METODOS DE DESARROLLO DE SOFTWARE: EL DESAFIO PENDIENTE DE LA ESTANDARIZACION

SOFTWARE DEVELOPMENT METHODOLOGIES: A DUEL PENDING FOR STANDARDIZATION

RICARDO A. GACITÚA BUSTOS

Depto. Sistemas de Información, Facultad de Ciencias Empresariales, Universidad del Bío-Bío, Avda. Collao 1202, Concepción, Chile, e-mail: rgacitua@ubiobio.cl

RESUMEN

Este artículo describe la evolución de los métodos de desarrollo de software. El foco está centrado en el desacuerdo en cómo debe crearse el software. El tema es como se considera el desarrollo de software: como un proceso de ingeniería o un proceso centrado en las personas. Se presenta el lenguaje de modelamiento unificado (UML) como una notación estándar del desarrollo de software. Actualmente es considerado como base para una metodología monumental (que incluye muchas reglas y prácticas) – RUP. Se menciona la reacción a las metodologías monumentales: los métodos ágiles. La cual es característica de un estado inmaduro del desarrollo de software como una disciplina. No solo hay desacuerdo en terminologías, enfoques y detalles de diferentes métodos, sino incluso en un esquema de clasificación común. La siguiente estructura está basada en la evolución de los principales conceptos y las distinciones claves que reflejan los cambios de paradigmas en la filosofía de métodos.

PALABRAS CLAVES: Métodos de desarrollo de software, metodología, UML, métodos ágiles.

ABSTRACT

This paper describes the evolution of software development methods. The focus is the disagreement about how programmers must make software. The issue is: How is software development considered as a engineering process or as a human centered process. Unified Model Language (UML) is considered as a standard notation for software development. At present, UML is considered as basis for a Monumental Methodology (which include a lot of rules and practices) - RUP. We mentioned the reactions to these Monumental methodologies: The Agile Methodologies. Which is characteristic of an immature state of software development as a discipline. There is not only disagreements in terminology, approach, and details of different methods, there is not even a commonly accepted classification scheme. The following structure is based on the evolution of the underlying concepts and the key distinctions that reflects the paradigmatic shifts the philosophy of methods.

KEYWORDS: Sofware development methodologies, methodology, UML, agiles methodologies.

Recibido: 05/05/2003 Aceptado: 24/10/2003

1. INTRODUCCION

El año 1994 un estudio dirigido por el Standish Group (The Standish Group, 2000; Jonson, 1994 y Jonson, 1995) –organización de prestigio mundial– analizó a más de 350 empresas norteamericanas y 8.000 proyectos de desarrollo de software. Dicho estudio arrojó que sólo el 16,2% de los proyectos en pequeñas compañías y el 9% en grandes compañías, finalizaban dentro de los costos y de los plazos establecidos, el 52,7 % de

los proyectos finalizaba excedido ampliamente en el presupuesto (sobre el 189 %) y con grandes retrasos de tiempo (sobre 122%), además de ofrecer menores características y funcionalidad de las que originalmente se había especificado. Finalmente, el 31,1% restante simplemente se cancelaba (81 billones de dólares botados). El estudio, en general, demostró la existencia de problemas serios de costos asociados al desarrollo de software por concepto de retrasos, presupuestos excedidos y, sobre todo, por la no-implementación de toda la funcionalidad especificada que, al menos, eso es lo que se espera de un proyecto de desarrollo de software. Lo anterior muestra que el desarrollo de software no sólo no ha logrado estándares de calidad aceptables –todo problema ocurrido en el sistema operativo Windows se resuelve reiniciando el computador y nadie cuestiona la falta de tolerancia a fallas del producto– sino que demostró empíricamente lo que todo programador de aplicaciones de software sabe: que el desarrollo de software es una actividad caótica a menudo caracterizada por la frase "programar y luego ajustar" (equivalente a construir un edificio y posteriormente, una vez construido, repararlo). En general el software es escrito sin un plan delineado para su construcción y el diseño de éste es "reparado" a partir de muchas decisiones de corto plazo, lo que demuestra que en general existe una falta de sistematización para construirlo. Esta forma de trabajo funciona bien cuando el sistema es pequeño pero, como el software debe ser ampliado y ajustado a nuevas condiciones, comienza a ser más difícil agregarle nuevas características. Además, los errores comienzan incrementalmente a prevalecer y a incrementar la dificultad de ajustarlo.

Para muchos usuarios no relacionados con el desarrollo de software la situación anterior no es visible, pues sólo se percibe la funcionalidad en el ámbito de interfaz y de resultados pero se desconoce cómo ha sido diseñado o cuál es la estructura a la que responde; mucho menos se conoce la cantidad de recursos utilizados, ni la innumerable cantidad de problemas que se ha debido enfrentar para presentar este producto de software. En este sentido, es común encontrar muchos productos de software carentes de diseño formal y entendido sólo por quien lo ha construido, sin documentación, escrito en lenguajes obsoletos, sin posibilidad de incorporar nuevos requisitos, con resultados no confiables, entre numerosos otros factores, que redundan finalmente en muchos problemas, tales como, por ejemplo, la imposibilidad de mantenerlo y adecuarlo a nuevos requisitos o condiciones de operación o asegurar la exactitud de los resultados, entre otros. Es decir, como se dice en Chile, un producto sostenido por "alambritos".

Quizás lo anterior pareciera ser poco relevante, pero si consideramos que hoy tenemos software en casi toda la actividad humana, desde simples dispositivos electrónicos hasta modernas naves aerospaciales y control de vuelo –pasando incluso por software de control de procesos de salud humana–, es fundamental hacer prevalecer la reducción de riesgos por concepto de pérdidas económicas, de fallas en determinadas áreas y, lo más importante, reducir el riesgo de pérdidas de vidas humanas.

Hemos vivido durante largo tiempo con este estilo de desarrollo de software, pero también se ha tenido una alternativa por un gran tiempo: métodos de desarrollo de software.

Un método, comúnmente llamado metodología, impone un proceso disciplinado sobre el desarrollo de software con el objetivo de hacer el desarrollo de software más predecible y eficiente. Por tanto, se plantea que un método define un camino reproducible para obtener resultados confiables.

Todas las actividades basadas en conocimiento utilizan métodos que varían en so-

fisticación y formalidad. Los cocineros se guían de recetas, los pilotos de avión a través de listas de chequeo antes de volar, los arquitectos utilizan planos y los músicos siguen reglas de composición. Similarmente un método de desarrollo de software describe cómo modelar y construir un sistema de software de una forma confiable y reproducible. En general, los métodos permiten la construcción de modelos desde elementos de modelado que constituyen los conceptos fundamentales para representar sistemas o fenómenos. Las escala de notas musicales es el elemento del modelo para la música. El enfoque de orientación a objeto para el desarrollo de software propone el equivalente a las notas –los objetos– para describir el software, para un enfoque funcional los elementos del modelado son las funciones del sistema.

Los métodos también definen una representación –a menudo gráfica– que permite facilitar la manipulación de modelos, y la comunicación e intercambio de información entre todas las partes involucradas. Una buena representación busca un balance entre la densidad de información y la legibilidad.

En relación con los elementos del modelo y su representación gráfica, un método define las reglas que describen la resolución de los diferentes puntos de vista, el orden de las tareas y la asignación de responsabilidades. Estas reglas definen un proceso que asegura armonía dentro de un grupo de elementos cooperativos y explican cómo debería ser usado el método. A medida que pasa el tiempo, los usuarios de un método desarrollan un cierto "know-how", así como de la forma en que debería ser usado. Este know-how, también llamado experiencia, no siempre está claramente formulado y no siempre es fácilmente traspasado.

El objetivo ideal, por tanto, es desarrollar un proceso de desarrollo detallado y con un fuerte énfasis en la planificación, inspirado por otras disciplinas de la ingeniería. Sin embargo, aun cuando han estado rondando por mucho tiempo, ellos no se destacan por su éxito ni por su popularidad. Sin embargo, en el caso del desarrollo de software esto es discutible. Un estudio reciente (Nandhakumar and Avison, 1999) argumenta que los métodos de desarrollo para los sistemas de información tradicional (IS) "son tratados principalmente como una ficción necesaria para presentar una imagen de control o proporcionar un status simbólico". El mismo estudio argumentaba que estas metodologías son muy mecánicas para ser usadas en detalle. Parnas y Clement (1986) habían mencionado los mismos argumentos anteriormente. El año 2000, Truex (Truex *et al.,* 2000) tomó una posición extrema diciendo que es posible que los métodos tradicionales son "meramente ideas insostenibles para hombres hipotéticos, proporcionando guías normadas para situaciones de desarrollo utópicas". Como resultado, los desarrolladores de software industrial han comenzado a ser escépticos acerca de las nuevas soluciones que son difíciles de contagiar y, además, no son utilizadas (Wiegers, 1998).

