(cophr) <> o(obj) <> generator
```

La primera regla hace que el código fuente quede disponible para compilar. El resto puede interpretarse fácilmente a través de las correspondencias entre las tres series de notaciones.

Disponibilidad de plataforma – CHAM es un modelo de máquina abstracta independiente de plataforma y del lenguaje o paradigma de programación que se vaya a utilizar en el sistema que se modela. En una reciente tesis de licenciatura de la Facultad de Ciencias Exactas de la Universidad de Buenos Aires, de hecho, se utilizó CHAM (y no cálculo JI) para modelar las prestaciones de XLANG del middleware de integración BizTalk de Microsoft. Aunque CHAM no es decididamente un ADL, nos ha parecido que su notación y su fundamentación subyacente son una contribución interesante para una mejor comprensión de las herramientas descriptivas de una arquitectura. En el ejemplo propuesto se discierne no sólo un estilo en toda su pureza, sino que se aprecia además la

potencia expresiva de un lenguaje de descripción arquitectónica y un patrón de diseño que se impone a la hora de definir el modelo de cualquier compilador secuencial.

Darwin

Darwin es un lenguaje de descripción arquitectónica desarrollado por Jeff Magee y Jeff Kramer [MEDK95, MK96]. Darwin describe un tipo de componente mediante una interfaz consistente en una colección de servicios que son ya sea provistos (declarados por ese componente) o requeridos (o sea, que se espera ocurran en el entorno). Las configuraciones se desarrollan instanciando las declaraciones de componentes y estableciendo vínculos entre ambas clases de servicios.

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

Cada servicio de Darwin se modeliza como un nombre de canal, y cada declaración de *binding* es un proceso que trasmite el nombre del canal al componente que requiere el servicio. En una implementación generada en Darwin, se presupone que cada componente primitivo está implementado en algún lenguaje de programación, y que para cada tipo de servicio se necesita un ligamento (*glue*) que depende de cada plataforma. El algoritmo de elaboración actúa, esencialmente, como un servidor de nombre que proporciona la ubicación de los servicios provistos a cualquier componentes que se ejecute. El ejemplo muestra la definición de una tubería en Darwin [MK96: 5].

```
component pipeline (int n) {
       provide input;
       require output;
       array F[n]:filter;
       forall k:0..n-1 {
              inst F[k];
              bind F[k].output -- output;
              when k<n-1 bind
                     F[k].next -- F[k+1].prev;
              }
              bind
                     input -- F[0].prev;
                     F[n-1].next -- output;
              }
```

Darwin no proporciona una base adecuada para el análisis de la conducta de una arquitectura, debido a que el modelo no dispone de ningún medio para describir las propiedades de un componente o de sus servicios más que como comentario. Los componentes de implementación se interpretan como cajas negras, mientras que la

colección de tipos de servicio es una colección dependiente de plataforma cuya semántica tampoco se encuentra interpretada en el *framework* de Darwin [All97].

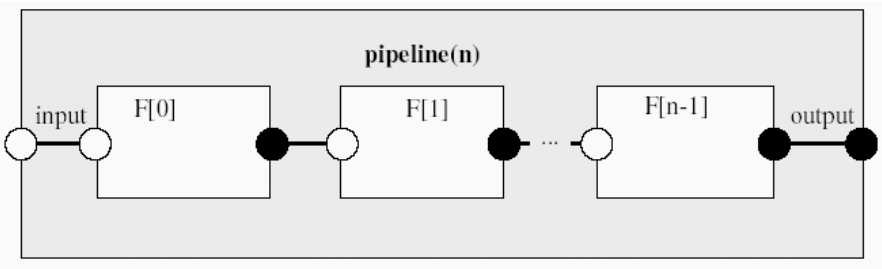


Fig. 4 - Diagrama de tubería en Darwin

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

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

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

Conectores – Al pertenecer a la clase en la que Medvidovic [Med96] agrupa a los lenguajes de configuración *in-line*, en Darwin (al igual que en Rapide) no es posible ponerle nombre, sub-tipear o reutilizar un conector. Tampoco se pueden describir patrones de interacción independientemente de los componentes que interactúan.

Semántica - Darwin proporciona una semántica para sus procesos estructurales mediante el cálculo Л. Cada servicio se modeliza como un nombre de canal, y cada declaración de enlace (*binding*) se entiende como un proceso que trasmite el nombre de ese canal a un componente que requiere el servicio. Este modelo se ha utilizado para demostrar la corrección lógica de las configuraciones de Darwin. Dado que el cálculo Л ha sido designado específicamente para procesos móviles, su uso como modelo semántico confiere a las configuraciones de Darwin un carácter potencialmente dinámico. En un escenario como el Web, en el que las entidades que interactúan no están ligadas por conexiones fijas ni caracterizadas por propiedades definidas de localización, esta clase de cálculo se presenta como un formalismo extremadamente útil. Ha sido, de hecho, una de las herramientas formales que estuvo en la base de los primeros modelos del XLANG de Microsoft, que ahora se encuentra derivando hacia BPEL4WS (Business Process Execution Language for Web Services).

Análisis y verificación – A pesar del uso de un modelo de cálculo Л para las descripciones estructurales, Darwin no proporciona una base adecuada para el análisis del comportamiento de una arquitectura. Esto es debido a que el modelo no posee

herramientas para describir las propiedades de un componente o de los servicios que presta. Las implementaciones de un componente vendrían a ser de este modo cajas negras no interpretadas, mientras que los tipos de servicio son una colección dependiente de la plataforma cuya semántica también se encuentra sin interpretar en el framework de Darwin.

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

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

Observaciones – Darwin (lo mismo que UniCon) carece de la capacidad de definir nuevos tipos, soportando sólo una amplia variedad de tipos de servicio predefinidos. Darwin presupone que la colección de tipos de servicio es suministrada por la plataforma para la cual se desarrolla una implementación, y confía en la existencia de nombres de tipos de servicio que se utilizan sin interpretación, sólo verificando su compatibilidad.

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

Jacal

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

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

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

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

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

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

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

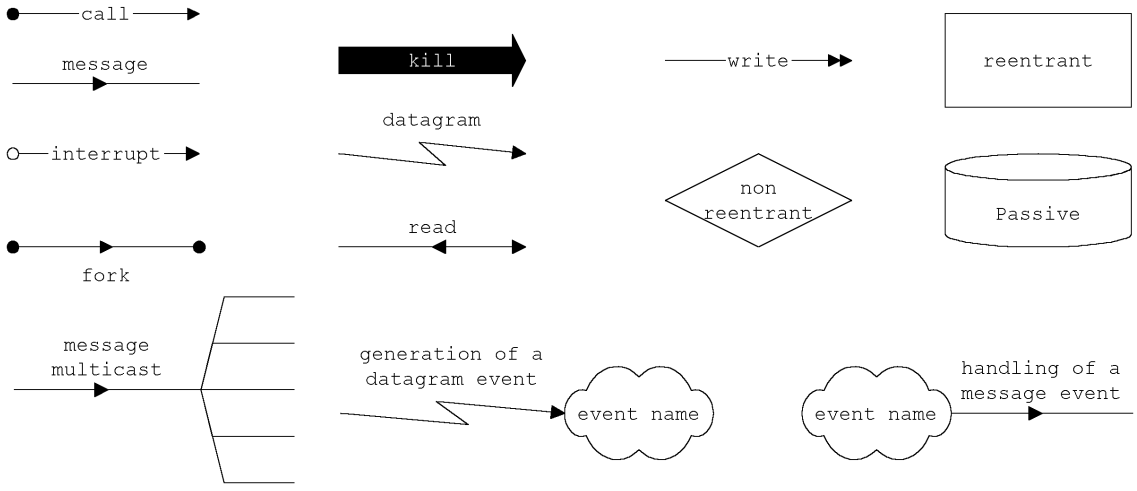
Además del nivel de interfaz, que corresponde a la configuración de una arquitectura ya que allí se determina la conectividad entre los distintos componentes, Jacal ofrece un nivel de descripción adicional, llamado nivel de comportamiento. En este nivel, se describe la relación entre las comunicaciones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

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

Interface gráfica – Como ya se ha dicho, la notación principal de Jacal es gráfica y hay una herramienta disponible en línea para editar y animar visualmente las arquitecturas.

En el nivel de interfaz, existen símbolos predeterminados para representar cada tipo de componente y cada tipo de conector, como se muestra en la siguiente figura.

A continuación, se da una explicación informal de la semántica de cada uno de los tipos de conectores mostrados en la figura.



`call`: transfiere el control y espera una respuesta.

`message`: coloca un mensaje en una cola y sigue ejecutando.

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

`fork`: agrega un flujo de control al receptor, el emisor continúa ejecutando.

`kill`: detiene todos los flujos de control en el receptor, el emisor continúa ejecutando.

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

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

`write`: cambia el estado del receptor, el emisor continúa su ejecución.

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

Disponibilidad de plataforma – La herramienta que actualmente está disponible para editar y animar arquitecturas en Jacal es una aplicación Win32, que no requiere instalación, basta con copiar el archivo ejecutable para comenzar a usarla.

El ambiente consiste en una interfaz gráfica de usuario, donde pueden dibujarse representaciones Jacal de sistemas, incluyendo tanto el nivel de interfaz como el de comportamiento. Se pueden editar múltiples sistemas simultáneamente y, abriendo distintas vistas, visualizar simultáneamente los dos niveles de un mismo sistema, para uno o más componentes.

El editor es orientado a la sintaxis, en el sentido de que no permite dibujar configuraciones inválidas. Por ejemplo, valida la compatibilidad entre el tipo de un componente y los conectores asociados. Para aumentar la flexibilidad (especialmente en el orden en que se dibuja una arquitectura), se dejan otras validaciones para el momento de la animación.

Cuando se anima una arquitectura, los flujos de control se generan en comunicaciones iniciadas por el usuario (representa una interacción con el mundo exterior) haciendo clic en el extremo de origen de un conector que no esté ligado a ningún puerto. Los flujos de control se muestran como círculos de colores aleatorios que se mueven a lo largo de los conectores y las transiciones. Durante la animación, el usuario puede abrir vistas adicionales para observar el comportamiento interno de uno o más componentes, sin dejar de tener en pantalla la vista global del nivel de interfaz.

La siguiente figura muestra la interfaz de la aplicación con un caso de estudio.

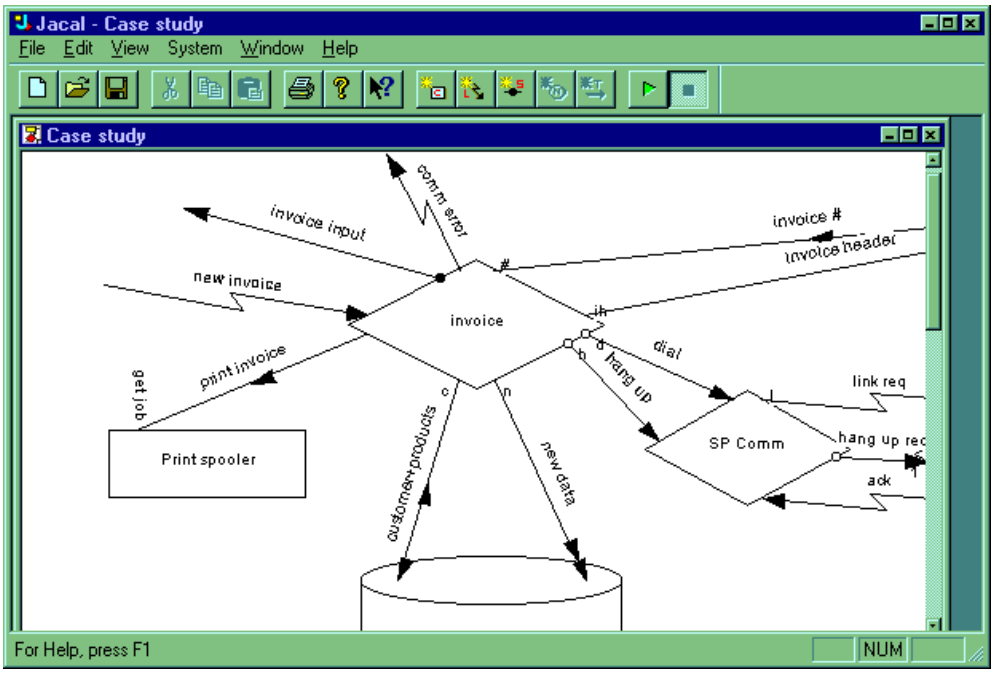


Fig. 5 – Estudio de caso en Jacal

Actualmente se encuentra en desarrollo una nueva versión del lenguaje y de la aplicación (Jacal 2). En esta nueva versión, se utilizará Microsoft Visio como interfaz del usuario, tanto para dibujar como para animar arquitecturas. La fecha estimada para su publicación es mediados de 2004.

LILEANNA

Al igual que en el caso de ArTek, en opinión de Medvidovic LILEANNA no es un genuino ADL, por cuanto la configuración es modelizada implícitamente mediante información de interconexión que se distribuye entre la definición de los componentes individuales y los conectores. En este sentido, aunque pueda no ser un ADL en sentido estricto, se le reconoce la capacidad de modelizar ciertos aspectos de una arquitectura.

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

LILEANNA es un ADL (o más estrictamente un MIL) que utiliza el lenguaje Ada para la implementación y Anna para la especificación. Fue desarrollado como parte del proyecto DSSA ADAGE, patrocinado por ARPA. La implementación fue llevada a cabo por Will

Tracz de Loral Federal Systems y se utilizó para producir software de navegación de helicópteros.

La semántica formal de LILEANNA se basa en la teoría de categorías, siguiendo ideas desarrolladas para el lenguaje de especificación Clear; posteriormente se le agregó una semántica basada en teoría de conjuntos. Las propiedades de interconexión de módulos se relacionan bastante directamente con las de los componentes efectivos a través de la semántica de las expresiones de módulo. Aunque una especificación en LILEANNA es varios órdenes de magnitud más verbosa de lo que ahora se estima deseable para visualizar una descripción, incluye un "editor de *layout*" gráfico basado en "una notación como la que usan típicamente los ingenieros, es decir cajas y flechas". Es significativo que el documento de referencia más importante sobre el proyecto [Gog96], tan puntilloso respecto de las formas sintácticas y los métodos formales concomitantes a la programación parametrizada, se refiera a su representación visual en estos términos.

Aunque parecería corresponderse con un paradigma peculiar de programación, el modelo parametrizado de LILEANNA soporta diferentes estilos de comunicación, tales como variables compartidas, tuberías, paso de mensajes y *blackboarding*. Mediante un sistema auxiliar llamado TOOR, se pudo implementar además el rastreo (*tracing*) de dependencias entre objetos potencialmente evolutivos y relaciones entre objetos en función del tiempo. Esta estrategia, pomposamente llamada hiper-requerimiento, se funda en métodos bien conocidos en programación parametrizada e hiperprogramación. En este contexto, "hiper" connota una semántica de vinculación con el mundo exterior análoga a la de la idea de hipertexto. La hiperprogramación y los hiper-requerimientos soportan reutilización. TOOR, la hiper-herramienta agregada a LILEANNA, está construida a imagen y semejanza de FOOPS, un lenguaje orientado a objetos de propósito general. TOOR utiliza módulos semejantes a los de FOOPS para declarar objetos de software y relaciones, y luego generar vínculos a medida que los objetos se interconectan y evolucionan. TOOR proporciona facilidades de hipermedia basados en HTML, de modo que se pueden agregar gráficos, grafos y videos, así como punteros a documentos tradicionales.

Dado que el modelo de programación parametrizada subyacente a LILEANNA (que habla de teorías, axiomas, grafos, vistas, *stacks*, aserciones, estructuras verticales y horizontales, *packages* y máquinas virtuales en un sentido idiosincrático a ese paradigma) no detallaré aquí las peculiaridades del lenguaje. Los fundamentos formales de LILEANNA son poderosos, pero no están en línea con el estilo de arquitectura orientada a servicios o con modelos igualmente robustos, como los de C2 y Wright. Aunque no pueda esperarse que LILEANNA tenga en el futuro próximo participación directa y protagónica en la corriente principal de los ADLs, encarna algunas ideas sobre lo que se puede hacer con un lenguaje descriptivo que pueden resultar más que informativas para los arquitectos de software.

MetaH/AADL

Así como LILEANNA es un ADL ligado a desarrollos que guardan relación específica con helicópteros, MetaH modela arquitecturas en los dominios de guía, navegación y control (GN&C) y en el diseño aeronáutico. Aunque en su origen estuvo ligado

estrechamente a un dominio, los requerimientos imperantes obligaron a implementar recursos susceptibles de extrapolarse productivamente a la tecnología de ADLs de propósito general. AADL (Avionics Architecture Description Language) está basado en la estructura textual de MetaH. Aunque comparte las mismas siglas, no debe confudirse este AADL con Axiomatic Architecture Description Language, una iniciativa algo más antigua que se refiere al diseño arquitectónico físico de computadoras paralelas.

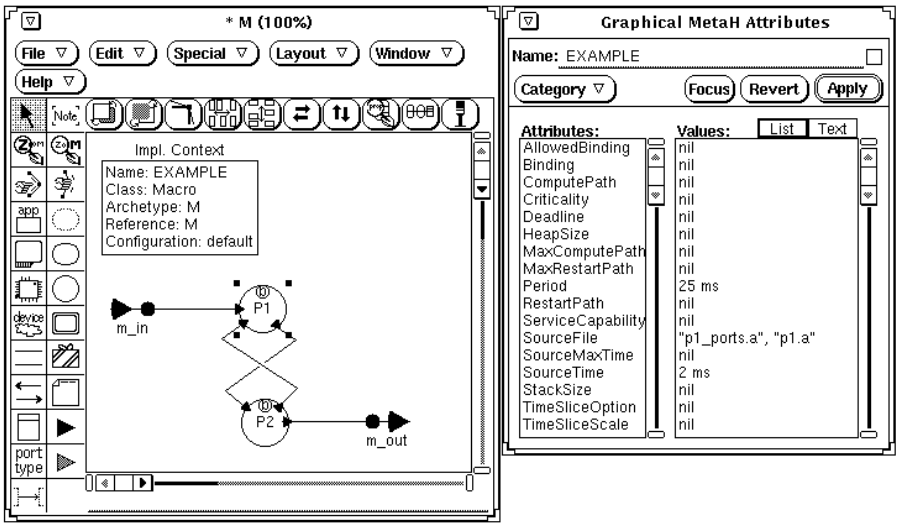


Fig. 6 - Ambiente gráfico MetaH con diagrama de macro

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

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

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

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

Rapide

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

Sitio de referencia – Universidad de Stanford - http://pavg.stanford.edu/rapide/

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

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

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

Semántica – Mientras muchos lenguajes de tipo ADL no soportan ninguna especificación semántica de sus componentes más allá de la descripción de sus interfaces, Wright y Rapide permiten modelar la conducta de sus componentes. Rapide define tipos de componentes (llamados interfaces) en términos de una colección de eventos de comunicación que pueden ser *observados* (acciones externas) o *iniciados* (acciones públicas). Las interfaces de Rapide definen el comportamiento computacional de un componente vinculando la observación de acciones externas con la iniciación de acciones públicas. Cada especificación posee una conducta asociada que se define a través de conjuntos de eventos parcialmente ordenados (*posets*); Rapide utiliza patrones de eventos para identificar *posets*, de manera análoga a la del método `match` de las expresiones regulares de .NET Framework. Para describir comportamientos Rapide también implementa un lenguaje cuyo modelo de interfaz se basa en Standard ML, extendido con eventos y patrones de eventos.

Análisis y verificación automática – En Rapide, el monitoreo de eventos y las herramientas nativas de filtrado facilitan el análisis de arquitectura. También es posible implementar verificación de consistencia y análisis mediante simulación. En esencia, en Rapide toda la arquitectura es simulada, generando un conjunto de eventos que se supone es compatible con las especificaciones de interfaz, conducta y restricciones. La simulación es entonces útil para detectar alternativas de ejecución. Rapide también proporciona una caja de herramientas específica para simular la arquitectura junto con la ejecución de la implementación. Sin embargo (como ha señalado Medvidovic en su *survey*) un proceso de ejecución solamente provee una idea del comportamiento con un juego particular de variables (un *poset* en particular), antes que una confirmación de la conducta frente a todos los valores y escenarios posibles. Esto implica que una corrida del proceso de simulación simplemente testea la arquitectura, y no proporciona un análisis exhaustivo del escenario. Nada garantiza que no pueda surgir una inconsistencia en una ejecución diferente. En general, la arquitectura de software mantiene una actitud de reserva crítica frente a la simulación. Escribe Paul Clements: “La simulación es inherentemente una herramienta de validación débil en la medida que sólo presenta una sola ejecución del sistema cada vez; igual que el *testing*, sólo puede mostrar la presencia antes que la ausencia de fallas. Más poderosos son los verificadores o probadores de teoremas que son capaces de comparar una aserción de seguridad contra todas las posibbles ejecuciones de un programa simultáneamente” [Cle96].

Interfaz gráfica – Rapide soporta notación gráfica.

Soporte de lenguajes – Rapide soporta construcción de sistemas ejecutables especificados en VHDL, C. C++, Ada y Rapide mismo. El siguiente es un ejemplo próximo a nuestro

caso de referencia de tubería y filtros, sólo que en este caso es bidireccional, ya que se ha definido una respuesta de notificación. Rapide no contempla estilos de la misma manera que la mayor parte de los otros ADLs. Nótese que Rapide establece tres clases de conexiones; el conector correspondiente a una conexión de tipo *pipe* es =>

```
type Producer (Max : Positive) is interface
     action out Send (N: Integer);
     action in Reply(N : Integer);
behavior
     Start => send(0);
     (?X in Integer) Reply(?X) where ?X<Max => Send(?X+1);
end Producer;
type Consumer is interface
     action in Receive(N: Integer);
     action out Ack(N : Integer);
behavior
    (?X in Integer) Receive(?X) => Ack(?X);
end Consumer
architecture ProdCon() return SomeType is
     Prod : Producer(100); Cons : Consumer;
connect
     (?n in Integer) Prod.Send(?n) => Cons.Receive(?n);
     Cons.Ack(?n) => Prod.Reply(?n);
end architecture ProdCon;
```

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

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

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

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

UML - De OMT al Modelado OO

UML forma parte del repertorio conocido como lenguajes semi-formales de modelado. Esta variedad de herramientas se remonta a una larga tradición que arrancó a mediados de la década de 1970 con PSL/PSA, SADT y el análisis estructurado. Alrededor de 1990 aparecieron los primeros lenguajes de especificación orientados a objeto propuestos por Grady Booch, Peter Coad, Edward Yourdon y James Rumbaugh. A instancias de Rumbaugh, Booch e Ivar Jacobson, finalmente, estos lenguajes se orientaron hacia lo que es hoy UML (Unified Modeling Language), que superaba la incapacidad de los primeros lenguajes de especificación OO para modelar aspectos dinámicos y de comportamiento de un sistema introduciendo la noción de casos de uso. De hecho, UML surgió de la convergencia de los métodos de Booch orientados a comportamiento, la Object Modeling Technique (OMT) de Rumbaugh y el OOSE/Objectory de Jacobson, así como de otras

tecnologías tales como los gráficos de estado de Jarel, los patrones de documentación de Gamma y otros formalismos.

En términos de número, la masa crítica de los conocedores de UML no admite comparación con la todavía modesta población de especialistas en ADLs. En la comunidad de arquitectos existen dos facciones claramente definidas; la primera, vinculada con el mundo de Rational y UML, impulsa el uso casi irrestricto de UML como si fuera un ADL normal; la segunda ha señalado reiteradas veces las limitaciones de UML no sólo como ADL sino como lenguaje universal de modelado. La literatura crítica de UML es ya un tópico clásico de la reciente arquitectura de software. Entre las deficiencias que se han señalado de UML como lenguaje de especificación están las siguientes:

(1) Un caso de uso de UML no puede especificar los requerimientos de interacción en situaciones en las que un sistema deba iniciar una interacción entre él mismo y un actor externo, puesto que proscribe asociaciones entre actores [Gli00].

(2) Al menos en algunas especificaciones, como UML 1.3, es imposible expresar relaciones secuenciales, paralelas o iterativas entre casos de uso, salvo mediante extensiones o estereotipos que introducen oscuridad en el diseño del sistema; UML, efectivamente, prohíbe la descomposición de casos de uso y la comunicación entre ellos.

(3) UML tampoco puede expresar una estructura entre casos de uso ni una jerarquía de casos de una forma fácil y directa; una precondición tal como "el caso de uso A requiere que el caso de uso X haya sido ejecutado previamente" no se puede expresar formalmente en ese lenguaje.

(4) Un modelo de casos de uso tampoco puede expresar adecuadamente la conducta de un sistema dependiente de estados, o la descomposición de un sub-sistema distribuido.

(5) El modelado del flujo de información en un sistema consistente en subsistemas es al menos confuso en UML, y no se puede expresar en una sola vista [Gli00].

(6) Otros autores, como Theodor Tempelmeier [Tem99, Tem01] han opuesto objeciones a UML como herramienta de diseño en el caso de sistemas embebidos, de alta concurrencia o de misión crítica y/o tiempo real, así como han señalado su inutilidad casi total para modelar sistemas con cualidades emergentes o sistemas que involucren representación del conocimiento.

(7) Otros autores dedicaron tesis enteras a las dificultades de UML referidas a componentes reflexivos y aspectos. Mientras que el modelado orientada a objetos especifica con claridad la interfaz de *inbound* de un objeto (el lado "tiene" de una interfaz), casi ninguna herramienta permite expresar con naturalidad la interfaz opuesta de *outbound*, popularmente conocida como "necesita". Sólo el diseño por contrato de Bertrand Meyer soporta este método de dos vías en su núcleo; en UML se debió implementar como extensión o metamodelo. UML tampoco considera los conectores como objetos de primera clase, por lo cual se deben implementar extensiones mediante ROOM o de alguna otra manera [Abd00].

(8) Klaus-Dieter Schewe, en particular, ha reseñado lo que él interpreta como un cúmulo de limitaciones y oscuridades de UML, popularizando además su consideración como un "dinosaurio moderno" [Sch00]. Al cabo de un análisis pormenorizado del que aquí no podemos dar cuenta sin excedernos de espacio, Schewe estima que, dando continuidad a una vieja polémica iniciada por E. F. Codd cuando éste cuestionó a los modelos en Entidad-Relación, UML es el ganador contemporáneo cuando se trata de falta de definiciones precisas, falta de una clara semántica, falta de claridad con respecto a los niveles de abstracción y falta de una metodología pragmática.

(9) Hoffmeister y otros [HNS99] alegan que UML es deficiente para describir correspondencias tales como el mapeo entre elementos en diferentes vistas, que serían mejor representadas en una tabla; faltan también en UML los elementos para modelar comunicación *peer-to-peer.* Los diagramas de secuencia de UML no son tampoco adecuados para soportar la configuración dinámica de un sistema, ni para describir secuencias generales de actividades.

A despecho que los responsables de UML insistan en la precisión y rigor de su semántica, se han organizado simposios y *workshops* enteros para analizar las limitaciones del modelo semántico de UML y su falta de recursos para detectar tempranamente errores de requerimiento y diseño [OOS99]. Si bien se reconoce que con UML es posible representar virtualmente cualquier cosa, incluso fenómenos y procesos que no son software, se debe admitir que muchas veces no existen formas *estándar* de materializar esas representaciones, de modo que no pueden intercambiarse modelos entre diversas herramientas y contextos sin pérdida de información. Se ha dicho que ni las clases, ni los componentes, ni los *packages*, ni los subsistemas de UML son unidades arquitectónicas adecuadas: las clases están tecnológicamente sesgadas hacia la OO y representan entidades de granularidad demasiado pequeña; los componentes "representan piezas físicas de implementación de un sistema" y por ser unidades de implementación y no de diseño, es evidente que existen en el nivel indebido de abstracción; un *package* carece de la estructura interna necesaria; los subsistemas pueden no aparecer en los diagramas de despliegue, o mezclarse con otras entidades de diferente granuralidad y propósito, carecen del concepto de puerto y su semántica y pragmática se han estimado caóticas [StöS/f].

Una debilidad más seria tiene que ver con la falta de modelos causales rigurosos; aunque UML proporciona herramientas para modelar requerimientos de comportamiento (diagramas de estado, de actividad y de secuencia), al menos en UML 1.x no existe diferencia alguna, por ejemplo, entre mensajes opcionales y mensajes requeridos; se pueden "colorear" los mensajes con restricciones, por cierto, pero es imposible hacerlo de una manera estándar. Asimismo, aunque los objetos de UML se pueden descomponer en piezas más pequeñas, no es posible hacer lo propio con los mensajes. También es palpable la sensación de que su soporte para componentes, subsistemas y servicios necesita ser mejorada sustancialmente (incluso en UML 2.0), que los modelos de implementación son inmaduros y que no hay una clara diferenciación o un orden claro de correspondencias entre modelos notacionales y meta-modelos, o entre análisis y diseño [Dou00] [AW99] [KroS/f].

Robert Allen [All97] ha señalado además que si bien se pueden diagramar comportamientos, los diagramas no distinguen entre un patrón de interacción y la conducta de los participantes en esa interacción. De este modo, es difícil razonar sobre un protocolo de interacción y la conformidad de un componente a ese protocolo, porque la interacción se define en términos de una determinada colección de componentes. Si los componentes coinciden exactamente con la conducta global de las máquinas de estados, entonces puede decirse que obdecen trivialmente el protocolo. Pero si no lo hacen, no hay forma claramente definida de separar las partes de la conducta del componente que son relevantes a una interacción en particular y analizarlas.

En una presentación poco conocida de uno de los líderes de la arquitectura de software, David Garlan, se señalan deficiencias de UML en relación con lo que podría ser la representación de estilos: los *profiles* de UML son demasiado densos y pesados para esa tarea, mientras que los *packages* proporcionan agregación, pero no restricciones. Por otra parte, las asociaciones de UML no sirven para denotar conectores, porque no pueden definirse independientemente del contexto y no es posible representar sub-estructuras. La información arquitectónica se termina construyendo de alguna manera, pero el resultado es un embrollo visual y conceptual; y aunque la simbología se puede visualizar con provecho y está bellamente lograda en imágenes, no hay nada más que hacer con ella que usarla a los efectos de la documentación [Gars/f].

Las limitaciones de UML en materia de *round-trip engineering* han sido documentadas con alguna frecuencia. En el meta-modelo de UML se definen numerosos conceptos que no aparecen en los modelos de implementación, que usualmente son lenguajes orientados a objeto; “Agregación”, “Asociación”, “Composición” y “Restricción” (*constraint*), “Estereotipo” son ejemplos de ello, pero hay muchos más. Herramientas de CASE y modelado industriales de alta calidad, como Rational Rose, TogetherSoft Together o JBuilder de Borland o productos de código abierto como ArgoUML, no poseen definiciones claras de las relaciones de clase binarias como asociación, agregación o composición; distinguen entre ellas a nivel gráfico, pero el código que se sintetiza para las diferentes clases binarias es el mismo. Las herramientas de reingeniería producen por lo tanto relaciones erróneas e inconsistentes; simplemente generando código a partir del diagrama y volviendo a generar el diagrama a partir del código se obtiene un modelo distinto [GAD+02]. De la misma manera, UML carece de conceptos de invocación y acceso; si bien las herramientas existentes resuelven de alguna manera estos problemas, en general lo hacen en forma idiosincrática y propietaria. En consecuencia, las herramientas no son interoperables. Las propuestas de solución existentes (por ejemplo, FAMIX) deben implementar meta-modelos que no son los de UML [DDT99].

Ya se ha dicho que UML no es en modo alguno un ADL en el sentido usual de la expresión. Los manuales contemporáneos de UML carecen del concepto de estilo y sus definiciones de “arquitectura” no guardan relación con lo que ella significa en el campo de los ADLs [BRJ99] [Lar03]. Sin embargo, debido al hecho de que constituye una herramienta de uso habitual en modelado, importantes autoridades en ADLs, siguiendo a Nenad Medvidovic, han investigado la posibilidad de utilizarlo como metalenguaje para implementar la semántica de al menos dos ADLs, que son C2 SADL y Wright [RMR+98]. También se ha afirmado que el meta-modelo de UML puede constituir un equivalente a la ontología arquitectónica de Acme.

Como quiera que sea, al no tipificar UML como un lenguaje homólogo a los ADLs que aquí se tratan no analizaremos sus rasgos y prestaciones en los términos que hasta aquí han sido habituales. Baste decir que, ya sea que se oponga o complemente al resto de las herramientas descriptivas, existen innumerables recursos alrededor de él para modelar sistemas en la plataforma de referencia. Con Visio Enterprise Architect de Microsoft se pueden crear, de hecho, los nueve tipos de diagramas de UML 1.*.

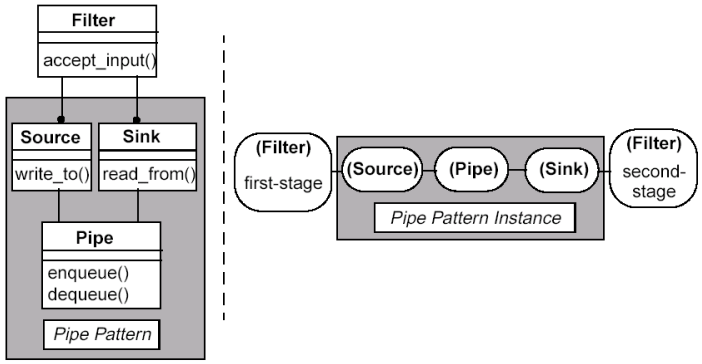


Fig. 7 – Tubería y filtro en UML

A fin de vincular el modelado basado en objetos con las problemáticas de ADLs y estilo que son propias de este estudio, nos ha parecido oportuno incluir una representación en notación OMT del modelo de referencia de tubería y filtros. En esta imagen, tomada de [MKM+96] se incorpora también la idea de concebir la tubería no sólo como una entidad de diseño de primer orden, sino como un patrón de diseño. Anticipamos de esta manera una cuestión que será tratada al cierre de este estudio, la cual concierne a la relación entre los ADLs y el campo de los patrones de diseño, en plena expansión a comienzos del siglo XXI.

UniCon

UniCon (Universal Connector Support) es un ADL desarrollado por Mary Shaw y otros [SDK+95]. Proporciona una herramienta de diseño para construir configuraciones ejecutables basadas en tipos de componentes, implementaciones y "conexiones expertas" que soportan tipos particulares de conectores. UniCon se asemeja a Darwin en la medida en que proporciona herramientas para desarrollar configuraciones ejecutables de caja negra y posee un número fijo de tipos de interacción, pero el modelo de conectores de ambos ADLs es distinto.

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

El siguiente ejemplo muestra un modelo completo de tubería y filtro en UniCon, definiendo un primer filtro para procesamiento primario y un segundo para eliminación de vocales:

```
COMPONENT System
       INTERFACE is
```

```
            TYPE Filter
            PLAYER input IS StreamIn
                  SIGNATURE (“line”)
                  PORTBINDING (stdin)
            END input
            PLAYER output IS StreamOut
                  SIGNATURE (“line”)
                  PORTBINDING (stdout)
            END output
            PLAYER error IS StreamOut
                  SIGNATURE (“line”)
                  PORTBINDING (stderr)
            END error
      END INTERFACE
      IMPLEMENTATION IS
            USES C INTERFACE CatFile
            USES R INTERFACE RemoveVowels
            USES P PROTOCOL Unix-pipe
            BIND input TO C.input
            BIND output TO R.output
            BIND C.error TO error
            BIND R.error TO error
            BIND P.err to error
            CONNECT C.output TO P.source
            CONNECT P.sink to R.input
      END IMPLEMENTATION
END System
COMPONENT CatFile
      INTERFACE is
            TYPE Filter - - como antes sin binding a puertos
      END INTERFACE
      IMPLEMENTATION IS
            VARIANT CatFile IN “catfile”
                  IMPLTYPE (Executable)
                  END CatFile
      END IMPLEMENTATION
END CatFile
COMPONENT RemoveVowels
      INTERFACE is
            TYPE Filter - - como antes sin binding a puertos
      END INTERFACE
      IMPLEMENTATION IS
            VARIANT RemoveVowels IN “remove”
            IMPLTYPE (Executable)
            END RemoveVowels
      END IMPLEMENTATION
END RemoveVowels
CONNECTOR Unix-pipe
      PROTOCOL IS
            TYPE Pipe
            ROLE source IS source
                  MAXCONNS (1)
                  END source
            ROLE sink IS sink
                  MAXCONNS (1)
                  END sink
            ROLE err IS sink
                  MAXCONNS (1)
                  END err
            END PROTOCOL
      IMPLEMENTATION IS
            BUILTIN
      END IMPLEMENTATION
```

```
END Unix-Pipe
```

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

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

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

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

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

Interfaz gráfica – UniCon soporta notación gráfica.

Generación de código – UniCon genera código C mediante el procedimiento de asociar elementos arquitectónicos a construcciones de implementación, que en este caso serían archivos que contienen código fuente. Sin embargo, habría algunos problemas con esta forma de implementación, dado que presuponer que esa asociación vaya a ser siempre uno-a-uno puede resultar poco razonable. Después de todo, se supone que los ADLs deben describir los sistemas a un nivel de abstracción más elevado que el que es propio del código fuente. Tampoco hay garantía que los módulos de código fuente implementen la conducta deseada o que se pueda seguir el rastro de futuros cambios en esos módulos hasta la arquitectura y viceversa.

Observaciones – UniCon (igual que Darwin) carece de la capacidad de definir nuevos tipos, soportando sólo una amplia variedad de tipos predefinidos. Los tipos de componente son definidos por enumeración, no siendo posible definir sub-tipos y careciendo por lo tanto de capacidad de evolución. Una ejemplificación razonable del caso testigo del estilo de tubería y filtros, a pesar de la simplicidad inherente a su estructura y de estar aquí tratándose de lenguajes descriptivos que deberían referirse a un elevado nivel de abstracción involucraría varias páginas de código; por esta razón omito el ejemplo, remitiendo a la documentación del manual de referencia del lenguaje que se encuentra en http://www-2.cs.cmu.edu/People/UniCon/reference-manual/Reference_Manual_1.html.

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

Weaves

Propuesto alrededor de 1991 [GR91], Weaves soporta la especificación de arquitecturas de flujo de datos. En particular, se especializa en el procesamiento en tiempo real de grandes volúmenes de datos emitidos por satélites meteorológicos. Es un ADL acaso demasiado ligado a un dominio específico, sobre el cual no hay abundancia de documentación disponible.

Wright

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

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

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

Estilos - En Wright se introduce un vocabulario común declarando un conjunto de tipos de componentes y conectores y un conjunto de restricciones. Cada una de las restricciones declaradas por un estilo representa un predicado que debe ser satisfecho por cualquier configuración de la que se declare que es miembro del estilo. La notación para las restricciones se basa en el cálculo de predicados de primer orden. Las restricciones se pueden referir a los conjuntos de componentes, conectores y *attachments*, a las puertos y a las computaciones de un componente específico y a los roles y ligamentos (*glue*) de un conector particular. Es asimismo posible definir sub-estilos que heredan todas las restricciones de los estilos de los que derivan. No existe, sin embargo, una manera de verificar la conformidad de una configuración a un estilo canónico estándar.

