Desarrollo Rápido de Aplicaciones: Guía rápida para el programador TOP

*José Miguel Santibáñez Allendes**

Resumen

Fuera de las aulas universitarias, y sin que se le haya dado la correspondiente importancia, los alumnos de carreras de Informática (Ingeniería en Informática, Análisis de Sistemas) están llevando a la práctica un modelo propio de desarrollo de aplicaciones. El propósito del presente artículo es realizar un estudio preliminar de esta tendencia que ha sido bautizada como "TOP", por la alegórica sigla en inglés de Table Oriented Programming.

Esta tendencia, que puede resumirse en la necesidad de diseñar interfases de usuario, basándose en las estructuras de datos, y diseñar las estructuras de datos con fuerte influencia de la interfaz de datos, es analizada desde una perspectiva histórica (que explica cómo se llega a ello), práctica (el diagnóstico preliminar de la situación actual) y técnica (sus riesgos y beneficios).

Se debe señalar que esta tendencia no ocurre sólo en el ámbito universitario (aunque allí se le considera de mayor gravedad); también es observable en el ámbito profesional, particularmente en el nivel de las PyMEs.

* Ingeniero Civil en Informática, Universidad de Santiago de Chile. Académico UCINF.

1. Introducción

Desde hace un par de años, se está imponiendo en los alumnos de las carreras de Informática una particular manera de enfrentar el desarrollo de los sistemas de información.

A falta de mejor nombre, se ha bautizado como Programación Orientada a la Tabla (Table Oriented Programming o TOP) la necesidad imperiosa que manifiestan los programadores de contar "con una tabla o campo" que permita establecer el desarrollo de la interfaz del sistema. Simultáneamente, el diseño de la base de datos se ve altamente influenciado por las decisiones de diseño de la interfaz, definiendo temas propios de bases de datos, en función de aspectos meramente estéticos.

Otro de los síntomas propios de TOP es la recurrente necesidad de utilizar campos "autoincrementantes" como llaves primarias de tablas. Incluso, en tablas donde existen muy buenos candidatos a llave primaria.

En primera instancia, en todo caso, es preciso analizar la historia de los paradigmas de programación, de manera de entender las fuerzas que provocan esta situación. El presente artículo se centra fundamentalmente en dicho análisis, para luego entrar en la descripción sintomática del paradigma detectado. El análisis preliminar de riesgos se somete a la discusión de la comunidad informática; no se pretende que este artículo cierre el proceso de análisis, sino ponga la discusión en el tapete, de modo de otorgarle su correspondiente importancia académica. Nada es más peligroso que quedarse en las definiciones teóricas, ignorando lo que ocurre en la realidad.

Un estudio de esta naturaleza es de mayor magnitud de lo que se puede abarcar en el presente artículo. Esta es la primera parte de una investigación necesaria, que debe ser conducida en distintos niveles. La información preliminar sugiere que resultados de esta tendencia son observables, además del ámbito universitario, a nivel de PyMEs, municipios y otras entidades cuyos limitados recursos los llevan a contratar programadores independientes.

2. Historia

Es conveniente hacer un rápido recuento de lo que han sido los

paradigmas aplicados en el desarrollo de los sistemas de información. La historia muestra cómo los distintos paradigmas surgen de dos grandes fuerzas: la de los programadores, y la de los jefes de proyecto y administradores. TOP es un paradigma propio de los programadores.

El presente análisis se dividirá en tres partes, según la correlación de fuerzas que afectan la impostura de uno u otro paradigma.

2.1 Grandes servidores (terreno de programadores)

Al principio de los tiempos de la Informática, los sistemas tenían fuertes restricciones, principalmente, la mínima disponibilidad de memoria y la crítica velocidad de cómputo. Baste recordar que la mayoría de los sistemas disponían de muy poca memoria (principal y secundaria) y que para procesar los programas se debía arrendar tiempo de CPU (no mucho —por costo—, tan sólo lo suficiente que requería el proceso a realizar).

Frente a estas condiciones, el programador era amo y señor de sus programas. De él dependía que todo se hiciera rápido y bien, así como a él se acusaba si las cosas fallaban. Bajo estas circunstancias, era normal que fuera el programador el que decidiera cómo, cuándo y por qué hacer las cosas. No se le sometía a mayor escrutinio ni auditoría, sólo se le pedían resultados.