2. LA HISTORIA DE LOS METODOS DE DESARROLLO DE SOFTWARE

2.1. Hacia una disciplina de programación

2.1.1. La indisciplina

Quizás para muchos es desconocido el hecho que las primeras actividades de "programación" consistían en dar vuelta diversos switch de control eléctrico, sobre la consola de un computador enorme, lo que permitía configurar una secuencia numérica de instrucciones, tal cual se muestra en la figura 1. El uso de los lenguajes de "alto-nivel" (con instrucciones nemotécnicas) sólo comenzó en los años 50 (Fortran y Cobol fueron los más populares de éstos) y, el hecho que és-

tos existieran, fue considerado un gran logro. Los programadores no pensaban mucho acerca del "estilo de programación". Por su parte, dado el limitado tamaño y velocidad de los primeros computadores, el gran problema de los programadores era cómo escribir código que fuera pequeño (pocas líneas de código) y eficiente en el uso de recursos (uso del espacio de memoria, tiempo de respuesta, etc.). Los compiladores a menudo no eran muy buenos, así que los programadores se enorgullecían de conocer trucos de cómo burlar el compilador para generar el mejor código posible.

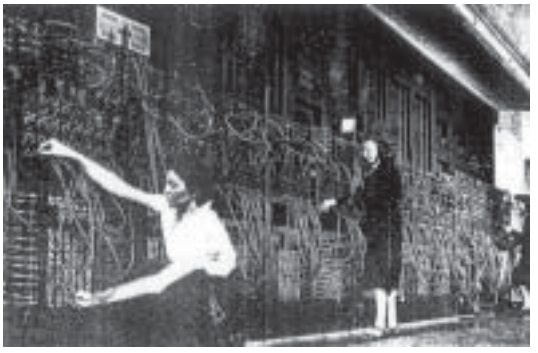

FIGURA 1. Programando en los años 40.

En 1968 los mini-computadores comenzaron a ser populares y, gracias al desarrollo de grandes compañías de computadores, los programadores comienzan a entender la Ley de Moore –"La densidad de los chips se dobla cada año"–, lo que había comenzado en 1964. Esto significaba que los computadores comenzaban a ser más grandes, más rápidos, el tamaño del programa y la velocidad dejaron de ser los principales criterios para medir la efectividad de los programas de computador. La aparición del popular sistema IBM /360 y la amplia variedad de lenguajes de programación de alto nivel implicó que los programas de computador eran durables y permanecían en el tiempo. La permanente baja del costo del hardware significó que el costo del desarrollo de software podría exceder el costo del hardware sobre el cual se ejecutaba. Por tanto, comienza a ponerse en evidencia un nuevo conjunto de criterios para medir el éxito del desarrollo de software. Estos criterios se mantienen incluso hoy. Un proyecto es juzgado como exitoso si el código producido:

– Tiene un costo relativamente bajo de desarrollo inicial.
– Es fácilmente mantenible.
– Es portable a un nuevo hardware.
– Cumple los requisitos del cliente, esto es, hace el trabajo que el cliente desea.
– Satisface criterios de calidad (seguridad, fiabilidad, etc.).

Mientras los lenguajes de alto nivel eran muy populares en 1968, no existían reglas para guiar a los programadores sobre el cómo escribir código que satisfaciera estos criterios. En efecto, en los primeros días, la programación se veía como un "arte" y los programadores se centraban en escribir código que fuera rápido y pequeño, y se aprendía el oficio de programador normalmente por prueba y error –tal cual ocurre aún en estos días–. En resumen, el mundo del software era virtualmente indisciplinado y muchos aprendices de entonces lo adoraban, pues parecería ser una actividad sumamente divertida.

2.1.2. Programación estructurada

En 1968 el profesor Edsger Dijkstra publicó una influyente carta al editor en la revista especializada *Communication of the Association for Computing Machinery (CACM)* llamada "Go to Statement Considered Harmful" ("La sentencia Go To se considera perjudicial"). Dijkstra era un académico quien veía a la ciencia de la computación como una rama de la matemática aplicada. Los programa-

dores de computadores, parecidos a otros ingenieros, deberían aplicar métodos matemáticos formales para crear programas efectivos y deben ser sometidos a pruebas formales. Dijkstra argumentaba que este constructor implementado en los lenguajes de programación de alto nivel ("go to": salto sin condición a otra sentencia en un programa de computador) era una abominación y llevaba a programas incorrectos, que no permitían pruebas formales. Las ideas de Dijkstra formaron las bases de la "programación estructurada", lo cual se constituyó en el primer método de desarrollo de software.

Básicamente, la programación estructurada se abocaba a:

– Desarrollar programas Top-Down (como opuesto a Bottom Up)
– Usar un conjunto específico de constructores formales de programación –secuencia, selección e iteración– (El "go to" era desterrado). Esto establecía que cualquier programa de computador podía ser construido sólo sobre la base de estos tres constructores.
– Seguir algunos pasos formales para descomponer grandes problemas.

Siguiendo esta metodología, se argumentaba, se aseguraría a los programadores satisfacer los criterios de éxito del software, enumerados anteriormente. Paralelamente, comienza a surgir un sinnúmero de conceptos que pretendían facilitar la conceptualización de los programas de computadores que se construían, entre ellos conceptos tales como: Programación modular, Refinamiento sucesivo, Information Hiding [19], entre otros. Como complemento, variadas son las notaciones que se proponen para modelar la estructura de los programas de computador (comúnmente denominada lógica de detalle) y que, en su mayoría, recogen la propuesta de Dijkstra. Entre las más conocidas se cuentan: Diagramas de flujos (Flowchart), Diagramas N-S, Pseucódigo, Tablas de decisión, Arboles de decisión, entre otros.

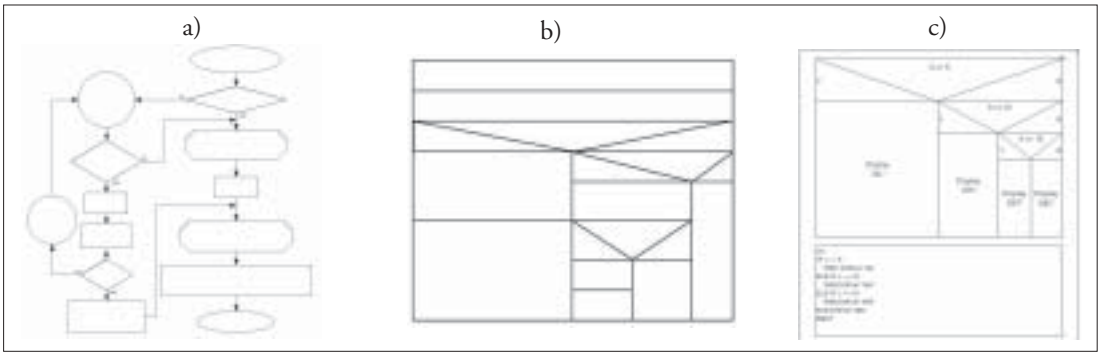

FIGURA 2. Ejemplo de notaciones para representar estructura de programas: a) Flowchart; b) Diagramas N-S, y c) Diagramas N-S con Pseudocódigo.

En 1971 el profesor Niklaus Wirth lanzó el lenguaje de programación Pascal, el cual no proporcionaba la sentencia "go to" y tenía las estructuras de control que implementaban el paradigma de la programación estructurada de Dijkstra. Posteriormente todos los lenguajes de programación habían sido influenciados por las ideas de Wirth y Dijkstra de crear programas bien estructurados y fáciles de leer. Sin embargo, fuertes detractores a las ideas de Dijkstra comienzan a surgir, y se origina un debate que se mantie-

ne hasta el día de hoy. Uno de los principales hitos fue marcado en 1974, cuando el profesor Donald Knuth, máximo opositor, publica un libro *The Art of Computer Programming,* que escribió programas estructurados que incluían sentencias "go to" en The ACM's Computing Survey. Lo cierto es que en ese tiempo se comenzó a hablar de una "guerra religiosa" entre los devotos de Dijkstra y los devotos de Knuth.

2.1.3. Diseño y análisis estructurado

La influencia de Dijkstra no paró con el gran debate del "go to". Las ideas de la programación estructurada se llevaron al ámbito de diseño y análisis estructurado. De hecho nació una nueva disciplina: Ingeniería de software. Los supuestos básicos detrás del análisis estructurado eran lograr obtener mayor control intelectual sobre el creciente aumento de la complejidad de los sistemas, utilizando como base el concepto de descomposición funcional propuesto por Hamilton y Zeldin (1979), sobre la base de los siguientes supuestos:

– Un modelo conceptual común para describir todos los problemas,
– Un conjunto de procedimientos para sugerir la dirección general del análisis y ordenado por pasos,
– Un conjunto de guías de acción o decisiones soportadas en heurísticas acerca del problema y su especificación y,
– Un conjunto de criterios para evaluar la calidad del producto.