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

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

Análisis y verificación automática – Al basarse en CSP como modelo semántico, Wright permite analizar los conectores para verificar que no haya *deadlocks*. Wright define verificaciones de consistencia que se aplican estáticamente, y no mediante simulación.

Esto lo hace más confiable que, por ejemplo, Rapide. También se puede especificar una forma de compatibilidad entre un puerto y un rol, de modo que se pueda cambiar el proceso de un puerto por el proceso de un rol sin que la interacción del conector detecte que algo ha sido modificado. Esto permite implementar relaciones de refinamiento entre procesos. En Wright existen además recaudos (basados en teoremas formales relacionados con CSP) que garantizan que un conector esté libre de *deadlocks* en todos los escenarios posibles. Esta garantía no se obtiene directamente sobre Wright, sino implementando un verificador de modelos comercial para CSP llamado FDR. Herramientas complementarias gene-ran datos de entrada para FDR a partir de una especificación Wright. Cualquier herramienta de análisis o técnica que pueda usarse para CSP se puede utilizar también para especificaciones en Wright.

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

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

El siguiente ejemplo en código Wright correspondiente a nuestro caso canónico de tubería y filtros ilustra la sintaxis para el estilo correspondiente y la remisión al proceso de verificación (no incluido). Presento primero la notación básica de Wright, la más elemental de las disponibles:

```
Style DataFiles
Component DataFile1
       Port File = Action [] Exit
       where {
              Exit = close -> Tick
              Action = read -> Action [] write -> Action
       }
       Computation = ...
       Component DataFile2
              Port File = Action [] Exit
              where {
                     Exit = close -> Tick
                     Action = read -> File [] write -> File
              }
              Computation = ...
       Connector Pipe
              Role Writer = write -> Writer |~| close -> Tick
              Role Reader = DoRead |~| ExitOnly
              where {
                     DoRead = read -> Reader [] readEOF -> ExitOnly
                     ExitOnly = close -> Tick
              }
              Glue = ...
End Style
```

La siguiente pieza de código, más realista, incluye la simbología *hardcore* de Wright según Robert Allen y David Garlan [AG96]:

```
Style pipe-and-filter
```

```
Interface Type DataInput = (read Æ → (data?x → DataInput
              [] end-of-data → close → √))
              ⌈⌉ (close →√)
Interface Type DataOutput = write → DataOutput ⌈⌉ close →√
       Connector Pipe
              Role Source = DataOutput
              Role Sink = DataInput
              Glue = Buf<>
                     where
              Buf<> = Source.write?x → Buf<x> [] Source.close → Closed<>
              Buf<s<x>> = Source.write?y → Buf<y>s<x>>
                     [] Source.close → Closeds<x>
                     [] Sink.read → Sink.data!x → Bufs
                     [] Sink.close → Killed
              Closed<s<x>> = Sink.read → Sink.data!x → Closed<s>
                     [] Sink.close → √
                     Closed<> = Sink.read → Sink.end-of-data → Sink.close → √
                     Killed = Source.write → Killed [] Source.close → √
              Constraints
                     ∀ c : Connectors • Type(c) = Pipe
                     ∀ c : Components • Filter(c)
                     where
                     Filter(c:Component) = ∀ p : Ports(c) • Type(p) = DataInput
                     ∨ Type(p) = DataOutput
End Style
```

La imagen ilustra la representación gráfica correspondiente al estilo:

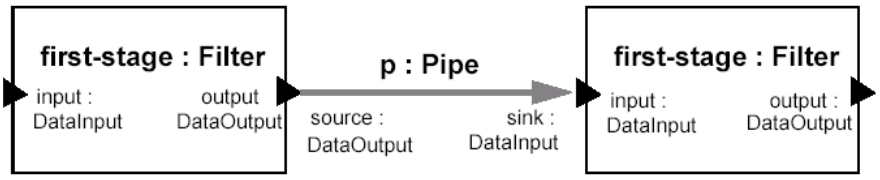


Fig. 8 – Diagrama correspondiente al código en Wright

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

Disponibilidad de plataforma – Wright es en principio independiente de plataforma y es un lenguaje formal, antes que una herramienta paquetizada. Debido a que se basa en CSP (Communicating Sequential Process), cualquier solución que pueda tratar código CSP en plataforma Windows es apta para obtener ya sea una visualización del modelo o verificar la ausencia de deadlocks o la consistencia del modelo. El código susceptible de ser manejado por herramientas de CSP académicas o comerciales requiere tratamiento previo por un módulo de Wright que por el momento existe sólo para Linux o SunOS; pero una vez que se ha generado el CSP (lo cual puede hacerse manualmente insertando unos pocos *tags*) se puede tratar en Windows con cualquier solución ligada a ese lenguaje, ya sea FDR o alguna otra. CSP no está ligado a plataforma alguna, y fue elaborado a

mediados de la década de 1970 por Sir Tony Hoare, actualmente Senior Researcher en Microsoft Research, Cambridge.

Modelos computacionales y paradigmas de modelado

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

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

Un concepto importante que surge de haber considerado este abanico de posibilidades, es que no todo lo que tiene que ver con diseño requiere compulsivamente notaciones y modelos de objeto. Considerando los ADLs en su conjunto, habrá podido comprobarse que la tradición de modelado OO, como la que se plasma a partir de [RBP+91], juega en este campo un papel muy modesto, si es que juega alguno; la reciente tendencia a redefinir los ADLs en términos de mensajes, servicios, integración, XML Schemas, xADL, SOAP y sus derivaciones no hará más que consolidar ese estado de cosas.

En el campo del modelado arquitectónico, UML/OOP tiende a ser antagónico a la modalidad “estructural” representada por los ADLs. Grandes proyectos recientes, como el del equipo de JPL que está definiendo el *framework* de Mission Data Systems para la NASA, ha decidido hace pocos meses reemplazar UML por ADLs basados en XML [MMM03]. Correlativamente, como bien destaca Paul Clements en uno de sus *surveys*, muy pocos ADLs pueden manejar conceptos básicos en el modelado OO tales como herencia de clases y polimorfismo [Cle96]. De más está decir que ningún ADL a la fecha permite incorporar inflexiones más complejas y sutiles como programación orientada a aspectos, otro tópico caliente en lo que va del siglo [MPM+99]. En la generalidad de los casos, tampoco es relevante que así sea.

ADLs en ambientes Windows

En este estudio se ha podido comprobar que existe un amplio repertorio de implementaciones de ADLs y sus herramientas colaterales de última generación en plataforma Windows. Específicamente señalo Saage, un entorno para diseño arquitectó-

nico que utiliza el ADL de C2SADEL y que requiere Visual J++ (o Visual J#), COM y eventualmente Rational Rose (o DRADEL); también existen extensiones de Visio para C2, xADL y naturalmente UML. Visio para xADL se utiliza como la interfaz usuario de preferencia para *data binding* de las bibliotecas de ArchEdit para xADL. En relación con xADL también se puede utilizar en Windows ArchStudio para modelado en gran escala, o Ménage para mantenimiento y gestión de arquitecturas cambiantes en tiempo de ejecución.

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

Algunas herramientas son nativas de Win32, otras corren en CygWin, Interix o algún Xserver, otras necesitan JRE. Mientras los ADLs de elección estén vivos y mantengan su impulso no faltarán recursos y complementos, a menos que se escoja una plataforma muy alejada de la corriente principal para ejecutar el software de modelado. No necesariamente habrá que lidiar entonces manualmente con notaciones textuales que quizá resulten tan complejas como las de los propios lenguajes de implementación. Si bien el trabajo del arquitecto discurre a un nivel de abstracción más elevado que la del operario que escribe código, en ninguna parte está escrito que vaya a ser más simple. Por el contrario, está claro que desde el punto de vista del modelado con ADLs, el arquitecto debe dominar no sólo una práctica sino también una teoría, o diversas teorías convergentes.

Pero los ambientes gráficos son sólo una parte de la cuestión, un capítulo que dudosamente merezca ocupar el foco de una estrategia arquitectónica o constituirse en su parámetro decisivo; poseen la misma importancia relativa y circunstancial que los nombres o los URLs de especificaciones que hoy pueden estar y mañana desaparecer sin dar explicaciones. El núcleo de la relevancia y conveniencia de los ADLs tiene que ver más bien con las diferentes cuestiones formales involucradas en los distintos escenarios y estilos, y eventualmente con la relación entre el modelado arquitectónico y el inmenso campo, en plena ebullición, de los patrones arquitectónicos en general y los patrones de diseño en particular.

ADLs y Patrones

A primera vista, daría la impresión que la estrategia arquitectónica de Microsoft reposa más en la idea de patrones y prácticas que en lenguajes de descripción de diseño; pero si

se examina con más cuidado el concepto de patrones se verá que dichos lenguajes engranan clara y armónicamente en un mismo contexto global. El concepto de patrón fue acuñado por Christopher Alexander entre 1977 y 1979. De allí en más fue enriqueciéndose en una multitud de sentidos, abarcando un conjunto de prácticas para capturar y comunicar la experiencia de los diseñadores expertos, documentar *best practices*, establecer formas recurrentes de solución de problemas comunes, etcétera.

Las clasificaciones usuales en el campo de los patrones reconocen habitualmente que hay diferentes clases de ellos; por lo común se hace referencia a patrones de análisis, organizacionales, procedurales y de diseño. Esta última categoría tiene que ver con recurrencias específicas de diseño de software y sistemas, formas relativamente fijas de interacción entre componentes, y técnicas relacionadas con familias y estilos. En este sentido, es evidente que los ADLs denotan una clara relación con los patrones de diseño, de los que puede decirse que son una de las formas posibles de notación. Como en el caso de las heurísticas y recetas de Acme, muchas veces esos patrones son explícitos y el ADL opera como una forma regular para expresarlos en la descripción arquitectónica.

La correspondencia entre patrones, ADLs y estilos, sin embargo, no está establecida de una vez y para siempre porque no existe ni una taxonomía uniforme, ni una especificación unívoca que defina taxativamente cuántas clases de patrones y cuántos niveles existen en ingeniería, arquitectura y diseño de software desde que se concibe conceptualmente un sistema hasta que se lo programa. En la percepción de Jørgen Thelin, quien a su vez se funda en la nomenclatura y tipificación de Martin Fowler, por ejemplo, distintos niveles en los proyectos de desarrollo se vincularían con otras tantas clases de patrones: el diseño con patrones de código, el análisis con los patrones de modelos y los estilos arquitectónicos con los patrones de arquitectura [The03]. Pero aunque cada escuela y cada autor organiza la articulación del campo a su manera y se sirve de denominaciones idiosincráticas, sin duda es legítimo plantear una relación entre patrones y lenguajes de descripción de arquitectura y tratar de esclarecerla.

Los ADLs mantienen asimismo una frontera difusa con lo que desde Alexander en adelante se llamaron lenguajes de patrones, que se definen como sistemas de patrones organizados en una estructura o plantilla que orienta su aplicación [Ale79] [AIS+77]. Ambas ideas se consolidaron en una nutrida bibliografía, en la cual la obra de la llamada "Banda de los Cuatro" ha alcanzado una fama legendaria [Gof95]. En algunas contadas ocasiones, los lenguajes de patrones fueron reformulados como ADLs y también viceversa [Shaw96]. Recientemente se ha propuesto un nuevo estilo arquitectónico "de nueva generación", ABAS (Attribute-Based Architectural Style) [KC99], que explícitamente se propone la convergencia entre la arquitectura de alto nivel expresada en los ADLs y en los estilos con el *expertise*, las *best practices,* los *building blocks* y los patrones decantados en la disciplina de diseño de aplicaciones.

Un estudio esencial de Mary Shaw y Paul Clements analiza la compleja influencia de la teoría y la práctica de los patrones sobre los ADLs [SC96]. Estos autores consideran que los ADLs han sido propios de la comunidad de arquitectos de software, mientras que los patrones de diseño y sus respectivos lenguajes han prosperado entre los diseñadores de software, particularmente entre los grupos más ligados a la orientación a objetos. Naturalmente, ambas comunidades se superponen en más de un respecto. En lo que

respecta a la relación entre arquitectura y diseño, las discusiones mantenidas en el seno de OOSPLA y en otros foros han consensuado que los diseñadores basados en patrones operan a niveles de abstracción más bajo que el de los arquitectos, pero por encima del propio de los programadores. Por otra parte, Buschmann [BMR+96] ha documentado patrones utilizados como estilos arquitectónicos regidos por ADLs. Shaw y Clements concluyen su análisis alegando que los ADLs pueden beneficiarse incorporando elementos de tipo análogo a los patrones en las secciones que se refieren a estilos, plantillas y reglas de diseño. A similares conclusiones llegan Robert T. Monroe, Drew Kompanek, Ralph Melton y David Garlan [MKM+96].

Al lado de los ADLs, existen otros recursos para abordar la cuestión de los patrones de diseño. La estrategia arquitectónica de Microsoft contempla niveles de abstracción que hoy en día se resuelven, por ejemplo, mediante Logidex para .Net de LogicLibrary, el cual se encuentra en http://www.logiclibrary.com/logidex.htm. Este es un *engine* que tipifica más como un *software development asset* (SDA) que como una entidad semejante a un ADL. Un *asset* de este tipo incluye componentes, servicios, frameworks, artefactos asociados con el ciclo de vida de un desarrollo de software, patrones de diseño y documentación de *best practices*. Este paquete en particular es más un producto para desarrolladores arquitectónicamente orientados que para arquitectos que prefieran situarse en una tesitura más abstracta y alejada del código. En futuras versiones de VisualStudio .Net se prevé la incorporación de otros recursos, algunos de ellos construidos en torno de la edición Architect y otros desarrollados por terceras partes. Los ADLs complementan a estos *assets* y herramientas, lo mismo que lo seguirán haciendo las soluciones de modelado relacionadas con la tradición de UML.

Conclusiones

Nuestra experiencia en conferencias de industria indica que cuando se presentan temas como los lenguajes de descripción arquitectónica y eventualmente los estilos de arquitectura ante audiencias de arquitectos, son muy pocos quienes han oído hablar de semejantes tópicos, y muchos menos aún quienes poseen alguna experiencia de modelado genuinamente arquitectónico. Por lo general tiende a confundirse este modelado abstracto con el diseño concreto de la solución a través de UML, o con la articulación de patrones. Este es un claro indicador del hecho de que la arquitectura de software no ha sabido comunicar su propia función mediadora entre el requerimiento por un lado y el diseño y la implementación por el otro. El presente estudio ha procurado describir herramientas disponibles para el desempeño de esa función. La idea no ha sido examinar productos concretos de modelado con ADL, sino referir problemáticas formales involucradas en esa clase de representación.

De más está decir que los ADLs son convenientes pero no han demostrado aún ser imprescindibles. Numerosas piezas de software, desde sistemas operativos a aplicaciones de misión crítica, se realizaron echando mano de recursos de diseño ad hoc, o de formalismos incapaces de soportar un *round trip* evolutivo, expresar un estilo, implementar patrones o generar código. Aunque algunos se perdieron en el camino o quedaron limitados a experiencias en dominios específicos, y aunque en la era de las arquitecturas orientadas a servicios se encuentran en estado de fluidez y transición, los

ADLs, nacidos hace más de una década, han llegado para quedarse. La demanda que los ha generado se ha instalado como un tópico permanente de la arquitectura de software, y por eso nos ha parecido útil elaborar este examen.

Referencias bibliográficas

[Abd00] Aynur Abdurazik. "Suitability of the UML as an Architecture Description Language with applications to testing". Reporte ISE-TR-00-01, George Mason University. Febrero de 2000.

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

[Ale79] Christopher Alexander. *The Timeless Way of Building*. Nueva York, Oxford University Press, 1979.

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

[AW99] David Akehurst y Gill Waters. "UML deficiencies from the perspective of automatic performance model generation". *OOSPLA'99 Workshop on Rigorous Modeling and Analysis with the UML: Challenges and Limitations*, Denver, http://www.cs.kent.ac.uk/pubs/1999/899/content.pdf, Noviembre de 1999.

[AIS+77] Christopher Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King y S. Angel. *A Pattern Language*. Nueva York, Oxford University Press, 1977.

[BEJ+93] Pam Binns, Matt Englehart, Mike Jackson y Steve Vestal. "Domain-Specific Software Architectures for Guidance, Navigation, and Control". *Technical report*, Honeywell Technology Center, http://www-ast.tds-gn.lmco.com/arch/arch-ref.html, 1993.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad y Michael Stahl. *Pattern-Oriented Software Architecture: A System of Patterns*. Nueva York, Wiley, 1996.

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

[Cle95] Paul C. Clements. "Formal Methods in Describing Architectures". En *Proceedings of the Workshop on Formal Methods and Architecture*, Monterey, California, 1995.

[DDT99] Serge Demeyer, Stéphane Ducasse y Sander Tichelaar. "Why FAMIX and not UML?: UML Shortcomings for coping with round-trop engineering". *UML'99 Conference Proceedings*, LNCS Series, Springer Verlag, 1999.

[DH01] Eric Dashofy y André van der Hoek. "Representing Product Family Architectures in an Extensible Architecture Description Language". En *Proceedings of the International Workshop on Product Family Engineering* (PFE-4), Bilbao, España, 2001.

[Dou00] Bruce Powel Douglas. "UML – The new language for real-timed embedded systems". http://wooddes.intranet.gr/papers/Douglass.pdf, 2000.

[Fie00] Roy Fielding. "Architectural styles and the design of network-based software architectures". Tesis de Doctorado. University of California at Irvine, 2000.

[Fux00] Ariel Fuxman. "A survey of architecture description languages". Febrero de 2000. Http://citeseer.nj.nec.com/fuxman00survey.html.

[GAD+02] Yann-Gaël Guéhéneuc, Hervé Albin-Amiot, Rémi Douence y Pierre Cointe. "Bridging the gap between modeling and programming languages". Reporte técnico, Object Technology International, Julio de 2002.

[GAO94] David Garlan, Robert Allen y John Ockerbloom. "Exploiting style in architectural design environments". En Proceedings of the SIGSOFT '94 Symposium on Foundations of Software Engineering. Nueva Orleans, diciembre de 1994.

[GarS/f] David Garlan. "Software architectures". Presentación en transparencias, http://www.sti.uniurb.it/events/sfm03sa/slides/garlan-B.ppt, sin fecha.

[GB99] Neil Goldman y Robert Balzer. "The ISI visual design editor". En *Proceedings of the 1999 IEEE International Conference on Visual Languages*, Setiembre de 1999.

[Gli00] Martin Glinz. "Problems and deficiencies of UML as a requirements specification language". En *Proceedings of the 10th International Workshop of Software Specification and Design* (IWSSD-10), pp. 11-22, San Diego, noviembre de 2000.

[GMW00] David Garlan, Robert Monroe y David Wile. "Acme: Architectural description of component-based systems". *Foundations of Component-Based Systems*, Gary T. Leavens y Murali Sitaraman (eds), Cambridge University Press, pp. 47-68, 2000.

[GoF95] Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GR91] Michael Gorlick y Rami Razouk. "Using Weaves for Software Construction and Analysis". En *13th International Conference on Software Engineering*, Austin, Mayo de 1991.

[HNS99] Christine Hofmeister, Robert Nord y Dilip Soni. "Describing Software Architecture with UML". En *Proceedings of the First Working IFIP Conference on Software Architecture*, IEEE Computer Society Press, pp. 145-160, San Antonio, Febrero de 1999.

[KC94] Paul Kogut y Paul Clements. "Features of Architecture Description Languages". Borrador de un CMU/SEI Technical Report, Diciembre de 1994.

[KC95] Paul Kogut y Paul Clements. “Feature Analysis of Architecture Description Languages”. En *Proceedings of the Software Technology Conference (STC’95)*, Salt Lake City, Abril de 1995.

[KK99] Mark Klein y Rick Kazman. “Attribute-Based Architectural Styles”, *Technical Report, CMU/SEI-99-TR-022, ESC-TR-99-022*, Octubre de 1999.

[KM95] Paul Kogut y Robert May. “Features of Architecture Description Languages”. *Software Technology Conference*, Salt Lake City, Abril de 1995.

[KroS/f] John Krogstie. “UML, a good basis for the development of models of high quality?”. Andersen Consulting Noruega & IDI, NTNU, http://dataforeningen.no/ostlandet/metoder/krogstie.pdf, sin fecha.

[Lar03] Craig Larman. *UML y Patrones*. 2ª edición, Madrid, Prentice Hall, 2003.

[LV95] David Luckham y James Vera. “An Event-Based Architecture Definition Language”. *IEEE Transactions on Software Engineering*, pp. 717-734, Setiembre de 1995.

[MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach y Jeff Kramer. “Specifying distributed software architectures”. En *Proceedings of the Fifth European Software Engineering Conference, ESEC’95*, Setiembre de 1995.

[Med96] Neno Medvidovic. “A classification and comparison framework for software Architecture Description Languages”. Technical Report UCI-ICS-97-02, 1996.

[MK96] Jeff Magee y Jeff Kramer. “Dynamic structure in software architectures”. En *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 3–14, San Fransisco, Octubre de 1996.

[MKM+96] Robert Monroe, Drew Kompanek, Ralph Melton, David Garlan. “Stylized architecture, design patterns, and objects”. *Research Report*. Setiembre de 1996.

[MMM03] Nenad Medvidovic, Sam Malek, Marija Mikic-Rakic. “Software architectures and embedded systems”. Presentación en *Workshop on Software Engineering for Embedded Systems: From Requirements to Implementation*. Chicago, Illinois, 24 al 26 de Setiembre de 2003.

[Mon98] Robert Monroe. “Capturing software architecture design expertise with Armani”. *Technical Report CMU-CS-163*, Carnegie Mellon University, Octubre de 1998.

[MPM+99] A. Navasa, M. A. Pérez, J. M. Murillo y J. Hernández. “Aspect oriented software architecture: A structural perspective”. Reporte del proyecto CICYT, TIC 99-1083-C2-02, 1999.

[OOS99] OOSPLA’99, Workshop #23: “Rigorous modeling and analysis with UML: Challenges and Limitations”. Denver, Noviembre de 1999.

[Par76] David Parnas. “On the Design and Development of Program Families.” *IEEE Transactions on Software Engineering* SE-2, pp. 1-9, Marzo de 1976.

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

[Rey04] Carlos Reynoso. "Estilos y patrones en la estrategia de arquitectura de Microsoft". [Ref**], Marzo de 2004.

[RMR+98] Jason E. Robbins, Nenad Medvidovic, David F. Redmiles y David S. Rosenblum. "Integrating Architecture Description Languages with a Standard Design Method." En *Proceedings of the 20th International Conference on Software Engineering* (ICSE'98), pp. 209-218, Kyoto, Japón, 19 al 25 de Abril de 1998.

[SC96] Mary Shaw y Paul Clements. "How should patterns influence Architecture Description Languages?". *Working Paper for DARPA EDCS Community*, 1996.

[Sch00] Klaus-Dieter Schewe. "UML: A modern dinosaur? – A critical analysis of the Unified Modelling Language". En H. Kangassalo, H. Jaakkola y E. Kawaguchi (eds.), *Proceedings of the 10th European-Japanese Conference on Information Modelling and Knowledge Bases*, Saariselk, Finlandia, 2000.

[SDK+95] Mary Shaw, Robert DeLine, Daniel Klein, Theodore Ross, David Young y Gregory Zelesnik. "Abstractions for Software Architecture and Tools to Support Them". *IEEE Transactions on Software Engineering*, pp. 314-335, Abril de 1995.

[SG94] Mary Shaw y David Garlan. "Characteristics of Higher-Level Languages for Software Architecture". *Technical Report CMU-CS-94-210*, Carnegie Mellon University, Diciembre de 1994.

[Shaw94] Mary Shaw. "Patterns for software architecture". *First Annual Conference on the Pattern Languages of Programming*, 1994.

[Shaw96] Mary Shaw. "Some patterns for software architecture". Vlissides, Coplien & Kerth (eds.), *Pattern Languages of Program Design, Vol. 2*, pp. 255-269, Addison-Wesley, 1996.

[SG95] Mary Shaw y David Garlan. "Formulations and Formalisms in Software Architecture". Springer-Verlag, *Lecture Notes in Computer Science*, Volumen 1000, 1995.

[StöS/f] Harald Störrle. "Turning UML subsystems into architectural units". Reporte, Ludwig-Maximilians-Universität München, http://www.pst.informatik.uni-muenchen.de/personen/stoerrle/Veroeffentlichungen/PosPaperICSEFormat.pdf, sin fecha.

[Tem01] Theodor Tempelmeier. "Comments on Design Patterns for Embedded and Real-Time Systems". En: A. Schürr (ed.): *OMER-2 ("Object-Oriented Modeling of Embedded Realtime systems") Workshop Proceedings*. Mayo 9-12, 2001, Herrsching, Alemania. Bericht Nr. 2001-03, Universität der Bundeswehr München, Fakultät für Informatik, Mayo de 2001.

[Tem99] Theodor Tempelmeier. “UML is great for Embedded Systems – Isn’t it?” En: P. Hofmann, A. Schürr (eds.): *OMER (“Object-Oriented Modeling of Embedded Realtime systems”) Workshop Proceedings*. Mayo 28-29, 1999, Herrsching (Ammersee), Alemania. Bericht Nr. 1999-01, Universität der Bundeswehr München, Fakultät für Informatik, Mayo de 1999.

[The03] Jørgen Thelin. “A comparison of service-oriented, resource-oriented, and object-oriented architecture styles”. Presentación de Cape Clear Software, 2003.

[TMA+S/f] Richard Taylor, Nenad Medvidovic, Kennet Anderson, James Whitehead Jr, Jason Robbins, Kari Nies, Peyman Oreizy y Deborah Dubrow. “A component- and message-based architectural style for GUI software”. Reporte para el proyecto F30602-94-C-0218, Advanced Research Projects Agency, sin fecha.

[Ves93] Steve Vestal. “A cursory overview and comparison of four Architecture Description Languages”. *Technical Report*, Honeywell Technology Center, Febrero de 1993.

[Wolf97] Alexander Wolf. “Succeedings of the Second International Software Architecture Workshop” (ISAW-2). *ACM SIGSOFT Software Engineering Notes*, pp. 42-56, enero de 1997.

Métodos Heterodoxos en Desarrollo de Software

Contenidos:

Métodos Heterodoxos en Desarrollo de Software

Versión 1.0 – Abril de 2004

Carlos Reynoso – UNIVERSIDAD DE BUENOS AIRES

Revisión técnica de Nicolás Kicillof – Universidad de Buenos Aires

Introducción

> Los métodos Ágiles, tales como Lean Development, eXtreme Programming y Adaptive Software Development, son estrategias de desarrollo de software que promueven prácticas que son adaptativas en vez de predictivas, centradas en la gente o en los equipos, iterativas, orientadas hacia prestaciones y hacia la entrega, de comunicación intensiva, y que requieren que el negocio se involucre en forma directa. Comparando esos atributos con los principios fundacionales de MSF, se encuentra que MSF y las metodologías ágiles están muy alineadas tanto en los principios como en las prácticas para el desarrollo de software en ambientes que requieren un alto grado de adaptabilidad.
>
> - Documentación de Microsoft Solutions Framework 3.0 [MS03]

Ni duda cabe que a finales de la década de 1990 dos grandes temas irrumpieron en las prácticas de la ingeniería de software y en los métodos de desarrollo: el diseño basado en patrones y los métodos ágiles. De estos últimos, el más resonante ha sido la Programación Extrema (XP), que algunos consideran una innovación extraordinaria y otros creen cínica [Rak01], extremista [McC02], falaz [Ber03] o perniciosa para la salud de la profesión [Kee03]. Patrones y XP se convirtieron de inmediato en *hypes* de discusión masiva en la industria y de fuerte presencia en la Red. Al primero de esos temas el mundo académico lo está tratando como un asunto respetable desde hace un tiempo; el otro recién ahora se está legitimando como tópico serio de investigación. La mayor parte de los documentos proviene todavía de los practicantes, los críticos y los consultores que impulsan o rechazan sus postulados. Pero el crecimiento de los métodos ágiles y su penetración ocurre a un ritmo pocas veces visto en la industria: en tres o cuatro años, según el Cutter Consortium, el 50% de las empresas define como "ágiles" más de la mitad de los métodos empleados en sus proyectos [Cha04].

Los métodos ágiles (en adelante MAs) constituyen un movimiento heterodoxo que confronta con las metodologías consagradas, acordadas en organismos y apreciadas por consultores, analistas de industria y corporaciones. Contra semejante adversario, los MAs se expresaron a través de manifiestos y libros en tono de proclama, rehuyendo (hasta hace poco) toda especificación formal. El efecto mediático de esos manifiestos ha sido explosivo y ocasionó que la contienda entre ambas formas haya sido y siga siendo enconada. Lejos de la frialdad burocrática que caracteriza a las crónicas de los argumentos ortodoxos, se ha tornado común referirse a sus polémicas invocando metáforas bélicas y apocalípticas que hablan de "la batalla de los gurúes" [Tra02], "el

gran debate de las metodologías" [Hig01], "las guerras religiosas", "el fin del mundo" [McC02], un "choque cultural" [Cha04], "la venganza de los programadores" [McB02] o "la muerte del diseño" [Fow01].

Lo que los MAs tienen en común (y lo que de aquí en más obrará como una definición de los mismos) es su modelo de desarrollo incremental (pequeñas entregas con ciclos rápidos), cooperativo (desarrolladores y usuarios trabajan juntos en estrecha comunicación), directo (el método es simple y fácil de aprender) y adaptativo (capaz de incorporar los cambios). Las claves de los MAs son la velocidad y la simplicidad. De acuerdo con ello, los equipos de trabajo se concentran en obtener lo antes posible una pieza útil que implemente sólo lo que sea más urgente; de inmediato requieren *feedback* de lo que han hecho y lo tienen muy en cuenta. Luego prosiguen con ciclos igualmente breves, desarrollando de manera incremental. Estructuralmente, los MAs se asemejan a los RADs (desarrollo rápido de aplicaciones) más clásicos y a otros modelos iterativos, pero sus énfasis son distintivos y su combinación de ideas es única.

Si hubo una sublevación no fue inmotivada. Diversos estudios habían revelado que la práctica metodológica fuerte, con sus exigencias de planeamiento y sus técnicas de control, en muchos casos no brindaba resultados que estuvieran a la altura de sus costos en tiempo, complejidad y dinero. Investigaciones como la de Joe Nandhakumar y David Avison [NA99], en un trabajo de campo sobre "la ficción del desarrollo metodológico", denunciaban que las metodologías clásicas de sistemas de información "se tratan primariamente como una ficción necesaria para presentar una imagen de control o para proporcionar estatus simbólico" y que dichas metodologías son demasiado ideales, rígidas y mecanicistas para ser utilizadas al pie de la letra. Duane Truex, Richard Baskerville y Julie Travis [TBT00] toman una posición aún más extrema y aseguran que es posible que los métodos tradicionales sean "meramente ideales inalcanzables y 'hombres de paja' hipotéticos que proporcionan guía normativa en situaciones de desarrollo utópicas". En su reclamo de un desarrollo a-metódico, consideran que las metodologías estándares se basan en una fijación de objetivos pasada de moda e incorrecta y que la obsesión de los ingenieros con los métodos puede ser inhibidora de una adecuada implementación, tanto a nivel de sistemas como en el plano de negocios.

Pero también hay cientos de casos documentados de éxitos logrados con metodologías rigurosas. En ocasiones, el entusiasmo de los promotores de las revueltas parece poco profesional, como si su programa de crítica, muchas veces bien fundado, fuera de mayor importancia que su contribución positiva. También ellas, sin embargo, tienen su buen catálogo de triunfos. Algunos nombres respetados (Martin Fowler con el respaldo de Cutter Consortium, Dee Hock con su visión caórdica, Philippe Kruchten con su RUP adaptado a los tiempos que corren) e incluso Ivar Jacobson [Jac02] consideran que los MAs constituyen un aporte que no sería sensato dejar de lado. No habría que tratarlo entonces como si fuera el capricho pasajero de unos pocos *hackers* que han leído más arengas posmodernas de lo aconsejable.

En este documento se presentará una breve reseña de los MAs junto con una descripción sucinta de la situación y las tendencias actuales. No se pretende que el texto brinde una orientación operativa ni tampoco una evaluación de los métodos; se trata sólo de una visión de conjunto que tal vez ayude a comprender una de las alternativas existentes en la organización del proceso de desarrollo. Se ha procurado mantener una distancia crítica;

no se encontrarán aquí las expresiones que suelen proliferar en los textos y en la Red en las que se "define" a XP afirmando que es "un conjunto de valores, principios y prácticas para el desarrollo rápido de software *de alta calidad* que proporciona *el valor más alto* para el cliente *en el menor tiempo posible*". No es que la postura de este estudio sea equidistante, como procuran serlo, por ejemplo, los exámenes de Robert Glass [Gla01] o Barry Boehm [Boe02a]; sencillamente, la presente es una introducción analítica y no una tabla de evaluación.

Se podrá advertir en el cuerpo de este texto que las referencias comparativas y contextuales a los MAs apuntan muchas más veces a la ingeniería de software que a la arquitectura; la razón es que la elaboración metodológica de la arquitectura es reciente y se está formulando en el SEI y en otros organismos contemporáneamente al desarrollo de los MAs. Recién ahora, en otras palabras, la arquitectura de software está elaborando sus metodologías para el ciclo de vida; si éstas se asemejan más a las ágiles que a las ortodoxas es una pregunta importante que se habrá de responderse en documentos separados.

Una vez más, se examinará con detenimiento la relación de concordancia y complementariedad entre los métodos ágiles y los principios que articulan Microsoft Solutions Framework (MSF), un tema sustancial al que se ha dedicado una sección específica [pág. 55]. También se ha incluido un capítulo sobre el uso de patrones en MAs y otro sobre la respuesta crítica de la comunidad metodológica frente al avance arrollador de los nuevos métodos [pág. 65]. Otros documentos de esta serie detallarán algunos aspectos que aquí se desarrollan concisamente, tales como la relación entre los métodos ágiles con el paradigma de las ciencias de la complejidad y el caos.

La ortodoxia metodológica

Los MAs no surgieron porque sí, sino que estuvieron motivados (a) por una conciencia particularmente aguda de la crisis del software, (b) por la responsabilidad que se imputa a las grandes metodologías en la gestación de esa crisis y (c) por el propósito de articular soluciones.

Los organismos y corporaciones han desarrollado una plétora de estándares comprensivos que han ido jalonando la historia y poblando los textos de metodologías e ingeniería de software: CMM, Spice, BootStrap, TickIt, derivaciones de ISO9000, SDCE, Trillium [Wie98] [Wie99] [She97]. Algunos son verdaderamente métodos; otros, metodologías de evaluación o estimación de conformidad; otros más, estándares para metodologías o meta-modelos. Al lado de ellos se encuentra lo que se ha llamado una ciénaga de estándares generales o específicos de industria a los que se someten organizaciones que desean (o deben) articular sus métodos conforme a diversos criterios de evaluación, vehículos de selección de contratistas o marcos de referencia. Hay muchos *frameworks* y disciplinas, por cierto; pero lo primordial no es la proliferación de variedades, sino la sujeción de todos los ejemplares a unos pocos tipos de configuraciones o flujos posibles. Llamaremos aquí *modelos* a esas configuraciones.

Los modelos de los métodos clásicos difieren bastante en su conformación y en su naturaleza, pero exaltan casi siempre las virtudes del planeamiento y poseen un espíritu normativo. Comienzan con la elicitación y el análisis completo de los requerimientos del

usuario. Después de un largo período de intensa interacción con usuarios y clientes, los ingenieros establecen un conjunto definitivo y exhaustivo de rasgos, requerimientos funcionales y no funcionales. Esta información se documenta en forma de especificaciones para la segunda etapa, el diseño, en el que los arquitectos, trabajando junto a otros expertos en temas puntuales (como ser estructuras y bases de datos), generan la arquitectura del sistema. Luego los programadores implementan ese diseño bien documentado y finalmente el sistema completo se prueba y se despacha.

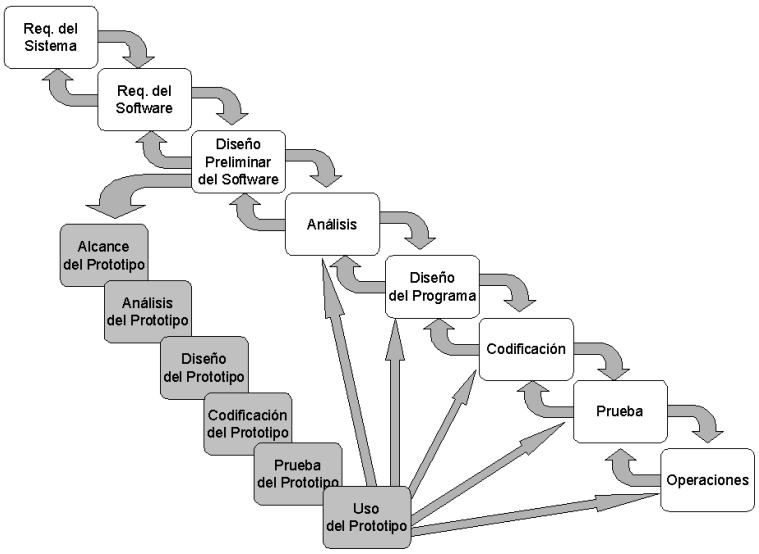
Modelo	Versión de origen	Características
Modelo en cascada	Secuencial: Bennington 1956 – Iterativo: Royce 1970 – Estándar DoD 2167-A	Secuencia de requerimiento, diseño del sistema, diseño de programa, codificación, pruebas, operación y mantenimiento
Modelo en cascada c/ fases superpuestas	Cf. McConnell 1996:143-144	Cascada con eventuales desarrollos en paralelo (Modelo Sashimi)
Modelo iterado con prototipado	Brooks 1975	Iterativo – Desarrollo incremental
Desarrollo rápido (RAD)	J. Martin 1991 – Kerr/Hunter 1994 – McConnell 1996	Modelo lineal secuencial con ciclos de desarrollo breves
Modelo V	Ministerio de Defensa de Alemania 1992	Coordinación de cascada con iteraciones
Modelo en espiral	Barry Boehm 1988	Iterativo – Desarrollo incremental. Cada fase no es lineal, pero el conjunto de fases sí lo es.
Modelo en espiral win-win	Barry Boehm 1998	Iterativo – Desarrollo incremental – Aplica teoría-W a cada etapa
Modelo de desarrollo concurrente	Davis y Sitaram 1994	Modelo cíclico con análisis de estado
Modelo de entrega incremental (*Staged delivery*)	McConnell 1996: 148	Fases tempranas en cascada – Fases ulteriores descompuestas en etapas

A fines del siglo XX había un abanico de tipos de procesos o modelos disponibles: el más convencional era el modelo en cascada o lineal-secuencial, pero al lado de él había otros como el modelo “V”, el modelo de construcción de prototipos, el de desarrollo rápido o RAD, el modelo incremental, el modelo en espiral básico, el espiral *win-win*, el modelo de desarrollo concurrente y un conjunto de modelos iterativos o evolutivos (basados en componentes, por ejemplo) que en un primer momento convivían apaciblemente con los restantes, aunque cada uno de ellos se originaba en la crítica o en la percepción de las limitaciones de algún otro. Algunos eran prescriptivos, otros descriptivos. Algunos de los modelos incluían iteraciones, incrementos, recursiones o bucles de retroalimentación; y algunos se preciaban también de rápidos, adaptables y expeditivos.