Con el paso del tiempo y el mayor uso de los sistemas computacionales, los programas empezaron a crecer. No era extraño que en una empresa se solicitara "un programita" que hiciera determinado procesamiento de información (un requerimiento relativamente simple) que se reciclaba en el tiempo al que se le iban agregando "nuevos programitas". El caldo de cultivo de estos programas eran los CPD (Centros de Procesamiento de Datos), lugares más asépticos que el mejor de los quirófanos, donde habitaba "el personal de computación", una casta especial que hablaba ese idioma extraño llamado COBOL y usaba esos "armarios" que eran los computadores IBM. Su capacitación era de técnico altamente especializado; normalmente había seguido algunos cursos (principalmente en la misma IBM) y en esos esquemas asentaba sus desarrollos, los cuales, al igual que en todos los grandes servidores, se basaban en el

desarrollo de pequeños programas, también altamente especializados (de acción específica sobre los datos) y que toman una entrada y producen una salida. Así, de existir varios requerimientos, se resuelven por la vía de tomar en cada programa la salida del programa anterior y producir una nueva salida para que sirva de entrada al programa siguiente. Con ello, los programas no precisan de mayor diseño y la resolución de requerimientos "grandes" es más un problema de la unidad de operaciones (que tiene que saber en qué secuencia ejecutar los programas y dónde conviene hacer respaldos de información) que un problema visto en su conjunto al momento de diseñar la solución (que, además, ha crecido en forma desordenada y dispersa en el tiempo).

Con el crecimiento del hardware y de los requerimientos, se comienza a advertir la necesidad de tener más programadores, sobre todo, porque la organización empieza a tener más requerimientos de los que el CPD puede desarrollar. Y con la llegada de más programadores hace aparición un nuevo personaje: el administrador (o jefe de proyecto).

2.2 Programación estructurada (zona de administradores)

La libertad que tenían los programadores, que les llevaba a definir sus algoritmos casi sobre el computador, se ve restringida. Empiezan a surgir las normas de programación, normas que indican, por ejemplo, cómo denominar las variables (el nombre de la novia deja de ser aceptable...) y que constituirán las bases de la "programación estructurada" (Pressman, 2002), paradigma bajo el cual se desarrollarán gran parte de los sistemas de información.

Con la programación estructurada se logran muchos de los objetivos de los administradores, partiendo por el de poder asignar adecuadamente los recursos humanos y computacionales, incluyendo el de poder traspasar un sistema de un programador a otro. Esto va a ser de mucho agrado para el administrador, pero de a poco para el programador, pues ahora se le exigen no sólo resultados, sino también respeto por los tiempos y formas de alcanzar dichos resultados. Y si el programador no cumple adecuadamente con ello hay argumentos reales para amonestarlo (e incluso despedirlo).

Siendo los programadores seres humanos con grandezas y debilidades, intentarán, de todos modos, tener cierto control sobre los sistemas, pero tal cosa se hará con mucho sigilo, casi como un desafío intelectual que dará lugar a una competencia entre programadores, que pretenden poner "puertas traseras",[1] y administradores, que intentan evitarlas a toda costa.

Uno de los "malentendidos" de la programación estructurada, surge del concepto de reutilización de código. "Los programadores que reutilizan el código ven reducidos sus tiempos de desarrollo", reza la publicidad, y suena razonable a quien la escucha. Sin embargo, no es llegar y hacer: se requiere trabajo para generar las bibliotecas de programas y garantizar que estén libres de errores o de "puertas traseras". No obstante, para muchos programadores la idea se simplifica al llevar a la práctica el "cortar y pegar" programas, es decir, hacer una reutilización informal del código: simplemente se abre un programa, se ve qué sirve de él y el resto se borra, para luego guardarlo con otro nombre en el sistema. Así, el código hecho previamente tiene utilidad actual, pero la informalidad con la que se usa permite que los programas transmitan libremente errores (que después cuesta mucho más corregir, pues no se sabe dónde fue utilizado tal o cual código) o "puertas traseras". Pero, más grave aún, hace que el programador empiece a desechar la idea de pensar los algoritmos como un todo y trate de enfrentarlos "sobre la marcha", sin mayor diseño que el que "este programa, se parece a este otro". No es raro encontrar respaldos de programas hechos de esta manera, que incluyen 20 o más versiones del código, cada una adaptada a una condición excepcional.

Aun así, los programas y/o sistemas están fuertemente controlados por la capacidad de procesamiento de la máquina. Aspectos de velocidad del proceso y uso eficiente de la memoria son muchas veces más relevantes que un desarrollo bien cuidado. En este sentido, los operadores de las máquinas le exigirán al programador información clara respecto de las necesidades del sistema: cuánta CPU puede requerir, cuánta memoria y cómo serán estructurados los datos. Ello se traducirá, durante años, en el principal punto de control de calidad de los programas.

Si bien la programación estructurada introdujo muchas prácticas positivas, que llevaron a un mejor desarrollo del software, también es necesario tener presente que en más de una ocasión se llevó a un extremo excesivo y tan poco deseable como su ausencia. En demasiados proyectos, los programadores sentían que su esfuerzo se diluía en grandes cantidades de documentación que les parecía innecesaria. Una metodología sólo es realmente positiva si ayuda al desarrollador en su labor (Santibáñez, 2002).