Surgen, pues, varios métodos denominados genéricamente como métodos SA/SD (Structured Análisis / Structured Design), los que incluyen una variedad de notaciones para la especificación formal de software. Durante la fase de análisis, muchas notaciones, entre ellas diagramas de flujo de datos, especificación de procesos, diccionario de datos, diagramas de transición de estados y diagramas de entidad-relación, entre otros, son usados para describir lógicamente el sistema, tal cual se presenta en las figuras 3, 4 y 5.

La mayor parte de los métodos propuestos en esta época se denominaban, además, métodos funcionales debido a que eran inspirados directamente por la arquitectura de computadores (un dominio probado bien conocido de los programadores de la computación). La separación de datos y código, tal como existía físicamente en el hardware, fue trasladada a los métodos; por tanto los programadores tendían a pensar en términos de funciones de sistemas. En este sentido, diversas propuestas para modelar datos y modelar procesos y/o funciones son definidas; entre las principales se cuentan: SA/SD- Ed. Yourdon & DeMarco (Yourdon, 1989), SA/SD- Gane & Sarson (1979), JSD- Jackson Structured Development (Jackson, 1983), Information Engineering (Martín, 1990), Warnier (1974), entre otros. En la misma época, las exigencias de la defensa nacional y las aplicaciones aero-espaciales resultan en una alta demanda por software complejo y de misión crítica. Los desarrolladores responden creando una variedad de enfoques de dominio-específico a través de la adaptación del análisis estructurado para soportar especificaciones de sistemas de control embebidos (insertos dentro de componentes electrónicos), agregando notaciones para capturar el control de comportamiento. Estas variaciones son conocidas como Análisis / Real-Time (SA/RT). Uno de los trabajos realizados para la armada de USA, en el sistema de defensa de mísiles, produjo el método SREM (Software Requirement Engineering Method) (Alford, 1977), y muchas otras variaciones que han sido descritas por Ward and Mellor (1986) y Hatley & Pirbhai (1987).

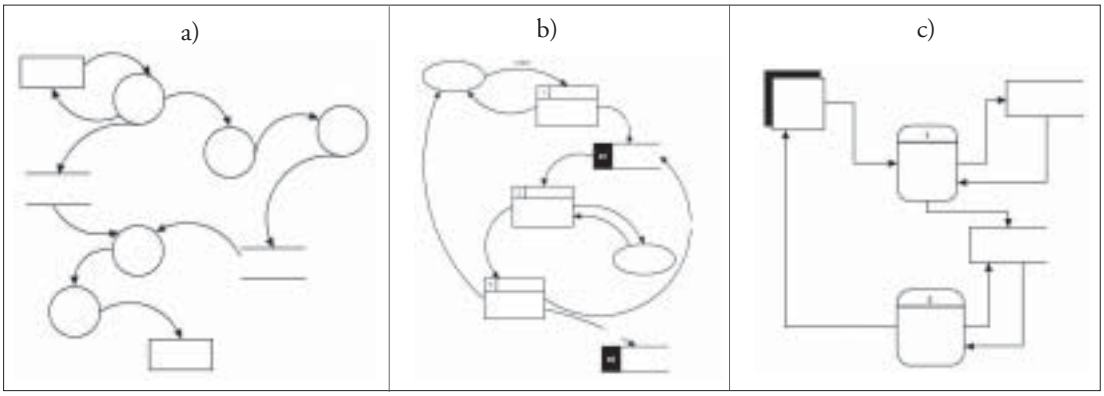

FIGURA 3. Ejemplo de notaciones para análisis estructurado –modelado de procesos–: a) Diagrama de flujo de datos - Método Yourdon & DeMarco; b) Diagrama de flujo de datos - Método SSADM; c) Diagrama de flujo de datos - Método Gane & Sarson.

Al igual que el modelamiento de procesos de transformación de datos, surge con fuerza la necesidad de modelar semánticamente los datos, para lo cual también surgen numerosas propuestas, entre ellas las de Chen (1976) y Codd (1976).

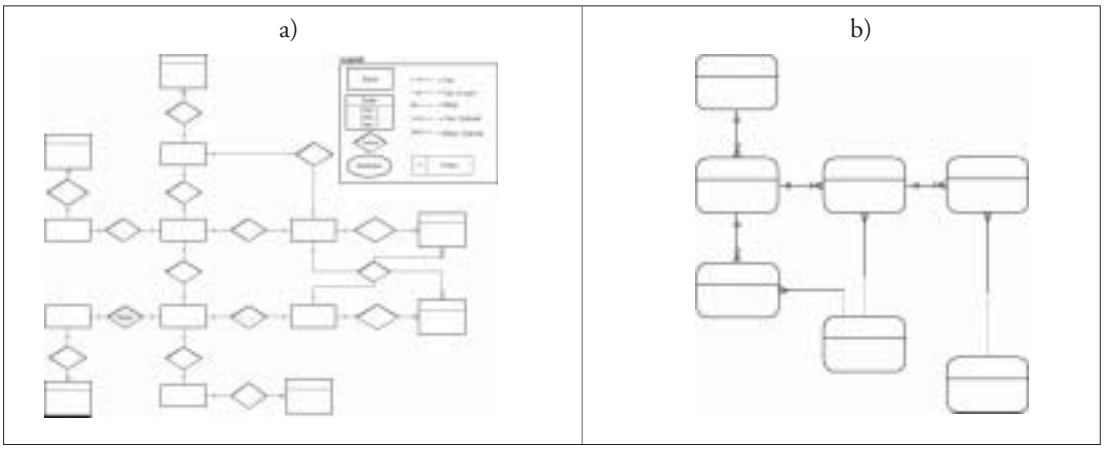

FIGURA 4. Ejemplo de notaciones para análisis estructurado - modelamiento semántico de datos-: a) Diagrama entidad-relación, Peter Chen; b) Diagrama modelo de datos, SSADM.

La necesidad de diseño surge del hecho de modelar la estructura de la solución que se había de proponer para un determinado problema. En la fase de diseño, se agregan detalles a los modelos de análisis y los diagramas de flujo de datos son convertidos en cartas de estructura y descripciones de código de lenguajes de programación.

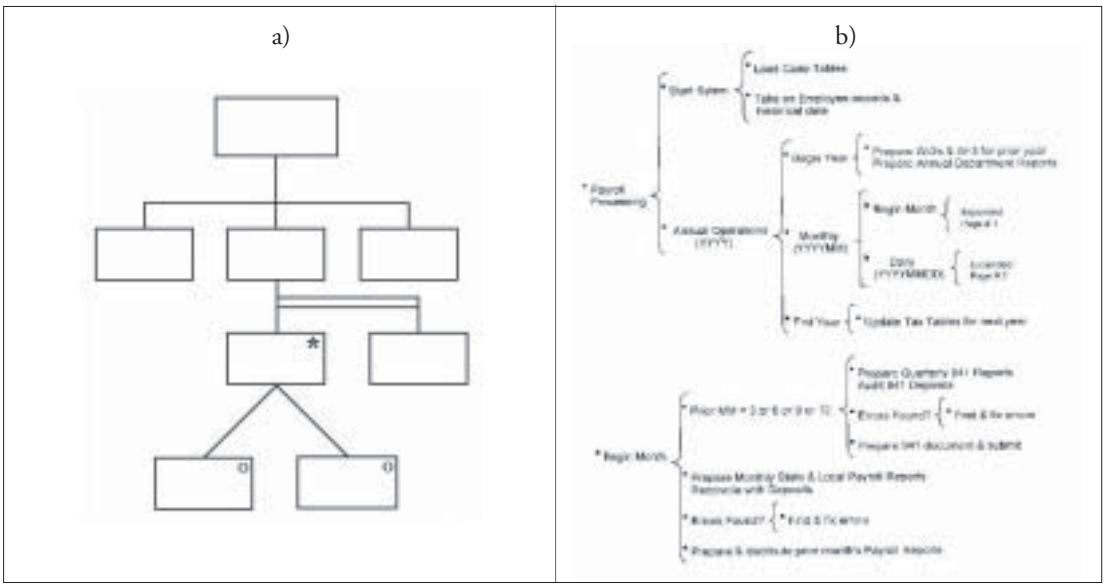

FIGURA 5. Ejemplo de notaciones para diseño estructurado: a) carta de estructura; b) método JSD, diagramas Warnier.

El desarrollo de software era visto como un proceso industrial. Muchos métodos fueron desarrollados y éstos proporcionaban una serie de guías, reglas detalladas y regulaciones para el proceso de desarrollo de software. Dijkstra volvió al mundo de la academia, pero el liderazgo de la "guerra religiosa" fue tomado por personas tales como Edward Yourdon, Peter Coad, James Martin, y Tom DeMarco, quienes hasta el día de hoy permanecen activos. Cada uno de ellos, a veces solos, a veces en parejas, establecieron valores (también conocidos como grupos de consultoría). Se publicaron biblias, muchos sermones fueron escritos y aclamados. Ellos prometían que, si incluso los programadores perezosos y buenos para nada, seguían su disciplina (como opuesto a los competidores charlatanes), entonces el desarrollo de software debería ser lejos menos costoso, el código debería ser fácil de mantener y portable, y los clientes deberían estar emocionados y deslumbrados con los resultados.