La tabla que se encuentra en estas páginas ilustra el repertorio de los métodos que podríamos llamar “de peso completo” o “de fuerza industrial”, que en sus variantes más rigurosas imponen una prolija especificación de técnicas correspondientes a los diferentes momentos del ciclo de vida. Es importante señalar que en esta tabla sólo se consignan las grandes formas de proceso metodológico, y no técnicas específicas que pueden aplicarse cualquiera sea esa forma, como las técnicas de ingeniería de software asistidas por computadoras (CASE), las herramientas de cuarta generación, las técnicas de modelado estático (Lai) o dinámico (Forrester), el modelo de software de sala limpia [MDL87] o los modelos evaluativos como el Capability Maturity Model (CMM) del SEI, que

popularmente se considera ligado a los modelos en cascada. De hecho ni CMM está sujeto a un modelo particular, ni constituye un modelo de proceso de ingeniería de software en el sentido estricto de la palabra; es un canon de evaluación que establece criterios para calificar la madurez de un proyecto en cinco niveles que van de lo caótico a la optimización continua [Pau02].

Ha habido una injusticia histórica con Winston Royce [Roy70] y su modelo en cascada: en su versión original, documentada en la imagen, era claramente iterativo [Ber03]. Por alguna razón pasó a la posteridad como la encarnación de una metodología secuencial.



El modelo original en cascada, basado en [PP03]

Hay unos cuantos modelos más o menos clásicos que no hemos considerado. Steve McConnell [McC96], por ejemplo, describe un modelo *code-and-fix*, un modelo en cascada con subproyectos, otro modelo en cascada con reducción de riesgo, un modelo de diseño sujeto a agenda y un modelo de prototipado evolutivo (el antepasado de Evo) que aquí hemos transferido al otro lado de la divisoria entre métodos pesados y ligeros. El tema de nuestro estudio, empero, no es la taxonomía de los modelos. La mejor referencia para las formas modélicas clásicas anteriores a 1990 sigue siendo el libro de Peter DeGrace y Leslie Stahl *Wicked problems, righteous solutions* [DS90], un verdadero catálogo de los paradigmas metodológicos principales en ingeniería de software. Lo importante aquí es que toda esta diversidad se va visto subsumida de golpe en un solo conjunto. Por diversas razones, legítimas o espurias, todo comenzó a verse bajo una nueva luz. Cuando estas cosas suceden no cabe sino hablar de una revolución.

Orígenes de la heterodoxia

No hace falta comulgar con las premisas más extremas de los MAs para encontrar aspectos cuestionables en todos y cada uno de los modelos clásicos. Aun los manuales introductorios de ingeniería de software incluyen la lista de limitaciones bien conocidas de la mayor parte de ellos, con el modelo en cascada, lejos, como el más castigado. Se le objeta su rigidez, su obstinación por exigir una estipulación previa y completa de los requerimientos, su modelo inapropiado de correcciones, su progresivo distanciamiento de la visión del cliente y su morosidad para liberar piezas ejecutables. En los últimos dos o

tres años de la década de 1990, las críticas a la metodología convencional aumentaron en intensidad y detonaron por todas partes.

Aunque la gestación de la rebelión fue larga y tortuosa, pareció que de pronto, para muchos, los modelos consagrados se tornaron inadmisibles. Circularon también unas cuantas evaluaciones que revelaron serias fallas en proyectos regidos por ellos, y se refrescó la idea de la crisis del software, que en realidad es un asunto que arrastra una historia de más de cuarenta años. Para colmo, en 1995 se publicó la edición de plata del clásico de Fred Brooks *The Mythical Man-Month* [Bro75], que con años de anticipación había revelado la estrechez de la mentalidad lineal. En *MMM* Brooks se había expedido en contra del método en cascada:

> Gran parte de los procedimientos actuales de adquisición de software reposan en el supuesto de que se puede especificar de antemano un sistema satisfactorio, obtener requisitos para su construcción, construirlo e instalarlo. Pienso que este supuesto es fundamentalmente equivocado y que muchos de los problemas de adquisición se originan en esta falacia.

En la vieja edición Brooks había dicho "Haga un plan para tirarlo. Lo hará de todos modos". En la nueva edición, fingió retractarse para atacar con más fuerza: "No construya uno para tirar. El modelo en cascada es erróneo!". Comenzaron a publicarse, una tras otra, historias de fracasos del método, o se refrescó el recuerdo de las que habían sido editadas. El Departamento de Defensa, que en la década de 1980 había promovido el ciclo de vida en cascada DOD-STD-2167, de infausta memoria, comenzó a asociarlo a una serie de reveses continuos. En 1987, se recomendó la implementación de métodos iterativos y evolutivos, de lo que resultó el MIL-STD-498. La NATO, la FDA y otros organismos han experimentado historias semejantes [Lar04]. En la segunda mitad de la década de 1990, también el reputado ISO 9000 cayó bajo fuego; se encontraron correlaciones negativas entre la adscripción a los estándares y la calidad de los productos, y la autoridad británica que regula la publicidad mandó que el British Standard Institute se abstuviera de proclamar que la adhesión a ISO mejoraría la calidad o la productividad de las soluciones [ESPI96].

Ante esa evidencia, los expertos, en masa, aconsejaron el abandono de los métodos en cascada y las normativas fuertes; eran nombres influyentes: Harlan Mills, Tom Gilb, Barry Boehm, James Martin, Tom DeMarco, Ed Yourdon y muchos más [Lar04]. Ya eran conocidos por diversos logros y muchos de ellos estaban, además, elaborando sus propias alternativas dinámicas, iterativas, evolutivas y en suma, ágiles. El surgimiento de los MAs no fue, mal que le pese a críticos como Steven Rakitin [Rak01], Gerold Keefer [Kee03], Stephen Mellor [Mel03], Edward Berard [Ber03], Matt Stephens o Doug Rosenberg [SR03], una revuelta de aficionados adictos al código sin destreza en el desarrollo corporativo. A los MAs les pueden caber muchos cuestionamientos, pero de ningún modo resulta razonable hacerles frente con argumentos *ad hominem*.

En las vísperas del estallido del Manifiesto ágil, todos los autores sensibles al encanto de los métodos incrementales que comenzaban a forjarse o a resurgir tenían un enemigo común. Aunque CMM no es estrictamente un método, ni un método en cascada en particular, lo cierto es que se lo comenzó a identificar como el modelo no-iterativo de peso pesado por excelencia. Hay razones para ello. CMM fue leído con espíritu de cascada a fines de los 80 y durante buena parte de la década siguiente. Aunque una

solución producida con una estrategia iterativa podría llegar a calificar como CMM-madura, las discusiones iniciales de CMM tienen un fuerte tono de planeamiento predictivo, desarrollo basado en planes y orientado por procesos [SEI03]. Craig Larman [Lar04] sostiene que muchos de los certificadores y consultores en CMM tienen una fuerte formación en los valores del modelo en cascada y en procesos prescriptivos, con poca experiencia en métodos iterativos e incrementales.

Los MAs difieren en sus particularidades, pero coinciden en adoptar un modelo iterativo que la literatura más agresiva opone dialécticamente al modelo ortodoxo dominante. El modelo iterativo en realidad es bastante antiguo, y se remonta a ideas plasmadas por Tom Gilb en los 60 y por Vic Basili y Joe Turner en 1975 [BT75]. Como quiera que se los llame, la metodología de la corriente principal se estima opuesta en casi todos los respectos a las nuevas, radicales, heterodoxas, extremas, adaptativas, minimalistas o marginales. El antagonismo entre ambas concepciones del mundo es, según los agilistas, extrema y abismal. Escribe Bob Charette, uno de los creadores de Lean Development: "La burocracia e inflexibilidad de organizaciones como el Software Engineering Institute y de prácticas como CMM las hacen cada vez menos relevantes para las circunstancias actuales del desarrollo de software" [en High00b]. Ken Orr, del Cutter Consortium, dice que la diferencia entre un asaltante de bancos y un metodólogo de estilo CMM es que con un asaltante se puede negociar [Orr03]. La certificación en CMM, agrega, depende más de una buena presentación en papel que de la calidad real de una pieza de software; tiene que ver más con el seguimiento a ciegas de una metodología que con el desarrollo y puesta en producción de un sistema de estado del arte.

La mayoría de los agilistas no cree que CMM satisfaga sus necesidades. Algunos de ellos lo exponen gráficamente: Turner y Jain afirman que si uno pregunta a un ingeniero de software típico si cree que CMM es aplicable a los MAs, la respuesta probablemente oscile entre un una mirada de sorpresa y una carcajada histérica [TJ02]. Una razón para ello es que CMM constituye una creencia en el desarrollo de software como un proceso definido que puede ser especificado en detalle, que dispone de algoritmos precisos y que los resultados pueden ser medidos con exactitud, usando las medidas para refinar el proceso hasta que se alcanza el óptimo. Para proyectos con algún grado de exploración e incertidumbre, los desarrolladores ágiles simplemente no creen que esos supuestos sean válidos. A Fred Brooks no le habría cabido la menor duda: en un método clásico puede haber bucles e iteraciones, pero la mentalidad que está detrás de su planificación es fatalmente lineal. Hay entre ambas posturas una división fundamental que muchos creen que no puede reconciliarse mediante el expediente de recurrir a algún cómodo término medio [Hig02a], aunque tanto desde CMM [Pau01] como desde el territorio ágil [Gla01] [Boe02a] se han intentado fórmulas de apaciguamiento.

Como arquetipo de la ortodoxia, CMM tiene compañía en su cuadrante. El Project Management Institute (PMI), la segunda *bête noire* de los iconoclastas, ha tenido influencia en niveles gerenciales a través de su prestigioso Body of Knowledge (PMBOK) [PMB04]. Aunque este modelo es una contribución valiosa y reconoce el mérito de los nuevos métodos, los contenidos más tempranos de PMBOK son también marcadamente prescriptivos y depositan mucha confianza en la elaboración del plan y el desenvolvimiento de un trabajo conforme a él.

A comienzos del siglo XXI estos modelos estaban ya desacreditados y se daban las condiciones para que irrumpiera el Manifiesto de los métodos ágiles. Kent Beck, el líder de XP, comentó que sería difícil encontrar una reunión de anarquistas organizacionales más grande que la de los 17 proponentes de MAs que se juntaron en marzo de 2001 en Salt Lake City para discutir los nuevos métodos. Lo que surgió de su reunión fue el Manifiesto Ágil de Desarrollo de Software [BBB+01a]. Fue todo un acontecimiento, y aunque ha sido muy reciente tiene la marca de los sucesos que quedan en la historia. Bob Charette y Ken Orr, independientemente, han comparado el sentido histórico del Manifiesto con las Tesis de Lutero; ambos desencadenaron guerras dogmáticas [Orr03].

En el encuentro había representantes de modelos muy distintos, pero todos compartían la idea de que era necesaria una alternativa a los métodos consagrados basados en una gestión burocrática, una documentación exhaustiva y procesos de peso pesado. Resumían su punto de vista argumentando que el movimiento Ágil no es una anti-metodología:

> De hecho, muchos de nosotros deseamos restaurar credibilidad a la palabra metodología. Queremos recuperar un equilibrio. Promovemos el modelado, pero no con el fin de archivar algún diagrama en un polvoriento repositorio corporativo. Promovemos la documentación, pero no cientos de páginas en tomos nunca mantenidos y rara vez usados. Planificamos, pero reconocemos los límites de la planificación en un entorno turbulento [BBB+01b].

El Manifiesto propiamente dicho [BBB+01a] reza como sigue:

> Estamos poniendo al descubierto formas mejores de desarrollo de software, haciéndolo y ayudando a otros a que lo hagan. A través de este trabajo hemos llegado a valorar:
>
> - Los individuos y la interacción por encima de los procesos y herramientas.
> - El software que funciona por encima de la documentación abarcadora.
> - La colaboración con el cliente por encima de la negociación contractual.
> - La respuesta al cambio por encima del seguimiento de un plan.
>
> Aunque hay valor en los elementos a la derecha, valorizamos más los de la izquierda.

Los firmantes del Manifiesto fueron Kent Beck (XP), Mike Beedle, Arie van Bennekum (DSDM), Alistair Cockburn (Crystal), Ward Cunnimgham (XP), Martin Fowler (XP), James Grenning (XP), Jim Highsmith (ASD), Andrew Hunt (Pragmatic Programming), Ron Jeffries (XP), Jon Kern (FDD), Brian Marick, Robert C. Martin (XP), Steve Mellor, Ken Schwaber (Scrum), Jeff Sutherland (Scrum) y Dave Thomas (Pragmatic Programming). Hay predominio demográfico de XP (6 sobre 17) y, considerando nuestra selección de diez MAs, se percibe la falta de delegados de Evo, Agile Modeling, Lean Development y RUP.

Al lado del Manifiesto se han especificado los principios que rigen a la comunidad de MAs [BBB+01b]:

1. Nuestra prioridad más alta es satisfacer al cliente a través de la entrega temprana y continua de software valioso.
2. Los requerimientos cambiantes son bienvenidos, incluso cuando llegan tarde en el desarrollo. Los procesos ágiles se pliegan al cambio en procura de una ventaja competitiva para el cliente.

3. Entregar con frecuencia software que funcione, desde un par de semanas hasta un par de meses, con preferencia por las escalas de tiempo más breves.
4. La gente de negocios y los desarrolladores deben trabajar juntos cotidianamente a través de todo el proyecto.
5. Construir proyectos en torno de individuos motivados. Darles la oportunidad y el respaldo que necesitan y procurarles confianza para que realicen la tarea.
6. La forma más eficiente y efectiva de comunicar información de ida y vuelta dentro de un equipo de desarrollo es mediante la conversación cara a cara.
7. El software que funciona es la medida primaria de progreso.
8. Los procesos ágiles promueven el desarrollo sostenido. Los patrocinadores, desarrolladores y usuarios deben mantener un ritmo constante indefinidamente.
9. La atención continua a la excelencia técnica enaltece la agilidad.
10. La simplicidad (el arte de maximizar la cantidad de trabajo que no se hace) es esencial.
11. Las mejores arquitecturas, requerimientos y diseños emergen de equipos que se auto-organizan.
12. A intervalos regulares, el equipo reflexiona sobre la forma de ser más efectivo, y ajusta su conducta en consecuencia.

El mero hecho de que los MAs se expresen a través de un Manifiesto urdido en una tertulia ocasional y no de un estándar elaborado durante años en un organismo, está señalando una tajante diferencia de actitud. No será la única. Mientras la terminología de CMM, EUP y otros métodos ortodoxos es prolija y circunspecta, las nomenclaturas de muchos de los MAs están plagadas de figuras del lenguaje procedentes de los dominios semánticos más diversos: "Scrum", por empezar, y luego "gallinas", "cerdos", "corridas" (*sprints*), pre-juego, juego, pos-juego y montaje (*staging*) en Scrum; "púas" (*spikes*) en Scrum y XP; "irradiadores" en Crystal.

Al lado de esos lexemas unitarios, en los nombres de principios, estrategias y técnicas abundan los aforismos y máximas como "El Minimalismo es Esencial", "Todo se puede cambiar", "80% hoy en vez de 100% mañana" en Lean Development, "Mirando basura" en LSD; el "Cono del Silencio" o el "Esqueleto Ambulante" en Crystal Clear; "Ligero de equipaje" en AM; "Documentos para tí y para mí" en ASD; "Gran Jefe" y la más publicitada de todas, "YAGNI" (acrónimo de "No vas a necesitarlo") en XP. También hay una producción destacada de citas citables o "agilismos" como éste de Ron Jeffries: "Lo lamento por su vaca; no sabía que era sagrada". Podría argumentarse que esta búsqueda de peculiaridad de expresión conspira contra la homogeneización de la terminología que ha sido uno de los motores de las iniciativas y lenguajes unificados como RUP o UML; pero también es un valor diferencial: nadie podrá jamás confundir un método ágil paradigmático con otro que no lo es.

Hay que subrayar que no todos los MAs comparten esta forma simbólica de locución, reminiscente de la neolengua de George Orwell; pues si bien la mayoría de ellos rubricó el Manifiesto, converge en el espíritu de ciertas ideas, toma prestadas prácticas de métodos aliados y hasta emprende una buena proporción de sus proyectos combinándose con otros, el conjunto ágil presenta ejemplares de naturaleza muy dispar. Hay en ese

conjunto dos mundos: uno es auto-organizativo, anárquico, igualitario o emergente de abajo hacia arriba; el otro es la expresión de una disciplina de *management*, innovadora por cierto, pero muchas veces más formal que transgresora, más rigurosa que temeraria. Se volverá a esta discusión en las conclusiones, pero se anticipa la idea ahora, para que en la lectura que sigue no se confunda lo más radical con lo más representativo.

Los Métodos Ágiles

Existen unos cuantos MAs reconocidos por los especialistas. En este estudio se analizarán con algún detenimiento los que han alcanzado a figurar en la bibliografía mayor y cuentan en su haber con casos sustanciales documentados. Ha habido unos cuantos más que típicamente se presentaron en ponencias de simposios de la comunidad y en seguida se esfumaron. Hacia el final se dedicará un párrafo para caracterizar a este género colateral, más que para describir sus ejemplares. Lo mismo se aplica a los métodos híbridos y a los dialectos (XBreed, XP@Scrum, dX, Grizzly, Enterprise XP, IndustrialXP). La idea no es hacer un censo de todo el campo, sino tratar las metodologías de real importancia.

Consideraremos entonces los diez MAs que han adquirido masa crítica: Extreme Programming, Scrum, Evo, Crystal Methods, Feature Driven Development, RUP, Dynamic Systems Development Method, Adaptive Software Development, Agile Modeling y Lean Development. Aunque hemos adoptado un foco restringido, de los *surveys* existentes a la fecha el presente estudio es el que contempla el mayor número de variedades. Los demás [Gla01] [ASR+02] [Hig02a] [CLC03] [Lar04] siempre omiten algunos importantes.

Dejaremos fuera de consideración un conjunto de formulaciones posmodernas, centradas en la acción, fractales, humanistas o caopléjicas porque entendemos que no han estipulado debidamente sus heurísticas ni las definiciones operacionales requeridas en la ingeniería de soluciones intensivas. Excluiremos Agile Model Driven Development de Scott Ambler (una versión ágil de MDD) y también el modelo en que se basa. Si se incluyera, se admitiría que un conjunto enorme de estrategias de desarrollo "guiadas" de alguna manera[1] (**-driven development*) se maquillaran un poco y se redefinieran como MAs, extendiendo la población de los géneros existentes más allá de lo que puede manejarse. AMDD es, por otra parte, una variante de AMD. Se excluirá también Pragmatic Programming, por más que sus responsables, Andrew Hunt y David Thomas, hayan firmado el Manifiesto; aunque en [ASR+02] se le consagra un capítulo, no se trata de una metodología articulada que se haya elaborado como método ágil, sino de un conjunto de *best practices* ágiles publicadas en *The Pragmatic Programmer* [HT00]. El mismo temperamento se aplicará a Internet-Speed Development (ISD). Por último, se hará caso omiso de Agile Database Methodology, porque es una metodología circunscripta y no un método de propósito general.

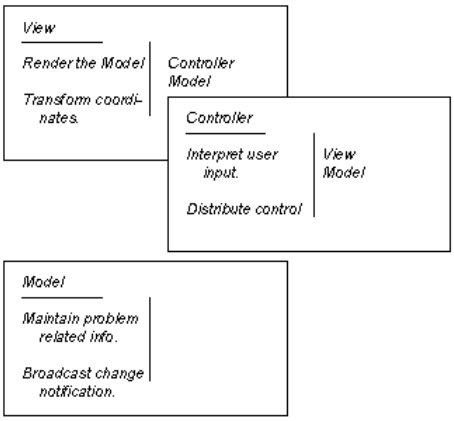
---

[1] Contract-Driven Development, Model Driven Development, User-Driven Development, Requirements-Driven Development, Design-Driven Development, Platform Driven Development, Use Case-Driven Development, Domain-Driven Development, Customer-Driven Development, Application-Driven Development, Pattern-Driven Development.

Con unas pocas excepciones, los *surveys* de MAs disponibles poseen estructuras demasiado complejas y heterogéneas para que puedan visualizarse con comodidad las regularidades del conjunto. Dado que el presente examen no es un estudio comparativo a fondo sino una visión preliminar, expondremos cada método de acuerdo con un plan estable que contemple su historia, sus valores, sus roles, sus artefactos, sus prácticas y su ciclo de vida. Cuando haya que representar diagramas de procesos, se tratará de reflejar el estilo iconográfico de los textos o páginas originales, apenas homogeneizado en su nivel de detalle para su mejor correlación.

eXtreme Programming (XP)

La Programación Extrema es sin duda alguna el método ágil que primero viene a la mente cuando se habla de modelos heterodoxos y el más transgresor entre ellos. Es, de acuerdo con el *survey* de Cutter Consortium, también el más popular entre los MAs: 38% del mercado ágil contra 23% de su competidor más cercano, FDD. Luego vienen Adaptive Software Development con 22%, DSDM con 19%, Crystal con 8%, Lean Development con 7% y Scrum con 3%. Todos los demás juntos suman 9% [Cha04].



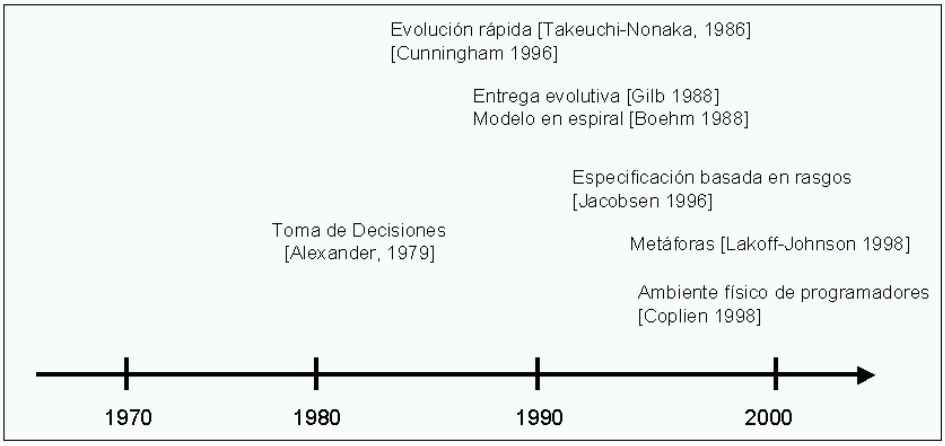
Tarjetas CRC del patrón MVC según [BC89]

A mediados de la década de 1980, Kent Beck y Ward Cunningham trabajaban en un grupo de investigación de Tektronix; allí idearon las tarjetas CRC y sentaron las bases de lo que después serían los patrones de diseño y XP. Los patrones y XP no necesitan presentación, pero las tarjetas CRC tal vez sí. Se trata de simples tarjetas de papel para fichado, de 4x6 pulgadas; CRC denota “Clase-Responsabilidad-Colaboración”, y es una técnica que reemplaza a los diagramas en la representación de modelos. En las tarjetas se escriben las Responsabilidades, una descripción de alto nivel del propósito de una clase y las dependencias primarias. En su economía sintáctica y en su abstracción, prefiguran a lo que más tarde serían las tarjetas de historias de XP. Beck y Cunningham prohibían escribir en las tarjetas más de lo que en ellas cabía [BC89]. Nos ha parecido útil ilustrar las tarjetas correspondientes al Modelo, la Vista y el Controlador del patrón MVC de Smalltalk, tal como lo han hecho los autores en el texto original. Hemos descripto ese patrón (o estilo) en el documento sobre Estilos en Arquitectura de Software.

Luego de esa tarea conjunta, Beck comenzó a desarrollar XP mientras Cunningham inventó el concepto de Wiki Web, el sistema de hipertexto de autoría colectiva antecesor

de los Weblogs. En la década siguiente, Beck fue empleado por Chrysler para colaborar en el sistema de liquidación de sueldos en Smalltalk que se conoció como el proyecto C3 (Chrysler Comprehensive Compensation). Más tarde se unieron al proyecto Ron Jeffries (luego ardiente partidario de LSD y del framework .NET de Microsoft) y Martin Fowler, ulterior líder de Cutter Consortium. Previsiblemente, terminaron desarrollando C3 en una especie de proto-XP y el proyecto terminó exitosamente. Hay una leyenda negra que afirma que aunque XP surgió de allí, el proyecto C3 fracasó [Lar04] [Kee03]; lo cierto es que estuvo en producción durante un tiempo, pero fue reemplazado por otro sistema y los equipos originales de programación y mantenimiento se disolvieron cuando Chrysler cambió de nombre y de dueños.

En la gestación de C3 se encuentra no sólo la raíz de XP, sino el núcleo primitivo de la comunidad ágil. Beck, Cunningham y Jeffries son llamados "*the three extremos*", por analogía con "*the three amigos*" de UML [Hig00b]. Muchos autores escriben "eXtreme Programming" para ser consistentes con la sigla, pero Kent Beck siempre lo mayusculiza como se debe. XP se funda en cuatro valores: comunicación, simplicidad, *feedback* y coraje. Pero tan conocidos como sus valores son sus prácticas. Beck sostiene que se trata más de lineamientos que de reglas:



Gestación de XP, basado en [ASR+02]

1. **Juego de Planeamiento**. Busca determinar rápidamente el alcance de la versión siguiente, combinando prioridades de negocio definidas por el cliente con las estimaciones técnicas de los programadores. Éstos estiman el esfuerzo necesario para implementar las historias del cliente y éste decide sobre el alcance y la agenda de las entregas. Las historias se escriben en pequeñas fichas, que algunas veces se tiran. Cuando esto sucede, lo único restante que se parece a un requerimiento es una multitud de pruebas automatizadas, las pruebas de aceptación.
2. **Entregas pequeñas y frecuentes**. Se "produccioniza" [sic] un pequeño sistema rápidamente, al menos uno cada dos o tres meses. Pueden liberarse nuevas versiones diariamente (como es práctica en Microsoft), pero al menos se debe liberar una cada mes. Se agregan pocos rasgos cada vez.
3. **Metáforas del sistema**. El sistema de define a través de una metáfora o un conjunto de metáforas, una "historia compartida" por clientes, *managers* y programadores que orienta todo el sistema describiendo como funciona. Una metáfora puede interpretarse como una arquitectura simplificada. La concepción de metáfora que se aplica en XP

deriva de los estudios de Lakoff y Johnson, bien conocidos en lingüística y psicología cognitiva.

4. **Diseño simple**. El énfasis se deposita en diseñar la solución más simple susceptible de implementarse en el momento. Se eliminan complejidades innecesarias y código extra, y se define la menor cantidad de clases posible. No debe duplicarse código. En un oxímoron deliberado, se urge a "decir todo una vez y una sola vez". Nadie en XP llega a prescribir que no haya diseño concreto, pero el diseño se limita a algunas tarjetas elaboradas en sesiones de diseño de 10 a 30 minutos. Esta es la práctica donde se impone el minimalismo de YAGNI: no implementar nada que no se necesite ahora; o bien, nunca implementar algo que vaya a necesitarse más adelante; minimizar diagramas y documentos.
5. **Prueba continua**. El desarrollo está orientado por las pruebas. Los clientes ayudan a escribir las pruebas funcionales *antes* que se escriba el código. Esto es *test-driven development*. El propósito del código real no es cumplir un requerimiento, sino pasar las pruebas. Las pruebas y el código son escritas por el mismo programador, pero la prueba debería realizarse sin intervención humana, y es a todo o nada. Hay dos clases de prueba: la prueba unitaria, que verifica una sola clase, o un pequeño conjunto de clases; la prueba de aceptación verifica todo el sistema, o una gran parte.
6. **Refactorización[2] continua**. Se refactoriza el sistema eliminando duplicación, mejorando la comunicación y agregando flexibilidad sin cambiar la funcionalidad. El proceso consiste en una serie de pequeñas transformaciones que modifican la estructura interna preservando su conducta aparente. La práctica también se conoce como **Mejora Continua de Código** o **Refactorización implacable**. Se lo ha parafraseado diciendo: "Si funciona bien, arréglelo de todos modos". Se recomiendan herramientas automáticas. En sus comentarios a [Hig00b] Ken Orr recomienda GeneXus de ARTech, Uruguay, virtuoso ejecutor de las mejores promesas incumplidas de las herramientas CASE.
7. **Programación en pares**. Todo el código está escrito por pares de programadores. Dos personas escriben código en una computadora, turnándose en el uso del ratón y el teclado. El que no está escribiendo, piensa desde un punto de vista más estratégico y realiza lo que podría llamarse revisión de código en tiempo real. Los roles pueden cambiarse varias veces al día. Esta práctica no es en absoluto nueva. Hay

---

[2] El término *refactoring*, introducido por W. F. Opdyke en su tesis doctoral [Opd92], se refiere "al proceso de cambiar un sistema de software [orientado a objetos] de tal manera que no se altere el comportamiento exterior del código, pero se mejore su estructura interna". Normalmente se utilizan herramientas automáticas para hacerlo (DMS, GeNexus), y/o se implementan técnicas tales como aserciones (invariantes, pre- y poscondiciones) para expresar propiedades que deben conservarse antes y después de la refactoría. Otras técnicas son transformaciones de grafos, métricas de software, refinamiento de programas y análisis formal de conceptos [MVD03]. Al igual que sucede con los patrones, existe un amplio catálogo de refactorizaciones más comunes: reemplazo de iteración por recursión; sustitución de un algoritmo por otro más claro; extracción de clase, interface o método; descomposición de condicionales; reemplazo de herencia por delegación, etcétera. La Biblia del método es *Refactoring* de Martin Fowler, Kent Beck, John Brant, William Opdyke y Don Roberts [FBB+99].

antecedentes de programación en pares anteriores a XP [Hig00b]. Steve McConnell la proponía en 1993 en su *Code Complete* [McC93: 528].

8. **Propiedad colectiva del código**. Cualquiera puede cambiar cualquier parte del código en cualquier momento, siempre que escriba antes la prueba correspondiente. Algunas veces los practicantes aplican el patrón organizacional *CodeOwnership* de Coplien [Cop95].
9. **Integración continua**. Cada pieza se integra a la base de código apenas está lista, varias veces al día. Debe correrse la prueba antes y después de la integración. Hay una máquina (solamente) dedicada a este propósito.
10. **Ritmo sostenible**, trabajando un máximo de 8 horas por día. Antes se llamaba a esta práctica **Semana de 40 horas**. Mientras en RAD las horas extras eran una *best practice* [McC96], en XP todo el mundo debe irse a casa a las cinco de la tarde. Dado que el desarrollo de software se considera un ejercicio creativo, se estima que hay que estar fresco y descansado para hacerlo eficientemente; con ello se motiva a los participantes, se evita la rotación del personal y se mejora la calidad del producto. Deben minimizarse los héroes y eliminar el "proceso neurótico". Aunque podrían admitirse excepciones, no se permiten dos semanas seguidas de tiempo adicional. Si esto sucede, se lo trata como problema a resolver.
11. **Todo el equipo en el mismo lugar**. El cliente debe estar presente y disponible a tiempo completo para el equipo. También se llama **El Cliente en el Sitio**. Como esto parecía no cumplirse (si el cliente era muy *junior* no servía para gran cosa, y si era muy *senior* no deseaba estar allí), se especificó que el representante del cliente debe ser preferentemente un analista. (Tampoco se aclara analista de qué; seguramente se definirá en una próxima versión).
12. **Estándares de codificación**. Se deben seguir reglas de codificación y comunicarse a través del código. Según las discusiones en Wiki, algunos practicantes se desconciertan con esta regla, prefiriendo recurrir a la tradición oral. Otros la resuelven poniéndose de acuerdo en estilos de notación, indentación y nomenclatura, así como en un valor apreciado en la práctica, el llamado "código revelador de intenciones". Como en XP rige un cierto purismo de codificación, los comentarios no son bien vistos. Si el código es tan oscuro que necesita comentario, se lo debe reescribir o refactorizar.
13. **Espacio abierto**. Es preferible una sala grande con pequeños cubículos o, mejor todavía, sin divisiones. Los pares de programadores deben estar en el centro. En la periferia se ubican las máquinas privadas. En un encuentro de espacio abierto la agenda no se establece verticalmente.
14. **Reglas justas**. El equipo tiene sus propias reglas a seguir, pero se pueden cambiar en cualquier momento. En XP se piensa que no existe un proceso que sirva para todos los proyectos; lo que se hace habitualmente es adaptar un conjunto de prácticas simples a la características de cada proyecto.

Las prácticas se han ido modificando con el tiempo. Originariamente eran doce; de inmediato, trece. Las versiones más recientes enumeran diecinueve prácticas, agrupadas en cuatro clases.

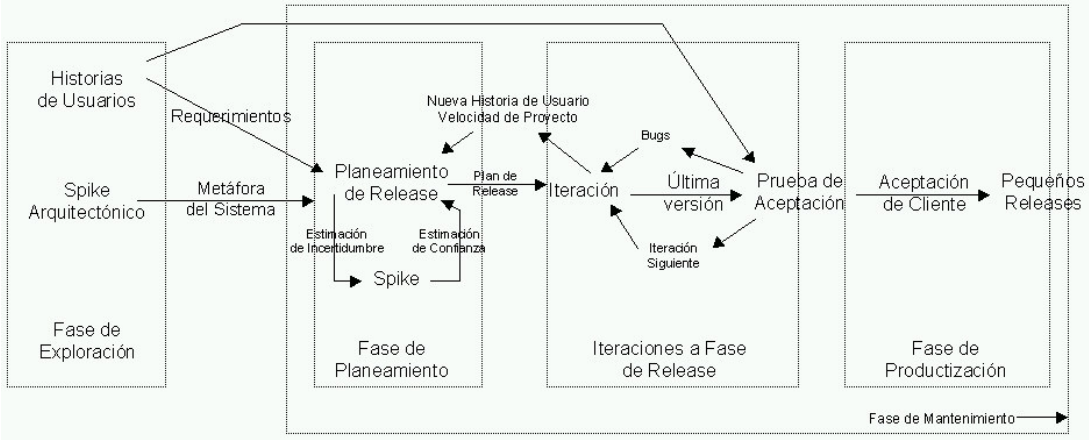
Prácticas conjuntas	Iteraciones Vocabulario Común – Reemplaza a Metáforas Espacio de trabajo abierto Retrospectivas
Prácticas de Programador	Desarrollo orientado a pruebas Programación en pares Refactorización Propiedad colectiva Integración continua YAGNI (“No habrás de necesitarlo”) – Equivale a Diseño Simple
Prácticas de Management	Responsabilidad aceptada Cobertura aérea para el equipo Revisión trimestral Espejo – El *manager* debe comunicar un fiel reflejo del estado de cosas Ritmo sostenible
Prácticas de Cliente	Narración de historias Planeamiento de entrega Prueba de aceptación Entregas frecuentes

Mientras esto se escribe, Ward Cunningham sugiere rebautizar las prácticas como Xtudios (*Xtudes*). Oficialmente las prácticas siguen siendo doce y su extensión se basa en documentos inéditos de Kent Beck. Las prácticas no tratan en detalle el proceso de entrega, que es casi un no-evento.

Al lado de los valores y las prácticas, XP ha abundado en acrónimos algo extravagantes pero eficaces, que sus seguidores intercambian con actitud de complicidad. YAGNI significa “You Aren’t Gonna Need It”; TETCPB, “Test Everything That Can Possibly Broke”; DTSTTCPW, “Do The Simplest Thing That Can Possibly Work”, etcétera. Del mismo modo, BUFD (“Big Up Front Design”) es el mote peyorativo que se aplica a los grandes diseños preliminares de los métodos en cascada, “el Libro Verde” es *Planning Extreme Programming* de Kent Beck y Martin Fowler [BF00], y “el Libro Blanco” es *Extreme Programming Explained* de Kent Beck [Beck99a]. Todo el mundo en XP sabe, además, qué significan GoF, *PLOPD* o *POSA*. Al menos en la concepción de Cunningham y Wiki, parecería que quien conoce más mantras y acrónimos, gana.

Beck [Beck99a] sugiere adoptar XP paulatinamente: “Si quiere intentar con XP, por Dios le pido que no se lo trague todo junto. Tome el peor problema de su proceso actual y trate de resolverlo a la manera de XP”. Las prácticas también pueden adaptarse a las características de los sistemas particulares. Pero no todo es negociable y la integridad del método se sabe frágil. La particularidad de XP radica en no requerir ninguna herramienta fuera del ambiente de programación y prueba. Al contrario de otros métodos, que permiten modelado, XP demanda comunicación oral tanto para los requerimientos como para el diseño. Beck [Beck99b] ha llegado a decir: “También yo creo en el modelado; sólo que lo llamo por su nombre propio, ‘mentir’, y trato de convertirlo en arte”.

El ciclo de vida es, naturalmente, iterativo. El siguiente diagrama describe su cuerpo principal:



Ciclo de vida de XP, adaptado de [Beck99a]

Algunos autores sugieren implementar *spikes* (o sea "púas" o "astillas") para estimar la duración y dificultad de una tarea inmediata [JAH01]. Un *spike* es un experimento dinámico de código, realizado para determinar cómo podría resolverse un problema. Es la versión ágil de la idea de prototipo. Se lo llama así porque "va de punta a punta, pero es muy fino" y porque en el recorrido de un árbol de opciones implementaría una opción de búsqueda *depth-first* (http://c2.com/cgi/wiki?SpikeSolution).

Entre los artefactos que se utilizan en XP vale la pena mencionar las tarjetas de historias (*story cards*); son tarjetas comunes de papel en que se escriben breves requerimientos de un rasgo, jamás casos de uso; pueden adoptar el esquema CRC. Las tarjetas tienen una granularidad de diez o veinte días. Las tarjetas se usan para estimar prioridades, alcance y tiempo de realización; en caso de discrepancia, gana la estimación más optimista. Otros productos son listas de tareas en papel o en una pizarra (jamás en computadora) y gráficos visibles pegados en la pared. Martin Fowler admite que la preferencia por esta clase de herramientas ocasiona que el mensaje entre líneas sea "Los XPertos no hacen diagramas" [Fow01].

Los roles de XP son pocos. Un cliente que escribe las historias y las pruebas de aceptación; programadores en pares; verificadores (que ayudan al cliente a desarrollar las pruebas); consultores técnicos; y, como parte del *management*, un *coach* o consejero que es la conciencia del grupo, interviene y enseña, y un seguidor de rastros (*tracker*) que colecta las métricas y avisa cuando hay una estimación alarmante, además de un Gran Jefe. El *management* es la parte menos satisfactoriamente caracterizada en la bibliografía, como si fuera un mal necesario; Beck comenta, por ejemplo, que la función más relevante del *coach* es la adquisición de juguetes y comida [Hig00b]; otros dicen que está para servir café. Lo importante es que el coach se vea como un facilitador, antes que como quien dá las órdenes. Los equipos de XP son típicamente pequeños, de tres a veinte personas, y en general no se cree que su escala se avenga al desarrollo de grandes sistemas de misión crítica con tanta comodidad como FDD.