2.3 Computadores personales (zona libre para programadores)

Un cambio sustancial se empezó a vivir con la llegada del computador personal, su masificación y correspondiente caída de precios. La mayor disponibilidad de memoria permitirá empezar a buscar otros esquemas de trabajo, que ya no estén tan limitados como antes. Comienzan a transformarse los conceptos de calidad de software, de sistemas, que deben ser desarrollados considerando las limitaciones de la máquina (lo que implica optimizar el uso de memoria y CPU); se pasa a la idea de que el software "pide" los requerimientos de la máquina (en otras palabras: "para usar este software, usted debe tener el siguiente hardware *mínimo*...").

Pero, sin duda, el principal cambio será la llegada de una nueva generación de programadores, los que programan para el computador personal, muchos de ellos, con capacitación técnica simple (y mucho menor que la del programador de servidores), pues requiere de menos conocimientos técnicos. Incluso, aparecen ahora los verdaderos autodidactas de la computación, que sólo saben usar el lenguaje que tienen en sus manos. Será una época en que el terreno del desarrollo de sistemas se disputa palmo a palmo por COBOL, en los grandes servidores, y Clipper,[2] en las máquinas personales.

Los programadores Clipper aún tienen importantes limitaciones en cuanto al tamaño de sus aplicaciones (hacen malabares definiendo *overlays*[3] que permitan ejecutar el programa dentro de las limitaciones de DOS). Si bien es cierto que el lenguaje tenía muchas deficiencias, era poco modular y permitía cometer graves faltas en cuanto a ciencia de la computación (alta dependencia entre

módulos, ausencia de un modelo de datos adecuado, etc.); se masifica su uso, facultando a pequeñas y medianas empresas acceder a los beneficios de la computación; sistemas de contabilidad, facturación y otros, llegarán a distintas partes y, con ello, aumentará la necesidad de programadores cerca de las empresas. Clipper impulsó el crecimiento acelerado de sistemas y la expansión de un mercado (el programador freelance) muy llamativo para mucha gente; su lenguaje simple, además, reducirá enormemente las barreras de entradas, permitiendo que cualquiera que tenga un computador personal (o que tenga acceso a uno), pueda incorporarse en este lucrativo negocio.

Pero las complejidades estaban a la vuelta de la esquina. Serán las grandes aplicaciones en Clipper las que obligarán al programador a sentarse a analizar y diseñar los pormenores del sistema, incluyendo los mecanismos para dar mayor modularidad al sistema, de modo que permita que los *overlays* no sean continuamente cargados en memoria, sino que sea posible hacer un conjunto de acciones relacionadas con un solo módulo, y que sólo al momento de cambiar de módulo sea necesario levantar el *overlay* siguiente. Con ello, muchas de las normas establecidas para la programación estructurada demuestran su valor intrínseco y se exhiben como una guía útil para el desarrollo de sistemas.

La principal desventaja de Clipper será la alta disponibilidad para el programador de las estructuras de datos. El hacer modificaciones en una tabla Clipper puede pasar desapercibido al administrador del proyecto; las tablas obran en poder del programador y mientras este no informe al administrador de las estructuras definitivas (lo que generalmente ocurre cuando se entregan los programas) puede hacer cuantas modificaciones estime conveniente. Bajo el paradigma de Clipper el diseño de los datos parece ser de menor importancia.

Por otro lado, Clipper tendrá muy buenos programadores, que son quienes se dan cuenta de las bondades de la programación estructurada altamente modular. Comienzan a reconocer que existen muchos elementos en común entre distintos sistemas y que es conveniente contar con módulos de programas potencialmente

reutilizables (bibliotecas de programación). Estos programadores verán reducidos sus tiempos de desarrollo y, por eso, los restantes programadores, al igual que los programadores de COBOL, empezarán a llevar a la práctica, el "cortar y pegar" programas y volver al concepto de "este programa, se parece a este otro".

2.4 Cliente-Servidor, OOP y otras técnicas (zona de importancia del jefe de proyecto)

La aparición en el terreno tradicional de las interfases visuales[4] (MS Windows) junto a la llegada de las redes locales, ofrecerá nuevas posibilidades. Los teóricos de la computación comenzarán a vislumbrar las potencialidades de la computación distribuida, permitiendo que distintas máquinas se dediquen a aquello que mejor pueden hacer. Uno de los esquemas que, a la larga, será de los más utilizados es el llamado: Cliente-Servidor. Bajo este esquema, el computador local se dedica a desplegar datos, mientras que el servidor los almacena, en forma más segura y compartida.