2.2. Brooks y el Mítico Hombre-Mes

En 1975, cuando la denominada "guerra de los métodos de Ingeniería de Software" estaba en la cima (denominada así, dada la enorme cantidad de métodos propuestos con notaciones y procesos diferentes, tratando de imponerse unos sobre otros), Fred Brooks publicó su famoso libro *The Mythical Man-Month* (El Mítico Hombre-Mes, en alusión directa a la unidad de medida de esfuerzo de Horas-Hombre, Hombre-Mes, etc.). Distinto al académico Dijkstra y sus seguidores consultores, Brook aprendía su lección acerca del desarrollo de software cuando él era el administrador del probablemente primer proyecto emprendido de desarrollo de software de gran escala –el desarrollo del sistema operativo IBM/360 (OS/360) a inicio de los años 60. El centro del mensaje de Brooks es que el desarrollo de software es un proceso centrado en el ser humano, y no una disciplina de ingeniería. Atendiendo a

los innumerables problemas y fracasos del desarrollo de software sumado a una baja productividad de los programadores y una mala calidad de los productos de software, Brooks acuñó la frase hasta hoy usada: "La crisis del software", aunque algunos plantean que se debería hablar de una "aflicción crónica", pues las crisis tienen un punto alto y luego se resuelven, a diferencia de lo que sucede con el desarrollo de software. Su libro dispuso la primera metodología de desarrollo de software útil, porque el método presionaba por administrar procesos de personas y no procesos de ingeniería.

2.2.1. Problemas claves de proyectos de desarrollo de software, según Brooks

Brooks, en un estilo humorístico, identifica los muchos problemas que presentan los proyectos de software. Cualquiera que haya estado involucrado en un proyecto de desarrollo de software los encontrará relevantes incluso hoy día.

1. *La mina de alquitrán:* "Los proyectos de software son quizás lo más intrincado y complejo de las cosas que hace la humanidad" (en términos de las distintas clases de partes que lo componen). Esto es más cierto hoy día en varias órdenes de magnitud, dado el aumento considerable de complejidad de las aplicaciones.
2. *El Mítico Hombre-Mes:* "Muchos proyectos fracasan más por pérdidas de tiempo calendario, que por todas las otras razones combinadas". Esta sentencia, que abre el libro de Brooks, es cierta hasta ahora. No se ha desarrollado hasta ahora una buena forma de estimar cuánto tiempo tomarán los proyectos de programación. La programación es un proceso creativo similar al arte y la música. Los proyectos de programación son tentativas comerciales. Esto lleva al corazón del problema: ¿Cómo administramos exitosamente un proceso humano creativo (como opuesto a procesos mecánicos y productivos)?
3. *La Ley de Brooks (la que exige ser validada por investigaciones):* "Agregar horas-hombre a un proyecto atrasado hace que se retrase más". Las personas y los tiempos no son intercambiables. Hay cierto procesos que no pueden ser apurados. Agregar más personas incrementa las intercomunicación y la sobrecarga en entrenamiento, tanto como interrumpir el avance.
4. *El Efecto del segundo sistema:* "El segundo es el sistema más peligroso de una persona que sobre diseña; la tendencia general es a sobre diseñarlo". En lenguaje moderno debería llamarse "featuritis".
5. *¿Por qué la Torre de Babel falla?:* "Desastres de calendario, inadaptación funcional y errores de sistema, todos crecen porque la mano izquierda no sabe que está haciendo la mano derecha. Los equipos definen supuestos". La comunicación es la parte más difícil de cualquier emprendimiento humano. Esto, por supuesto, es el centro del Mítico Hombre-Mes.
6. *Un paso adelante y un paso atrás:* "La entropía del sistema crece en el tiempo". Esto también es conocido como "Pedazos de tonterías". Reparar tiende a destruir estructuras e incrementar el desorden.

2.2.2. El Impacto del Mítico Hombre-Mes

Junto al crecimiento de los problemas en el desarrollo de software, el libro ofrece algunas soluciones importantes que, cuando son tomadas juntas, proporcionan un método universal para el desarrollo de software. Sin embargo, *The Mythical Man-Month*, a diferencia de la programación estructurada, no formó las bases de ninguna metodología promovida por algún gurú en los años 70 u 80.

Quizás, porque Brooks se orientaba a una carrera académica y no se definía como gurú para realizar consultorías. Mientras el libro fue de una importancia increíble, no fue hasta finales de los años 90 que las ideas de Brooks fueron tomadas por los proponentes de los denominados métodos ágiles. En 1995, en el aniversario veinte, se editó una nueva edición del libro. Brooks trató de actualizarlo, pero como él mismo admite, estando en la academia se alejó de los asuntos, problemas y soluciones del mundo real. Sin embargo, el corazón del libro es hasta hora relevante, muchos años después de que fue publicado.

2.3. 1975 al presente

2.3.1. Las herramientas CASE y la Orientación a Objeto

En los años 70 y principio de los 80, el análisis y diseño estructurado dominaba la mente de los desarrolladores de software. Como se mencionó anteriormente, varias sectas y subsectas se reunían alrededor de los gurús más conocidos. Un modelo estándar de desarrollo de "ciclo de vida" para sistemas de software, llamado "Modelo en Cascada" (Waterfall Model), fue el primer modelo de ciclo de vida documentado públicamente en 1970 e inmediatamente comenzó a ser el paradigma dominante (Este modelo presume un desarrollo lineal, que consiste en que el resultado de una fase se constituye en la entrada de la otra fase). Desgraciadamente, el análisis y diseño estructurado falló en cumplir la promesa de reducir los costos de desarrollo e incremento de la confiabilidad del software, debido a que la complejidad de los sistemas aumentaba y se orientaba principalmente a lenguajes de tercera generación. Muy pronto, muchos "herejes" atacaron el modelo de Cascada y desarrollaron toda clase de nuevos modelos de ciclo de vida, es decir, modelos acerca de los estadios o etapas de vida por las cuales atraviesa un producto de software y la estrategia para componer el producto final. Tal como se aprecia en la figura 6.

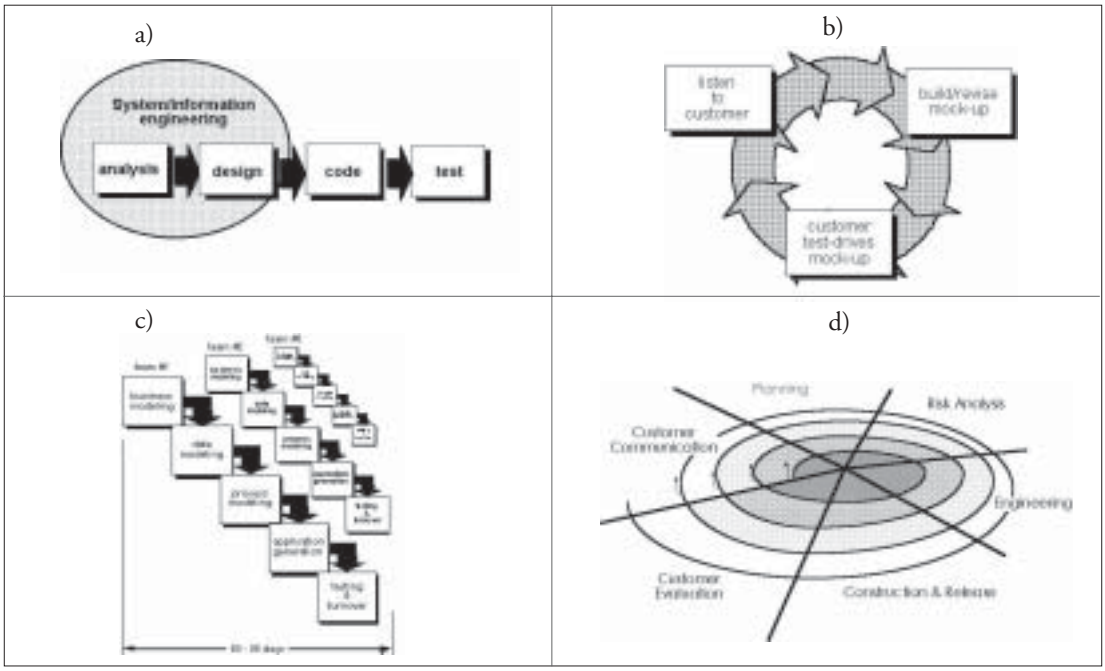

FIGURA 6. Ejemplo de diferentes ciclos de vida: a) Waterfall (en cascada); b) Prototipo c) RAD; d) Espiral.

Pese a esta disputa, dos ideas interesantes en los métodos de desarrollo de software emergen en los años 80.