Algunas empresas han creado sus propias versiones de XP, como es el caso de Standard & Poor's, que ha elaborado plantillas para proyectos futuros. XP se ha combinado con Evo, a pesar que ambos difieren en su política de especificaciones; también se estima compatible con Scrum, al punto que existe una forma híbrida, inventada por Mike Beedle, que se llama XBreed [http://www.xbreed.net] y otra bautizada XP@Scrum

[http://www.controlchaos.com/xpScrum.htm] en la que Scrum se usa como envoltorio de gestión alrededor de prácticas de ingeniería de XP. Se lo ha combinado muchas veces con UP, aunque éste recomienda pasar medio día de discusión de pizarra antes de programar y XP no admite más de 10 o 20 minutos. La diferencia mayor concierne a los casos de uso, que son norma en UP y que en XP se reemplazan por tarjetas de papel con historias simples que se refieren a rasgos. En las herramientas de RUP ahora hay *plug-ins* para XP.

Los obstáculos más comunes surgidos en proyectos XP son la "fantasía" [Lar04] de pretender que el cliente se quede en el sitio y la resistencia de muchos programadores a trabajar en pares. Craig Larman señala como factores negativos la ausencia de énfasis en la arquitectura durante las primeras iteraciones (no hay arquitectos en XP) y la consiguiente falta de métodos de diseño arquitectónico. Un estudio de casos de Bernhard Rumpe y Astrid Schröder sobre 45 ejemplares de uso reveló que las prácticas menos satisfactorias de XP han sido la presencia del usuario en el lugar de ejecución y las metáforas, que el 40% de los encuestados no comprende para qué sirven o cómo usarlas [RS01]. Las metáforas han sido reemplazadas por un Vocabulario Común (equivalente al "Sistema de Nombres" de Cunningham) en las últimas revisiones del método.

XP ha sido, de todos los MAs, el que más resistencia ha encontrado [Rak01] [McB02] [Baer03] [Mel03] [SR03]. Algunos han sugerido que esa beligerancia es un efecto de su nombre, que debería ser menos intimidante. Jim Highsmith [Hig02a] argumenta, sin embargo, que es improbable que muchos se entusiasmen por un libro que se titule, por ejemplo, *Programación Moderada*. Los nuevos mercados, tecnologías e ideas –piensa Highsmith– no se forjan a fuerza de moderación sino con giros radicales y con el coraje necesario para desafiar al status quo; XP, en todo caso, ha abierto el camino.

Scrum

Esta es, después de XP, la metodología ágil mejor conocida y la que otros métodos ágiles recomiendan como complemento, aunque su porción del mercado (3% según el Cutter Consortium) es más modesta que el ruido que hace. La primera referencia a la palabra "*scrum*" en la literatura aparece en un artículo de Hirotaka Takeuchi e Ikujiro Nonaka, "The New Product Development Game" en el que se presentaron diversas *best practices* de empresas innovadoras de Japón que siempre resultaban ser adaptativas, rápidas y con capacidad de auto-organización [TN86].

Otra palabra del mismo texto relacionada con modelos japoneses es *Sashimi*, el cual se inspira en una estrategia japonesa de desarrollo de hardware con fases solapadas utilizada por Fuji-Xerox. La palabra Scrum, empero, nada tiene que ver con Japón, sino que procede de la terminología del juego de rugby, donde designa al acto de preparar el avance del equipo en unidad pasando la pelota a uno y otro jugador (aunque hay otras acepciones en circulación). Igual que el juego, Scrum es adaptativo, ágil, auto-organizante y con pocos tiempos muertos.

Como metodología ágil específicamente referida a ingeniería de software, Scrum fue aplicado por Jeff Sutherland y elaborado más formalizadamente por Ken Schwaber [Sch95]. Poco después Sutherland y Schwaber se unieron para refinar y extender Scrum [BDS+98] [SB02]. En Scrum se aplicaron principios de procesos de control industrial, junto con experiencias metodológicas de Microsoft, Borland y Hewlett-Packard.

Schwaber, en particular, había trabajado con científicos de Du Pont para comprender mejor los procesos definidos de antemano, y ellos le dijeron que a pesar que CMM se concentraba en hacer que los procesos de desarrollo se tornaran repetibles, definidos y predecibles, muchos de ellos eran formalmente impredecibles e irrepetibles porque cuando se está planificando no hay primeros principios aplicables, los procesos recién comienzan a ser comprendidos y son complejos por naturaleza. Schwaber se dió cuenta entonces de que un proceso necesita aceptar el cambio, en lugar de esperar predictibilidad.

Al igual que Agile Modeling, Scrum no está concebido como método independiente, sino que se promueve como complemento de otras metodologías, incluyendo XP, MSF o RUP. Como método, Scrum enfatiza valores y prácticas de gestión, sin pronunciarse sobre requerimientos, implementación y demás cuestiones técnicas; de allí su deliberada insuficiencia y su complementariedad. Scrum se define como un proceso de *management* y control que implementa técnicas de control de procesos; se lo puede considerar un conjunto de patrones organizacionales, como se verá en un capítulo posterior (p. 60).

Los valores de Scrum son:

- Equipos auto-dirigidos y auto-organizados. No hay *manager* que decida, ni otros títulos que "miembros del equipo" o "cerdos"; la excepción es el *Scrum Master* que debe ser 50% programador y que resuelve problemas, pero no manda. Los observadores externos se llaman "gallinas"; pueden observar, pero no interferir ni opinar.
- Una vez elegida una tarea, no se agrega trabajo extra. En caso que se agregue algo, se recomienda quitar alguna otra cosa.
- Encuentros diarios con las tres preguntas indicadas en la figura. Se realizan siempre en el mismo lugar, en círculo. El encuentro diario impide caer en el dilema señalado por Fred Brooks: "¿Cómo es que un proyecto puede atrasarse un año?: Un día a la vez" [Bro95].
- Iteraciones de treinta días; se admite que sean más frecuentes.
- Demostración a participantes externos al fin de cada iteración.
- Al principio de cada iteración, planeamiento adaptativo guiado por el cliente.

El nombre de los miembros del equipo y los externos se deriva de una típica fábula agilista: un cerdo y una gallina discutían qué nombre ponerle a un nuevo restaurant; la gallina propuso "Jamón y Huevos". "No, gracias", respondió el cerdo, "Yo estaré comprometido, pero tú sólo involucrada".

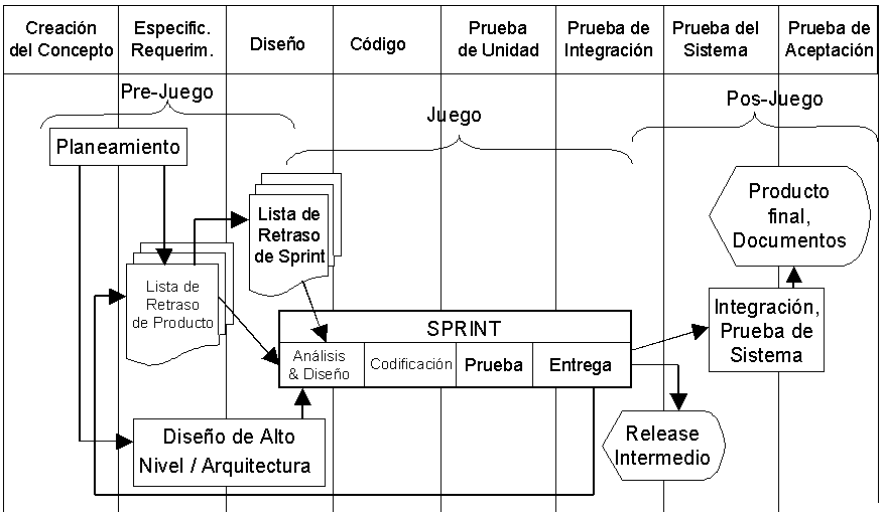
Scrum define seis roles:

1. **El Scrum Master**. Interactúa con el cliente y el equipo. Es responsable de asegurarse que el proyecto se lleve a cabo de acuerdo con las prácticas, valores y reglas de Scrum y que progrese según lo previsto. Coordina los encuentros diarios, formula las tres preguntas canónicas y se encarga de eliminar eventuales obstáculos. Debe ser miembro del equipo y trabajar a la par.
2. **Propietario del Proyecto**. Es el responsable oficial del proyecto, gestión, control y visibilidad de la lista de acumulación o lista de retraso del producto (*product backlog*). Es elegido por el Scrum Master, el cliente y los ejecutivos a cargo. Toma

las decisiones finales de las tareas asignadas al registro y convierte sus elementos en rasgos a desarrollar.

3. **Equipo de Scrum**. Tiene autoridad para reorganizarse y definir las acciones necesarias o sugerir remoción de impedimentos. El equipo posee la misma estructura del "equipo quirúrgico" desarrollado por IBM y comentado en el *MMM* de Brooks [Bro75], aunque Schwaber y Beedle [SB02] destacan que su naturaleza auto-organizadora la hace distinta.
4. **Cliente**. Participa en las tareas relacionadas con los ítems del registro.
5. **Management**. Está a cargo de las decisiones fundamentales y participa en la definición de los objetivos y requerimientos. Por ejemplo, selecciona al Dueño del Producto, evalúa el progreso y reduce el registro de acumulación junto con el Scrum Master.
6. **Usuario**.

La dimensión del equipo total de Scrum no debería ser superior a diez ingenieros. El número ideal es siete, más o menos dos, una cifra canónica en ciencia cognitiva [Mil56]. Si hay más, lo más recomendable es formar varios equipos. No hay una técnica oficial para coordinar equipos múltiples, pero se han documentado experiencias de hasta 800 miembros, divididos en Scrums de Scrum, definiendo un equipo central que se encarga de la coordinación, las pruebas cruzadas y la rotación de los miembros. El texto que relata esa experiencia es *Agile Software Development Ecosystems*, de Jim Highsmith [Hig02b].



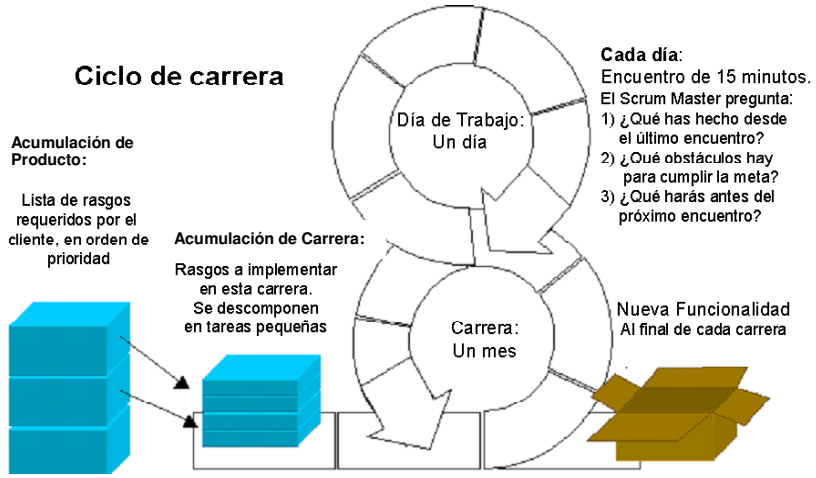
Ciclo de Scrum, basado en [Abr02]

El ciclo de vida de Scrum es el siguiente:

1. **Pre-Juego: Planeamiento**. El propósito es establecer la visión, definir expectativas y asegurarse la financiación. Las actividades son la escritura de la visión, el presupuesto, el registro de acumulación o retraso (*backlog*) del producto inicial y los ítems estimados, así como la arquitectura de alto nivel, el diseño exploratorio y los prototipos. El registro de acumulación es de alto nivel de abstracción.

2. **Pre-Juego: Montaje** (*Staging*). El propósito es identificar más requerimientos y priorizar las tareas para la primera iteración. Las actividades son planificación, diseño exploratorio y prototipos.
3. **Juego o Desarrollo**. El propósito es implementar un sistema listo para entrega en una serie de iteraciones de treinta días llamadas "corridas" (*sprints*). Las actividades son un encuentro de planeamiento de corridas en cada iteración, la definición del registro de acumulación de corridas y los estimados, y encuentros diarios de Scrum.
4. **Pos-Juego: Liberación**. El propósito es el despliegue operacional. Las actividades, documentación, entrenamiento, mercadeo y venta

Usualmente los registros de acumulación se llevan en planillas de cálculo comunes, antes que en una herramienta sofisticada de gestión de proyectos. Los elementos del registro pueden ser prestaciones del software, funciones, corrección de *bugs*, mejoras requeridas y actualizaciones de tecnología. Hay un registro total del producto y otro específico para cada corrida de 30 días. En la jerga de Scrum se llaman "paquetes" a los objetos o componentes que necesitan cambiarse en la siguiente iteración.



Ciclo de Carrera (*Sprint*) de Scrum, basado en [http://www.Controlchaos.com]

La lista de Acumulación del Producto contiene todos los rasgos, tecnología, mejoras y lista de *bugs* que, a medida que se desenvuelven, constituyen las futuras entregas del producto. Los rasgos más urgentes merecerán mayor detalle, los que pueden esperar se tratarán de manera más sumaria. La lista se origina a partir de una variedad de fuentes. El grupo de mercadeo genera los rasgos y la función; la gente de ventas genera elementos que harán que el producto sea más competitivo; los de ingeniería aportarán paquetes que lo harán más robusto; el cliente ingresará debilidades o problemas que deberán resolverse.

El propietario de la administración y el control del *backlog* en productos comerciales bien puede ser el *product manager*; para desarrollos *in-house* podría ser el *project manager*, o alguien designado por él. Se recomienda que una sola persona defina la prioridad de una tarea; si alguien tiene otra opinión, deberá convencer al responsable. Se estima que priorizar adecuadamente una lista de producto puede resultar dificultoso al principio del desarrollo, pero deviene más fácil con el correr del tiempo.

Acumulación de Producto:	Fecha:
	Estimado:
Tipo: Nuevo __ Mejora __ Arreglo: __	Fuente:
Descripción	
Notas	

Product backlog de Scrum, basado en [http://www.controlchaos.com/pbacklog.htm]

La lista de acumulación de corrida sugerida tiene este formato:

Acumulación de Corrida:	Fecha:
Propietario:	Trabajo Pendiente/Fecha
Status: Pendiente___ Activo___Completo___	
Descripción:	
Notas:	

Sprint backlog de Scrum, basado en [http://www.controlchaos.com/sbacklog.htm]

El registro incluye los valores que representan las horas de trabajo pendiente; en función de esos valores se acostumbra elaborar un gráfico de quemado, cuyo modelo y fundamentación se encuentra en [http://www.controlchaos.com/burndown.htm]. Los gráficos de quemado, usados también en Crystal Methods, se describen en la página 33.

Al fin de cada iteración de treinta días hay una demostración a cargo del Scrum Master. Las presentaciones en PowerPoint están prohibidas. En los encuentros diarios, las gallinas deben estar fuera del círculo. Todos tienen que ser puntuales; si alguien llega tarde, se le cobra una multa que se destinará a obras de caridad. Es permitido usar artefactos de los métodos a los que Scrum acompañe, por ejemplo Listas de Riesgos si se utiliza UP, Planguage si el método es Evo, o los Planes de Proyecto sugeridos en la disciplina de Gestión de Proyectos de Microsoft Solutions Framework [MS02b]. No es legal, en cambio, el uso de instrumentos tales como diagramas PERT, porque éstos parten del supuesto de que las tareas de un proyecto se pueden identificar y ordenar; en Scrum el supuesto dominante es que el desarrollo es semi-caótico, cambiante y tiene demasiado ruido como para que se le aplique un proceso definido.

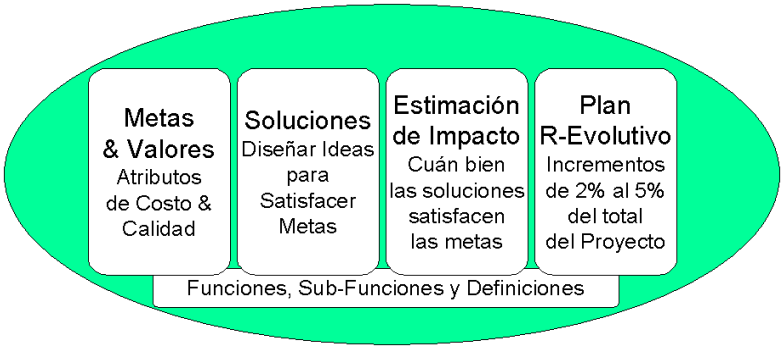
Algunos textos sobre Scrum establecen una arquitectura global en la fase de pre-juego; otros dicen que no hay una arquitectura global en Scrum, sino que la arquitectura y el diseño emanan de múltiples corridas [SB02]. No hay una ingeniería del software prescripta para Scrum; cada quien puede escoger entonces las prácticas de automatización, inspección de código, prueba unitaria, análisis o programación en pares que le resulten adecuadas.

Es habitual que Scrum se complemente con XP; en estos casos, Scrum suministra un marco de *management* basado en patrones organizacionales, mientras XP constituye la práctica de programación, usualmente orientada a objetos y con fuerte uso de patrones de diseño. Uno de los nombres que se utiliza para esta alianza es XP@Scrum. También son viables los híbridos con otros MAs.

Evolutionary Project Management (Evo)

Evo, creado por Tom Gilb, es el método iterativo ágil más antiguo. Se lo llama también Evolutionary Delivery, Evolutionary Management, Requirements Driven Project Management y Competitive Engineering. Fue elaborado inicialmente en Europa. En 1976 Gilb trató temas de desarrollo iterativo y gestión evolutiva en su clásico *Software metrics* [Gilb76], el texto que acuñó el concepto e inauguró el campo de las métricas de software. Luego desarrolló en profundidad esos temas en una serie de columnas en *Computer Weekly UK*. En 1981 publicó "Evolutionary Development" en *ACM Software Engineering Notes* y "Evolutionary Devivery versus the 'Waterfall Model'" en *ACM Sigsoft Software Requirements Engineering Notes*.

En la década de 1980 Gilb cayó bajo la influencia de los valores de W. Edward Deming y del método Planear-Hacer-Estudiar-Actuar (PDSA) de Walter Shewhart, que se constituyeron en modelos conceptuales subyacentes a Evo. Deming y Schewhart son, incidentalmente, considerados los padres del control estadístico de calidad; sus ideas, desarrolladas en las décadas de 1940 y 1950, se han aprovechado, por ejemplo, en la elaboración de estrategias como Six Sigma, en Lean Development o en la industria japonesa. En los 90s Gilb continuó el desarrollo de Evo, que tal vez fue más influyente por las ideas que éste proporcionara a XP, Scrum e incluso UP que por su éxito como método particular. El texto de Microsoft Press *Rapid Development* de Steve McConnell [McC96], que examina prácticas iterativas e incrementales en desarrollo de software, menciona ideas de Gilb en 14 secciones [Lar04]; todas esas referencias están ligadas a *best practices*.



Elementos de Evo [Gilb03b]

En las breves iteraciones de Evo, se efectúa un progreso hacia las máximas prioridades definidas por el cliente, liberando algunas piezas útiles para algunos participantes y solicitando su *feedback*. Esta es la práctica que se ha llamado Planeamiento Adaptativo Orientado al Cliente y Entrega Evolutiva. Otra idea distintiva de Evo es la clara definición, cuantificación, estimación y medida de los requerimientos de performance que necesitan mejoras. La performance incluye requisitos de calidad tales como robustez y tolerancia a fallas, al lado de estipulaciones cuantitativas de capacidad de carga y de ahorro de recursos. En Evo se espera que cada iteración constituya una re-evaluación de las soluciones en procura de la más alta relación de valor contra costo, teniendo en cuenta tanto el *feedback* como un amplio conjunto de estimaciones métricas. Evo requiere, igual que otros MAs, activa participación de los clientes. Todo debe cuantificarse; se

desalientan las apreciaciones cualitativas o subjetivas como “usable”, “mantenible” o “ergonómico”. A diferencia de otros MAs como Agile Modeling, donde la metodología es puntillosa pero discursiva, en Evo hay una especificación semántica y una pragmática rigurosa, completamente alejadas del sentido común, pero con la fundamentación que les presta derivarse de prácticas productivas suficientemente probadas.

Los diez principios fundamentales de Evo son:

1. Se entregarán temprano y con frecuencia resultados verdaderos, de valor para los participantes reales.
2. El siguiente paso de entrega de Evo será el que proporcione el mayor valor para el participante en ese momento.
3. Los pasos de Evo entregan los requerimientos especificados de manera evolutiva.
4. No podemos saber cuáles son los requerimientos por anticipado, pero podemos descubrirlos más rápidamente intentando proporcionar valor real a participantes reales.
5. Evo es ingeniería de sistemas holística (todos los aspectos necesarios del sistema deben ser completos y correctos) y con entrega a un ambiente de participantes reales (no es sólo sobre programación; es sobre satisfacción del cliente).
6. Los proyectos de Evo requieren una arquitectura abierta, porque habremos de cambiar las ideas del proyecto tan a menudo como se necesite hacerlo, para entregar realmente valor a nuestros participantes.
7. El equipo de proyecto de Evo concentrará su energía como equipo hacia el éxito del paso actual. En este paso tendrán éxito o fracasarán todos juntos. No gastarán energías en pasos futuros hasta que hayan dominado los pasos actuales satisfactoriamente.
8. Evo tiene que ver con aprendizaje a partir de la dura experiencia, tan rápido como se pueda: qué es lo que verdaderamente funciona, qué es lo que realmente entrega valor. Evo es una disciplina que nos hace confrontar nuestros problemas tempranamente, pero que nos permite progresar rápido cuando probadamente lo hemos hecho bien.
9. Evo conduce a una entrega temprana, a tiempo, tanto porque se lo ha priorizado así desde el inicio, y porque aprendemos desde el principio a hacer las cosas bien.
10. Evo debería permitirnos poner a prueba nuevos procesos de trabajo y deshacernos tempranamente de los que funcionan mal.

El modelo de Evo consiste en cinco elementos mayores [Gilb03b]:

1. **Metas, Valores y Costos** – Cuánto y cuántos recursos. Las Metas y Valores de los Participantes se llaman también, según la cultura, objetivos, metas estratégicas, requerimientos, propósitos, fines, ambiciones, cualidades e intenciones.
2. **Soluciones** – Banco de ideas sobre la forma de alcanzar Metas y Valores dentro del rango de los Costos.
3. **Estimación de Impacto** – Mapear las Soluciones a Metas y Costos para averiguar si se tienen ideas adecuadas para lograr las Metas dentro de los Costos.

4. **Plan Evolutivo** – Inicialmente una idea general de la secuencia a desarrollar y evolucionar hacia las Metas. Los detalles necesarios evolucionan junto con el resto del plan a medida que se desarrolla el producto/servicio.
5. **Funciones** – Describen qué hace el sistema. Son extremadamente secundarias, más de lo que se piensa, y deben mantenerse al mínimo.

A diferencia de lo que es el caso en IEEE 1471, donde todos, clientes y técnicos, son Participantes (*Stakeholders*), en Evo se llama Participante sólo al cliente. Cuando se inicia el ciclo, primero se definen los Valores y Metas del Participante; esta es una lista tradicional de recursos tales como dinero, tiempo y gente. Una vez que se comprende hacia dónde se quiere ir y cuándo se podría llegar ahí, se definen Soluciones para lograrlo. Utilizando una Tabla de Estimación de Impacto, se realiza la ingeniería de las Soluciones para satisfacer óptimamente las Metas y Valores de los Participantes. Se desarrolla un plan paso a paso llamado Entrega Evolutiva para entregar no soluciones, sino mejoras a dichas Metas y Valores. Inicialmente las Soluciones y el Plan de Entrega Evolutiva se delinean a un alto nivel de abstracción. Tomando ideas de las Soluciones y del Plan se detallan, desarrollan, verifican y entregan a los Participantes reales o a quién se encuentre tan cerca de ellos como se pueda llegar.

A medida que se desenvuelve el proyecto, se obtiene *feedback* en tiempo real sobre las mejoras que implica la Entrega Evolutiva sobre las Metas y Valores del Participante, así como sobre el consumo de Recursos. Esta información se usa para establecer qué es lo que está bien y lo que no, cuáles son los desafíos y qué es lo que no se sabía desde un principio; también se aprende sobre las nuevas tecnologías y técnicas que no estaban disponibles cuando el proyecto empezó. Se ajusta luego todo según se necesite, pero sin detallar las Soluciones o las Entregas Evolutivas hasta que se esté próximo a la entrega. Por último vienen las Funciones y Sub-Funciones, de las que se razona teniendo en cuenta que en rigor, en un nivel puro, son de hecho Soluciones a Metas.

El hijo de Tom, Kai Gilb, ha sido crítico sobre la necesidad de comprender bien las metas y no mezclarlas con otra cosa. Bajo el nombre de requerimientos se suelen mezclar soluciones y finalidades sin que nadie se sienta culpable. En Evo, cada una de las categorías ha sido objeto de un cuidadoso análisis semántico que busca establecer, además, por qué la gente suele comunicarse sobre el "Cómo" de una solución y no sobre "Cuán bien". En este marco, esa clase de distinciones (demasiado escrupulosas para detallarlas aquí) suele ser esencial. En esa lectura, por ejemplo, una Meta es un nivel de Calidad que se ha decidido alcanzar; como en inglés hay confusión entre calidad y cualidad (que se designan ambos con la misma palabra) Gilb luego dice que cada producto tiene muchas cualidades, y las considera atributos del sistema. Cuando se desarrolla un sistema, entonces, se decide qué niveles de calidad desearíamos que tuviera el sistema, y se las llama Metas.

Igual tratamiento se concede en Evo a las Funciones y Sub-Funciones, que se conciben más bien como Soluciones para Metas de los Participantes. Aunque hay un proceso bien definido para establecerlas y tratar con ellas, en Evo se recomienda mantener las funciones al mínimo, porque se estima además que una especificación funcional es una concepción obsoleta. La diferencia entre un procesador de texto como Word y la escritura con lápiz y papel, ilustra Gilb, *no* es funcional; en ambos casos se puede hacer lo mismo, sólo que se lo hace de distinta manera. Lo que el Participante requiere no debe ser

interpretado como la adquisición de nueva funcionalidad; por el contrario, el desastre de la industria de software, se razona aquí, se origina en que ha dependido demasiado de la funcionalidad. En Evo, sin duda, se debe razonar de otro modo: todo tiene que ver con las Metas y Valores de los Participantes, expresadas en términos tales que una Solución (o como se la llama también Diseño, Estrategia, Táctica, Proceso, Funcionalidad, Arquitectura y Método) pueda definirse como un medio para un fin, a través del cual se lleve de la situación en la que se está a otra situación que se desea.

Una herramienta desarrollada finamente en Evo es la que se llama "Herramienta-?"; consiste en preguntar "por qué" a cada meta o requerimiento aparente, haciéndolo además iterativamente sobre las respuestas que se van dando, a fin de deslindar el requisito real por detrás de la fachada de la contestación inicial. Unos pocos ejemplos demuestran la impensada eficacia de un principio que a primera vista parecería ser circular [Gilb03b]. Tras presentar otras herramientas de notación y detallar sus usos, llega la hora de poner en marcha un recurso fundamental de Evo, que es el Planguage. Se trata de un lenguaje estructurado de especificación que sirve para formalizar el requerimiento. El lenguaje es simple pero rico, así como son persuasivas las pautas que orientan su uso. Este es un ejemplo sencillo de la especificación de una meta de satisfacción del cliente en Planguage:

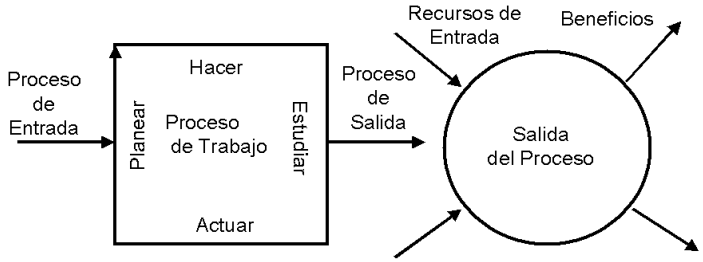
```
CUSTOMER.SATISFACTION
SCALE: evaluación promedio de la satisfacción de un cliente, de 1 a 5,
       siendo 1 la peor y 5 la mejor
PAST [2003] 2.5
GOAL [2004] 3.5
```

Planguage integra todas las disciplinas de solución de problemas. Mediante él, y usando el Método Planguage (PL), se pueden sujetar los medios a los fines. El método consiste en un lenguaje para Planificación y un conjunto de procesos de trabajo, el Lenguaje de Procesos. Proporciona un conjunto rico de formas de expresar propósito, restricciones, estrategia, diseño, bondad, inadecuación, riesgo, logro y credibilidad. Se inspira ampliamente en el ciclo Planear-Hacer-Estudiar-Actuar (PDSA) que Walter Shewhart aplicara desde 1930 y su discípulo Walter Deming impusiera en Japón.

En [Gilb03b] Planguage es elaborado y enseñado a medida que se describe el desarrollo de la metodología. En la descripción de los pasos de una solución, Gilb va cuestionando idea por idea el concepto tradicional de método, modelo, diseño y solución, así como las herramientas que se suponen inevitablemente ligadas a ellos. Un tratamiento crítico ejemplar merece, por ejemplo, el modelo en cascada, cuya idea subyacente de "flujo" atraviesa incluso a los modelos que creen basarse en una concepción distinta. Gilb lo ilustra con una caso real: "Los ingenieros de Microsoft no usan su propia herramienta tipo PERT [MS Project], porque éstas se basan en el modelo en cascada; no pueden hacerlo, porque ellos tienen proyectos reales, con desafíos, incógnitas, requerimientos cambiantes, nuevas tecnologías, competidores, etcétera. Ellos usan métodos que proporcionan *feedback* y aceptan cambios" [Gilb03b: 72].

Esencial en el tratamiento metodológico de Evo es la concepción que liga la evolución con el aprendizaje, lo que por otra parte es consistente con las modernas teorías evolutivas de John Holland [Hol95] y con las teorías que rigen los sistemas adaptativos complejos. Los ciclos de aprendizaje de Evo son exactamente lo mismo que el ciclo

Planear-Hacer-Estudiar-Actuar. Los principios que orientan la idea de la Entrega Evolutiva del Proyecto son:



Conceptos del Método Planguage, basado en [Gilb03a]

1. **Aprendizaje**: La Entrega Evolutiva es un ciclo de aprendizaje. Se aprende de la realidad y la experiencia; se aprende lo que funciona y lo que no.
2. **Temprano**: Aprender lo suficiente para cambiar lo que necesita cambiarse antes que sea tarde.
3. **Pequeño**: Pequeños pasos acarrean pequeños riesgos. Manteniendo el ciclo de entrega breve, es poco lo que se pierde cuando algo anda mal.
4. **Más simple**: Lo complejo se hace más simple y más fácil de manejar cuando se lo descompone en pequeñas partes.

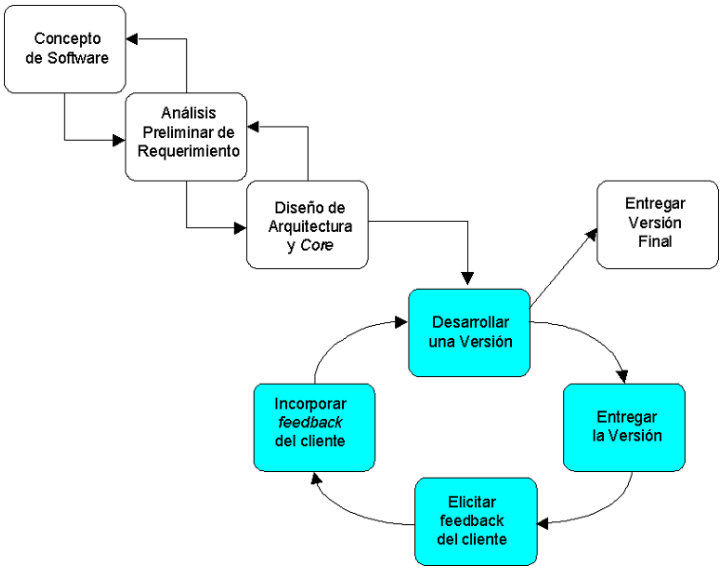
El carácter de aprendizaje que tiene el modelo de ciclos es evidente en el ejemplo:

> Plan a: Hacer un plan de alto nivel.
>
> Plan b: Descomponer ese plan en incrementos. Seleccionar un incremento a realizar primero.
>
> Hacer: Hacer el primer incremento. Entregarlo a los Participantes en un breve período de tiempo.
>
> Estudio: Mediar y estudiar cuán bien se comportó el incremento, comparándolo con las expectativas y aprendiendo de la experiencia.
>
> Actuar: Basado en lo que se aprendió, mantener el incremento o desecharlo, o introducir los cambios necesarios.

Otro marco que ha establecido la analogía entre evolución (o adaptación) y aprendizaje es, como se verá, el Adaptive Software Development de Jim Highsmith [Hig00a].

La evaluación que se realiza en el estudio debe apoyarse en una herramienta objetiva de inspección o control de calidad (un tema muy amplio de ingeniería de software); Evo implementa para ello Specification Quality Control (SQC), un método probado durante décadas, capaz de descubrir fallas que otros métodos pasan por alto. Se estima que SQC puede captar el 95% de las fallas y reducir tiempos de proyecto en un 50% [Gilb03a]; el uso de SQC, así como su rendimiento, está documentado en docenas de proyectos de misión crítica. Si bien SQC como lenguaje de especificación es complejo, como se trata de un estándar de industria existen herramientas automatizadas de alto nivel, como SQC for Excel™, un *add*-in desarrollado por BaRaN Systems (http://www.baran-systems.com/New/Products/SQC_For_Excel/index.htm), el *shareware* Premium SQC for Excel, SPC for MS Excel de Business Process Improvement (BPI) y otros productos similares.

Tom Gilb distingue claramente entre los Pasos de la Entrega Evolutiva y las iteraciones propias de los modelos iterativos tradicionales o de algún método ágil como RUP. Las iteraciones siguen un modelo de flujo, tienen un diseño prestablecido y son una secuencia de construcciones definidas desde el principio; los pasos evolutivos se definen primero de una manera genérica y luego se van refinando en cada ciclo, adoptando un carácter cada vez más formal. Las iteraciones no se realizan sólo para corregir los errores de código mediante refinamiento convencional, sino en función de la aplicación de SQC u otras herramientas con capacidades métricas.



Modelo de entrega evolutiva – Basado en [McC96]

En proyectos evolutivos, las Metas se desarrollan tratando de comprender de quiénes vienen (Participantes), qué es lo que son (medios y fines) y cómo expresarlas (cuantificables, medibles y verificables). Se procura pasar el menor tiempo posible en tareas de documentación. En las formas más simples y relajadas de Entrega Evolutiva, a veces llamada Entrega Incremental, liberar parte de una solución es un Paso de Entrega Evolutiva. En la Entrega Evolutiva más pura y más rica, sólo las mejoras en las Metas de los Participantes se consideran un Paso de Entrega Evolutiva. Hay que distinguir bien, asimismo, entre los Medios y los Fines, por un lado, y las Metas y las Soluciones por el otro. Las Metas deben separarse de las Soluciones; las Metas deben ser sagradas, y deben alcanzarse por cualquier medio; las Soluciones son sólo posibles, contingentes: son caballos de trabajo, y deben cambiarse cuando se consigue un caballo mejor. Evo incluye, al lado de Planguage, lineamientos de métrica (todo es cuantificable en él), procedimientos formales y orientaciones para descomponer procesos grandes en pasos, políticas de planeamiento y un conjunto de plantillas y tablas de Estimación de Impacto.

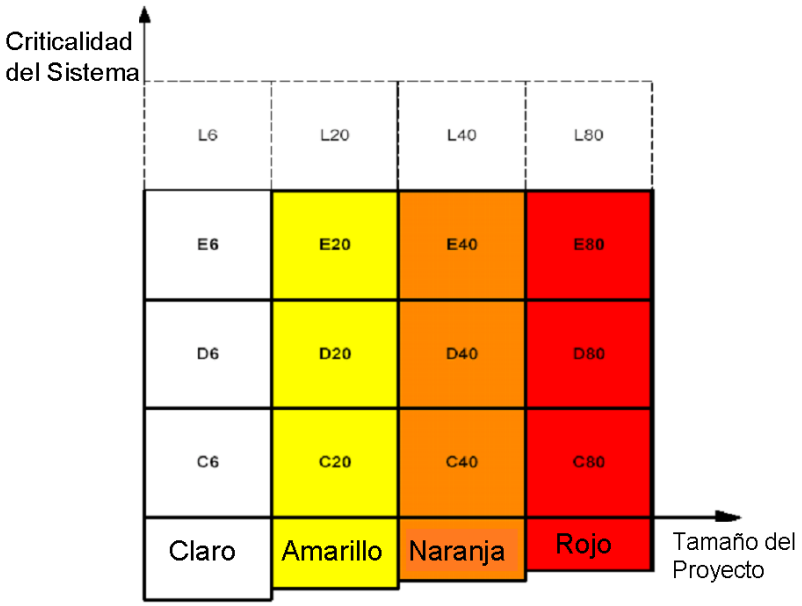
A diferencia de otras MAs que son más experimentales y que no tienen mucho respaldo de casos sistemáticamente documentados, Evo es una metodología probada desde hace mucho tiempo en numerosos clientes corporativos: la NASA, Lockheed Martin, Hewlett Packard, Douglas Aircraft, la Marina británica. El estándar MIL-STD-498 del Departamento de Defensa y su correspondiente estándar civil IEEE doc 12207 homologan el uso del modelo de entrega evolutiva. Sólo DSDM y RUP han logrado un reconocimiento comparable. Tom Gilb considera que las metodologías de Microsoft

reveladas en el best-seller *Microsoft Secrets* de Michael Cusumano y Richard Selby [CS95a], con su ciclo de construcción cotidiano a las 5 de la tarde y demás prácticas organizacionales, demuestra la vigencia de un método evolutivo. Todos los ejemplos de métodos evolutivos de uno de los libros mayores de Gilb [Gilb97] se ilustran con ejemplos de prácticas de Microsoft. En Evo no hay, por último, ni la sombra del carácter lúdico y la pintura de brocha gorda que se encuentran con frecuencia en Scrum o en XP. Se trata, por el contrario, de una metodología que impresiona por la precisión, relevancia e imaginación de sus fundamentaciones.

Crystal Methods

Las metodologías Crystal fueron creadas por el "antropólogo de proyectos" Alistair Cockburn, el autor que ha escrito los que tal vez sean los textos más utilizados, influyentes y recomendables sobre casos de uso, *Writing effective Use Cases* [Coc01] [Lar03]. Cockburn (quien siempre insiste que su apellido debe pronunciarse "Coburn" a la manera escocesa), escribe que mucha gente piensa que el desarrollo de software es una actividad de ingeniería. Esa comparación, piensa, es de hecho más perniciosa que útil, y nos lleva en una dirección equivocada.

Comparar el software con la ingeniería nos conduce a preguntarnos sobre "especificaciones" y "modelos" del software, sobre su completitud, corrección y vigencia. Esas preguntas son inconducentes, porque cuando pasa cierto tiempo no nos interesa que los modelos sean completos, que coincidan con el mundo "real" (sea ello lo que fuere) o que estén al día con la versión actual del lenguaje. Intentar que así sea es una pérdida de tiempo [Coc00]. En contra de esa visión ingenieril a la manera de un Bertrand Meyer, Cockburn ha alternado diversas visiones despreocupadamente contradictorias que alternativamente lo condujeron a adoptar XP en el sentido más radical, a sinergizarse con DSDM o LSD, a concebir el desarrollo de software como una forma comunitaria de poesía [Coc97a] o a elaborar su propia familia de Metodologías Crystal.