La primera de estas ideas es la herramienta de Ayuda Asistida por Computer (CASE: Computer Aided Software Engineering – Ayuda Asistida por Computador a la Ingeniería de Software). Cada gurú tiene una compleja notación y un proceso, el cual deben usar los diseñadores para modelar sus sistemas de software. Muchos gurúes afirman que sus métodos deberían comenzar a retornar valor solamente cuando las herramientas de ayuda asistida por computador (CASE) se integren con el método. Las primeras herramientas CASE fueron programas de dibujos gráficos primitivos vinculado a una notación de un método específico.

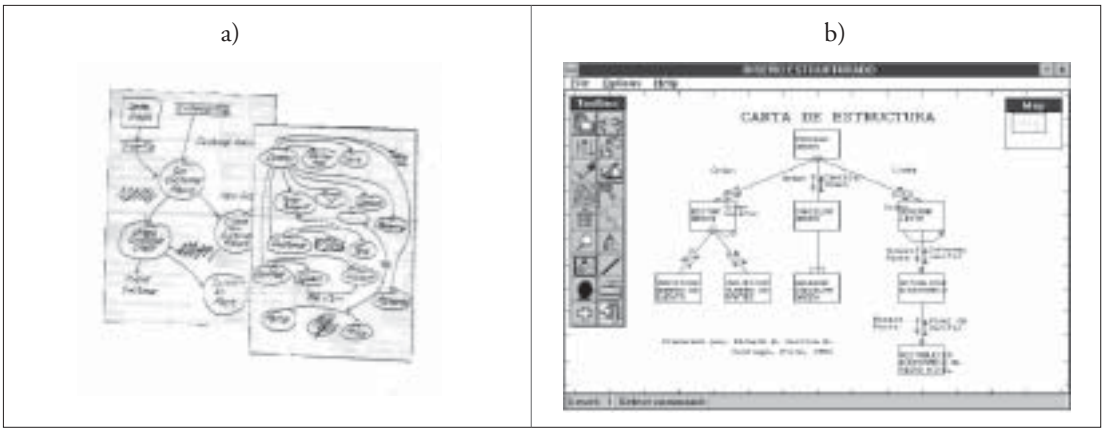

FIGURA 7. a) Diagramas manuales y b) diagramas elaborados en un CASE.

Pero, a la persona a quien realmente el CASE ayudó a despegar fue a Philippe Kahn, el fundador de Borland International. En 1983 Kahn lanzó un producto revolucionario: Un ambiente integrado de Desarrollo (IDE – Integration Development Environment) llamado Turbo Pascal. La idea de un IDE no era nueva –Emacs había estado rondando por un gran tiempo. Pero Turbo Pascal comenzó justo cuando el computador personal estaba despegando y comenzando a ampliar la plataforma de desarrollo de software. Turbo Pascal manejó mucha de las tareas tediosas y repetitivas del desarrollo de software, permitiendo al desarrollador concentrarse en los problemas del diseño de alto nivel. En efecto, Turbo Pascal fue el precursor de Visual Basic y otros ambientes de desarrollo integrado visuales, tales como PowerBuilder. Sin embargo, ninguna de estas herramientas está asociada con la idea tradicional de una "metodología de software" –guía de acciones y modelos para crear software–, sino que conforman una parte importante de las Toolkits (Cajas de Herramientas –software de ayuda al diseño que contiene apoyo a labores específicas de los desarrolladores de software). A diferencia de las IDES's, muchas de las herramientas CASE de los gurú han sido olvidadas.

La segunda idea interesante y útil es la idea del desarrollo de software Orientado a Objetos (OO). Uno de los padres de la OO es Alan Kay, el hombre quien dijo: "La mejor forma de predecir el futuro es inventarlo". En 1971 él comenzó desarrollando las ideas detrás del lenguaje de programación Smalltalk. Este fue desarrollado muy lejos en Xerox Palo Alto Research Center (PARC) durante los años 70 y 80. El objetivo clave de Kay era hacer el mundo del software mucho más cercano al mundo real. En el

mundo real, los objetos se comunican enviando mensajes hacia atrás, para allá y para acá. Cuando un objeto interactúa con otro, éste no tiene indicación del funcionamiento interno del otro objeto. Cada objeto conoce los protocolos de interacción y comunicación de los otros objetos. Sistemas muy complejos pueden ser construidos combinando objetos y permitiéndoles que interactúen naturalmente. El lenguaje Smalltalk proporcionaba una sofisticada implementación de estas ideas.

Los proponentes de OO argumentan que su amplia adopción permitirá una mayor flexibilidad en el desarrollo de software que con las técnicas estructuradas iniciales. Esto debería permitir el re-uso de software y, una vez que el objeto haya sido bien definido y creado, éste debería ser usado en diferentes sistemas. Las ideas detrás de la OO comienzan a prender en la comunidad de desarrollo de software. En los inicios de los años 80, el Departamento de Defensa de los Estados Unidos decidió que podría reducir millones de dólares y asegurar la confiabilidad del software imponiendo que todos los desarrollos de software sean hechos en un lenguaje OO. Cerca de 1983, el Departamento de Defensa gastó millones desarrollando una nueva generación de Pascal con aspectos de OO. Esto fue llamado ADA. En el mismo tiempo, Bjarne Stroustrup, de Bell Labs, creó una nueva generación del Lenguaje C con aspectos OO, llamado C++ pero, ninguno de estos lenguajes implementó completamente el poder de Smalltalk. Quizás por esta razón (aunque no la única), por diez años tuvo que combatir por la aceptación final. Esto permitió la creación de la Web, que llevó a la explosión en la adopción de los lenguajes y técnicas OO. Mientras Smalltalk nunca despegó, su sucesor Java y Phyton han sido extremadamente exitosos.

Al igual que en el caso anterior del enfoque estructurado, numerosas son las propuestas surgidas de métodos orientados a objeto y notaciones, y su evolución comenzó desde la programación, pasando por el diseño y finalmente abordando el análisis, tal cual se muestra en la figura 8.

FIGURA 8. Evolución de métodos.

Muchas son las propuestas realizadas, tanto de Diseño como de Análisis OO. En el caso del diseño, los métodos OO comparten los siguientes paso básicos de diseño, aunque los detalles varíen mucho:

– Se identifican los objetos y sus atributos.
– Se establece la visibilidad de cada objeto en relación con los demás objetos.
– Se establece la interfaz de cada objeto y el tratamiento de excepciones.
– Se realizan y comprueban los objetos.

Algunos ejemplos de métodos de diseño OO se encuentran en Booch (1994), Seidewitz y Stark (1986), Wasserman *et al.* (1990), entre otros.

Por su parte, el análisis OO evolucionó considerando dos fuentes: El modelamiento de semántico de datos y el diseño OO y, con el transcurrir del tiempo, surgen muchas propuestas orientadas al Análisis OO, y muchos gurús se incorporan a este nuevo campo de batalla, entre los que se cuentan los trabajos

de Rumbaugh-OMT (Rumbaugh *et al.*, 1991), Coad y Yourdon (1990), Shlaer y Mellor (1988), Booch (1986, 1994 y 1999), Jacobson *et al.* (1992), Reenskaug-OORASS (Reenskaug *et al.,* 1991), Martín y O'dell (1992), entre muchos otros. Como consecuencia de lo anterior, numerosas son también las propuestas de notaciones para modelar distintos componentes de un sistema OO, tal como lo muestra la figura 9.

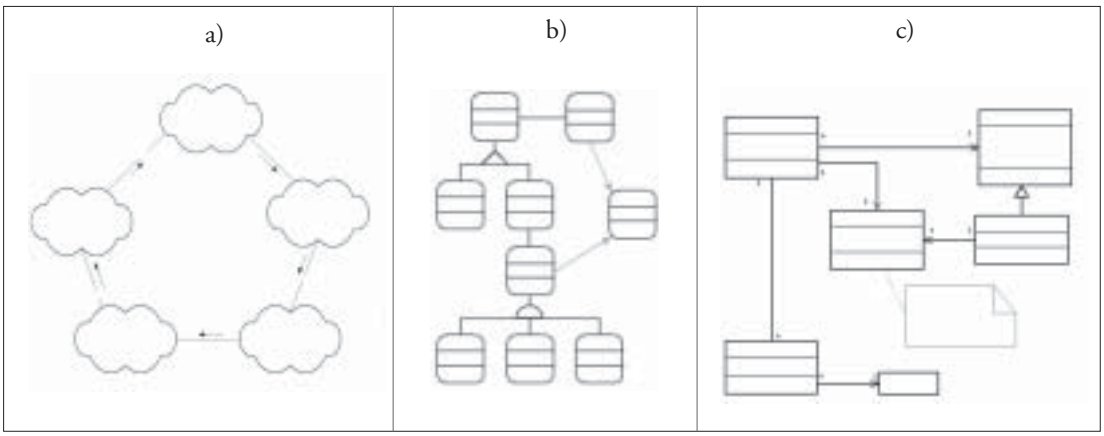

FIGURA 9. Notaciones de diferentes Métodos OO para representar "clases" de objetos: a) Método de Booch; b) Método de Coad & Yourdon; c) Método OMT, Rumbaught.

Aun cuando existen muchas propuestas de métodos para el desarrollo de software OO, en la práctica la situación es más compleja –los métodos a menudo no cubren todo el ciclo de vida, dado que muchas propuestas son orientadas sólo a diseño y especificación OO y muchas otras se orientan al Análisis OO. Más aún, no se adecuan a todos los dominios específicos requeridos tal como, por ejemplo, el desarrollo para Internet. Por tanto, en la práctica, los métodos están siendo mezclados y entrelazados: un método A es usado para análisis seguido de un método B, el cual es usado para el diseño. A lo largo del desarrollo, se utiliza un único paradigma –el enfoque funcional o el enfoque orientado a objeto–, esta división resulta razonable. Una muestra de ello lo constituye el método FUSION (Coleman *et al.*, 1993), desarrollado por el Object-Oriented Design Group de los Laboratorios de Hewlett-Packard, Bristol. Este método, por ejemplo, se construye sobre la base de la primera generación de métodos, incluyendo Booch, OMT y CRC y provee una ruta completa desde la definición de requisitos hasta la implementación en un lenguaje de programación.

Durante mucho tiempo, en las empresas se ha desarrollado un fuerte conocimiento de análisis funcional y de métodos de modelamiento semántico de datos (por ejemplo modelos entidad-relación). Muchos desarrolladores de software se inclinaban por seguir una fase de análisis funcional-estructurado seguido de una fase de diseño orientado a objeto. Aun cuando puede ser no entendible, el hecho de mezclar paradigmas (funcional, OO, etc.) es claramente menos razonable y aconsejable, debido al serio inconveniente asociado al cambio de paradigma: moverse de un enfoque funcional a un enfoque orientado a objeto requiere un traslado de los elementos de modelo funcional a elementos de modelo de objetos, lo cual está lejos de ser natural o ser una ventaja. Más aún, no hay

una relación directa entre los dos conjuntos de elementos y es necesario romper los elementos de modelo desde uno de los enfoques para crear modelos de fragmentos de elementos que puedan ser usado por otros (Por ejemplo, diseñar con un enfoque estructurado y posteriormente programar con el paradigma OO o modelar clases que incluye atributos y métodos y posteriormente implementar en una Base de Datos Relacional).

Este cambio de paradigma, centrado en el medio del esfuerzo de desarrollo, puede impedir enormemente la navegación desde las sentencias de requisitos obtenidos tempranamente en las fases de análisis a la satisfacción de aquellos requisitos en la fase de diseño. Más aún, un diseño orientado a objeto obtenido después de la translación puede llevar a perder abstracción, y limitar el encapsulamiento de los objetos de bajo nivel disponibles en la implementación y ambiente de ejecución, todo lo cual implica un gran acuerdo de esfuerzo en orden a obtener resultados que no son muy satisfactorios.

La combinación de un enfoque funcional y un enfoque orientado a objeto para diseño e implementación, como equivalente a un moderno método orientado a objeto que cubre todo el ciclo de desarrollo de un software, no necesita existir hoy día, aun cuando existen numerosos trabajos con intención de establecer una integración, tal como se muestra en Ward (1989).

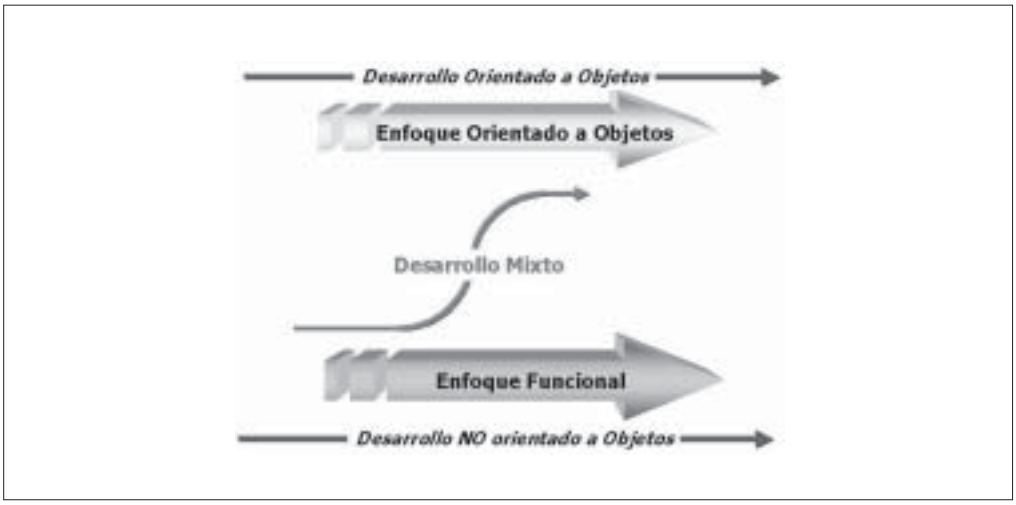

FIGURA 10. Combinación de enfoques para modelamiento e implementación de sistemas.

Durante la pasada década, las aplicaciones orientadas a objeto –que fueron desde el análisis de requisitos a la implementación– han sido desarrolladas en todos los sectores de programación. La experiencia adquirida en estos proyectos ha ayudado al entendimiento de cómo unir las muchas y variadas actividades requeridas para soportar un enfoque completo orientado a objetos.

2.3.2. Lenguaje de Modelamiento Unificado (UML)

Los gurús rápidamente recogieron la idea de la Orientación a Objeto. Quizás, porque ellos entendieron que lo conceptual cambiaba desde la programación secuencial al enfoque OO y sería difícil y doloroso el cambio, pero que el entrenamiento de las personas debe-

ría ser una tendencia útil y lucrativa. El número de métodos orientados a objeto se incrementó desde menos de 10 a más de 50 durante el período entre 1989 y 1994, por esta razón se hablaba la guerra de los métodos. Muchos usuarios de estos métodos tenían dificultad para encontrar algún lenguaje de modelamiento que satisficiera completamente sus necesidades.

Una gran concentración de ideas críticas comenzó a formarse a mediados de los 90's, cuando Grady Booch (Rational Software Corporation), Ivar Jacobson (Objectory) y James Rumbaugh (General Electric), comenzaron a adoptar ideas de cada uno de los otros métodos, los cuales colectivamente comenzaron a reconocerse como líderes de los métodos orientados a objeto en el ámbito mundial. Así es como los principales autores de los métodos de Booch, OOSE y OMT, se motivaron a crear un lenguaje de modelamiento unificado, atendiendo a tres razones:

– Primero, los métodos estaban evolucionando realmente hacia el otro independientemente. Esto crea la sensación de que la evolución, juntos más que aparte, eliminaría la potencial diferencia que podría llevar a confundir a los usuarios.
– Segundo, unificado los métodos, se podría proporcionar alguna estabilidad al mercado de la orientación a objeto, permitiendo a los proyectos utilizar un lenguaje de modelamiento maduro y llevando a las herramientas de construcción a focalizarse sobre las características de salida más útiles.
– Tercero, se espera que la colaboración podría producir mejoramiento en los tres métodos iniciales, ayudando a recoger las lecciones aprendidas y resolver los problemas que ninguno de los métodos había manejado previamente bien.

El esfuerzo de UML partió oficialmente en octubre de 1994, cuando Rumbaugh se unió a Booch de Rational. El proyecto inicial se concentró en la unificación de los métodos de Booch y OMT. La versión draft 0.8 del Método Unificado (así fue llamado) se lanzó en octubre de 1995. Alrededor del mismo tiempo, Jacobson se unía a Rational y el ámbito del proyecto UML fue expandido para incorporar OOSE. El esfuerzo resultó en el nuevo lanzamiento de UML versión 0.9 en junio de 1996. Durante 1996, los autores invitaron y recibieron retroalimentación de la comunidad de ingeniería de software. Durante este tiempo comienza a ser claro que muchas organizaciones veían a UML como una estrategia para sus negocios. Se estableció un consorcio UML con varias organizaciones destinando recursos para trabajar hacia una definición de UML más completa y fuerte. Estos socios contribuyeron a la definición de UML 1.0, incluyendo Digital Equipment Corporation, Hewlett-Packard, i-Logic, Intellicorp y James Martín y Cia., IBM, ICON Computing, MCI Systemhouse, Microsoft, Oracle, Rational, Texas Instruments y Unisys. Esta colaboración, resultó en UML 1.0, un lenguaje de modelamiento que estaba bien definido, explícito y poderoso, y aplicable a una amplia variedad de dominios de problemas. En respuesta a su petición para propuestas, por parte de Object Management Group (OMG), UML 1.0 fue ofrecido para la estandarización en enero de 1997. Entre enero de 1997 y julio de 1997, el grupo original de socios fue ampliado para incluir virtualmente a todos los socios y contribuyentes de la oferta original de OMG, incluyendo Andersen Consulting, Ericsson, ObjectTime Limited, Platinum Technology, Ptech, Reicht Techno- logies, Softeam, Sterling Software y Taskon. Una forzada tarea de semántica fue definida, liderada por Cris Kobryn de MCI System-house y administrado por Ed Eykholt de Rational, para formalizar la especificación

de UML e integrar el UML con otros esfuerzos de estandarización. Una versión revisada de la versión UML 1.0 fue ofrecida a la estandarización de OMG en julio de 1997. En septiembre de 1997 esta versión fue aceptada por "The OMG Análisis and Design Task Force" (ADTF) y el OMG Architecture Board, y lo pusieron para votación de todos los miembros de OMG. La versión UML 1.0, la cual es la más descrita, fue aceptada por la OMG en noviembre de 1997.