Familia de Crystal Methods [Crystalmethodologies.org]

La familia Crystal dispone un código de color para marcar la complejidad de una metodología: cuanto más oscuro un color, más "pesado" es el método. Cuanto más crítico es un sistema, más rigor se requiere. El código cromático se aplica a una forma tabular elaborada por Cockburn que se usa en muchos MAs para situar el rango de complejidad al cual se aplica una metodología. En la figura se muestra una evaluación de las pérdidas que puede ocasionar la falla de un sistema y el método requerido según este criterio. Los parámetros son Comodidad (C), Dinero Discrecional (D), Dinero Esencial (E) y Vidas (L). En otras palabras, la caída de un sistema que ocasione incomodidades indica que su nivel de criticalidad es C, mientras que si causa pérdidas de vidas su nivel es L. Los números del cuadro indican el número de personas afectadas a un proyecto, ±20%.

Los métodos se llaman Crystal evocando las facetas de una gema: cada faceta es otra versión del proceso, y todas se sitúan en torno a un núcleo idéntico. Hay cuatro variantes de metodologías: Crystal Clear ("Claro como el cristal") para equipos de 8 o menos integrantes; Amarillo, para 8 a 20; Naranja, para 20 a 50; Rojo, para 50 a 100. Se promete seguir con Marrón, Azul y Violeta. La más exhaustivamente documentada es Crystal Clear (CC), y es la que se ha de describir a continuación. CC puede ser usado en proyectos pequeños de categoría D6, aunque con alguna extensión se aplica también a niveles E8 a D10. El otro método elaborado en profundidad es el Naranja, apto para proyectos de duración estimada en 2 años, descripto en *Surviving Object-Oriented Projects* [Coc97b]. Los otros dos aún se están desarrollando. Como casi todos los otros métodos, CC consiste en valores, técnicas y procesos.

Los siete valores o propiedades de CC [Coc02] son:

1. **Entrega frecuente**. Consiste en entregar software a los clientes con frecuencia, no solamente en compilar el código. La frecuencia dependerá del proyecto, pero puede ser diaria, semanal, mensual o lo que fuere. La entrega puede hacerse sin despliegue, si es que se consigue algún usuario cortés o curioso que suministre *feedback*.
2. **Comunicación osmótica**. Todos juntos en el mismo cuarto. Una variante especial es disponer en la sala de un diseñador *senior*; eso se llama **Experto al Alcance de la Oreja**. Una reunión separada para que los concurrentes se concentren mejor es descripta como **El Cono del Silencio**.
3. **Mejora reflexiva**. Tomarse un pequeño tiempo (unas pocas horas cada algunas semanas o una vez al mes) para pensar bien qué se está haciendo, cotejar notas, reflexionar, discutir.
4. **Seguridad personal**. Hablar cuando algo molesta: decirle amigablemente al *manager* que la agenda no es realista, o a un colega que su código necesita mejorarse, o que sería conveniente que se bañase más seguido. Esto es importante porque el equipo puede descubrir y reparar sus debilidades. No es provechoso encubrir los desacuerdos con gentileza y conciliación. Técnicamente, estas cuestiones se han caracterizado como una importante variable de *confianza* y han sido estudiadas con seriedad en la literatura.
5. **Foco**. Saber lo que se está haciendo y tener la tranquilidad y el tiempo para hacerlo. Lo primero debe venir de la comunicación sobre dirección y prioridades, típicamente con el Patrocinador Ejecutivo. Lo segundo, de un ambiente en que la gente no se vea compelida a hacer otras cosas incompatibles.

6. **Fácil acceso a usuarios expertos**. Una comunicación de Keil a la ACM demostró hace tiempo, sobre una amplia muestra estadística, la importancia del contacto directo con expertos en el desarrollo de un proyecto. No hay un dogma de vida o muerte sobre esto, como sí lo hay en XP. Un encuentro semanal o semi-semanal con llamados telefónicos adicionales parece ser una buena pauta. Otra variante es que los programadores se entrenen para ser usuarios durante un tiempo. El equipo de desarrollo, de todas maneras, incluye un *Experto en Negocios*.
7. **Ambiente técnico con prueba automatizada, *management* de configuración e integración frecuente**. Microsoft estableció la idea de los *builds* cotidianos, y no es una mala práctica. Muchos equipos ágiles compilan e integran varias veces al día.

Crystal Clear no requiere ninguna estrategia o técnica, pero conviene tener unas cuantas a mano para empezar. Las estrategias documentadas favoritas, comunes a otros MAs, son:

1. **Exploración de 360°**. Verificar o tomar una muestra del valor de negocios del proyecto, los requerimientos, el modelo de dominio, la tecnología, el plan del proyecto y el proceso. La exploración es preliminar al desarrollo y equivale al período de incepción de RUP. Mientras en RUP esto puede demandar algunas semanas o meses, en Crystal Clear debe insumir unos pocos días; como mucho dos semanas si se requiere usar una tecnología nueva o inusual. El muestreo de valor de negocios se puede hacer verbalmente, con casos de uso u otros mecanismos de listas, pero debe resultar en una lista de los casos de uso esenciales del sistema. Respecto de la tecnología conviene correr unos pocos experimentos en base a lo que Cunningham y Beck han llamado *Spikes* (http://c2.com/cgi/wiki?SpikeSolution).
2. **Victoria temprana**. Es mejor buscar pequeños triunfos iniciales que aspirar a una gran victoria tardía. La fundamentación de esta estrategia proviene de algunos estudios sociológicos específicos. Usualmente la primera victoria temprana consiste en la construcción de un Esqueleto Ambulante. Conviene no utilizar la técnica de "lo peor primero" de XP, porque puede bajar la moral. La preferencia de Cockburn es "lo más fácil primero, lo más difícil segundo".
3. **Esqueleto ambulante**. Es una transacción que debe ser simple pero completa. Podría ser una rutina de consulta y actualización en un sistema cliente-servidor, o la ejecución de una transacción en un sistema transaccional de negocios. Un Esqueleto Ambulante no suele ser robusto; sólo *camina*, y carece de la carne de la funcionalidad de la aplicación real, que se agregará incrementalmente. Es diferente de un *Spike*, porque éste es "la más pequeña implementación que demuestra un éxito técnico plausible" y después se tira, porque puede ser desprolijo y peligroso; un *Spike* sólo se usa para saber si se está en la dirección correcta. El Esqueleto debe producirse con buenos hábitos de producción y pruebas de regresión, y está destinado a crecer con el sistema.
4. **Rearquitectura incremental**. Se ha demostrado que no es conveniente interrumpir el desarrollo para corregir la arquitectura. Más bien la arquitectura debe evolucionar en etapas, manteniendo el sistema en ejecución mientras ella se modifica.
5. **Radiadores de información**. Es una lámina pegada en algún lugar que el equipo pueda observar mientras trabaja o camina. Tiene que ser comprensible para el observador casual, entendida de un vistazo y renovada periódicamente para que valga

la pena visitarla. Puede estar en una página Web, pero es mejor si está en una pared, porque en una máquina se acumulan tantas cosas que ninguna llama la atención. Podría mostrar el conjunto de la iteración actual, el número de pruebas pasadas o pendientes, el número de casos de uso o historias entregado, el estado de los servidores, los resultados del último Taller de Reflexión. Una variante creativa es un sistema de semáforos implementado por Freeman, Benson y Borning.

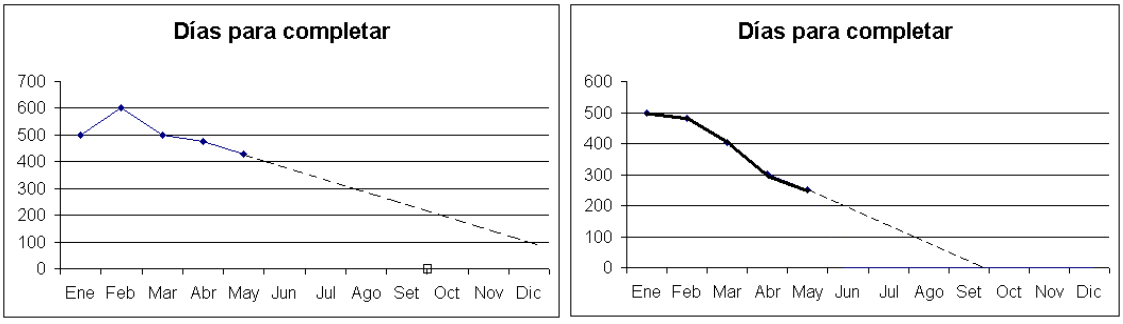
En cuanto a las técnicas, se favorecen:

1. **Entrevistas de proyectos**. Se suele entrevistar a más de un responsable para tener visiones más ricas. Cockburn ha elaborado una útil plantilla de dos páginas con entrevistas que probó ser útil en diversos proyectos. La idea es averiguar cuales son las prioridades, obtener una lista de rasgos deseados, saber cuáles son los requerimientos más críticos y cuáles los más negociables. Si se trata de una actualización o corrección, saber cuáles son las cosas que se hicieron bien y merecen preservarse y los errores que no se quieren repetir.
2. **Talleres de reflexión**. El equipo debe detenerse treinta minutos o una hora para reflexionar sobre sus convenciones de trabajo, discutir inconvenientes y mejoras y planear para el período siguiente. De aquí puede salir material para poner en un poster como Radiador de Información.
3. **Planeamiento Blitz**. Una técnica puede ser el Juego de Planeamiento de XP. En este juego, se ponen tarjetas indexadas en una mesa, con una historia de usuario o función visible en cada una. El grupo finge que no hay dependencias entre tarjetas, y las alinea en secuencias de desarrollo preferidas. Los programadores escriben en cada tarjeta el tiempo estimado para desarrollar cada función. El patrocinador o embajador del usuario escribe la secuencia de prioridades, teniendo en cuenta los tiempos referidos y el valor de negocio de cada función. Las tarjetas se agrupan en períodos de tres semanas llamados iteraciones que se agrupan en entregas (*releases*), usualmente no más largas de tres meses [Beck01]. Pueden usarse tarjetas CRC. Cockburn propone otras variantes del juego, como la Jam Session de Planeamiento del Proyecto[3] [Coc02]. Las diferencias entre la versión de Cockburn y el juego de XP son varias: en XP las tarjetas tienen historias, en CC listas de tareas; el juego de XP asume que no hay dependencias, el de CC que sí las hay; en XP hay iteraciones de duración fija, en CC no presupone nada sobre la longitud de la iteración.
4. **Estimación Delphi con estimaciones de pericia**. La técnica se llama así por analogía con el oráculo de Delfos; se la describió por primera vez en el clásico *Surviving Object-Oriented Projects* de Cockburn [Coc97b], reputado como uno de los mejores libros sobre el paradigma de objetos. En el proceso Delphi se reúnen los expertos responsables y proceden como en un remate para proponer el tamaño del sistema, su tiempo de ejecución, la fecha de las entregas según dependencias técnicas y de

[3] Hay aquí un juego de palabras deliberado que contrasta esta "Jam Session" con las "JAD Sessions" propias del RAD tradicional. Éstas eran reuniones intensivas donde participaba todo el mundo (técnicos y gente de negocios); duraban todo un día y se hacían lejos de la oficina. Estas excursiones lejos del lugar de trabajo no son comunes en los MAs. Las sesiones JAD son un proceso complejo y definido, sobre el cual hay abundante literatura [WS95].

negocios y para equilibrar las entregas en paquetes de igual tamaño. La descripción del remate está en [Coc02] e insume tres o cuatro páginas .

5. **Encuentros diarios de pie**. La palabra clave es "brevedad", cinco a diez minutos como máximo. No se trata de discutir problemas, sino de identificarlos. Los problemas sólo se discuten en otros encuentros posteriores, con la gente que tiene que ver en ellos. La técnica se origina en Scrum. Se deben hacer de pie para que la gente no escriba en sus *notebooks*, garabatee papeles o se quede dormida.
6. **Miniatura de procesos**. La "Hora Extrema" fue inventada por Peter Merel para introducir a la gente en XP en 60 minutos (http://c2.com/cgi/wiki?ExtremeHour) y proporciona lineamientos canónicos que pueden usarse para articular esta práctica. Una forma de presentar Crystal Clear puede insumir entre 90 minutos y un día. La idea es que la gente pueda "degustar" la nueva metodología.
7. **Gráficos de quemado**. Su nombre viene de los gráficos de quemado de calorías de los regímenes dietéticos; se usan también en Scrum. Se trata de una técnica de graficación para descubrir demoras y problemas tempranamente en el proceso, evitando que se descubra demasiado tarde que todavía no se sabe cuánto falta. Para ello se hace una estimación del tiempo faltante para programar lo que resta al ritmo actual, lo cual sirve para tener dominio de proyectos en los cuales las prioridades cambian bruscamente y con frecuencia. Esta técnica se asocia con algunos recursos ingeniosos, como la *Lista Témpana*, llamada así porque se refiere al agregado de ítems con alta prioridad en el tope de las listas de trabajos pendientes, esperando que los demás elementos se hundan bajo la línea de flotación; los elementos que están sobre la línea se entregarán en la iteración siguiente, los que están por debajo en las restantes. En otros MAs la *Lista Témpana* no es otra cosa que un gráfico de retraso. Los gráficos de quemado ilustran la velocidad del proceso, analizando la diferencia entre las líneas proyectadas y efectivas de cada entrega, como se ilustra en las figuras.
8. **Programación lado a lado**. Mucha gente siente que la programación en pares de XP involucra una presión excesiva; la versión de Crystal Clear establece proximidad, pero cada quien se aboca a su trabajo asignado, prestando un ojo a lo que hace su compañero, quien tiene su propia máquina. Esta es una ampliación de la Comunicación Osmótica al contexto de la programación.

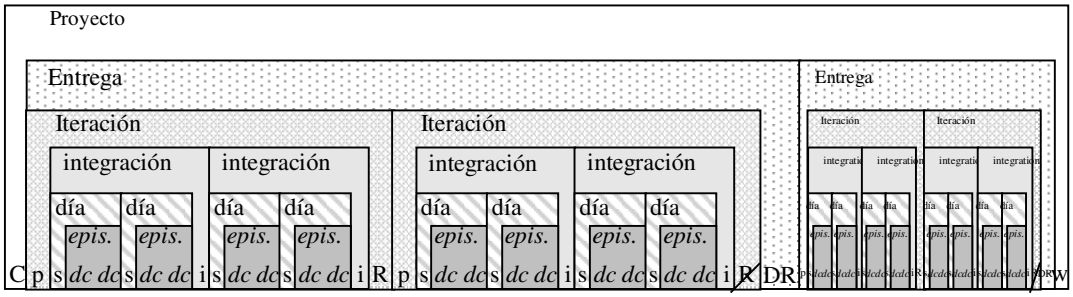


Gráficos de quemado – Con necesidad de recortar retrasos (izq.) y con entrega proyectada en término. Medición realizada en mayo – La fecha de entrega proyectada es el 1° de octubre

El proceso de CC se basa en una exploración refinada de los inconvenientes de los modelos clásicos. Dice Cockburn que la mayoría de los modelos de proceso propuestos

entre 1970 y 2000 se describían como secuencias de pasos. Aún cuando se recomendaran iteraciones e incrementos (que no hacían más que agregar confusión a la interpretación) los modelos parecían dictar un proceso en cascada, por más que los autores aseguraran que no era así. El problema con estos procesos es que realmente están describiendo un *workflow* requerido, un grafo de dependencia: el equipo no puede entregar un sistema hasta que está integrado y corre. No puede integrar y verificar hasta que el código no está escrito y corriendo. Y no puede diseñar y escribir el código hasta que se le dice cuáles son los requerimientos. Un grafo de dependencia se interpreta necesariamente en ese sentido, aunque no haya sido la intención original.

En lugar de esta interpretación lineal, CC enfatiza el proceso como un conjunto de ciclos anidados. En la mayoría de los proyectos se perciben siete ciclos: (1) el proyecto, (2) el ciclo de entrega de una unidad, (3) la iteración (nótese que CC requiere múltiples entregas por proyecto pero no muchas iteraciones por entrega), (4) la semana laboral, (5) el período de integración, de 30 minutos a tres días, (6) el día de trabajo, (7) el episodio de desarrollo de una sección de código, de pocos minutos a pocas horas.



Ciclos anidados de Crystal Clear [Coc02]

Cockburn subraya que interpretar no linealmente un modelo de ciclos es difícil; la mente se resiste a hacerlo. Él mismo asegura que estuvo diez años tratando de explicar el uso de un proceso cíclico y sólo recientemente ha logrado intuir cómo hacerlo. La figura muestra los ciclos y las actividades conectadas a ellos. Las letras denotan *Chartering*, planeamiento de iteración, reunión diaria de pie (*standup*), desarrollo, *check-in*, integración, taller de Reflexión, Entrega (*Delivery*), y empaquetado del proyecto (*Wrapup*).

Hay ocho roles nominados en CC: Patrocinador, Usuario Experto, Diseñador Principal, Diseñador-Programador, Experto en Negocios, Coordinador, Verificador, Escritor. En Crystal Naranja se agregan aun más roles: Diseñador de IU, Diseñador de Base de Datos, Experto en Uso, Facilitador Técnico, Analista/Diseñador de Negocios, Arquitecto, Mentor de Diseño, Punto de Reutilización. A continuación se describen en bastardilla los artefactos de los que son responsables los roles de CC, detalladamente descriptos en la documentación.

1. **Patrocinador**. Produce la *Declaración de Misión con Prioridades de Compromiso (Tradeoff)*. Consigue los recursos y define la totalidad del proyecto.
2. **Usuario Experto**. Junto con el Experto en Negocios produce la *Lista de Actores-Objetivos* y el *Archivo de Casos de Uso y Requerimientos*. Debe familiarizarse con el uso del sistema, sugerir atajos de teclado, modos de operación, información a visualizar simultáneamente, navegación, etcétera.

3. **Diseñador Principal**. Produce la *Descripción Arquitectónica*. Se supone que debe ser al menos un profesional de Nivel 3. (En MAs se definen tres niveles de experiencia: Nivel 1 es capaz de "seguir los procedimientos"; Nivel 2 es capaz de "apartarse de los procedimientos específicos" y encontrar otros distintos; Nivel 3 es capaz de manejar con fluidez, mezclar e inventar procedimientos). El Diseñador Principal tiene roles de coordinador, arquitecto, mentor y programador más experto.
4. **Diseñador-Programador**. Produce, junto con el Diseñador Principal, los *Borradores de Pantallas*, el *Modelo Común de Dominio*, las *Notas y Diagramas de Diseño*, el *Código Fuente*, el *Código de Migración*, las *Pruebas* y el *Sistema Empaquetado*. Cockburn no distingue entre diseñadores y programadores. Un programa en CC es "diseño y programa"; sus programadores son diseñadores-programadores. En CC un diseñador que no programe no tiene cabida.
5. **Experto en Negocios**. Junto con el Usuario Experto produce la *Lista de Actores-Objetivos* y el *Archivo de Casos de Uso y Requerimientos*. Debe conocer las reglas y políticas del negocio.
6. **Coordinador**. Con la ayuda del equipo, produce el *Mapa de Proyecto*, el *Plan de Entrega*, el *Estado del Proyecto*, la *Lista de Riesgos*, el *Plan y Estado de Iteración* y la *Agenda de Visualización*.
7. **Verificador**. Produce el *Reporte de Bugs*. Puede ser un programador en tiempo parcial, o un equipo de varias personas.
8. **Escritor**. Produce el *Manual de Usuario*.

El **Equipo como Grupo** es responsable de producir la *Estructura y Convenciones del Equipo* y los *Resultados del Taller de Reflexión*.

A pesar que no contempla el desarrollo de software propiamente dicho, CC involucra unos veinte productos de trabajo o artefactos. Mencionamos los más importantes:

1. Declaración de la misión. Documento de un párrafo a una página, describiendo el propósito.
2. Estructura del equipo. Lista de equipos y miembros.
3. Metodología. Comprende roles, estructura, proceso, productos de trabajo que mantienen, métodos de revisión.
4. Secuencia de entrega. Declaración o diagrama de dependencia; muestra el orden de las entregas y lo que hay en cada una.
5. Cronograma de visualización y entrega. Lista, planilla de hoja de cálculo o herramienta de gestión de proyectos.
6. Lista de riesgos. Descripción de riesgos por orden descendente de prioridad.
7. Estatus del proyecto. Lista hitos, fecha prevista, fecha efectiva y comentarios.
8. Lista de actores-objetivos. Lista de dos columnas, planilla de hoja de cálculo, diagrama de caso de uso o similar.
9. Casos de uso anotados. Requerimientos funcionales.
10. Archivo de requerimientos. Colección de información indicando qué se debe construir, quiénes han de utilizarlo, de qué manera proporciona valor y qué restricciones afectan al diseño.

Los métodos Crystal no prescriben las prácticas de desarrollo, las herramientas o los productos que pueden usarse, pudiendo combinarse con otros métodos como Scrum, XP y Microsoft Solutions Framework. En su comentario a [Hig00b], Cockburn confiesa que cuando imaginó CC pensaba proporcionar un método ligero; comparado con XP, sin embargo, CC resulta muy pesado. CC es más fácil de aprender e implementar; a pesar de su jerga chocante XP es más disciplinado, piensa Cockburn; pero si un equipo ligero puede tolerar sus rigores, lo mejor será que se mude a XP.

Feature Driven Development (FDD)

Feature Oriented Programming (FOP) es una técnica de programación guiada por rasgos o características (*features*) y centrada en el usuario, no en el programador; su objetivo es sintetizar un programa conforme a los rasgos requeridos [Bat03]. En un desarrollo en términos de FOP, los objetos se organizan en módulos o capas conforme a rasgos. FDD, en cambio, es un método ágil, iterativo y adaptativo. A diferencia de otros MAs, no cubre todo el ciclo de vida sino sólo las fases de diseño y construcción y se considera adecuado para proyectos mayores y de misión crítica. FDD es, además, marca registrada de una empresa, Nebulon Pty. Aunque hay coincidencias entre la programación orientada por rasgos y el desarrollo guidado por rasgos, FDD no necesariamente implementa FOP.

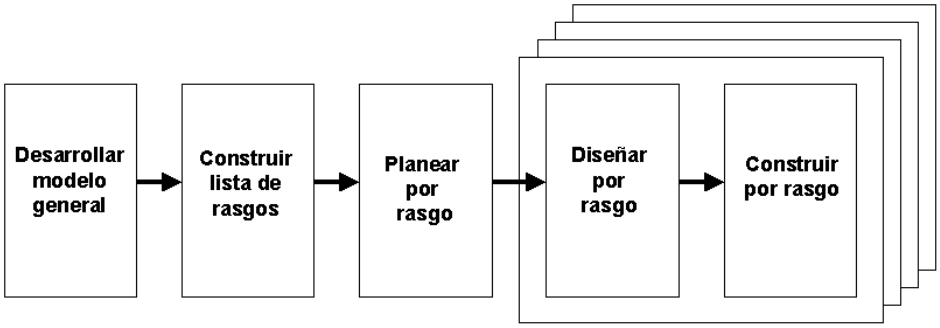
FDD no requiere un modelo específico de proceso y se complementa con otras metodologías. Enfatiza cuestiones de calidad y define claramente entregas tangibles y formas de evaluación del progreso. Se lo reportó por primera vez en un libro de Peter Coad, Eric Lefebvre y Jeff DeLuca *Java Modeling in Color with UML* [CLD00]; luego fue desarrollado con amplitud en un proyecto mayor de desarrollo por DeLuca, Coad y Stephen Palmer. Su implementación de referencia, análogo al C3 de XP, fue el Singapore Project; DeLuca había sido contratado para salvar un sistema muy complicado para el cual el contratista anterior había producido, luego de dos años, 3500 páginas de documentación y ninguna línea de código. Naturalmente, el proyecto basado en FDD fue todo un éxito, y permitió fundar el método en un caso real de misión crítica.

Los principios de FDD son pocos y simples:

- Se requiere un sistema para construir sistemas si se pretende escalar a proyectos grandes.
- Un proceso simple y bien definido trabaja mejor.
- Los pasos de un proceso deben ser lógicos y su mérito inmediatamente obvio para cada miembro del equipo.
- Vanagloriarse del proceso puede impedir el trabajo real.
- Los buenos procesos van hasta el fondo del asunto, de modo que los miembros del equipo se puedan concentrar en los resultados.
- Los ciclos cortos, iterativos, orientados por rasgos (*features*) son mejores.

Hay tres categorías de rol en FDD: roles claves, roles de soporte y roles adicionales. Los seis roles claves de un proyecto son: (1) administrador del proyecto, quien tiene la última palabra en materia de visión, cronograma y asignación del personal; (2) arquitecto jefe (puede dividirse en arquitecto de dominio y arquitecto técnico); (3) *manager* de desarrollo, que puede combinarse con arquitecto jefe o *manager* de proyecto; (4)

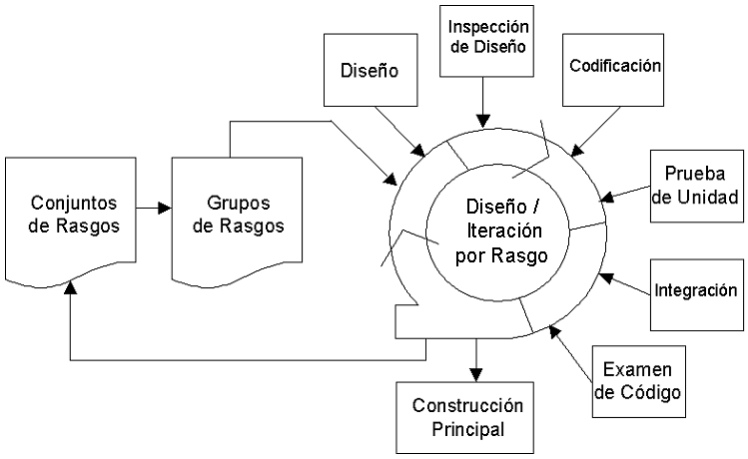
programador jefe, que participa en el análisis del requerimiento y selecciona rasgos del conjunto a desarrollar en la siguiente iteración; (5) propietarios de clases, que trabajan bajo la guía del programador jefe en diseño, codificación, prueba y documentación, repartidos por rasgos y (6) experto de dominio, que puede ser un cliente, patrocinador, analista de negocios o una mezcla de todo eso.



Proceso FDD, basado en [http://togethercommunities.com]

Los cinco roles de soporte comprenden (1) administrador de entrega, que controla el progreso del proceso revisando los reportes del programador jefe y manteniendo reuniones breves con él; reporta al manager del proyecto; (2) abogado/guru de lenguaje, que conoce a la perfección el lenguaje y la tecnología; (3) ingeniero de construcción, que se encarga del control de versiones de los *builds* y publica la documentación; (4) herramientista (*toolsmith*), que construye herramientas ad hoc o mantiene bases de datos y sitios Web y (5) administrador del sistema, que controla el ambiente de trabajo o productiza el sistema cuando se lo entrega.

Los tres roles adicionales son los de verificadores, encargados del despliegue y escritores técnicos. Un miembro de un equipo puede tener otros roles a cargo, y un solo rol puede ser compartido por varias personas.



Ciclo de FDD, basado en [ASR+02]

FDD consiste en cinco procesos secuenciales durante los cuales se diseña y construye el sistema. La parte iterativa soporta desarrollo ágil con rápidas adaptaciones a cambios en requerimientos y necesidades del negocio. Cada fase del proceso tiene un criterio de entrada, tareas, pruebas y un criterio de salida. Típicamente, la iteración de un rasgo insume de una a tres semanas. Las fases son:

1) **Desarrollo de un modelo general**. Cuando comienza este desarrollo, los expertos de dominio ya están al tanto de la visión, el contexto y los requerimientos del sistema a construir. A esta altura se espera que existan requerimientos tales como casos de uso o especificaciones funcionales. FDD, sin embargo, no cubre este aspecto. Los expertos de dominio presentan un ensayo (*walkthrough*) en el que los miembros del equipo y el arquitecto principal se informan de la descripción de alto nivel del sistema. El dominio general se subdivide en áreas más específicas y se define un ensayo más detallado para cada uno de los miembros del dominio. Luego de cada ensayo, un equipo de desarrollo trabaja en pequeños grupos para producir modelos de objeto de cada área de dominio. Simultáneamente, se construye un gran modelo general para todo el sistema.
2) **Construcción de la lista de rasgos**. Los ensayos, modelos de objeto y documentación de requerimientos proporcionan la base para construir una amplia lista de rasgos. Los rasgos son pequeños ítems útiles a los ojos del cliente. Son similares a las tarjetas de historias de XP y se escriben en un lenguaje que todas las partes puedan entender. Las funciones se agrupan conforme a diversas actividades en áreas de dominio específicas. La lista de rasgos es revisada por los usuarios y patrocinadores para asegurar su validez y exhaustividad. Los rasgos que requieran más de diez días se descomponen en otros más pequeños.
3) **Planeamiento por rasgo**. Incluye la creación de un plan de alto nivel, en el que los conjuntos de rasgos se ponen en secuencia conforme a su prioridad y dependencia, y se asigna a los programadores jefes. Las listas se priorizan en secciones que se llaman paquetes de diseño. Luego se asignan las clases definidas en la selección del modelo general a programadores individuales, o sea propietarios de clases. Se pone fecha para los conjuntos de rasgos.
4) **Diseño por rasgo y Construcción por rasgo**. Se selecciona un pequeño conjunto de rasgos del conjunto y los propietarios de clases seleccionan los correspondientes equipos dispuestos por rasgos. Se procede luego iterativamente hasta que se producen los rasgos seleccionados. Una iteración puede tomar de unos pocos días a un máximo de dos semanas. Puede haber varios grupos trabajando en paralelo. El proceso iterativo incluye inspección de diseño, codificación, prueba de unidad, integración e inspección de código. Luego de una iteración exitosa, los rasgos completos se promueven al *build* principal. Este proceso puede demorar una o dos semanas en implementarse.

FDD consiste en un conjunto de "mejores prácticas" que distan de ser nuevas pero se combinan de manera original. Las prácticas canónicas son:

- Modelado de objetos del dominio, resultante en un *framework* cuando se agregan los rasgos. Esta forma de modelado descompone un problema mayor en otros menores; el diseño y la implementación de cada clase u objeto es un problema pequeño a resolver. Cuando se combinan las clases completas, constituyen la solución al problema mayor. Una forma particular de la técnica es el modelado en colores [CLD00], que agrega una dimensión adicional de visualización. Si bien se puede modelar en blanco y negro, en FDD el modelado basado en objetos es imperativo.

- Desarrollo por rasgo. Hacer simplemente que las clases y objetos funcionen no refleja lo que el cliente pide. El seguimiento del progreso se realiza mediante examen de pequeñas funcionalidades descompuestas y funciones valoradas por el cliente. Un rasgo en FDD es una función pequeña expresada en la forma <acción> <resultado> <por | para | de | a> <objeto> con los operadores adecuados entre los términos. Por ejemplo, calcular el importe total de una venta; determinar la última operación de un cajero; validar la contraseña de un usuario.
- Propiedad individual de clases (código). Cada clase tiene una sola persona nominada como responsable por su consistencia, performance e integridad conceptual.
- Equipos de Rasgos, pequeños y dinámicamente formados. La existencia de un equipo garantiza que un conjunto de mentes se apliquen a cada decisión y se tomen en cuenta múltiples alternativas.
- Inspección. Se refiere al uso de los mejores mecanismos de detección conocidos. FDD es tan escrupuloso en materia de inspección como lo es Evo.
- *Builds* regulares. Siempre se tiene un sistema disponible. Los *builds* forman la base a partir de la cual se van agregando nuevos rasgos.
- Administración de configuración. Permite realizar seguimiento histórico de las últimas versiones completas de código fuente.
- Reporte de progreso. Se comunica a todos los niveles organizacionales necesarios.

FDD suministra un rico conjunto de artefactos para la planificación y control de los proyectos. En http://www.nebulon.com/articles/fdd/fddimplementations.html se encuentran diversos formularios y tablas con información real de implementaciones de FDD: Vistas de desarrollo, Vistas de planificación, Reportes de progreso, Reportes de tendencia, Vista de plan (<acción><resultado><objeto>), etcétera. Se han desarrollado también algunas herramientas que generan vistas combinadas o específicas.

Id	Descripción	Prog. Jefe.	Prop. Clase	Ensayo		Diseño		Inspección de Diseño		Código		Inspección de Código		Promover a Build	
				Plan	Actual	Plan	Actual	Plan	Actual	Plan	Actual	Plan	Actual	Plan	Actual
MD125	Validar los límites transaccionales de un CAO contra una instrucción de implementación	CP	ABC	23/12/98	23/12/98	31/01/99	31/01/99	01/02/99	01/02/99	10/02/99		18/02/99		20/02/99	
MD126	Definir el estado de una instrucción de implementación	CP	ABC	23/12/98	23/12/98	31/01/99	31/01/99	01/02/99	01/02/99	10/02/99		18/02/99		20/02/99	
MD127	Especificar el oficial de autorización de una instrucción de implementación	CP	ABC	23/12/98	23/12/98	31/01/99	31/01/99	01/02/99	01/02/99	10/02/99		18/02/99		20/02/99	
MD128	Rechazar una instrucción de implementación para un conjunto de líneas	CP	ABC	STATUS: Inactivo NOTA: [agregado por CK: 3/2/99] No aplicable											
MD129	Confirmar una instrucción de implementación para un conjunto de líneas	CP	ABC	23/12/98	23/12/98	31/01/99	31/01/99	01/02/99	01/02/99	10/02/99		18/02/99		20/02/99	
MD130	Determinar si todos los documentos se han completado para un prestatario	CP	ABC	23/12/98	23/12/98	31/01/99	31/01/99	01/02/99	01/02/99	05/02/99		08/02/99		10/02/99	
				NOTA: [agregado por SL: 3/2/99] Bloqueado en AS											
MD131	Validar los límites transaccionales de un CAO contra una instrucción de desembolso	CP	ABC	23/12/98	23/12/98	31/01/99	31/01/99	01/02/99	01/02/99	05/02/99		08/02/99		10/02/99	
				NOTA: [agregado por: 3/2/99] Atrasado según estimaciones iniciales											
MD132	Enviar para autorización una instrucción de implementación	CP	ABC	23/12/98	23/12/98	31/01/99	31/01/99	01/02/99	01/02/99	05/02/99	05/02/99	06/02/99	06/02/99	08/02/99	08/02/99
MD133	Validar fecha de baja de una instrucción de implementación	CP	ABC	23/12/98	23/12/98	31/01/99	31/01/99	01/02/99	01/02/99	05/02/99	05/02/99	06/02/99	06/02/99	08/02/99	08/02/99

Plan de rasgo – Implementación – Basado en http://www.nebulon.com/articles/fdd/planview.html

La matriz muestra un ejemplo de vista de un plan de rasgo, con la típica codificación en colores.

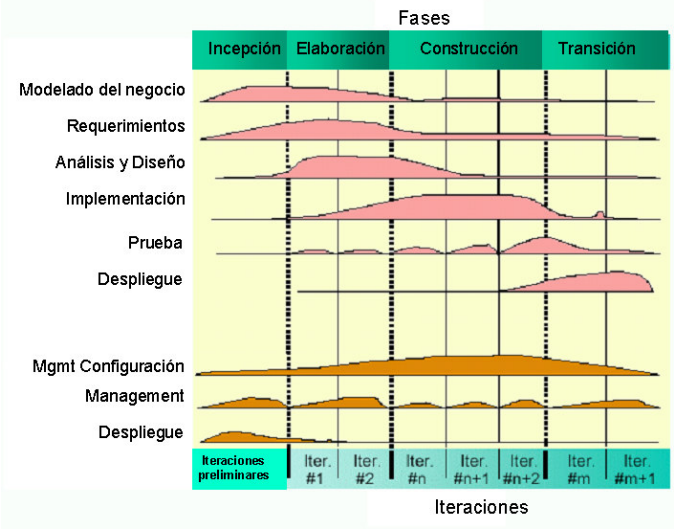
En síntesis, FDD es un método de desarrollo de ciclos cortos que se concentra en la fase de diseño y construcción. En la primera fase, el modelo global de dominio es elaborado por expertos del dominio y desarrolladores; el modelo de dominio consiste en diagramas de clases con clases, relaciones, métodos y atributos. Los métodos no reflejan conveniencias de programación sino rasgos funcionales.

Algunos agilistas sienten que FDD es demasiado jerárquico para ser un método ágil, porque demanda un programador jefe, quien dirige a los propietarios de clases, quienes dirigen equipos de rasgos. Otros críticos sienten que la ausencia de procedimientos detallados de prueba en FDD es llamativa e impropia. Los promotores del método aducen que las empresas ya tienen implementadas sus herramientas de prueba, pero subsiste el problema de su adecuación a FDD. Un rasgo llamativo de FDD es que no exige la presencia del cliente.

FDD se utilizó por primera vez en grandes aplicaciones bancarias a fines de la década de 1990. Los autores sugieren su uso para proyectos nuevos o actualizaciones de sistemas existentes, y recomiendan adoptarlo en forma gradual. En [ASR+02] se asegura que aunque no hay evidencia amplia que documente sus éxitos, las grandes consultoras suelen recomendarlo incluso para delicados proyectos de misión crítica.

Rational Unified Process (RUP)

Uno podría preguntarse legítimamente qué hace RUP en el contexto de los MAs ¿No es más bien representativo de la filosofía a la que el Manifiesto se opone? ¿Están tratando los métodos clásicos de cooptar a los métodos nuevos? El hecho es que existe una polémica aún en curso respecto de si los métodos asociados al Proceso Unificado, y en particular RUP, representan técnicas convencionales y pesadas o si por el contrario son adaptables al programa de los MAs.



Fases y *workflows* de RUP, basado en [BMP98]

Philippe Kruchten, impulsor tanto del famoso modelo de vistas 4+1 como de RUP, ha participado en el célebre "debate de los gurúes" [Hig01] afirmando que RUP es particularmente apto para ambas clases de escenarios. Kruchten ha sido en general conciliador; aunque el diseño orientado a objetos suele otorgar a la Arquitectura de Software académica un lugar modesto, el modelo de 4+1 [Kru95] se inspira explícitamente en el modelo arquitectónico fundado por Dewayne Perry y Alexander Wolf [PW92]. Recíprocamente, 4+1 tiene cabida en casi todos los métodos basados en arquitectura desarrollados recientemente por el SEI. Highsmith, comentando la movilidad de RUP de un campo a otro, señaló que nadie, en apariencia, está dispuesto a conceder "agilidad" a sus rivales; RUP pretende ser ágil, y tal vez lo sea. De hecho, los principales textos que analizan MAs acostumbran a incluir al RUP entre los más representativos, o como un método que no se puede ignorar [ASR+02] [Lar04].