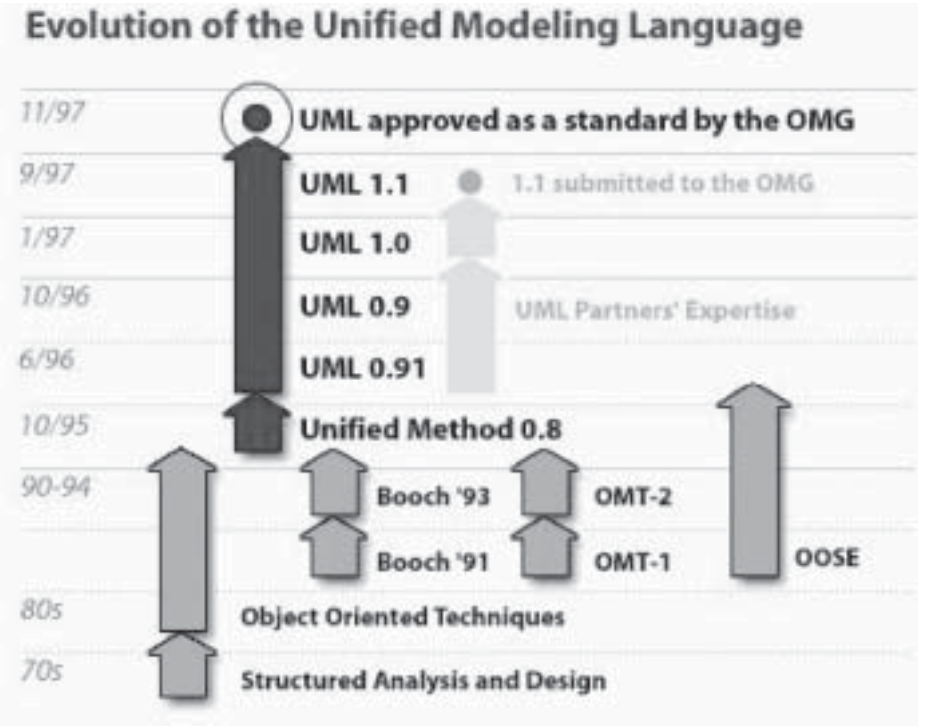

FIGURA 11. Evolución de UML (Fuente: Rational).

A pesar de este acuerdo, y algún interés comercial de muchos miembros del consorcio sobre una *notación* estandarizada para modelamiento de sistemas de software, es importante notar que los antiguos gurús tenían un mensaje centrado en lo humano de gran importancia que se perdió en el fragor de la guerra.

Uno de los principales argumentos para la programación estructurada es que el desarrollo de software está centrado principalmente sobre la comunicación entre personas, y no tanto en la comunicación de las personas a las máquinas. Los programas necesitan ser bien escritos y bien organizados, de modo tal que los desarrolladores de software puedan comunicarse más fácilmente entre ellos mismos y con sus clientes. Estas ideas tan buenas, son hasta ahora relevantes.

Lo cierto es que la guerra de los métodos no finalizó con la creación de UML. Incluso con una notación unificada, hay muchos nuevos métodos alternativos propuestos para modelamiento de sistemas de software que incluyen, incluso a UML como notación. Pero debemos destacar, sin embargo, que UML es un gran logro –UML es una notación no es un método–. Desde entonces se

considera un primer estándar. Sirve como un lenguaje útil y universal que pueda ser usado para comunicar diseños e ideas acerca de sistemas de software, no importa que método sea escogido para usarlo. Incluso, la principal organización que promueve UML, Rational, ha formulado un Proceso de Desarrollo soportado en UML (Ver figura 11), llamado RUP (Rational Unified Process), en el que se integran herramientas CASE y herramientas de asistencia al proceso riguroso, tales como aseguramiento de calidad, administración de requisitos entre otros, aun cuando se establece como premisa que es posible adecuar las actividades y reglas como mejor acomode a los desarrolladores, es decir, es adaptable al gusto del desarrollador. Esto pareciera ser un problema, pues en ocasiones se cree necesario establecer con claridad qué hacer, que disponer de una enorme cantidad de opciones para decidir que realizar.

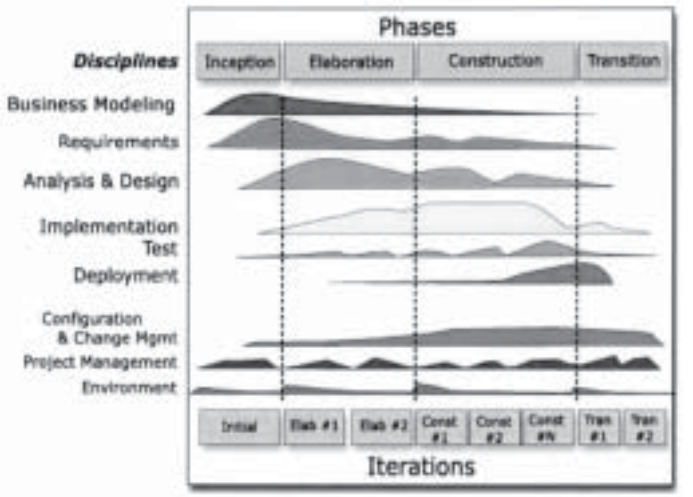

FIGURA 12. Proceso RUP – Rational Unified Process.

2.3.3. Una nueva alternativa: Los Métodos Agiles

Una de las principales críticas realizadas a los métodos propuestos hasta ahora es que son burocráticos, es decir, hay tantas cosas que hacer que el desarrollo de software se vuelve lento. Más aún, estos métodos han sido llamados "Heavy Methodologies" (Métodos pesados) o "Monumental Methodologies" (Métodos monumentales). Como una reacción a estas metodologías, a finales de los años 90, ha surgido un nuevo grupo de metodologías sustentadas en las antiguas ideas de Brooks, las que fueron conocidas por un tiempo como "lightweight methodologies", pero ahora el término aceptado es "Agiles Methodologies" (Métodos ágiles).

En este sentido, la guerra de los métodos de software ha venido ha completar un círculo peligroso. Mientras el manifiesto de Dijkstra llamaba por "más" disciplina en el desarrollo de software, los principales disidentes de los "tres amigos" (como se les llama a los creadores de UML) han lanzado un manifiesto que llama por "menos", llamado "Manifesto for Agile Software Development" (Manifiesto por el Desarrollo de Software Agil). El que se sustenta en los siguientes postulados:

– *Los individuos y sus interacciones,* son más importantes que los procesos y herramientas.
– *Un software que funcione,* es más importante que una abundante documentación.
– *La colaboración con los clientes,* es más importante que la negociación de contratos.
– *La respuesta ante el cambio,* es más importante que el seguimiento de un plan.

Hay varios paralelos entre las dos escuelas. El autodenominado estilo ágil ha comenzado a presentar una serie de libros en *The Agile Software Development Series*, similar a lo propuesto por los tres amigos, *Object Technology Series*. Los gurús ágiles han creado su propia empresa de consultoría y entrenamiento orientando sus principales esfuerzos hacia tomar ventajas de "la nueva economía", es decir, el desarrollo sobre Internet. Es destacable, sin duda, que la principal contribución de los métodos ágiles es que ellos están recogiendo ampliamente el enfoque centrado en la persona propuesta por Brooks. Más aún, están agregando elementos adicionales para el entendimiento de la problemática humana detrás del desarrollo de software en general. Highsmith (2002) utiliza la palabra "ecosistema" en vez de método o metodología para indicar que el desarrollo de software trata acerca de personas, sus interacciones y adaptaciones a un ambiente amplio y no sobre procesos de ingeniería. Algunos importantes ejemplos de métodos ágiles son: XP (Extreme Programming), Cockburn's Crystal Family, Open Source, Highsmith's Adaptive Software Development, Scrum, Feature Driven Development y DSDM (Dynamic System Development Method). Se debe destacar que la carencia de documentación es un síntoma de dos profundas diferencias, según Martín Fowler, uno de los líderes de estas propuestas:

1. *Los métodos ágiles son más adaptativos que predictivos*. Los métodos monumentales tienden a tratar de planear una gran parte del proceso de desarrollo de software, con gran detalle por un gran lapso de tiempo. Esto está bien, hasta que las cosas cambian. Así es que su naturaleza es resistir el cambio. Los métodos ágiles, en cambio, reciben los cambios. Ellos tratan de procesar y hacer propicio los cambios.
2. *Los métodos ágiles son más orientados a las personas que al proceso*. Ellos explícitamente manifiestan que se ha de "tratar" con el trabajo y la naturaleza de las personas más que contra ellos, y enfatizan que el desarrollo de software debería ser una actividad entretenida.