Algunos autores no admiten esta concesión. En un interesante análisis crítico, Wolfgang Hesse, de la Universidad de Marburg [HesS/f] ha señalado que aunque el RUP se precia de sus cualidades iterativas, incrementales y centradas en la arquitectura, de hecho responde subterráneamente a un rígido modelo de fases, no posee capacidades recursivas suficientes y sus definiciones dinámicas son demasiado engorrosas y recargadas como para ser de utilidad en un contexto cambiante.

El proceso de ciclo de vida de RUP se divide en cuatro fases bien conocidas llamadas Incepción, Elaboración, Construcción y Transición. Esas fases se dividen en iteraciones, cada una de las cuales produce una pieza de software demostrable. La duración de cada iteración puede extenderse desde dos semanas hasta seis meses. Las fases son:

1. **Incepción**. Significa "comienzo", pero la palabra original (de origen latino y casi en desuso como sustantivo) es sugestiva y por ello la traducimos así. Se especifican los objetivos del ciclo de vida del proyecto y las necesidades de cada participante. Esto entraña establecer el alcance y las condiciones de límite y los criterios de aceptabilidad. Se identifican los casos de uso que orientarán la funcionalidad. Se diseñan las arquitecturas candidatas y se estima la agenda y el presupuesto de todo el proyecto, en particular para la siguiente fase de elaboración. Típicamente es una fase breve que puede durar unos pocos días o unas pocas semanas.
2. **Elaboración**. Se analiza el dominio del problema y se define el plan del proyecto. RUP presupone que la fase de elaboración brinda una arquitectura suficientemente sólida junto con requerimientos y planes bastante estables. Se describen en detalle la infraestructura y el ambiente de desarrollo, así como el soporte de herramientas de automatización. Al cabo de esta fase, debe estar identificada la mayoría de los casos de uso y los actores, debe quedar descripta la arquitectura de software y se debe crear un prototipo de ella. Al final de la fase se realiza un análisis para determinar los riesgos y se evalúan los gastos hechos contra los originalmente planeados.
3. **Construcción**. Se desarrollan, integran y verifican todos los componentes y rasgos de la aplicación. RUP considera que esta fase es un proceso de manufactura, en el que se debe poner énfasis en la administración de los recursos y el control de costos, agenda y calidad. Los resultados de esta fase (las versiones alfa, beta y otras versiones de prueba) se crean tan rápido como sea posible. Se debe compilar también una versión de entrega. Es la fase más prolongada de todas.

4. **Transición**. Comienza cuando el producto está suficientemente maduro para ser entregado. Se corrigen los últimos errores y se agregan los rasgos pospuestos. La fase consiste en prueba beta, piloto, entrenamiento a usuarios y despacho del producto a mercadeo, distribución y ventas. Se produce también la documentación. Se llama transición porque se transfiere a las manos del usuario, pasando del entorno de desarrollo al de producción.

A través de las fases se desarrollan en paralelo nueve *workflows* o disciplinas: Modelado de Negocios, Requerimientos, Análisis & Diseño, Implementación, Prueba, Gestión de Configuración & Cambio, Gestión del Proyecto y Entorno. Además de estos *workflows*, RUP define algunas prácticas comunes:

1. **Desarrollo iterativo de software**. Las iteraciones deben ser breves y proceder por incrementos pequeños. Esto permite identificar riesgos y problemas tempranamente y reaccionar frente a ellos en consecuencia.
2. **Administración de requerimientos**. Identifica requerimientos cambiantes y postula una estrategia disciplinada para administrarlos.
3. **Uso de arquitecturas basadas en componentes**. La reutilización de componentes permite asimismo ahorros sustanciales en tiempo, recursos y esfuerzo.
4. **Modelado visual del software.** Se deben construir modelos visuales, porque los sistemas complejos no podrían comprenderse de otra manera. Utilizando una herramienta como UML, la arquitectura y el diseño se pueden especificar sin ambigüedad y comunicar a todas las partes involucradas.
5. **Prueba de calidad del software**. RUP pone bastante énfasis en la calidad del producto entregado.
6. **Control de cambios y trazabilidad**. La madurez del software se puede medir por la frecuencia y tipos de cambios realizados.

Aunque RUP es extremadamente locuaz en muchos respectos, no proporciona lineamientos claros de implementación que puedan compararse, por ejemplo, a los métodos Crystal, en los que se detalla la documentación requerida y los roles según diversas escalas de proyecto. En RUP esas importantes decisiones de dejan a criterio del usuario. Se asegura [Kru00] que RUP puede implementarse “sacándolo de la caja”, pero dado que el número de sus artefactos y herramientas es inmenso, siempre se dice que hay que recortarlo y adaptarlo a cada caso. El proceso de implementación mismo es complejo, dividiéndose en seis fases cíclicas.

Existe una versión recortada de RUP, dX de Robert Martin, en la cual se han tomado en consideración experiencias de diversos MAs, reduciendo los artefactos de RUP a sus mínimos esenciales y (en un gesto heroico) usando tarjetas de fichado en lugar de UML. Es como si fuera RUP imitando los principios de XP; algunos piensan que dX es XP de cabo a rabo, sólo que con algunos nombres cambiados [ASR+02]. RUP se ha combinado con Evo, Scrum, MSF y cualquier metodología imaginable. Dado que RUP es suficientemente conocido y su estructura es más amplia y compleja que el de cualquier otro método ágil, su tratamiento en este texto concluye en este punto.

Dynamic Systems Development Method (DSDM)

Originado en los trabajos de Jennifer Stapleton, directora del DSDM Consortium, DSDM se ha convertido en el framework de desarrollo rápido de aplicaciones (RAD) más popular de Gran Bretaña [Sta97] y se ha llegado a promover como el estándar de facto para desarrollo de soluciones de negocios sujetas a márgenes de tiempo estrechos. Se calcula que uno de cada cinco desarrolladores en Gran Bretaña utiliza DSDM y que más de 500 empresas mayores lo han adoptado.

Además de un método, DSDM proporciona un framework completo de controles para RAD y lineamientos para su uso. DSDM puede complementar metodologías de XP, RUP o Microsoft Solutions Framework, o combinaciones de todas ellas. DSDM es relativamente antiguo en el campo de los MAs y constituye una metodología madura, que ya va por su cuarta versión. Se dice que ahora las iniciales DSDM significan Dynamic Solutions Delivery Method. Ya no se habla de sistemas sino de soluciones, y en lugar de priorizar el desarrollo se prefiere enfatizar la entrega. El libro más reciente que sintetiza las prácticas se llama *DSDM: Business Focused Development* [DS03].

La idea dominante detrás de DSDM es explícitamente inversa a la que se encuentra en otras partes, y al principio resulta contraria a la intuición; en lugar de ajustar tiempo y recursos para lograr cada funcionalidad, en esta metodología tiempo y recursos se mantienen como constantes y se ajusta la funcionalidad de acuerdo con ello. Esto se expresa a través de reglas que se conocen como “reglas MoSCoW” por las iniciales de su estipulación en inglés. Las reglas se refieren a rasgos del requerimiento:

1. **Must have: Debe tener**. Son los requerimientos fundamentales del sistema. De éstos, el subconjunto mínimo ha de ser satisfecho por completo.
2. **Should have: Debería tener**. Son requerimientos importantes para los que habrá una resolución en el corto plazo.
3. **Could have: Podría tener**. Podrían quedar fuera del sistema si no hay más remedio.
4. **Want to have but won’t have this time around: Se desea que tenga, pero no lo tendrá esta vuelta**. Son requerimientos valorados, pero pueden esperar.

DSDM consiste en cinco fases:

1. Estudio de viabilidad.
2. Estudio del negocio.
3. Iteración del modelo funcional.
4. Iteración de diseño y versión.
5. Implementación.

Las últimas tres fases son iterativas e incrementales. De acuerdo con la iniciativa de mantener el tiempo constante, las iteraciones de DSDM son cajas de tiempo. La iteración acaba cuando el tiempo se consume. Se supone que al cabo de la iteración los resultados están garantizados. Una caja de tiempo puede durar de unos pocos días a unas pocas semanas.

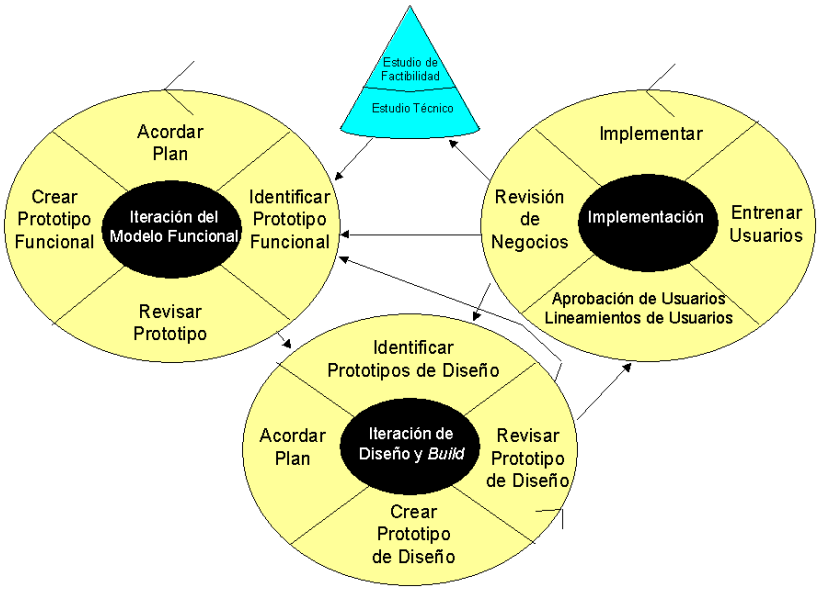
A diferencia de otros AMs, DSDM ha desarrollado sistemáticamente el problema de su propia implantación en una empresa. El proceso de Examen de Salud (*Health Check*) de DSDM se divide en dos partes que se interrogan, sucesivamente, sobre la capacidad de

una organización para adoptar el método y sobre la forma en que éste responde a las necesidades una vez que el proyecto está encaminado. Un Examen de Salud puede insumir entre tres días y un mes de trabajo de consultoría.

1. **Estudio de factibilidad**. Se evalúa el uso de DSDM o de otra metodología conforme al tipo de proyecto, variables organizacionales y de personal. Si se opta por DSDM, se analizan las posibilidades técnicas y los riesgos. Se preparan como productos un reporte de viabilidad y un plan sumario para el desarrollo. Si la tecnología no se conoce bien, se hace un pequeño prototipo para ver qué pasa. No se espera que el estudio completo insuma más de unas pocas semanas. Es mucho para un método ágil, pero menos de lo que demandan algunos métodos clásicos.
2. **Estudio del negocio**. Se analizan las características del negocio y la tecnología. La estrategia recomendada consiste en el desarrollo de talleres, donde se espera que los expertos del cliente consideren las facetas del sistema y acuerden sus prioridades de desarrollo. Se describen los procesos de negocio y las clases de usuario en una Definición del Área de Negocios. Se espera así reconocer e involucrar a gente clave de la organización en una etapa temprana. La Definición utiliza descripciones de alto nivel, como diagramas de entidad-relación o modelos de objetos de negocios. Otros productos son la Definición de Arquitectura del Sistema y el Plan de Bosquejo de Prototipado. La definición arquitectónica es un primer bosquejo y se admite que cambie en el curso del proyecto DSDM. El plan debe establecer la estrategia de prototipado de las siguientes etapas y un plan para la gestión de configuración.
3. **Iteración del modelo funcional**. En cada iteración se planea el contenido y la estrategia, se realiza la iteración y se analizan los resultados pensando en las siguientes. Se lleva a cabo tanto el análisis como el código; se construyen los prototipos y en base a la experiencia se mejoran los modelos de análisis. Los prototipos no han de ser descartados por completo, sino gradualmente mejorados hacia la calidad que debe tener el producto final. Se produce como resultado un Modelo Funcional, conteniendo el código del prototipo y los modelos de análisis. También se realizan pruebas constantemente. Hay otros cuatro productos emergentes: (1) Funciones Priorizadas es una lista de funciones entregadas al fin de cada iteración; (2) los Documentos de Revisión del Prototipado Funcional reúnen los comentarios de los usuarios sobre el incremento actual para ser considerados en iteraciones posteriores; (3) los Requerimientos Funcionales son listas que se construyen para ser tratadas en fases siguientes; (4) el Análisis de Riesgo de Desarrollo Ulterior es un documento importante en la fase de iteración del modelo, porque desde la fase siguiente en adelante los problemas que se encuentren serán más difíciles de tratar.
4. **Iteración de diseño y construcción**. Aquí es donde se construye la mayor parte del sistema. El producto es un Sistema Probado que cumplimenta por lo menos el conjunto mínimo de requerimientos acordados conforme a las reglas MoSCoW. El diseño y la construcción son iterativos y el diseño y los prototipos funcionales son revisados por usuarios. El desarrollo ulterior se atiene a sus comentarios.
5. **Despliegue**. El sistema se transfiere del ambiente de desarrollo al de producción. Se entrena a los usuarios, que ponen las manos en el sistema. Eventualmente la fase puede llegar a iterarse. Otros productos son el Manual de Usuario y el Reporte de Revisión del Sistema. A partir de aquí hay cuatro cursos de acción posibles: (1) Si el

sistema satisface todos los requerimientos, el desarrollo ha terminado. (2) Si quedan muchos requerimientos sin resolver, se puede correr el proceso nuevamente desde el comienzo. (3) Si se ha dejado de lado alguna prestación no crítica, el proceso se puede correr desde la iteración funcional del modelo en adelante. (4) si algunas cuestiones técnicas no pudieron resolverse por falta de tiempo se puede iterar desde la fase de diseño y construcción.

La configuración del ciclo de vida de DSDM se representa con un diagrama característico (del cual hay una evocación en el logotipo del consorcio) que vale la pena reproducir:



Proceso de desarrollo DSDM, basado en [http://www.dsdm.org]

DSDM define quince roles, algo más que el promedio de los MAs. Los más importantes son:

1. **Programadores y Programadores Senior**. Son los únicos roles de desarrollo. El título de Senior indica también nivel de liderazgo dentro del equipo. Equivale a Nivel 3 de Cockburn. Ambos títulos cubren todos los roles de desarrollo, incluyendo analistas, diseñadores, programadores y verificadores.
2. **Coordinador técnico**. Define la arquitectura del sistema y es responsable por la calidad técnica del proyecto, el control técnico y la configuración del sistema.
3. **Usuario embajador**. Proporciona al proyecto conocimiento de la comunidad de usuarios y disemina información sobre el progreso del sistema hacia otros usuarios. Se define adicionalmente un rol de **Usuario Asesor** (*Advisor*) que representa otros puntos de vista importantes; puede ser alguien del personal de IT o un auditor funcional.
4. **Visionario**. Es un usuario participante que tiene la percepción más exacta de los objetivos del sistema y el proyecto. Asegura que los requerimientos esenciales se cumplan y que el proyecto vaya en la dirección adecuada desde el punto de vista de aquéllos.

5. **Patrocinador Ejecutivo**. Es la persona de la organización que detenta autoridad y responsabilidad financiera, y es quien tiene la última palabra en las decisiones importantes.
6. **Facilitador**. Es responsable de administrar el progreso del taller y el motor de la preparación y la comunicación.
7. **Escriba**. Registra los requerimientos, acuerdos y decisiones alcanzadas en las reuniones, talleres y sesiones de prototipado.

En DSDM las prácticas se llaman Principios, y son nueve:

1. Es imperativo el compromiso activo del usuario.
2. Los equipos de DSDM deben tener el poder de tomar decisiones.
3. El foco radica en la frecuente entrega de productos.
4. El criterio esencial para la aceptación de los entregables es la adecuación a los propósitos de negocios.
5. Se requiere desarrollo iterativo e incremental.
6. Todos los cambios durante el desarrollo son reversibles.
7. La línea de base de los requerimientos es de alto nivel. Esto permite que los requerimientos de detalle se cambien según se necesite y que los esenciales se capten tempranamente.
8. La prueba está integrada a través de todo el ciclo de vida. La prueba también es incremental. Se recomienda particularmente la prueba de regresión, de acuerdo con el estilo evolutivo de desarrollo.
9. Es esencial una estrategia colaborativa y cooperativa entre todos los participantes. Las responsabilidades son compartidas y la colaboración entre usuario y desarrolladores no debe tener fisuras.

Desde mediados de la década de 1990 hay abundantes estudios de casos, sobre todo en Gran Bretaña, y la adecuación de DSDM para desarrollo rápido está suficientemente probada [ASR+02]. El equipo mínimo de DSDM es de dos personas y puede llegar a seis, pero puede haber varios equipos en un proyecto. El mínimo de dos personas involucra que un equipo consiste de un programador y un usuario. El máximo de seis es el valor que se encuentra en la práctica. DSDM se ha aplicado a proyectos grandes y pequeños. La precondición para su uso en sistemas grandes es su partición en componentes que pueden ser desarrollados por equipos normales.

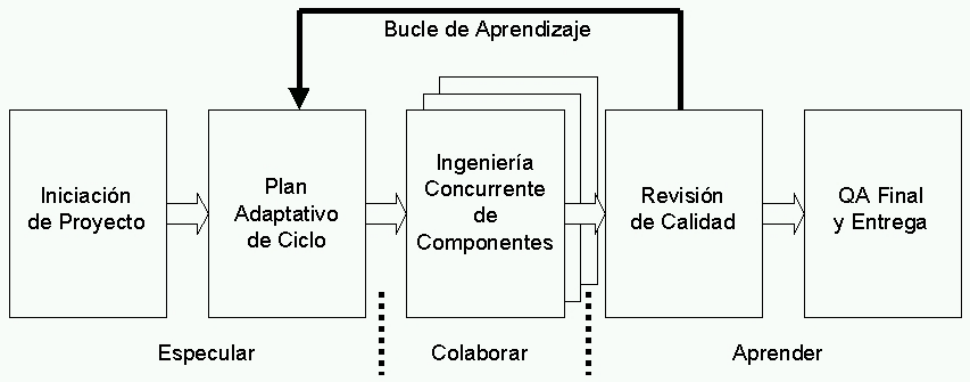
Se ha elaborado en particular la combinación de DSDM con XP y se ha llamado a esta mixtura EnterpriseXP, término acuñado por Mike Griffiths de Quadrus Developments (http://www.enterprisexp.org). Se atribuye a Kent Beck haber afirmado que la comunidad de DSDM ha construido una imagen corporativa mejor que la del mundo XP y que sería conveniente aprender de esa experiencia. También hay documentos conjuntos de DSDM y Rational, con participación de Jennifer Stapleton, que demuestran la compatibilidad del modelo DSDM con RUP, a despecho de sus fuertes diferencias terminológicas.

En el sitio del DSDM Consortium hay documentos específicos sobre la convivencia de la metodología con Microsoft Solutions Framework; se tratará detalladamente el particular en el capítulo sobre MSF y los MAs (pág. 55). También hay casos de éxito

(particularmente el de Fujitsu European Repair Centre) en que se emplearon Visual Basic como lenguaje, SQL Server como base de datos y Windows como plataforma de desarrollo e implementación (http://www.dsdm.org). Los métodos y manuales completos de DSDM sólo están disponibles para miembros del consorcio; la membrecía involucra el pago de un canon que oscila entre 550 € y 4400 € anuales.

Adaptive Software Development

James Highsmith III, consultor de Cutter Consortium, desarrolló ASD hacia el año 2000 con la intención primaria de ofrecer una alternativa a la idea, propia de CMM Nivel 5, de que la optimización es la única solución para problemas de complejidad creciente. Este método ágil pretende abrir una tercera vía entre el "desarrollo monumental de software" y el "desarrollo accidental", o entre la burocracia y la adhocracia. Deberíamos buscar más bien, afirma Highsmith, "el rigor estrictamente necesario"; para ello hay que situarse en coordenadas apenas un poco fuera del caos y ejercer menos control que el que se cree necesario [Hig00a].



Fases del ciclo de vida de ASD, basado en Highsmith [Hig00a: 84]

La estrategia entera se basa en el concepto de emergencia, una propiedad de los sistemas adaptativos complejos que describe la forma en que la interacción de las partes genera una propiedad que no puede ser explicada en función de los componentes individuales. El pensador de quien Highsmith toma estas ideas es John Holland, el creador del algoritmo genético y probablemente el investigador actual más importante en materia de procesos emergentes [Hol95]. Holland se pregunta, entre otras cosas, cómo hace un macro-sistema extremadamente complejo, no controlado de arriba hacia abajo en todas las variables intervinientes (como por ejemplo la ciudad de Nueva York o la Web) para mantenerse funcionando en un aparente equilibrio sin colapsar.

La respuesta, que tiene que ver con la auto-organización, la adaptación al cambio y el orden que emerge de la interacción entre las partes, remite a examinar analogías con los sistemas adaptativos complejos por excelencia, esto es: los organismos vivientes (o sus análogos digitales, como las redes neuronales auto-organizativas de Teuvo Kohonen y los autómatas celulares desde Von Neumann a Stephen Wolfram). Para Highsmith, los proyectos de software son sistemas adaptativos complejos y la optimización no hace más que sofocar la emergencia necesaria para afrontar el cambio. Llevando más allá la analogía, Highsmith interpreta la organización empresarial que emprende un desarrollo como si fuera un ambiente, sus miembros como agentes y el producto como el resultado

emergente de relaciones de competencia y cooperación. En los sistemas complejos no es aplicable el análisis, porque no puede *deducirse* el comportamiento del todo a partir de la conducta de las partes, ni sumarse las propiedades individuales para determinar las características del conjunto: el oxígeno es combustible, el hidrógeno también, pero cuando se combinan se obtiene agua, la cual no tiene esa propiedad.

Highsmith indaga además la economía del retorno creciente según Brian Arthur. Esta economía se caracteriza por la alta velocidad y la elevada tasa de cambio. Velocidad y cambio introducen una complejidad que no puede ser manipulada por las estrategias convencionales. En condiciones de complejidad el mercado es formalmente impredecible y el proceso de desarrollo deviene imposible de planificar bajo el supuesto de que después se tendrá control del proceso. El razonamiento simple de causa y efecto no puede dar cuenta de la situación y mucho menos controlarla. En este contexto, los sistemas adaptativos complejos suministran los conceptos requeridos: agentes autónomos que compiten y cooperan, los ambientes mutables y la emergencia. Tomando como punto de partida estas ideas, Highsmith elabora un modelo de gestión que llama Modelo de Liderazgo-Colaboración Adaptativo (L-C), el cual tiene puntos en común con el modelado basado en agentes autónomos; en este modelo se considera que la adaptabilidad no puede ser comandada, sino que debe ser nutrida: nutriendo de conducta adaptativa a cada agente, el sistema global deviene adaptativo.

ASD presupone que las necesidades del cliente son siempre cambiantes. La iniciación de un proyecto involucra definir una misión para él, determinar las características y las fechas y descomponer el proyecto en una serie de pasos individuales, cada uno de los cuales puede abarcar entre cuatro y ocho semanas. Los pasos iniciales deben verificar el alcance del proyecto; los tardíos tienen que ver con el diseño de una arquitectura, la construcción del código, la ejecución de las pruebas finales y el despliegue.

Aspectos claves de ASD son:

1. Un conjunto no estándar de “artefactos de misión” (documentos para tí y para mí), incluyendo una visión del proyecto, una hoja de datos, un perfil de misión del producto y un esquema de su especificación
2. Un ciclo de vida, inherentemente iterativo.
3. Cajas de tiempo, con ciclos cortos de entrega orientados por riesgo.

Un ciclo de vida es una iteración; este ciclo se basa en componentes y no en tareas, es limitado en el tiempo, orientado por riesgos y tolerante al cambio. Que se base en componentes implica concentrarse en el desarrollo de software que trabaje, construyendo el sistema pieza por pieza. En este paradigma, el cambio es bienvenido y necesario, pues se concibe como la oportunidad de aprender y ganar así una ventaja competitiva; de ningún modo es algo que pueda ir en detrimento del proceso y sus resultados.

Highsmith piensa que los procesos rigurosos (repetibles, visibles, medibles) son encomiables porque proporcionan estabilidad en un entorno complejo, pero muchos procesos en el desarrollo (por ejemplo, el diseño del proyecto) deberían ser flexibles. La clave para mantener el control radica en los “estados de trabajo” (la colección de los productos de trabajo) y no en el flujo de trabajo (*workflow*). Demasiado rigor, por otra parte, acarrea *rigor mortis,* el cual impide cambiar el producto cuando se introducen las inevitables modificaciones. En la moderna teoría económica del retorno creciente, ser

capaz de adaptarse es significativamente más importante que ser capaz de optimizar [Hig00a].

La idea subyacente a ASD (y de ahí su particularidad) radica en que no proporciona un método para el desarrollo de software sino que más bien suministra la forma de implementar una cultura adaptativa en la empresa, con capacidad para reconocer que la incertidumbre y el cambio son el estado natural. El problema inicial es que la empresa *no sabe que no sabe,* y por tal razón debe aprender. Los cuatro objetivos de este proceso de aprendizaje son entonces:

1. Prestar soporte a una cultura adaptativa o un conjunto mental para que se espere cambio e incertidumbre y no se tenga una falsa expectativa de orden.
2. Introducir marcos de referencia para orientar el proceso iterativo de gestión del cambio.
3. Establecer la colaboración y la interacción de la gente en tres niveles: interpersonal, cultural y estructural.
4. Agregar rigor y disciplina a una estrategia RAD, haciéndola escalable a la complejidad de los emprendimientos de la vida real.

ASD se concentra más en los componentes que en las tareas; en la práctica, esto se traduce en ocuparse más de la calidad que en los procesos usados para producir un resultado. En los ciclos adaptativos de la fase de Colaboración, el planeamiento es parte del proceso iterativo, y las definiciones de los componentes se refinan continuamente. La base para los ciclos posteriores (el bucle de Aprendizaje) se obtiene a través de repetidas revisiones de calidad con presencia del cliente como experto, constituyendo un grupo de foco de cliente. Esto ocurre solamente al final de las fases, por lo que la presencia del cliente se suplementa con sesiones de desarrollo conjunto de aplicaciones (JAD). Hemos visto que una sesión JAD, común en el antiguo RAD, es un taller en el que programadores y representantes del cliente se encuentran para discutir rasgos del producto en términos no técnicos, sino de negocios.

El modelo de Highsmith es, naturalmente, complementario a cualquier concepción dinámica del método; no podría ser otra cosa que adaptable, después de todo, y por ello admite y promueve integración con otros modelos y marcos. Un estudio de Dirk Riehle [Rie00] compara ASD con XP, encontrando similitudes y diferencias de principio que pueden conciliarse con relativa facilidad, al lado de otras variables que son incompatibles. La actitud de ambos métodos frente a la redundancia de código, por ejemplo, es distinta; en XP se debe hacer todo "una vez y sólo una vez", mientras que en ASD la redundancia puede ser un subproducto táctico inevitable en un ambiente competitivo y debe aceptarse en tanto el producto sea "suficientemente bueno". En materia de técnicas, ASD las considera importantes pero no más que eso; para XP, en cambio, los patrones y la refactorización son balas de plata [Bro87].

Hay ausencia de estudios de casos del método adaptativo, aunque las referencias literarias a sus principios son abundantes. Como ASD no constituye un método de ingeniería de ciclo de vida sino una visión cultural o una epistemología, no califica como framework suficiente para articular un proyecto. Más visible es la participación de Highsmith en el respetado Cutter Consortium, del cual es director del Agile Project Management Advisory Service. Entre las empresas que han requerido consultoría adaptativa se cuentan

AS Bank de Nueva Zelanda, CNET, GlaxoSmithKline, Landmark, Nextel, Nike, Phoenix International Health, Thoughworks y Microsoft. Los consultores que se vinculan a los lineamientos ágiles de Cutter Consortium, por su parte, son innumerables. Jim Highsmith es citado en un epígrafe referido a los métodos ágiles en la documentación de Microsoft Solutions Framework [MS03].

Agile Modeling

Agile Modeling (AM) fue propuesto por Scott Ambler [Amb02a] no tanto como un método ágil cerrado en sí mismo, sino como complemento de otras metodologías, sean éstas ágiles o convencionales. Ambler recomienda su uso con XP, Microsoft Solutions Framework, RUP o EUP. En el caso de XP y MSF los practicantes podrían definir mejor los procesos de modelado que en ellos faltan, y en el caso de RUP y EUP el modelado ágil permite hacer más ligeros los procesos que ya usan. AM es una estrategia de modelado (de clases, de datos, de procesos) pensada para contrarrestar la sospecha de que los métodos ágiles no modelan y no documentan. Se lo podría definir como un proceso de software basado en prácticas cuyo objetivo es orientar el modelado de una manera efectiva y ágil.

Los principales objetivos de AM son:

1. Definir y mostrar de qué manera se debe poner en práctica una colección de valores, principios y prácticas que conducen al modelado de peso ligero.
2. Enfrentar el problema de la aplicación de técnicas de modelado en procesos de desarrollo ágiles.
3. Enfrentar el problema de la aplicación de las técnicas de modelado independientemente del proceso de software que se utilice.

Los valores de AM incluyen a los de XP: comunicación, simplicidad, *feedback* y coraje, añadiendo humildad. Una de las mejores caracterizaciones de los principios subyacentes a AM está en la definición de sus alcances:

1. AM es una actitud, no un proceso prescriptivo. Comprende una colección de valores a los que los modeladores ágiles adhieren, principios en los que creen y prácticas que aplican. Describe un estilo de modelado; no es un recetario de cocina.
2. AM es suplemento de otros métodos. El primer foco es el modelado y el segundo la documentación.
3. AM es una tarea de conjunto de los participantes. No hay “yo” en AM.
4. La prioridad es la efectividad. AM ayuda a crear un modelo o proceso cuando se tiene un propósito claro y se comprenden las necesidades de la audiencia; contribuye a aplicar los artefactos correctos para afrontar la situación inmediata y a crear los modelos más simples que sea posible.
5. AM es algo que funciona en la práctica, no una teoría académica. Las prácticas han sido discutidas desde 2001 en comunidad (http://www.agilemodeling.com/feedback.htm).
6. AM no es una bala de plata.
7. AM es para el programador promedio, pero no reemplaza a la gente competente.

8. AM no es un ataque a la documentación. La documentación debe ser mínima y relevante.
9. AM no es un ataque a las herramientas CASE.
10. AM no es para cualquiera.

Los principios de AM especificados por Ambler [Amb02a] incluyen:

1. **Presuponer simplicidad**. La solución más simple es la mejor.
2. **El contenido es más importante que la representación**. Pueden ser notas, pizarras o documentos formales. Lo que importa no es el soporte físico o la técnica de representación, sino el contenido.
3. **Abrazar el cambio**. Aceptar que los requerimientos cambian.
4. **Habilitar el esfuerzo siguiente**. Garantizar que el sistema es suficientemente robusto para admitir mejoras ulteriores; debe ser un objetivo, pero no el primordial.
5. **Todo el mundo puede aprender de algún otro**. Reconocer que nunca se domina realmente algo.
6. **Cambio incremental**. No esperar hacerlo bien la primera vez.
7. **Conocer tus modelos**. Saber cuáles son sus fuerzas y sus debilidades.
8. **Adaptación local**. Producir sólo el modelo que resulte suficiente para el propósito.
9. **Maximizar la inversión del cliente**.
10. **Modelar con un propósito**. Si no se puede identificar para qué se está haciendo algo ¿para qué molestarse?
11. **Modelos múltiples**. Múltiples paradigmas en convivencia, según se requiera.
12. **Comunicación abierta y honesta**.
13. **Trabajo de calidad**.
14. **Realimentación rápida**. No esperar que sea demasiado tarde.
15. **El software es el objetivo primario**. Debe ser de alta calidad y coincidir con lo que el usuario espera.
16. **Viajar ligero de equipaje**. No crear más modelos de los necesarios.
17. **Trabajar con los instintos de la gente**.

Lo más concreto de AM es su rico conjunto de prácticas [Amb02b], cada una de las cuales se asocia a lineamientos decididamente narrativos, articulados con minuciosidad, pero muy lejos de los rigores del aparato cuantitativo de Evo:

1. Colaboración activa de los participantes.
2. Aplicación de estándares de modelado.
3. Aplicación adecuada de patrones de modelado.
4. Aplicación de los artefactos correctos.
5. Propiedad colectiva de todos los elementos.
6. Considerar la verificabilidad.
7. Crear diversos modelos en paralelo.
8. Crear contenido simple.
9. Diseñar modelos de manera simple.

10. Descartar los modelos temporarios.
11. Exhibir públicamente los modelos.
12. Formalizar modelos de contrato.
13. Iterar sobre otro artefacto.
14. Modelo en incrementos pequeños.
15. Modelar para comunicar.
16. Modelar para comprender.
17. Modelar con otros.
18. Poner a prueba con código.
19. Reutilizar los recursos existentes.
20. Actualizar sólo cuando duele.
21. Utilizar las herramientas más simples (CASE, o mejor pizarras, tarjetas, *post-its*)**.**

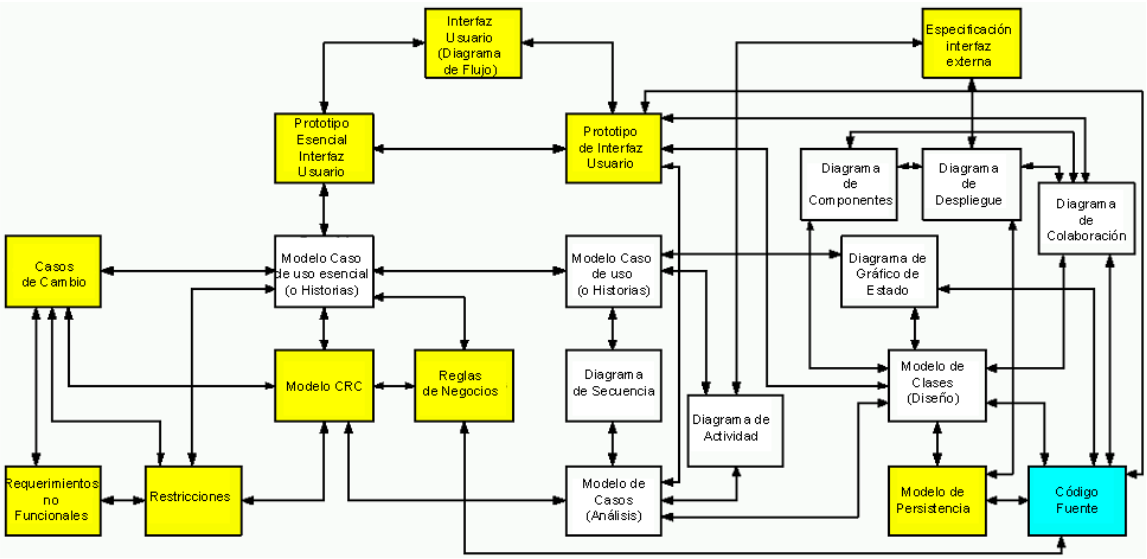


Diagrama de artefactos de Agile Modeling / EUP, basado en [Amb02c]

Como AM se debe usar como complemento de otras metodologías, nada se especifica sobre métodos de desarrollo, tamaño del equipo, roles, duración de iteraciones, trabajo distribuido y criticalidad, todo lo cual dependerá del método que se utilice. Los principios y prácticas que hemos citado puede dar impresión de puro sentido común, sólo que en variante caórdica; en realidad los aspectos atinentes a uso de herramientas de modelado y documentación, así como la articulación con metodologías específicas, las técnicas de bases de datos y el uso de artefactos están especificados cuidadosamente, como se puede constatar en el sitio de AM y en los textos más extensos de Ambler [Amb02a] [Amb02b] [Amb03] [Amb04].

Los diagramas de UML y los artefactos del Proceso Unificado, por ejemplo, han sido explorados en extremo detalle describiendo cómo debería ser su tratamiento en un proceso ágil EUP, regido por principios caórdicos. EUP es simplemente UP+AM. Se han documentado algunos estudios de casos de AM en proyectos de mediano calado [Amb02b]. Aún cuando uno no pretenda integrarse al bando de los MAs, puede que valga

la pena echar una mirada a ese material y considerar la razonabilidad de algunas de sus sugerencias.

Lean Development (LD) y Lean Software Development (LSD)

Lean Development (LD) es el método menos divulgado entre los reconocidamente importantes. La palabra "*lean*" significa magro, enjuto; en su sentido técnico apareció por primera vez en 1990 en el libro de James Womack *La Máquina que Cambió al Mundo* [WTR91]. LD, iniciado por Bob Charette [Cha01], se inspira en el éxito del proceso industrial Lean Manufacturing, bien conocido en la producción automotriz y en manufactura desde la década de 1980. Este proceso tiene como precepto la eliminación de residuos a través de la mejora constante, haciendo que el producto fluya a instancias del cliente para hacerlo lo más perfecto posible.

Los procesos a la manera americana corrían con sus máquinas al 100% de capacidad y mantenían inventarios gigantescos de productos y suministros; Toyota , en contra de la intuición, resultaba más eficiente manteniendo suministros en planta para un solo día, y produciendo sólo lo necesario para cubrir las órdenes pendientes. Esto es lo que se llama *Just in Time Production*. Con JIT se evita además que el inventario degrade o se torne obsoleto, o empiece a actuar como un freno para el cambio. Toyota implementaba además las técnicas innovadoras del Total Quality Management de Edward Deming, que sólo algunos matemáticos y empresarios de avanzada conocían en Estados Unidos. Hasta el día de hoy la foto de Deming en Toyota es más grande y esplendorosa que la del fundador, Toyoda Sakichi.

Otros aspectos esenciales de Lean Manufacturing son la relación participativa con el empleado y el trato que le brinda la compañía, así como una especificación de principios, disciplinas y métodos iterativos, adaptativos, auto-organizativos e interdependientes en un patrón de ciclos de corta duración que tiene algo más que un aire de familia con el patrón de procesos de los MAs (http://www.strategosinc.com/principles.htm). Existe unanimidad de intereses, consistencia de discurso y complementariedad entre las comunidades Lean de manufactura y desarrollo de software.

Mientras que otros MAs se concentran en el proceso de desarrollo, Charette sostenía que para ser verdaderamente ágil se debía conocer además el negocio de punta a punta. LD se inspira en doce valores centrados en estrategias de gestión [Hig02b]:

1. Satisfacer al cliente es la máxima prioridad.
2. Proporcionar siempre el mejor valor por la inversión.
3. El éxito depende de la activa participación del cliente.
4. Cada proyecto LD es un esfuerzo de equipo.
5. Todo se puede cambiar.
6. Soluciones de dominio, no puntos.
7. Completar, no construir.
8. Una solución al 80% hoy, en vez de una al 100% mañana.
9. El minimalismo es esencial.
10. La necesidad determina la tecnología.
11. El crecimiento del producto es el incremento de sus prestaciones, no de su tamaño.

12. Nunca empujes LD más allá de sus límites.

Dado que LD es más una filosofía de *management* que un proceso de desarrollo no hay mucho que decir del tamaño del equipo, la duración de las iteraciones, los roles o la naturaleza de sus etapas. Últimamente LD ha evolucionado como Lean Software Development (LSD); su figura de referencia es Mary Poppendieck [Pop01].

Uno de los sitios primordiales del modelo son las páginas consagradas a LSD que mantiene Darrell Norton [Nor04], donde se promueve el desarrollo del método aplicando el framework .NET de Microsoft. Norton ha reformulado los valores de Charette reduciéndolos a siete y suministrando más de veinte herramientas análogas a patrones organizacionales para su implementación en ingeniería de software. Los nuevos principios son:

1. Eliminar basura (las herramientas son *Seeing Waste, Value Stream Mapping*). Basura es todo lo que no agregue valor a un producto, desde la óptica del sistema de valores del cliente. Este principio equivale a la reducción del inventario en manufactura. El inventario del desarrollo de software es el conjunto de artefactos intermedios. Un estudio del Standish Group reveló que en un sistema típico, las prestaciones que se usan siempre suman el 7%, las que se usan a menudo el 13%, "algunas veces" el 16%, "raras veces" el 19% y "nunca" el 45%. Esto es un claro 80/20: el 80% del valor proviene del 20% de los rasgos. Concentrarse en el 20% útil es una aplicación del mismo principio que subyace a la idea de YAGNI.
2. Amplificar el conocimiento (*Feedback, Iterations, Synchronization, Set-based Development*). El desarrollo se considera un ejercicio de descubrimiento.
3. Decidir tan tarde como sea posible (*Options Thinking, The Last Responsible Moment, Making Decisions*). Las prácticas de desarrollo que proporcionan toma de decisiones tardías son efectivas en todos los dominios que involucran incertidumbre porque brindan una estrategia basada en opciones fundadas en la realidad, no en especulaciones. En un mercado que cambia, la decisión tardía, que mantiene las opciones abiertas, es más eficiente que un compromiso prematuro. En términos metodológicos, este principio se traduce también en la renuencia a planificarlo todo antes de comenzar. En un entorno cambiante, los requerimientos detallados corren el riesgo de estar equivocados o ser anacrónicos.
4. Entregar tan rápido como sea posible (*Pull Systems, Queueing Theory, Cost of Delay*). Se deben favorecer ciclos cortos de diseño → implementación → *feedback* → mejora. El cliente recibe lo que necesita hoy, no lo que necesitaba ayer.
5. Otorgar poder al equipo (*Self Determination, Motivation, Leadership, Expertise*). Los desarrolladores que mejor conocen los elementos de juicio son los que pueden tomar las decisiones más adecuadas.
6. Integridad incorporada (*Perceived Integrity, Conceptual Integrity, Refactoring, Testing*). La integridad conceptual significa que los conceptos del sistema trabajan como una totalidad armónica de arquitectura coherente. La investigación ha demostrado que la integridad viene con el liderazgo, la experiencia relevante, la comunicación efectiva y la disciplina saludable. Los procesos, los procedimientos y las medidas no son substitutos adecuados.

7. Ver la totalidad (*Measurements, Contracts*). Uno de los problemas más intratables del desarrollo de software convencional es que los expertos en áreas específicas (por ejemplo, bases de datos o GUIs) maximizan la corrección de la parte que les interesa, sin percibir la totalidad.

Otra preceptiva algo más amplia es la de Mary Poppendieck [Pop01], cuidadosamente decantadas del Lean Manufacturing y de Total Quality Management (TQM), que sólo coincide con la de Norton en algunos puntos:

1. Eliminar basura – Entre la basura se cuentan diagramas y modelos que no agregan valor al producto.
2. Minimizar inventario – Igualmente, suprimir artefactos tales como documentos de requerimiento y diseño.
3. Maximizar el flujo – Utilizar desarrollo iterativo.
4. Solicitar demanda – Soportar requerimientos flexibles.
5. Otorgar poder a los trabajadores.
6. Satisfacer los requerimientos del cliente – Trabajar junto a él, permitiéndole cambiar de ideas.
7. Hacerlo bien la primera vez – Verificar temprano y refactorizar cuando sea preciso.
8. Abolir la optimización local – Alcance de gestión flexible.
9. Asociarse con quienes suministran – Evitar relaciones de adversidad.
10. Crear una cultura de mejora continua.

Las herramientas, junto con el prolijo desarrollo de la metodología, se detallan en un texto de Mary y Tom Poppendieck [PP03], consistentemente encomiado por sus lectores.

Igual que Agile Modeling, que cubría sobre todo aspectos de modelado y documentación, LD y LSD han sido pensados como complemento de otros métodos, y no como una metodología excluyente a implementar en la empresa. LD prefiere concentrarse en las premisas y modelos derivados de Lean Production, que hoy constituyen lo que se conoce como el canon de la Escuela de Negocios de Harvard. Para las técnicas concretas de programación, LD promueve el uso de otros MAs que sean consistentes con su visión, como XP o sobre todo Scrum.

Aunque la formulación del método es relativamente reciente, la familiaridad de muchas empresas con los principios de Lean Production & Lean Manufacturing ha facilitado la penetración en el mercado de su análogo en ingeniería de software. LD se encuentra hoy en América del Norte en una situación similar a la de DSDM en Gran Bretaña, llegando al 7% entre los MAs a nivel mundial (algo menos que Crystal pero el doble que Scrum). Existen abundantes casos de éxito documentados empleando LD y LSD, la mayoría en Canadá. Algunos de ellos son los de Canadian Pacific Railway, Autodesk y PowerEx Corporation. Se ha aplicado prácticamente a todo el espectro de la industria.

Microsoft Solutions Framework y los Métodos Agiles

Los métodos ágiles de desarrollo tienen una larga tradición en las prácticas de Microsoft. Uno de los textos clásicos de RAD, *Rapid Development* de Steve McConnell [McC96], se remonta a una tradición más temprana que los MAs contemporáneos; al igual que

éstos, reconoce como “mejores prácticas” al modelo de ciclo de vida evolutivo, a los encuentros y talleres de equipo, las revisiones frecuentes, el diseño para el cambio, la entrega evolutiva, la reutilización, el prototipado evolutivo, la comunicación intensa, el desarrollo iterativo e incremental.

McConnell cita con generosidad el pensamiento de algunos de los precursores de los MAs: Barry Boehm, Frederick Brooks, Tom DeMarco, Harlan Mills. Cada vez que se utiliza la expresión “evolutivo” en RAD, se lo hace en el sentido que le imprimiera Tom Gilb, quien viene desarrollando el método Evo desde hace más de cuarenta años. El mismo McConnell, autor de *Code Complete* [McC93], es acreditado como una influencia importante en la literatura ágil de la actualidad [Lar04], aunque sus métodos carezcan de personalidad al lado de la estridente idiosincracia de algunos MAs. Algunas prácticas de RAD, sin embargo, divergen sensiblemente de lo que hoy se consideraría correcto en la comunidad ágil, como la recomendación de establecer metas fijas de antemano, contratar a terceros para realizar parte del código (*outsourcing*), trabajar en oficinas privadas u ofrecerse a permanecer horas extraordinarias.

Microsoft Solutions Framework es “un conjunto de principios, modelos, disciplinas, conceptos, lineamientos y prácticas probadas” elaborado por Microsoft. Como hemos entrevisto desde el epígrafe inicial, MSF se encuentra en estrecha comunión de principios con las metodologías ágiles, algunas de cuyas intuiciones y prácticas han sido anticipadas en las primeras versiones de su canon, hacia 1994. La documentación de MSF no deja sombra de dudas sobre esta concordancia; merece ser citada con amplitud y continuidad para apreciar el flujo de las palabras textuales y la intensidad de las referencias. Tras una cita del líder del movimiento ágil y ASD, Jim Highsmith, dice el documento principal de MSF 3.0:

> Las estrategias tradicionales de gestión de proyectos y los modelos de proceso en “cascada” presuponen un nivel de predictibilidad que no es tan común en proyectos de tecnología como puede serlo en otras industrias. A menudo, ni los productos ni los recursos para generarlos son bien entendidos y la exploración deviene parte del proyecto. Cuanto más busca una organización maximizar el impacto de negocios de la inversión tecnológica, más se aventura en nuevos territorios. Esta nueva base es inherentemente incierta y sujeta a cambios a medida que la exploración y la experimentación resultan en nuevas necesidades y métodos. Pretender o demandar certidumbre ante esta incertidumbre sería, en el mejor de los casos, irreal, y en el peor, disfuncional.
>
> MSF reconoce la naturaleza *caórdica* (una combinación de caos y orden, según lo acuñara Dee Hock, fundador y anterior CEO de Visa International) de los proyectos de tecnología. Toma como punto de partida el supuesto de que debe esperarse cambio continuo y de que es imposible aislar un proyecto de solución de estos cambios. Además de los cambios procedentes del exterior, MSF aconseja a los equipos que esperen cambios originados por los participantes e incluso por el propio equipo. Por ejemplo, admite que los requerimientos de un proyecto pueden ser difíciles de articular de antemano y que a menudo sufren modificaciones significativas a medida que los participantes van viendo sus posibilidades con mayor claridad.

> MSF ha diseñado tanto su Modelo de Equipo como su Modelo de Proceso para anticiparse al cambio y controlarlo. El Modelo de Equipo de MSF alienta la agilidad para hacer frente a nuevos cambios involucrando a todo el equipo en las decisiones fundamentales, asegurándose así que se exploran y revisan los elementos de juicio desde todas las perspectivas críticas. El Modelo de Proceso de MSF, a través de su estrategia iterativa en la construcción de productos del proyecto, suministra una imagen más clara del estado de los mismos en cada etapa sucesiva. El equipo puede identificar con mayor facilidad el impacto de cualquier cambio y lidiar con él efectivamente, minimizando los efectos colaterales negativos mientras optimiza los beneficios.
>
> En años recientes se han visto surgir estrategias específicas para desarrollar software que buscan maximizar el principio de agilidad y la preparación para el cambio. Compartiendo esta filosofía, MSF alienta la aplicación de esas estrategias donde resulte apropiado. […] Los métodos Ágiles, tales como Lean Development, eXtreme Programming y Adaptive Software Development, son estrategias de desarrollo de software que alientan prácticas que son adaptativas en lugar de predictivas, centradas en la gente y el equipo, iterativas, orientadas por prestaciones y entregas, de comunicación intensiva, y que requieren que el negocio se involucre en forma directa. Comparando esos atributos con los principios fundacionales de MSF, se encuentra que MSF y las metodologías ágiles están muy alineados tanto en los principios como en la práctica en ambientes que requieran alto grado de adaptabilidad [MS03].

Aunque MSF también se presta como marco para un desarrollo fuertemente documentado y a escala de misión crítica que requiere niveles más altos de estructura, como el que se articula en CMMI, PMBOK o ISO9000, sus disciplinas de ningún modo admiten la validez (o sustentan la vigencia) de modelos no iterativos o no incrementales.

La *Disciplina de Gestión de Riesgo* que se aplica a todos los miembros del *Modelo de Equipo* de MSF emplea el principio fundacional que literalmente se expresa como *permanecer ágil – esperar el cambio*. Esta ha sido una de las premisas de Jim Highsmith, quien tiene a Microsoft en un lugar prominente entre las empresas que han recurrido a su consultoría en materia de desarrollo adaptativo (ASD). También la utilización de la experiencia como oportunidad de aprendizaje está explícitamente en línea con las ideas de Highsmith. Los ocho principios fundacionales de MSF se asemejan a más de una de las declaraciones de principios que hemos visto expresadas en los manifiestos de los diversos MAs:

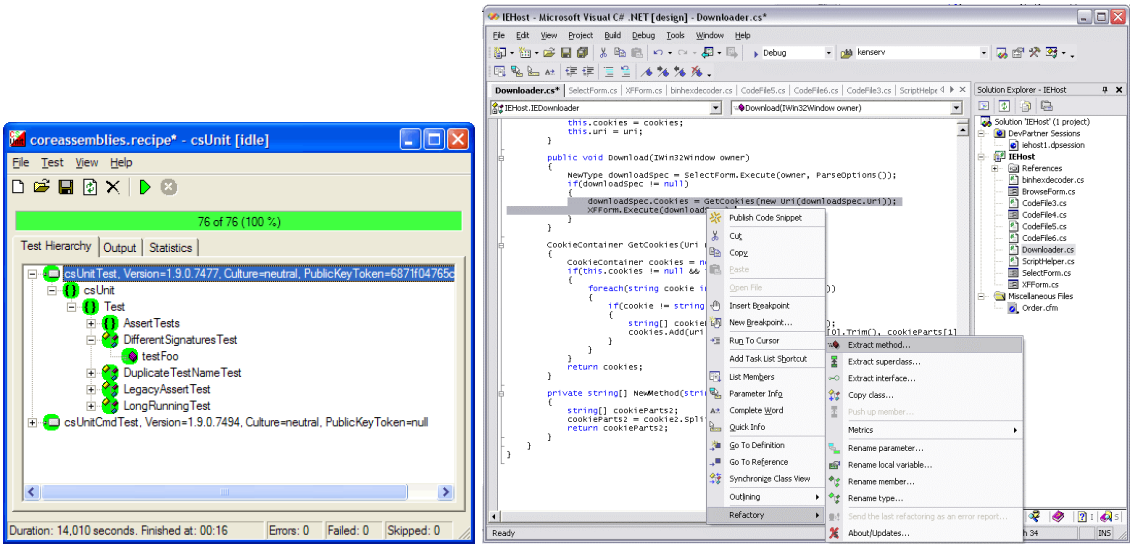
1. Alentar comunicaciones abiertas.
2. Trabajar hacia una visión compartida.
3. Otorgar poder a los miembros del equipo.
4. Establecer responsabilidad clara y compartida.
5. Concentrarse en la entrega de valor de negocios.
6. Permanecer ágil, esperar el cambio.
7. Invertir en calidad.
8. Aprender de todas las experiencias.

El primer principio, *Alentar la comunicación abierta*, utiliza como epígrafe para establecer el tono de sus ideas una referencia a la nueva edición de *The Mythical Man-Month* de Frederick Brooks [Bro95: 74-75], el ángel tutelar de todos los MAs y el disparador de las críticas al pensamiento de sentido común. Decía Brooks que este pensamiento (subyacente todavía en muchos usos de las *pivot-tables*) no titubearía en suscribir la idea de que nueve mujeres tardarían un mes en tener un hijo. Brooks ha sido uno de los primeros en estudiar las variables humanas en la producción de software, y su *MMM* es un libro de cabecera de todos los practicantes ágiles.

El segundo principio de MSF, *Trabajar hacia una visión compartida*, emplea para los mismos fines una cita de uno de los codificadores de la Programación Rápida, Steve McConnell [McC96]. *Permanecer ágil, esperar el cambio* comparte una vez más el pensamiento de Highsmith: "Los *managers* ágiles deben comprender que demandar certidumbre frente a la incertidumbre es disfuncional. Ellos deben establecer metas y restricciones que proporcionen límites dentro de los cuales puedan florecer la creatividad y la innovación".

En la *Disciplina de Management de Proyectos*, asimismo, se reconoce como uno de los cuerpos de conocimiento a los que se ha recurrido a Prince2 (***Pr**ojects **i**n **C**ontrolled **E**nvironments*), un estándar de industria para *management* y control de proyectos. El *Modelo de Procesos de MSF*, asimismo, recomienda una estructura de proceso fundamentalmente iterativa similar a la de todos los MAs que han elaborado modelos de ciclo de vida [MS02a], y a lo largo de todo el marco se insiste en otro postulado común, el de la participación activa del cliente en todas las etapas del proceso.

Recíprocamente, muchos de los codificadores de MAs han procurado poner en claro sus vínculos con MSF, entendiendo que éste es uno de los *players* importantes en la arena metodológica, y que en estas prácticas es común que disciplinas de alto nivel se articulen y orquesten con metodologías más precisas, e incluso con más de una de ellas. Por ejemplo, Agile Modeling es, en las palabras de Scott Ambler, un complemento adecuado a la disciplina de MSF, en la misma medida en que es también armónico con RUP o XP.



csUnit probando unidad en un proyecto XP en C# (izq) – Refactorización con Xtreme Simplicity (der)

Mas allá de MSF, Ron Jeffries, uno de los padres de XP, mantiene en su sitio de MAs múltiples referencias al Framework .NET y a C#. Mucha gente ignora que uno de los "*three* extremos" que inventaron XP es el mismo que publicó recientemente *Extreme Programming Adventures in C#* [Jef04]. Incidentalmente, hay herramientas para trabajar en términos a la vez conformes a MSF y a las prácticas ágiles, como csUnit para .NET, un verificador de regresión del tipo que exigen los procesos de XP (figura), o NUnitAsp para Asp.NET [NV04]. Instrumentos de este tipo permiten implementar la práctica de prueba automática y diseño orientado por pruebas requeridos por los MAs. También hay disponibilidad de herramientas comerciales de refactorización para .NET (VB y C#), como Xtreme Simplicity (http://www.xtreme-simplicity.net) y .NET Refactoring (http://www.dotnetrefactoring.com), así como un proyecto de código abierto, Opnieuw (http://sourceforge.net/projects/opnieuw).

Tanto en los MAs como en la estrategia arquitectónica de Microsoft coinciden en su apreciación de los patrones. Toda la práctica de Patterns & Practices de Microsoft, igual que es el caso en Scrum y en XP, utiliza numerosos patrones organizativos y de diseño propuestos por Martin Fowler [MS04]. Por su parte, Tom Gilb considera que las prácticas metodológicas internas de Microsoft, desde hace ya mucho tiempo, aplican principios que son compatibles con los de su modelo de Evolutionary Project Management (Evo), al punto que sirven para ilustrarlos, uno por uno, en viñetas destacadas [Gilb97].

David Preedy (de Microsoft Consulting Services) y Paul Turner (miembro del DSDM Consortium) elaboraron los principios subyacentes a sus respectivos modelos y también encontraron convergencia y armonía entre DSDM y MSF [PT03]. En el área de aplicabilidad, MSF se caracteriza como un marco de desarrollo para una era caracterizada por tecnología cambiante y servicios distribuidos; de la misma manera, DSDM se aplica a los mismos escenarios, excluyendo situaciones de requerimientos fijos y tecnologías monolíticas.

La participación de los usuarios en el proyecto es considerada en parecidos términos en ambos marcos, siendo MSF un poco más permisivo respecto de su eventual no-participación. En ambos marcos se observan similares temperamentos en relación con la capacidad de decisión del equipo, la producción regular de entregables frecuentes, los criterios de aceptabilidad, el papel de los prototipos en el desarrollo iterativo, la posibilidad de volver atrás (línea temprana de base y congelamiento tardío en MSF, reversibilidad de cambios en DSDM) y el papel crítico de los verificadores durante todo el ciclo de vida.

Hay algunas diferencias menores en la denominación de los roles, que son 6 en MSF y 10 en DSDM, pero no es difícil conciliar sus especificaciones porque ambas contemplan complementos. Lo mismo vale para la prueba de control de calidad, aunque en DSDM no hay un rol específico para QA. Los hitos externos de MSF son cuatro y los de DSDM son cinco, pero su correspondencia es notable: el primero establece Visión & Alcance, Aprobación de la Especificación Funcional, Alcance Completo y Entrega; el segundo, Viabilidad, Estudio de Negocios, Modelo Funcional, Diseño & Construcción e Implementación.

En MSF se prioriza la gestión de riesgos, como una forma de tratar las áreas más riesgosas primero; en DSDM la prioridad se basa en el beneficio de negocios antes que en el riesgo. Pero son semejantes los criterios sobre el tamaño de los equipos, la

importancia que se concede a la entrega en fecha, la gestión de versiones con funcionalidad incremental, el desarrollo basado en componentes (en la versión 4.x de DSDM, no así en la 3), así como el papel que se otorga al usuario en los requerimientos [PT03]. En DSDM Consortium también se ha investigado y puesto a prueba la integración de DSDM, MSF y Prince2 en proyectos ágiles complejos. De la combinación de estos métodos y estándares ha dicho Paul Turner que es incluso superior a la suma de las partes [Tur03].

En suma, muchos de los principios de los MAs son consistentes no sólo con los marcos teóricos, sino con las prácticas de Microsoft, las cuales vienen empleando ideas adaptativas, desarrollos incrementales y metodologías de agilidad (algunas de ellas radicales) desde hace mucho tiempo. Se puede hacer entonces diseño y programación ágil en ambientes Microsoft, o en conformidad con MSF (ya sea en modalidad corporativa como en DSDM, o en modo *hacker* como en XP), sin desnaturalizar a ninguna de las partes.

Métodos y Patrones

Una búsqueda de “extreme programming” en Google devuelve hoy algo así como 433.000 punteros; “design patterns” retorna, por su parte, 982.000. Son grandes números, aún en contraste con los 3.940.000 de “object oriented”, los 2.420.000 de “uml”, los 1.720.000 de “visual c++” o los 4.510.000 de “visual basic”, que vienen desarrollándose desde hace mucho más tiempo. Siendo la Programación Extrema en particular y los MAs en general dos de los tópicos más novedosos y expansivos en la actualidad, cabría preguntarse cuál es su relación con los patrones de diseño, que sin duda constituyen el otro tema de conversación dominante en tecnología de software.

Ante semejante pregunta no cabe una respuesta uniforme: los MAs de estilo *hacker*, como XP o Scrum, y en otro orden FDD, celebran a los patrones como una de sus herramientas cardinales, a tono con un paradigma de programación orientada a objetos; los MAs inclinados a una visión general del proceso, pensados como lecturas para el staff de *management* o candidatos a ser tenidos en cuenta en las metodologías corporativas, pocas veces o nunca se refieren a ellos. No hablan siquiera de programación concreta, en última instancia. La programación con patrones en ambientes XP se trata en infinitos sitios de la Web; una idea semejante en DSDM sería impensable. Esta situación corrobora que el concepto de MAs engloba dos niveles de análisis, dos lugares distintos en el ciclo de vida y dos mundos de *stakeholders* diferentes, por más que todos sus responsables se hayan unido alguna vez (y lo sigan haciendo) para elaborar el Manifiesto y promover la visión ágil en su conjunto.

En esta serie de estudios de Arquitectura de Software, los patrones se definen y estudian en un documento separado, por lo que obviamos aquí su historia y su definición. Hay diversas clases de patrones: de arquitectura, de diseño, organizacionales. En XP los patrones de diseño son la forma nativa de la refactorización, así como la orientación a objetos es la modalidad por defecto de su paradigma de programación. Highsmith estima que los patrones proporcionan a la programación actual lo que el diseño estructural de Larry Constantine y Ed Yourdon suministraban una generación antes: guías efectivas para la estructura de programas [Hig00b]. Highsmith piensa que los lineamientos

provistos por los patrones son, hoy por hoy, la técnica de programación más productiva y estructurada de que se dispone.

Ahora bien, si se piensa que la idea de patrones de la informática fue tomada de la arquitectura de edificios y ciudades (de donde procede también el concepto, a más alto nivel, de arquitectura de software), puede concluirse que hay patrones no solamente en la funcionalidad del código, sino, como decía Christopher Alexander [Ale77], en todas partes. Algunos de los que desarrollaron métodos percibieron lo mismo, y de eso trata el resto del capítulo: cuáles son los patrones propios de una metodología ágil.

En el mismo espíritu alexanderiano de Coplien [Cop95], que más abajo se examinará en detalle, Jim Highsmith propone reemplazar la idea de procesos por la de patrones en su Adaptive Management, un modelo anterior a la elaboración de su método ASD y a la firma del Manifiesto Ágil [Hig99]. A diferencia de la noción de proceso, que la imaginación vincula con circuitos cibernéticos de entrada-salida y control, los patrones proporcionan un marco para pensar antes que un conjunto de reglas para obedecer. En esa tesitura, Highsmith propone tres patrones de gestión de Liderazgo-Colaboración que llama *Encontrando el Equilibrio en el Filo del Caos*, *Encontrando el Equilibrio en el Filo del Tiempo*, y *Encontrando el Equilibrio en el Filo del Poder*.

- El primero sugiere el ejercicio del control justo para evitar precipitarse en el caos, pero no tan estrecho que obstaculice el juicio creativo. El caos es fácil: simplemente precipítese en él. La estabilidad es fácil: sólo tiene que seguir los pasos. El equilibrio, en cambio, es difícil y delicado.
- Encontrar el equilibrio en el filo del tiempo implica hacer una arquitectura no de las estructuras, sino del devenir. Si se concibe el mundo como estable y predecible, el cambio se verá como una aberración temporaria en la transformación de un estado al otro. Si se presume que todo es impredecible, sobrevendrá la inmovilidad. Los arquitectos del tiempo presuponen que todo cambia, pero en diferentes ciclos de tiempo. Separan las fuerzas estabilizadoras de las que desestabilizan y manejan el cambio en vez de ser manejados por él. Lo que hay que preguntarse en consecuencia no es "¿Cual es el mejor XYZ?", sino "¿Cuál es nuestra mejor estrategia cuando algo cambia?". Cuanto más turbulento el ambiente, más corto deberá ser el ciclo. La adaptación consiste en mejoras, migración e integración. Los arquitectos del tiempo no están inmovilizados en el presente; por el contrario, equilibran el aprendizaje del pasado con la anticipación del futuro y entienden cuál puede ser el manejo del tiempo más adecuado. Hay que aprender a crear un *timing* apropiado para manejar el tiempo y convertir un impedimento en un beneficio.
- La arquitectura del poder implica adaptabilidad. En opinión de Highsmith, los billones de dólares gastados en "*change management*" constituyen un despilfarro. Hay una sutil diferencia entre adaptarse y cambiar. Adaptarse involucra poner el foco en lo que pasa afuera (el ambiente, el mercado), en vez de dejarse llevar por las políticas internas. En tiempos turbulentos, esperar que el pináculo de la jerarquía de la empresa se digne a involucrarse es una receta para el fracaso. La adaptación necesita ocurrir en todos los niveles, lo que significa libre flujo de la información y toma de decisiones distribuida. Compartir el poder es la característica fundamental de las organizaciones adaptables.

En otra aplicación del concepto, Alistair Cockburn, experto reconocido en programación orientada a objetos y en casos de uso, desarrolló para Crystal un atractivo uso de los patrones. El argumento de Cockburn es que las formas tradicionales de organizar los artefactos, que en algunos casos son docenas, mezclan conceptos de factoría con propiedades del producto (visión, declaraciones de misión, prototipos conceptuales) e idealizan la asignación de roles a las personas. Las formas de expresar propiedades (diagramas HIPO, gráficos estructurales, diagramas de flujo, casos de uso, diagramas de clase) van y vienen con las modas y la tecnología. En las formas tradicionales, no se consideran ni las reglas culturales ni la situación ambiental.

Cockburn concibe su modelo alternativo como una factoría en la cual las reglas que describen la metodología se expresan mediante patrones. Estudiando la literatura de patrones, Cockburn encontró que ellos corresponden a dos grandes clases: propiedades y estrategias. Algunos elementos de CC pueden ser fácilmente tipificados: la Comunicación Osmótica y la Entrega Frecuente tienen que ver con propiedades, mientras que La Exploración de 360°, el Esqueleto Ambulante y la Rearquitectura Incremental son más bien estrategias. Se puede concebir entonces un modelo de procesos como un conjunto de patrones que expresan su naturaleza; de esa manera, el equipo de programación puede entender el modelo e involucrarse mejor [Coc02]. La idea no está desarrollada más allá de ese punto, pero parece ser un buen elemento para ponerse a pensar.

Podría sospecharse que el examen de Cockburn de la literatura de patrones no fue suficientemente amplia. No menciona, por ejemplo, los patrones organizacionales de James Coplien (de Lucent Technologies, ex-Bell Labs), que sistematizan la misma idea que Cockburn estaba tratando de articular [Cop95]. El principio de estos patrones es el mismo que subyace a las ideas de Christopher Alexander: hay propiedades y elementos que se repiten en diferentes ejemplares de un determinado dominio. Cada ejemplar se puede comprender como una organización particular, compuesta por miembros de un conjunto finito de patrones reutilizables. Un lenguaje de patrones consiste en un repertorio de patrones y reglas para organizarlos. Un patrón es también una instrucción que muestra cómo se puede usar una configuración una y otra vez para resolver determinado problema, dondequiera que el contexto lo haga relevante.

El lenguaje de patrones de Coplien expresa tanto procedimientos metodológicos (*IncrementalIntegation, WorkSplit*), como artefactos (*CRC-CardsAndRoles*), propiedades (*HolisticDiversity*) y circunstancias ambientales (*SmokeFilledRoom, CodeOwnership*). Los Patlets que complementan a los patrones son como patrones sumarios o de alto nivel y sirven también para encontrar los patrones que se necesitan. Los patrones de Coplien, por ende, permiten expresar la configuración de una metodología como una organización de patrones reutilizables, considerados como sus componentes. Idea poderosa, sin duda, que recién comienza a explorarse.

Algunos practicantes de Scrum de primera línea se han acostumbrado a pensar su propio marco como un conjunto o sistema de patrones organizacionales. Mike Beedle, Ken Schwaber, Jeff Sutherland y otros [BDS+98], por ejemplo, suplementaron algunos patrones organizacionales de Coplien (*Firewall, FormFollowsFunction*, *NamedStableBases*) y los del repositorio del sitio de patrones de organización de Lucent Technologies (http://www.bell-labs.com/cgi-user/OrgPatterns/OrgPatterns?ProjectIndex)

con nuevas extensiones del lenguaje, a fin de suministrar una descripción total de Scrum orientada a patrones.

La vigencia de los patrones se percibe además a través de reformulaciones o casos especiales de algunas metodologías nativas, como es el caso de XP@Scrum y sobre todo de XBreed, la combinación de XP, Scrum y patrones ideada por Mike Beedle. XBreed suplanta el Juego de Planeamiento por Scrum, auspicia algo de YAGNI pero en pequeñas dosis, admite CRC para las historias simples pero recomienda casos de uso para las historias complicadas, define un rol de arquitecto e impulsa con entusiasmo, casi excluyentemente, un fuerte uso de patrones.

Como Craig McCormick [McC02] no parece haber leído tampoco a [BDS+98], el brillo de una idea muy parecida a las anteriores se empañó debido a un síndrome que Coplien llamaría *InvenciónIndependiente*. En efecto, McCormick sugiere considerar los MAs, al lado de las prácticas ortodoxas, como una especie de repositorio de patrones de proceso o componentes metodológicos (entregables, técnicas, flujos de proceso, etcétera), junto con lineamientos sobre la manera de ensamblarlos para configurar la metodología de un proyecto determinado. McCormick encuentra que más útil que embarcarse en el objetivo de forjar un método unificado para todos los escenarios, sería articular una serie de frameworks de proceso, una meta-metodología, basada en abstracciones de patrones de procesos reutilizables, fundamentados en la experiencia. Bajo ciertas condiciones de proyecto, Extreme Programming (o RUP, o DSDM, o CMMI) puede ser una instancia del framework. La idea ya existía, por cierto, pues si se lo piensa un poco es evidente que se trata de un patrón recurrente de razonamiento; pero resulta sugerente verla expresada de esta otra forma.

Agilidad, Caos y Complejidad

Muchas de las manifestaciones heterodoxas de los MAs tienen como fuente de inspiración los nuevos paradigmas epistemológicos cristalizados en las teorías del caos determinista de la década de 1980 y las teorías de la complejidad de los 90s. Esas teorías tratan de diversas clases de procesos, fenómenos y sistemas: autómatas celulares, redes booleanas aleatorias, redes neuronales, programación evolutiva, memética, dinámica no lineal, criticalidad auto-organizada, modelos basados en agentes autónomos, metaheurísticas, atractores extraños, gramáticas recursivas, series de Fibonacci, orden emergente, fractales. La cuestión merece tratarse en un estudio específico; aquí sólo brindaremos un puñado de indicios sobre esta peculiar convergencia.

- Algunos de los ideólogos de Scrum [ADM96] se inspiran en conceptos tales como filo del caos y control de procesos caóticos, tal como se los describe en el conocido libro *Chaos* de James Gleick. También afirman que Microsoft (igual que Scrum) implementa en sus prácticas de desarrollo una estrategia de caos controlado, aunque no proporcionan mucho detalle a su razonamiento. Los autores también se refieren al modelo de L. B. S. Racoon [Rac95] referido al caos y al ciclo de vida caótico.
- En MSF 3.0 hay, como se ha visto, un comentario sobre la naturaleza caórdica de los sistemas, en el sentido del concepto acuñado por Dee Hock [Hock00], quien llevara a la práctica este concepto en la gestión de su propia compañía, Visa. Hock sostiene que la ciencia de la complejidad, ligada a la comprensión de los sistemas

autocatalíticos, no lineales, complejos y adaptativos, habrá de ser la ciencia más importante del nuevo siglo. La visión caórdica de Hock (cuyas ideas sobre no linealidad tienen más de un punto en común con las de Fred Brooks) ha inspirado un movimiento caórdico de amplios alcances que va mucho más allá de las metodologías de desarrollo de software. El modelo de gestión Liderazgo-Colaboración de Jim Highsmith, por ejemplo, se funda en las ideas de Hock.

- En Scrum se habla de caos permanentemente. Haciendo honor al nombre, el sitio de Scrum está en http://www.controlchaos.com. La descripción de Craig Larman sobre Scrum [Lar04: 136] incluye como bibliografía de referencia para comprender procesos adaptativos y autoorganización los dos textos clásicos de John Holland, el padre del algoritmo genético, quien también ha ejercido influencia sobre Highsmith, como ya se ha visto. Jeff Sutherland [Sut01] ha señalado la analogía entre los saltos y discontinuidades de los procesos de desarrollo ágiles y el "equilibrio puntuado" de los procesos evolutivos complejos.
- En las páginas de su sitio empresarial y en su bibliografía, Jeff DeLuca establece la estrecha relación entre las premisas de FDD y las teorías del caos determinista y de la complejidad, vinculándolas con el pensamiento no lineal de Fred Brooks y con la concepción del control de caos de Jim Highsmith.
- En su presentación de Adaptive Software Development, Highsmith afirma que su práctica se basa en "estar en equilibrio en el filo del caos": proporcionar suficiente guía para evitar caer en el caos, pero no demasiada, para no suprimir la emergencia y la creatividad. Este modelo basado en la analogía entre empresas/equipos/proyectos y sistemas adaptativos complejos, es, por mucho, el más vinculado a las nuevas ciencias de la complejidad y el caos [Hig00a]. Un ejemplo de estos sistemas podría ser una bandada de pájaros que se sincroniza sin que exista un líder que ejerza control y determine su dirección; cada pájaro sigue reglas simples y locales, pero en la macro-escala la bandada exhibe orden y un claro comportamiento colectivo emergente. Estas ideas han inspirado un modelo de gestión basado en cooperación y competencia [Hig02b].
- Una visión semejante se presenta en el modelo de Mary y Tom Poppendieck de Lean Development [PP03], donde se considera el comportamiento de sistemas sociales de insectos como una manifestación de trabajo en equipo, soluciones adaptativas y capacidades emergentes. LD promueve un diseño semejante a los de XP y Scrum, enfatizando sus aspectos evolutivos.
- Duane Truex [TBT00] de la Universidad de Georgia en Atlanta, promotor de una especie de método ágil llamado Desarrollo Ametódico, se basa en conceptos de dinámica y emergencia organizacional. Su modelo, desarrollado tres años antes de la publicación del Manifiesto, anticipa ideas caórdicas que se encuentran implícitas en los MAs.
- En una entrevista a Kent Beck y Martin Fowler, el primero afirmó que mientras SEI se encuentra en el campo modernista, XP es más bien posmoderno. "Sus raíces filosóficas se encuentran en la teoría de los sistemas complejos" y sus capacidades se generan a través de procesos emergentes. [Beck01]. Es significativo que este juicio haya sido expresado por el creador de la Programación Extrema.

- Barry Boehm, promotor temprano de los MAs y creador de COCOMO y del modelo en espiral que prevalece en MSF [MS03], ha expresado que en el futuro la ingeniería de software deberá explorar los sistemas adaptativos, los agentes autónomos que se auto-organizan, los paisajes de adecuación evolutivos y el orden emergente. Cita al respecto el pensamiento de Stuart Kauffman, estudioso de los fenómenos adaptativos e inventor de las redes booleanas aleatorias, una variante de los sistemas adaptativos complejos a medio camino entre los autómatas celulares y los modelos de agentes [Boe02b].

En fin, hay más de una razón para elaborar en detalle las relaciones entre los MAs (incluyendo MSF) y el paradigma del caos y la complejidad. Lo haremos en otro documento de esta serie, donde también se expondrán los ricos fundamentos y los logros prácticos de esa epistemología.

Anti-agilidad: La crítica de los Métodos Ágiles

A pesar de que nadie en sus cabales se opondría a que se lo considere ágil, adaptable y sensible a los requerimientos del cliente, es natural que se generara oposición a los MAs y que en ocasiones la reacción frente a ellos adoptara una fuerte actitud combativa, para deleite de los cronistas que gustan hablar de “conflagraciones”, “cruzadas” y “acometidas”. Si los ágiles han sido implacables en su caricatura de las metodologías rigurosas, los adversarios que se han ganado no se han quedado atrás en sus réplicas. En esta contienda hay algunas observaciones de carácter técnico, pero la ironía prevalece. Lo que sigue es apenas una muestra.

Uno de los ataques más duros y tempranos contra los MAs proviene de una carta de Steven Rakitin a la revista *Computer*, bajo la rúbrica “El Manifiesto genera cinismo” [Rak01]. Refiriéndose a la estructura opositiva del Manifiesto, Rakitin expresa que en su experiencia, los elementos de la derecha (vinculados a la mentalidad planificadora) son esenciales, mientras que los de la izquierda sólo sirven como excusas para que los *hackers* escupan código irresponsablemente sin ningún cuidado por la disciplina de ingeniería. Así como hay una lectura literal y una estrecha del canon de CMM, Rakitin practica una “interpretación *hacker*” de propuestas de valor tales como “responder al cambio en vez de seguir un plan” y encuentra que es un generador de caos. La interpretación *hacker* de ese mandamiento sería algo así como: “¡Genial! Ahora tenemos una razón para evitar la planificación y codificar como nos dé la gana”.

Más tarde, un par de libros de buena venta, *Questioning Extreme Programming* de Pete McBreen [McB02] y *Extreme Programming Refactored: The case against XP* de Matt Stephens y Doug Rosenberg [SR03] promovieron algunas dudas sobre XP pero no llegaron a constituirse como impugnaciones convincentes por su uso de evidencia circunstancial, su estilo redundante, su tono socarrón y su falta de método argumentativo. Que a último momento se redima a XP sugiriendo mejoras y refactorizaciones no ayuda a la causa de un dictamen anunciado desde sus títulos. Mucho más fundamentado es el estudio puramente crítico de Edward Berard, quien señala un buen número de “falacias” y medias verdades en el discurso ágil pero no logra, por alguna razón que se nos escapa, que su crítica levante vuelo [Ber03].

Gerold Keefer, de AVOCA GmbH, ha publicado ya dos versiones de un estudio titulado (con guiño a los expertos que reconozcan la alusión a Dijkstra) "Extreme Programming considerado dañino para el desarrollo confiable de software" [Kee03]. Algunos de los hechos tenidos en cuenta son que no sólo el primero, sino también el segundo proyecto de referencia de XP fueron cancelados; que las cifras que proporciona Beck sobre el costo del cambio metodológico no son tomadas en serio ni siquiera en el seno de la comunidad XP; que resultados de investigaciones en curso cuestionan la viabilidad de la programación orientada por pruebas[4], cuyo proceso puede consumir hasta el 30% o 40% de los recursos de un proyecto; y que reportes de proyectos de gran envergadura demuestran su insuperable problema de escalabilidad, que se manifiesta antes de lo que se cree.