Muchas personas apelan a estas metodologías ágiles como reacción a las metodologías burocráticas o monumentales. Estos nuevos métodos intentan establecer un justo equilibro entre "sin proceso" y "demasiado proceso", proporcionando sólo el proceso suficiente para obtener un retorno razonable. De muchas formas, estos métodos están más orientados al código: siguiendo la idea de que plantean que la parte clave de la documentación es el código fuente.

CONCLUSIONES

Desde que Dijkstra planteó que el desarrollo de software debería estar centrado fuertemente en las matemáticas para producir productos confiables y con costos predecibles, muchos esfuerzos se han realizado para definir un proceso de desarrollo de software en forma disciplinada y rigurosa. Importantes organizaciones, tales como Motorola, NASA, entre otros, basan su desarrollo en tales prácticas disciplinadas. Por su parte, la innumerable cantidad de métodos de desarrollo de software propuestos, algunos quedando en el olvido y los menos –en su afán de adecuarse a los cambios de enfoques, a la innumerable cantidad de situaciones que

deben enfrentar y a las diversas áreas de aplicación en que se deben desenvolver– han incorporado una gran cantidad de reglas, notaciones, prácticas y documentos que requieren mucha disciplina y tiempo para seguirla correctamente. Esto ha llevado a ser definidas como Metodologías monumentales o Heavy Methodologies. Sin embargo, son muchas las áreas en las cuales este tipo de desarrollo no se condice con las exigencias del problema. Desarrollar, por ejemplo, aplicaciones para la Web establece como premisa que el tiempo es corto, los requisitos cambiantes, se requieren equipos multidisciplinarios, entre otros aspectos. Por tanto, es difícil planificar dada la innumerable cantidad de cambios y se requiere una interacción muy fuerte entre el equipo de desarrolladores.

Por otra parte, pese al logro de la comunidad internacional de ingeniería de software de aceptar finalmente una notación estándar como lo es UML y su posterior inserción en un gran proceso de desarrollo de software (Rational Unified Process) –aun cuando existan intereses económicos detrás–, existen muchos detractores que se oponen a la aceptación de dichos métodos monumentales sobre la base de que hacen más burocrático y lento el desarrollo software y que este tipo de desarrollo es un proceso centrado en las personas y en sus interrelaciones, y no entre las personas y las máquinas, por tanto, no es equivalente a un proceso de ingeniería tradicional.

Sin duda, ambos enfoques tienen su área de aplicación y sus exigencias. Quizás, por el hecho de que los productos de software se requieran en áreas tan diferentes, distintos tipos de requisitos, distinta volatilidad de requisitos, diferentes niveles de riesgos, diversos clientes, diferentes niveles de calidad, entre muchos otros aspectos, hace que ambos tipos de enfoques metodológicos tengan su validez en el contexto en que se usan. Sumado a lo anterior, se debe incluir como "método" ampliamente usado, el estilo "hacker" o programación por prueba y error, sin disciplina ni sistematización alguna, tal como lo definen algunos autores. Por lo tanto, se ve difícil, por el momento, tener reconciliadas estas propuestas y disponer de estandarización de métodos de desarrollo de software, menos aún cuando siguen apareciendo nuevas propuestas de métodos. Por lo demás, nos vemos obligado a sumar esta preocupación a una de las tantas paradojas de este mundo actual, entre ellas: "Saber de todo, pero ser especialista en algo", y, en el caso de los métodos de desarrollo de software: "Servir para todas las áreas de aplicación y en todos los casos, pero simple y fácil de usar".

REFERENCIAS

ALFORD, M. (1977) "A Requirement Engineering Methodology for Real Time Processing Requirements," IEEE Trans. Software Eng., Vol. 3, Nº 1, Jan., pp. 60-69.

BOOCH, G. (1994) Object-Oriented Análisis and Design, Benjamin/Cumming, Redwood City, Calif.

BOOCH, G. (1986) "Object-Oriented development". IEEE Trans. On Software Eng., Vol. SE-12(2), 211-221.

CHEN, P. (1976) The Entity-Relationship model: toward a unified view of data. ACM Trans. On DataBase System, 1(1), 9-36.

COAD, P. and YOURDON, E. (1990) Object Oriented Analysis, Prentice-Hall, Englewood Cliffs, N.J.

CODD, E.F. (1976) A Relational Model of Data for Large Share Data Banks. Comm. ACM, 13 (6), 377-387.

COLEMAN, D., ARNOLD P., BODOFF, C., DOLLIN, C., HAYES, F. and JEREMAES, P. (1993) Object - Oriented Development: The Fusion Method. Prentice-Hall.

GANE, C. and SARSON, T. (1979) Structured System Anlysis. Prentice-Hall, New Jersey.

HAMILTON, M. and ZELDIN, S. (1979) "Higher Orde Software - A Methodology for Defining Software", IEEE Trans. Software Eng., Vol. 2, Nº 1, Jan., pp. 9-32.

HATLEY, D. and PIRBHAI, I. (1987) Strategies for Real-Time Specification. Dorset House, New York, N.Y.

HIGHSMITH, J. (2002) Agile Software Ecosystem, Adisson-Wesley, IBSN 0-201-76043-6.

JACKSON, M. (1983) System Development. Englewood Cliffs, New Jersey: Prentice Hall International.

JACOBSON, I., CHRISTERSON, M., JONSON P. and OVERGARD, G. (1992) Object-Oriented Software Engineering, Addison-Wesley, Reading, M.A.

JONSON, J. (1994) "Failure is Its Own Reward", Application Development Trends, Vol 1., Nº 9, August, p. 70.

JONSON, J. (1995) "CHAOS: The Dollar Drain of IT Project Failure", Application Development Trends, Vol. 2, Nº 1, January, pp. 41-47.

MARTÍN, J. (1990) Information Engineering, Vol. 1, 2 y 3. PTR Prentice Hall, Englewood Cliffs, New Jersey 07632.

MARTÍN J. and O'DELL J. (1992) Object –Oriented Análisis and Design. Englewood Cliffs, NJ, Prentice-Hall.

NANDHAKUMAR, J. and AVISON, J. (1999). The fiction of methodological development: a field study of information system development. Information Technology & People 12(2): 176-191.

PARNAS, D. (1972) "On the Criteria to be Used in Decomposing Systems into Modules", Comm. ACM, Vol. 15, Nº 12, Dec., pp. 1.053-1.058.

PARNAS, D. L. and CLEMENTS, P.C. (1986) A rational design process: How and why to fake it. IEEE Transactions on Software Engineering 12(2): 251-257.

REENSKAUG, T., ANDERSON, E.P. and BERRE, A.J. (1991) Seamless support for the creation and maintenance of object - oriented system. Taskon A/S Gandstadalteen 21, N-0371 Oslo 3.

RUMBAUGH, J. BLAHA, M., LORENSEN, W., HEDI, F. and PREMERLANJ, W. (1991) Object Oriented Modeling and Design, Prentice-Hall, Englewood Cliffs, N.J.,

SEIDEWITZ, E. and STARK, M. (1986) General Object-Oriented Software Development, Software Engineering Letters, 86-102.

SHLAER, S. and MELLOR S. (1988) Object-Oriented System Analysis: Modeling the World in Data, Prentice-Hall, Englewood, Cliffs, N.J.

THE STANDISH GROUP (2000) The CHAOS Report. Dennis, MA: The Standish Group,1994

TRUEX, D. P., BASKERVILLE, R. and TRAVIS, J. A (2000) Methodical systems development: The deferred meaning of system development methods. Accounting, Management and Information Technology 10: 53-79.

WARD, P. and MELLOR S. (1986) Structured Development for Real-Time, Vols. 1, 2 y 3, Prentice-Hall, Englewood Cliffs, N.J.

WARD, P. (1989) How integrated object-orientation with Structured Analysis and Design. IEEE Software, 6 March.

WARNIER, J. (1974) Logical Construction of Programs. New York: Van Nostrand Reinhold.

WASSERMAN A.L., PIRCHER P.A. and MULLER R.J. (1990) The Object-Oriented Structured design notation for software design representation, IEEE Computer, 50-62, March.

WIEGERS, K.E. (1998) Read my lips: No new models. IEEE Software 15(5): 10-13.

YOURDON, E. (1989) Modern Structured Analysis. Englewood Cliffs, New Yersey: Yourdon Press.