Otros argumentos de Keefer apuntan a los altos costos y magros resultados de la programación en pares; a las malas prácticas resultantes de la negación a documentar; al retorno a la "programación de garage" y a la idea de que "el código se documenta a sí mismo" cuestionadas por Brooks [Bro75] hace treinta años; a la escasa verosimilitud de las historias de éxito de C3 y VCAPS, etcétera. Por añadidura, XP no se expide sobre proyectos con alcances, precios y fechas fijas, ni sobre requerimientos no funcionales como performance y seguridad, ni sobre ambientes de desarrollo físicamente distribuidos, ni (a pesar de su insistencia en reutilización y patrones) sobre integración de componentes listos para usar (COTS). A Keefer, por último, no le persuade la premisa de "hacer la cosa más simple que pueda funcionar", sino que prefiere la postura de Einstein: "Que sea lo más simple posible, pero no más simple que eso".

Inesperadamente, el promotor de RAD SteveMcConnell se ha opuesto con vehemencia a las ideas más radicales del movimiento ágil. Se ha manifestado contrario tanto a las estrategias técnicas como a las tácticas promocionales; dice McConnell: "Esta industria tiene una larga historia en cuanto a pegarse a cuanta moda sale", pero "luego se encuentra, cuando los bombos y platillos se enfrían, que estas tendencias no logran el éxito que sus evangelistas prometieron". Los cargos de McConnell contra XP son numerosos. En primer lugar, diferentes proyectos requieren distintos procesos; no se puede escribir software de aviónica para un contrato de defensa de la misma manera en que se escribe un sistema de inventario para un negocio de alquiler de videos.

Además, las reglas de XP son excesivamente rígidas; casi nadie aplica las 12 en su totalidad, y pocas de ellas son nuevas: Tom Gilb proponía lineamientos similares muchos años atrás. Dadas las premisas desaliñadas y las falsas publicidades, McConnell expresa que "el fenómeno parece un caso de hipnosis colectiva" [Baer03]. En diversas presentaciones públicas y en particular en SD West 2004, McConnell ha considerado la programación sin diseño previo, el uso atropellado de XP, la programación automática y la costumbre de llamar "ágil" a cualquier práctica como algunas de las peores ideas en construcción de software del año 2000.

---

[4] La programación orientada por pruebas (*test-driven development*) en su modalidad convencional es en efecto resistida en ciertos ambientes académicos. Dijkstra afirmaba que "la prueba de programas puede ser una manera muy efectiva para mostrar la presencia de *bugs*, pero es irremediablemente inadecuada para mostrar su ausencia" [Dij72]. En la academia se prefieren los métodos formales de demostración, a los que en el movimiento ágil rara vez se hace referencia.

Un miembro de Rational, John Smith [SmiS/f], opina que la terminología de XP encubre una complejidad no reconocida; mientras las palabras "artefacto" y "producto de trabajo" no figuran en los índices de sus libros canónicos, Smith cuenta más de 30 artefactos encubiertos: Historias, Restricciones, Tareas, Tareas técnicas, Pruebas de aceptación, Código de software, Entregas, Metáforas, Diseños, Documentos de diseño, Estándares de codificación, Unidades de prueba, Espacio de trabajo, Plan de entrega, Plan de iteración, Reportes y notas, Plan general y presupuesto, Reportes de progreso, Estimaciones de historias, Estimaciones de tareas, Defectos, Documentación adicional, Datos de prueba, Herramientas de prueba, Herramientas de gestión de código, Resultados de pruebas, *Spikes* (soluciones), Registros de tiempo de trabajo, Datos métricos, Resultados de seguimiento. La lista, dice Smith, es indicativa, no exhaustiva. En un proyecto pequeño, RUP demanda menos que eso. Al no tratar sus artefactos como tales, XP hace difícil operarlos de una manera disciplinada y pasa por ser más ligero de lo que en realidad es.

Hay otras dificultades también; a diferencia de RUP, en XP las actividades no están ni identificadas ni descriptas. Las "cosas que se hacen" están tratadas anecdóticamente, con diversos grados de prescripción y detalle. Finalmente hay prácticas en XP que decididamente escalan muy mal, como la refactorización, la propiedad colectiva del código, las metáforas en lugar de una arquitectura tangible y las entregas rápidas, que no tienen en cuenta siquiera cuestiones elementales de la logística de despliegue. Smith afirma no estar formulando una crítica, pero pone en duda incluso, por su falta de escalabilidad, que XP esté haciendo honor a su nombre. Por otra parte, en toda la industria se sabe que la refactorización *en general* no escala muy bien. Eso se manifiesta sobre todo en proyectos grandes en los que hay exigencias no funcionales que son clásicas "quebradoras de arquitectura", como ser procedimientos ad hoc para resolver problemas de performance, seguridad o tolerancia a fallas. En estos casos, como dice Barry Boehm [Boe02a], ninguna dosis de refactorización será capaz de armar a Humpty Dumpty nuevamente.

Mientras Ivar Jacobson [Jac02] y Grady Booch (ahora miembro de la Agile Alliance) han saludado la aparición de los métodos innovadores (siempre que se los combine con RUP), un gran patriarca del análisis orientado a objetos, Stephen Mellor [Mel03], ha cuestionado con acrimonia sus puntos oscuros demostrando, como al pasar, que UML ejecutable sería mejor. Mellor impugna la definición de "cliente" que hacen los MAs: no existe, dice, semejante cosa. En las empresas hay expertos en negocios y en tecnología, en mercadeo y en productos; los MAs no especifican quién de ellos debe estar en el sitio, cuándo y con qué frecuencia. Mellor también deplora las consignas estridentes de los agilistas y escenifica una parodia de ellas: "¡Programadores del mundo, uníos!", "¡Aplastemos la ortodoxia globalizadora!", "¡Basta de exportar puestos de trabajo en programación!", "Un espectro recorre Europa ¡El espectro es Extreme!"… Además se pregunta: en un sistema muy distribuido ¿quién es el usuario de un determinado dominio? ¿Qué cosa constituye una "historia simple"? ¿Cómo se mantiene una pieza para que sea reutilizable?

Como este no es un estudio evaluativo, dejamos las preguntas de Mellor en suspenso. El lector que participa en esta comunidad tal vez pueda ofrecer sus propias respuestas, o pensar en otros dilemas todavía más acuciantes.

Conclusiones

La tabla siguiente permite apreciar las convergencias y divergencias en la definición de los MAs de primera línea, así como resumir sus características claves, los nombres de los promotores iniciales y sus fechas de aparición. En la comunidad de los MAs, ninguno de los autores se identifica solamente con su propia criatura. Muchos teóricos y practicantes se mueven con relativa frecuencia de un método ágil a otro, y a veces se los encuentra defendiendo y perfeccionando modelos que no necesariamente son los suyos.

Metodología	Acrónimo	Creación	Tipo de modelo	Característica
Adaptive Software Development	ASD	Highsmith 2000	Prácticas + Ciclo de vida	Inspirado en sistemas adaptativos complejos
Agile Modeling	AM	Ambler 2002	"Metodología basada en la práctica"	Suministra modelado ágil a otros métodos
Crystal Methods	CM	Cockburn 1998	"Familia de metodologías"	MA con énfasis en modelo de ciclos
Agile RUP	dX	Booch, Martin, Newkirk 1998	Framework / Disciplina	XP dado vuelta con artefactos RUP
Dynamic Solutions Delivery Model	DSDM	Stapleton 1997	Framework / Modelo de ciclo de vida	Creado por 16 expertos en RAD
Evolutionary Project Management	Evo	Gilb 1976	Framework adaptativo	Primer método ágil existente
Extreme Programming	XP	Beck 1999	"Disciplina en prácticas de ingeniería"	Método ágil radical
Feature-driven development	FDD	De Luca & Coad 1998 Palmer & Felsing 2002	"Metodología"	Método ágil de diseño y construcción
Lean Development	LD	Charette 2001, Mary y Tom Poppendieck	"Forma de pensar" – Modelo logístico	Metodología basada en procesos productivos
Microsoft Solutions Framework	MSF	Microsoft 1994	Lineamientos, Disciplinas, Prácticas	Framework de desarrollo de soluciones
Rapid Development	RAD	McConnell 1996	Survey de técnicas y modelos	Selección de *best practices*, no método
Rational Unified Process	RUP	Kruchten 1996	Proceso unificado	Método (¿ágil?) con modelado
Scrum	Scrum	Sutherland 1994 - Schwaber 1995	"Proceso" (framework de management)	Complemento de otros métodos, ágiles o no

Se han omitido algunas corrientes cuyas prácticas reflejan parecidos ocasionales, así como la mayor parte de los modelos híbidos (XBreed, XP@Scrum, EUP, EnterpriseXP, DSDM+Prince2+MSF, IndustrialXP), aunque se ha dejado la referencia a Agile RUP (o dX) debido al valor connotativo que tiene la importancia de sus promotores. Los métodos ágiles híbridos y los dialectos industriales merecen un estudio separado. Aparte de los mencionados quedan por mencionar Grizzly de Ron Crocker [Cro04] y Dispersed eXtreme Programming (DXP), creado por Michael Kircher en Siemens.

Más allá de la búsqueda de coincidencias triviales (por ejemplo, la predilección por el número 12 en los principios asociados al Manifiesto, en Lean Development, en las prácticas del primer XP), queda por hacer un trabajo comparativo más sistemático, que ponga en claro cuáles son los elementos de juicio que definen la complementariedad, cuál es el grado de acatamiento de los principios ágiles en cada corriente, cuáles son los escenarios que claman por uno u otro método, o cómo engrana cada uno (por ejemplo) con CMMI y los métodos de escala industrial.

Tras la revisión que hemos hecho, surge de inmediato que entre los diversos modelos se presenta una regularidad inherente a la lógica de las fases, por distintas que sean las premisas, los ciclos, los detalles y los nombres. Dado que el ciclo de vida es de diferente longitud en cada caso, las fases de la tabla no son estrictamente proporcionales.

Los métodos que hemos examinado no son fáciles de comparar entre sí conforme a un pequeño conjunto de criterios. Algunos, como XP, han definido claramente sus procesos, mientras que otros, como Scrum, son bastante más difusos en ese sentido, limitándose a un conjunto de principios y valores. Lean Development también presenta más principios que prácticas; unos cuantos que ellos, como la satisfacción del cliente, están menos articulados que en Scrum, por ejemplo. Ciertos MAs, como FDD, no cubren todos los pasos del ciclo de vida, sino unos pocos de ellos. Varios métodos dejan librada toda la ingeniería concreta a algún otro método sin especificar.

Modelo	→ Fases →				
Modelo en Cascada	Especificar requerimientos	Especificar funciones	Especificar diseño	Codificación y Prueba	Prueba y validación
ASD	Especular	Colaborar	Aprender		
DSDM	Estudio de Viabilidad	Estudio del Negocio	Iteración del Modelo Funcional	Iteración Diseño & Versión	Implementación
Evo	Concepto	Análisis prelim. Req.	Diseño Arquitect.	Desarrollo iterativo	Entrega
Extreme Programming	Exploración	Planeamiento	Iteraciones hasta entrega		Productización
FDD	Modelo general	Lista de Rasgos	Planear por Rasgo	Diseñar por Rasgo	Construir por Rasgo
Microsoft Sync & Stab.	Planeamiento	Desarrollo			Estabilización
Proceso Unificado	Incepción	Elaboración	Construcción		Transición
Scrum	Pre-juego: Planeamiento	Pre-juego: Montaje	Juego		Liberación

Por más que exista una homología estructural en su tratamiento del proceso, se diría que en un primer análisis hay dos rangos o conjuntos distintos de MAs en la escala de complejidad. Por un lado están los MAs declarativos y programáticos como XP, Scrum, LD, AM y RAD; por el otro, las piezas mayores finamente elaboradas como Evo, DSDM y Crystal. En una posición intermedia estaría ASD, cuya naturaleza es muy peculiar. No calificaríamos a RUP como metodología ágil en plenitud, sino más bien como un conjunto enorme de herramientas y artefactos, al lado de unos (pocos) lineamientos de uso, que acaso están mejor articulados en AM que en la propia documentación nativa de RUP. Los recursos de RUP se pueden utilizar con mayor o menor conflicto en cualquier otra estrategia, sin necesidad de crear productos de trabajo nuevos una y otra vez. Como sea, habría que indagar en qué casos los métodos admiten hibridación por ser semejantes y complementarios en un mismo plano, y en qué otros escenarios lo hacen porque se refieren a distintos niveles de abstracción.

La intención de este trabajo ha sido describir la naturaleza de los MAs, establecer su estado actual y poner en claro sus relaciones con otras metodologías, en particular Microsoft Solutions Framework. Esperamos que la escala de tratamiento haya permitido

apreciar que el conjunto de los MAs es más heterogéneo de lo que sostiene la prensa de ambos bandos, que los estereotipos sobre su espíritu transgresor no condicen mucho con los rigores de algunas de sus formulaciones, y que la magnitud de sus novedades se atempera un poco cuando se comprueba que algunas de sus mejores ideas son también las más antiguas y seguras.

No ha sido parte del programa de este estudio la promoción o la crítica de los métodos; si se han documentado las protestas en su contra ha sido solamente para ilustrar el contexto. No se ha buscado tampoco marcar su contraste o su afinidad con los métodos pesados; sólo se ha señalado su complementariedad allí donde se la ha hecho explícita. Un ejercicio pendiente sería tomar el modelo de ciclos de cada método particular, o sus artefactos, o su definición de los equipos de trabajo, y mapear cada elemento de juicio contra los de MSF, porque hasta hoy esto se ha hecho solamente a propósito de DSDM [PT03] o, a nivel de código, con XP [Jef04].

Esta operación podría establecer correspondencias encubiertas por diferencias meramente terminológicas y señalar los puntos de inflexión para el uso de los MAs en las disciplinas de MSF. También habría que considerar combinaciones múltiples. No sería insensato proponer que MSF se utilice como marco general, Planguage como lenguaje de especificación de requerimiento, Scrum (con sus patrones organizacionales) como método de management, XP (con patrones de diseño, programación guiada por pruebas y refactorización) como metodología de desarrollo, RUP como abastecedor de artefactos, ASD como cultura empresarial y (¿por qué no?) CMM como metodología de evaluación de madurez.

Hay mucho por hacer en la comparación de los métodos; los *surveys* disponibles en general han tratado sólo pequeños conjuntos de ellos, sin que pueda visualizarse todavía una correlación sistemática sobre una muestra más amplia y menos sesgada. También hay tareas por hacer en la coordinación de los MAs con un conjunto de técnicas, prácticas y frameworks que guardan con ellos relaciones diferenciales: metodologías basadas en arquitectura, programación guiada por rasgos y por pruebas, métodos formales de verificación, lenguajes de descripción arquitectónica, modelos de línea de productos, patrones de diseño y organizacionales, refactorización.

Como lo testimonian los acercamientos desde los cuarteles de UML/RUP, o la reciente aceptación de muchos de sus principios por parte de los promotores de CMMI o ISO9000, o la prisa por integrarlos a otros métodos, o por encontrar términos equidistantes, los MAs, al lado de los patrones, han llegado para quedarse. Han democratizado también la discusión metodológica, antes restringida a unos pocos especialistas corporativos y a clubes organizacionales de acceso circunscripto. Se puede disentir con muchos de ellos, pero cabe reconocer su obra de transformación: de ahora en adelante, la metodología no volverá a ser la misma.

Vínculos ágiles

Existen innumerables sitios dedicados a MAs, más o menos permanentes. La disponibilidad de documentos y discusiones sobre esos métodos es abrumadora. Aquí sólo hemos consignado los sitios esenciales, activos en Abril de 2004.

http://agilemanifesto.org/ – El Manifiesto Ágil.

http://alistair.cockburn.us/crystal/aboutmethodologies.html – Página inicial de Crystal Methodologies, de Alistair Cockburn.

http://c2.com/cgi/wiki?ExtremeProgrammingRoadmap – Materiales de XP en el sitio de Ward Cunnimgham, creador del Wiki Wiki Web.

http://comunit.sourceforge.net – Página de COMUnit en Source Forge. Herramienta para prueba de unidad y prueba funcional requeridas por XP para programas en Visual Basic .NET.

http://dotnetjunkies.com/WebLog/darrell.norton/articles/4306.aspx – Weblog consagrado a Lean Software Development (LSD) con abundantes referencias a tecnología .NET y vínculos adicionales.

http://dotnetrefactoring.com – Herramientas de refactorización para .NET.

http://industrialxp.org – Páginas de IndustrialXP (IXP), de Industrial Logic.

http://jeffsutherland.com – Sitio de Jeff Sutherland, con vínculos a Scrum y artículos sobre tecnologías ligadas a XML y SOAP.

http://members.aol.com/humansandt/crystal/clear – Descripción de Crystal Clear.

http://name.case.unibz.it – Sitio de NAME (Network for Agile Methodologies Experience) de Italia. Contiene vínculo a csUnit.

http://www.nunit.org – Herramienta de desarrollo orientado a prueba y refactorización para .NET.

http://sourceforge.net/projects/nunit – Herramienta de prueba de unidad para lenguajes del framework .NET.

http://sourceforge.net/projects/opnieuw – Proyecto de código abierto para refactorización en lenguaje C#.

http://www.adaptivesd.com – Página de Jim Highsmith y Adaptive Software Development.

http://www.agilealliance.com – Numerosos artículos específicos de MAs y vínculos.

http://www.agilemodeling.com – Sitio oficial de Agile Modeling de Scott Ambler.

http://www.controlchaos.com – Sitio de Ken Schwaber. Referencias a Scrum y otros métodos.

http://www.csunit.org/index.php – Sitio de csUnit, una herramienta de control para VisualStudio .NET conforme a las prácticas de prueba automatizada de unidades en XP.

http://www.cutter.com – Página de Cutter Consortium – Incluye publicaciones especializadas en MAs.

http://www.dotnetrefactoring.com – Vínculo a C# Refactoring Tool, herramienta comercial de refactorización para .NET.

http://www.dsdm.org – Sitio de DSDM. Hay documentos específicos de comparación e integración con Microsoft Solutions Framework.

http://www.enterprisexp.org – Página de Enterprise XP en Quadrus Developments.

http://www.extremeprogramming.org – Página semi-oficial de XP.

http://www.gilb.com – Es el sitio de Tom Gilb, creador de Evo. Dispone de una buena cantidad de documentos, gráficas y glosarios, incluyendo el manuscrito de 600 páginas de Competitive Engineering [Gilb03a].

http://www.iturls.com/English/SoftwareEngineering/SE_Agile.asp – Sitio de IT Source sobre ingeniería de software ágil. Innumerables artículos y vínculos.

http://www.iturls.com/English/SoftwareEngineering/SE_fop.asp – Sitio de IT Source sobre Feature Oriented Programming y FDD, con nutrido repertorio de vínculos relevantes.

http://www.jimhighsmith.com – Páginas de Jim Highsmith y Adaptive Software Development.

http://www.mapnp.org/library/quality/tqm/tqm.htm – Vínculos a información sobre Total Quality Management (TQM).

http://www.nebulon.com/fdd/index.html – Sitio de FDD en la empresa Nebulon de Jeff DeLuca, "consultor, autor e inventor" que creó el método. Abundante información del modelo, diagramas y plantillas.

http://www.objectmentor.com – Sitio de la compañía de Robert C. Martin, signatario del Manifiesto, con vínculos a sitios de XP, C#, Visual Basic .NET, .NET Enterprise Solution Patterns.

http://www.poppendieck.com/ld.htm – Página principal de los creadores de Lean Development.

http://www.refactoring.com – Página de Martin Fowler consagrada a refactorización. En http://www.refactoring.com/catalog/index.html se encuentran las refactorizaciones más comunes.

http://www.vbunit.org – vbUnit 3, herramienta de prueba de unidad para Visual Basic.

http://www.xbreed.net/ – Página de XP+Scrum+Patrones por Mike Beedle

http://www.xprogramming.com/index.htm – Recursos comunitarios para eXtreme Programming, mantenido por Ron Jeffries. Hay materiales relacionados con .NET y C#.

http://www.xprogramming.com/software.htm – Software para desarrollo en XP sobre diversas plataformas, con sección de herramientas para .NET.

http://www.xtreme-simplicity.net – Herramientas para refactorización conformes a XP para Visual Basic.NET y C#.

Referencias bibliográficas

[Abr02] Pekka Abrahamsson. "Agile Software development methods: A minitutorial". VTT Technical Research Centre of Finland, http://www.vtt.fi/virtual/agile/seminar2002/Abrahamsson_agile_methods_minitutorial.pdf, 2002.

[ADM96] Advanced Development Methods. "Controlled chaos: Living on the Edge", http://www.controlchaos.com/ap.htm, 1996.

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

[Amb02a] Scott Ambler. *Agile Modeling: Effective practices for Extreme Programming and the Unified Process*. John Wiley & Sons, 2002.

[Amb02b] Scott Ambler. "Agile Modeling and the Unified Process". http://www.agilemodeling.com/essays/agileModelingRUP.htm, 2002.

[Amb02c] Scott Ambler. "The Enterprise Unified Process (EUP): Extending the RUP for real-world organizations". http://szakkoli.sch.bme.hu/ooffk/oookea/Scott_Ambler_EUPOverview.pdf, 2003.

[Amb03] Scott Ambler. "Artifacts for Agile Modeling: The UML and beyond", http://www.agilemodeling.com/essays/modelingTechniques.htm, 2003.

[Amb04] Scott Ambler. "Agile Modeling Essays", http://www.agilemodeling.com/essays.htm, 2004.

[ASR+02] Pekka Abrahamsson, Outi Salo, Jussi Ronkainen y Juhani Warsta. "Agile Software Development Methods". *VTT Publications* 478, Universidad de Oulu, Suecia, 2002.

[Baer03] Martha Baer. "The new X-Men". *Wired 11.09*, Setiembre de 2003.

[Bat03] Don Batory. "A tutorial on Feature Oriented Programming and Product Lines". *Proceedings of the 25th International Conference on Software Engineering, ICSE'03*, 2003.

[BBB+01a] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunnimgham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland y Dave Thomas. "Agile Manifesto". http://agilemanifesto.org/, 2001.

[BBB+01b] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunnimgham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland y Dave Thomas. "Principles behind the Agile Manifesto". http://agilemanifesto.org/principles.html, 2001.

[BC89] Kent Beck y Ward Cunningham. "A laboratory for teaching Object-Oriented thinking". *OOPSLA'89 Conference Proceedings, SIGPLAN Notices*, 24(10), Octubre de 1989.

[BDS+98] Mike Beedle, Martine Devos, Yonat Sharon, Ken Schwaber y Jeff Sutherland. "SCRUM: A pattern language for hyperproductive software development". En N. Harrison, B. Foote, and H. Rohnert, eds., *Pattern Languages of Program Design*, vol. 4, pp. 637-651. Reading, Addison-Wesley, 1998.

[Beck99a] Kent Beck. *Extreme Programming Explained: Embrace Change*. Reading, Addison Wesley, 1999.

[Beck99b] Kent Beck, "RE: (OTUG) XP and Documentation". *Rational's Object Technology User Group Mailing List*, 23 de Marzo de 1999, 18:26:04 +0100.

[Beck01] Kent Beck. "Interview with Kent Beck and Martin Fowler". Addison-Wesley, http://www.awprofessional.com/articles/article.asp?p=20972&redir=1, 23 de Marzo de 2001.

[Ber03] Edward Berard. "Misconceptions of the Agile zealots". The Object Agency, http://www.svspin.org/Events/Presentations/MisconceptionsArticle20030827.pdf, 2003.

[BF00] Kent Beck y Martin Fowler. *Planning Extreme Programming*. Reading, Addison Wesley, 2000.

[BMP98] Grady Booch, Robert Martin y James Newkirk. *Object Oriented Analysis and Design With Applications*, 2ª edición, Addison-Wesley, 1998

[Boe02a] Barry Boehm. "Get ready for Agile Methods, with care". *Computer*, pp. 64-69, Enero de 2002.

[Boe02b] Barry Boehm. "The future of Software Engineering". USC-CSE, Boeing Presentation, 10 de Mayo de 1992.

[Bro75] Frederick Brooks. *The Mythical Man-Month*. Reading, Addison-Wesley.

[Bro87] Frederick Brooks. "No silver bullets – Essence and accidents of software engineering". *Computer*, pp. 10-19, Abril de 1987.

[Bro95] Frederick Brooks. *The Mythical Man-Month*. Boston, Addison-Wesley.

[BT75] Vic Basili y Joe Turner. "Iterative Enhancement: A Practical Technique for Software Development", *IEEE Transactions on Software Engineering*, 1(4), pp. 390-396, 1975.

[Cha01] Robert Charette, *The Foundation Series on Risk Management, Volume II: Foundations of Lean Development*, Cutter Consortium, Arlington, 2001.

[Cha04] Robert Charette. "The decision is in: Agile versus Heavy Methodologies". Cutter Consortium, *Executive Update*, 2(19), http://www.cutter.com/freestuff/apmupdate.html, 2004.

[CLC03] David Cohen, Mikael Lindvall y Patricia Costa. "Agile Software Development. A DACS State-of-the-Art Report", *DACS Report*, The University of Maryland, College Park, 2003.

[CLD00] Peter Coad, Eric Lefebvre y Jeff DeLuca. *Java modeling in color with UML: Enterprise components and process*. Prentice Hall, 2000.

[Coc97a] Alistair Cockburn. “Software development as Community Poetry Writing. Cognitive, cultural, sociological, human aspects of software development”. *Annual Meeting of the Central Ohio Chapter of the ACM*, http://alistair.cockburn.us/crystal/articles/sdacpw/softwaredevelopmentascommunitypoetrywriting.html, 1997.

[Coc97b] Alistair Cockburn. *Surviving Object-Oriented Projects*. Addison-Wesley.

[Coc00] Alistair Cockburn. “Balancing Lightness with Sufficiency”. http://alistair.cockburn.us/crystal/articles/blws/balancinglightnesswithsufficiency.html, Setiembre de 2000.

[Coc01] Alistair Cockburn. *Writing effective Use Cases*. Reading, Addison-Wesley.

[Coc02] Alistair Cockburn. “Crystal Clear. A human-powered methodology for small teams, including The Seven Properties of Effective Software Projects”. Borrador. *Humans and Technology*, versión del 27 de febrero de 2002.

[Cop01] Lee Copeland. “Developers approach Extreme Programming with caution”. *Computerworld*, p. 7, 22 de Octubre de 2001.

[Cro04] Ron Crocker. *Large-scale Agile Software Development*. Addison-Wesley, 2004.

[CS95a] Michael Cusumano y Richard Selby. *Microsoft secrets: How the world’s most powerful software company creates technology, shapes markets, and manages people*. The Free Press, 1995.

[CS95b] James O. Coplien y Douglas C. Schmidt, *Pattern Languages of Program Design* (*A Generative Development-Process Pattern Language)*, Reading, Addison Wesley, 1995.

[Dij72] Edsger Dijkstra. “The Humble Programmer,” 1972 ACM Turing Award Lecture, ACM Annual Conference, Boston, 14 de Agosto - *Communications of the ACM*, 15(10), pp. 859-866, Octubre de 1972.

[DS90] Peter DeGrace y Leslie Hulet Stahl. *Wicked problems, righteous solutions*. Englewood Cliffs, Yourdon Press, 1990.

[DS03] DSDM Consortium y Jennifer Stapleton. *DSDM: Business Focused Development*. 2ª edición, Addison-Wesley, 2003.

[ESPI96] ESPI Exchange. “Productivity claims for ISO 9000 ruled untrue”. Londres, European Software Process Improvement Foundation, p. 1, 1996.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke y Don Roberts. *Refactoring: Improving the design of existing code*. Addison Wesley, 1999.

[Fow01] Martin Fowler. “Is design dead?”. *XP2000 Proceedings*, http://www.martinfowler.com/articles/designDead.html, 2001.

[Gilb76] Tom Gilb. *Software metrics*. Chartwell-Bratt, 1976.

[Gilb88] Tom Gilb. *Principles of Software Engineering Management*. Reading, Addison-Wesley, 1988.

[Gilb97] Tom Gilb. *The Evolutionary Project Manager Handbook*. Evo Mini-Manuscript, http://www.ida.liu.se/~TDDB02/pkval01vt/EvoBook.pdf.

[Gilb99] Tom Gilb. "Evolutionary Project Management". Manuscrito inédito, material para el curso DC SPIN, junio de 1999.

[Gilb02] Tom Gilb. "10 Evolutionary Project Management (Evo) Principles". http://www.xs4all.nl/~nrm/EvoPrinc/EvoPrinciples.pdf, Abril de 2002.

[Gilb03a] Tom Gilb. *Competitive Engineering*. [Borrador] www.gilb.com, 2003.

[Gilb03b] Kai Gilb. *Evolutionary Project Management & Product Development (or The Whirlwind Manuscript)*, http://www.gilb.com/Download/EvoProjectMan.pdf, 2003.

[Gla01] Robert Glass. "Agile versus Traditional: Make love, not war". *Cutter IT Journal*, 14(12), diciembre de 2001.

[HesS/f] Wolfgang Hesse. "Dinosaur meets Archaeopteryx? Seven Theses on Rational's Unified Process (RUP)", http://citeseer.ist.psu.edu/571547.html, sin fecha.

[HHR+96] Pat Hall, Fiona Hovenden, Janet Rachel y Hugh Robinson. "Postmodern Software Development". *MCS Technical Reports*, 1996.

[Hig99] Jim Highsmith. "Adaptive Management Patterns". *Cutter IT Journal*, 12(9), Setiembre de 1999.

[Hig00a] Jim Highsmith. *Adaptive software development: A collaborative approach to managing complex systems*. Nueva York, Dorset House, 2000.

[Hig00b] Jim Highsmith. "Extreme Programming". *EBusiness Application Development*, Cutter Consortium, Febrero de 2000.

[Hig01] Jim Highsmith. "The Great Methodologies Debate. Part 1". *Cutter IT Journal*, 14(12), diciembre de 2001.

[Hig02a] Jim Highsmith. "What is Agile Software Development". *Crosstalk*, http://www.stsc.hill.af.mil/crosstalk/2002/10/highsmith.html, Octubre de 2002.

[Hig02b] Jim Highsmith. *Agile Software Development Ecosystems*. Boston, Addison Wesley, 2002.

[Hock00] Dee Hock. *Birth of the chaordic age*. San Francisco, Berrett-Koehler, 2000.

[Hol95] John Holland. *Hidden Order: How Adaptation builds Complexity*. Addison Wesley, 1995.

[HT00] Andrew Hunt y David Thomas. *The Pragmatic Programmer*. Reading, Addison Wesley, 2000.

[Jac02] Ivar Jacobson. "A resounding Yes to Agile Processes – But also to more". *The Rational Edge*, Marzo de 2002.

[JAH01] Ron Jeffries, Ann Anderson, Chet Hendrikson. *Extreme Programming Installed*. Upper Saddle River, Addison-Wesley, 2001.

[Jef04] Ron Jeffries. *Extreme Programming adventures in C#*. Redmond, Microsoft Press, 2004.

[Kee03] Gerold Keefer. "Extreme Programming considered harmful for reliable software development 2.0". AVOCA GmbH, Documento público, 2001.

[Kru95] Philippe Kruchten. "The 4+1 View Model of Architecture." *IEEE Software* 12(6), pp. 42-50, Noviembre de 1995.

[Kru00] Philippe Kruchten. *The Rational Unified Process: An introduction*. Addison-Wesley, 2000.

[Kru01] Philippe Kruchten. "Agility with the RUP". *Cutter IT Journal*, 14(12), diciembre de 2001.

[Lar03] Craig Larman. *UML y patrones*. Madrid, Prentice Hall.

[Lar04] Craig Larman. *Agile & Iterative Development. A Manager's Guide*. Reading, Addison-Wesley 2004.

[McB02] Pete McBreen. *Questioning Extreme Programming*. Addison Wesley, 2002.

[McC93] Steve McConnell. *Code complete*. Redmond, Microsoft Press, 1993.

[McC96] Steve McConnell. *Rapid Development. Taming wild software schedules*. Redmond, Microsoft Press, 1996.

[McC02] Craig McCormick. "Programming Extremism". *Upgrade*, 3(29, http://www.upgrade-cepis.org/issues/2002/2/up3-2McCormick.pdf, Abril de 2002. (Original en *Communications of the ACM*, vol. 44, Junio de 2001).

[MDL87] Harlan Mills, Michael Dyer y Richard Linger. "Cleanroom software engineering", *IEEE Software*, Setiembre de 1987.

[Mel03] Stephen Mellor. "What's wrong with Agile?", Project Technology, http://www.projtech.com/pdfs/shows/2003/wwwa.pdf, 2003.

[Mil56] George Miller. "The magical number seven, plus or minus two: Some limits on our capacity for processing information". *Psychology Review*, 63, pp. 81-97, 1956.

[MS02a] Microsoft Solutions Framework Process Model, v. 3.1. http://www.microsoft.com/technet/itsolutions/techguide/msf/msfpm31.mspx, 2002.

[MS02b] Microsoft Solutions Framework – MSF Project Management Discipline. http://www.microsoft.com/technet/treeview/default.asp?url=/technet/itsolutions/tandp/innsol/default.asp, 2002.

[MS03] Microsoft Solutions Framework Version 3.0 Overview, http://www.microsoft.com/technet/itsolutions/techguide/msf/msfovrvw.mspx, 2003.

[MS04] Microsoft Patterns & Practices. "Organizaing patterns version 1.0.0", http://msdn.microsoft.com/architecture/patterns/02/default.aspx, 2004.

[MVD03] Tom Mens y Arie Van Deursen. “Refactoring: Emerging trends and open problems”. http://homepages.cwi.nl/~arie/papers/refactoring/reface03.pdf, Octubre de 2003.

[NA99] Joe Nandhakumar y David Avison. “The fiction of methodological development: a field study of information systems development”. *Information Technology & People*, 12(2), pp. 176-191, 1999.

[Nor04] Darrell Norton. “Lean Software Development Overview”, http://dotnetjunkies.com/WebLog/darrell.norton/articles/4306.aspx, 2004.

[NV04] James Newkirk y Alexei Vorontsov. *Test-driven development in Microsoft .Net*. Microsoft Press, 2004.

[Op92] William F. Opdyke. *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks*. Tesis de Doctorado, University of Illinois at Urbana-Champaign, 1992.

[Orr03] Ken Orr. “CMM versus Agile Development: Religious wars and software development”. Cutter Consortium, Executive Reports, 3(7), 2003.

[Pau01] Mark Paulk. “Extreme Programming from a CMM Perspective”. *IEEE Software*, 2001.

[Pau02] Mark Paulk. “Agile Methodologies from a CMM Perspective”. SEI-CMU, Stiembre de 2002.

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

[PMB04] PMBOK® Guide – 2000 Edition. Project Management Institute, http://www.pmi.org/prod/groups/public/documents/info/pp_pmbok2000welcome.asp, 2004.

[Pop01] Mary Poppendieck. “Lean Programming”. http://www.agilealliance.org/articles/articles/LeanProgramming.htm, 2001.

[PP03] Mary Poppendieck y Tom Poppendieck. *Lean Software Development: An Agile Toolkit for Software Development Managers*. Addison-Wesley, 2003.

[PT03] David Preedy y Paul Turner. “DSDM & MSF: A Joint Paper”. http://www.dsdm.org, 2003.

[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.

[Rac95] L. B. S. Racoon. “The Chaos Model and the Chaos Life Cycle”. *Software Engineering Notes*, 20(1), Enero de 1995.

[Rak01] Steven Rakitin. “Manifesto elicits cynicism”. *Computer*, pp. 4-7, Diciembre de 2001.

[Rie00] Dirk Riehle. “A comparison of the value systems of Adaptive Software Development and Extreme Programming: How methodologies may learn from

each other". http://www.riehle.org/computer-science/research/2000/xp-2000.pdf, 2000.

[Roy70] Winston Royce. "Managing the development of large software systems: Concepts and techniques". *Proceedings of WESCON*, Agosto de 1970.

[RS01] Bernhard Rumpe y Astrid Schröder. "Quantitative survey on Extreme Programming Projects". Informe, Universidad de Tecnología de Munich, 2001.

[SB02] Ken Schwaber y Mike Beedle. *Agile software development with Scrum*. Prentice-Hall, 2002.

[Sch95] Ken Schwaber. "The Scrum development process". *OOPSLA '95 Workshop on Business Object Design and Implementation*, Austin, 1995.

[SEI03] Capability Maturity Model® for Software (SW-CMM®). http://www.sei.cmu.edu/cmm/cmm.html, 2003.

[She97] Sarah Sheard. "The frameworks quagmire, a brief look". Software Productivy Consortium, NFP, http://www.software.org/quagmire/frampapr/FRAMPAPR.HTML, 1997.

[Shi03] Shine Technologies. "Agile Methodologies Survey Result". http://www.shinetech.com/ShineTechAgileSurvey2003-01-17.pdf, Enero de 2003.

[SmiS/f] John Smith. "A comparison of RUP® and XP". *Rational Software White Paper,* sin fecha.

[SN04] Will Stott y James Newkirk. "Test-driven C#: Improve the design and flexibility of your project with Extreme Programming techniques". *MSDN Magazine*, http://msdn.microsoft.com/msdnmag/issues/04/04/ExtremeProgramming/default.aspx. Abril de 2004.

[SR03] Matt Stephens y Doug Rosenberg. *Extreme Programming Refactored: The case against XP*. Apress, 2003.

[Sta97] Jennifer Stapleton. *Dynamic Systems Development Method – The method in practice*. Addison Wesley, 1997.

[Sut01] Jeff Sutherland. "Agile *can* scale: Inventing and reinventing SCRUM in five companies". *Cutter IT Journal*, Diciembre de 2001.

[TBT00] Duane Truex, Richard Baskerville y Julie Travis. "Amethodical Systems Development: The deferred meaning of systems development methods". *Accounting, Management and Information Technology*, 10, pp. 53-79, 2000.

[TJ02] Richard Turner y Apurva Jain. "Agile meets CMMI: Culture clash or common cause". En Don Wells y Laurie A. Williams (Eds.), *Proceedings of Extreme Programming and Agile Methods - XP/Agile Universe 2002*, *Lecture Notes in Computer Science 2418*, Springer Verlag, 2002.

[TN86] Hirotaka Takeuchi e Ikujiro Nonaka. "The new product development game". *Harvard Business Review*, pp. 137-146, Enero-Febrero de 1986.

[Tra02] Aron Trauring. “Software methodologies: The battle of the Gurus”. *Info-Tech White Papers*, 2002.

[Tur03] Paul Turner. “DSDM, Prince2 and Microsoft Solutions Framework”. DSDM Consortium, http://www.dsdm.org/kss/details.asp?fileid=274, Noviembre de 2003.

[Wie98] Karl Wiegers. “Read my lips: No new models!”. *IEEE Software*. Setiembre-Octubre de 1998.

[Wie99b] Karl Wiegers. “Molding the CMM to your organization”. *Software Development*, Mayo de 1998.

[WS95] Jane Wood y Denise Silver. *Joint Application Development*. 2ª edición, Nueva York, John Wiley & Sons, 1995.

[WTR91] James Womack, Daniel Jones y Daniel Roos. *The machine that changed the world: The story of Lean Production*. HarperBusiness, 1991.