La transición no será instantánea; durante mucho tiempo (e incluso en la actualidad) convivirán sistemas desarrollados en Clipper (y con datos peligrosamente compartidos, lo que no pocas veces concluye en la pérdida de información) con sistemas Cliente-Servidor.

Una de las principales ventajas, desde la perspectiva de los paradigmas, será la necesidad de volver a contar con buenos diseños de las estructuras de datos. El tema del alcance de memoria sigue presente; las primeras versiones de Windows heredan las restricciones de DOS.

El cambio sustancial que vive Windows de sus versiones 3.x[5] a sus versiones de 32 bits (Win95, 98 y siguientes) no sólo estará dado por la libertad en cuanto al uso de memoria (si el equipo la tiene, puede usarla), sino por la aparición, en el mercado tradicional de los sistemas de información (Pressman, 2002), del paradigma de la orientación a objeto[6] (Object Oriented Programming u OOP).

Este paradigma, que no es más que la evolución natural de la metodología estructurada (ahora, con apoyo del lenguaje, la característica de encapsulamiento[7] tiene una garantía mucho mayor), es de gran ayuda

para la creación de interfaces gráficas. En principio, la orientación a objeto debiera facilitar más la creación de sistemas de información, pero no será hasta la aparición de lenguajes de programación especialmente habilitados para operar bajo ambiente gráfico, que el paradigma sea adoptado como estándar por la industria de Desarrollo de Sistemas de Información.

2.5 Programación dirigida a eventos (recuperación de los programadores)

La programación orientada a objetos traerá un hijo de menor complejidad técnica (y, por lo tanto, más cercano a los programadores): la programación dirigida a eventos (Davek et al., 2002) (Event Driven Programming o EDP). Este esquema, muy propio de las interfases gráficas, quiebra uno de los esquemas más tradicionales de la programación estructurada: el "hilo" del programa. Hasta antes de la programación dirigida a eventos, los sistemas estaban en un estado determinado y sólo pasaban a otro bajo condiciones muy controladas (sólo cuando el programador definiera que era posible o aconsejable hacerlo). Bajo la programación dirigida a eventos, el sistema ya no tiene un único (o reducida cantidad de) camino(s) posible(s); en cualquier momento, el usuario puede dar "clic" y saltar a otro módulo, abandonando lo que estaba haciendo. El principal "caballo de batalla" de la EDP será la herramienta de desarrollo de Microsoft: Visual Basic.

Por el costado, con menos fama, la empresa Borland apostará con todo a la Programación Orientada a Objetos, creando su propio esquema de árbol de objetos visuales (para ser usados en el ambiente gráfico). De forma muy consecuente, permitirán a los usuarios crear fácilmente sus propios objetos y compartirlos con el resto de la gente.

Uno de los grandes problemas de la orientación a objeto es que, si bien todos los que la discuten comparten el fondo de los conceptos de OOP, el nivel de acuerdo en cómo llegar a programar en OOP, es bastante bajo. Cada autor, al menos durante años, manifestó su propia versión de cómo hacer OOP.

No obstante, tanto en OOP como en EDP, el acostumbrado uso de bases de datos relacionales en servidores

conlleva la necesidad de que el programador diseñe muy bien sus estructuras de datos, pero empieza un "decaimiento" en la capacidad de generar algoritmos propios.

Los programadores de MS Visual Basic[8] aprenderán, en muchos casos, simplemente por la vía de copiar y adecuar programas (algoritmos) ya hechos. Aun así, sin capacitación formal, el concepto de desarrollo rápido de aplicaciones empieza a tomar forma.

Por otra parte, a los estudiantes de Informática (Ingeniería, Análisis y Programación), habitualmente se les instruye en los modelos de programación pero rara vez se les enseña a programar en un lenguaje determinado (mal que mal, los lenguajes están siempre evolucionando, no permanecen inalterables el suficiente tiempo como para que tenga suficiente utilidad el enseñarlos) y se les pide que sean capaces de aprender, por sí mismos, el uso de nuevos lenguajes. El problema se expone cuando se ve a un alumno que no es capaz de conciliar los conceptos teóricos que se le han enseñado con las prácticas de programación que acostumbra a utilizar.

Es importante señalar que mucha de la gente que estudia Informática, ha tenido acceso previo a computadores y ha establecido sus propias bases de entendimiento. Existe una muy reiterada costumbre de aprender por la vía de la imitación. Si, además, el estudiante tiene experiencia laboral en el campo del desarrollo de sistemas, es altamente posible que se vea constantemente exigido por cumplir con el desarrollo de sistemas, sin mayor preocupación por la forma en que lo consigue. Y si hay alguna preocupación es que el desarrollo sea ejecutado lo más rápido posible... Ojalá "Just-in-time".

Surge así un nuevo concepto, el Desarrollo Rápido de Aplicaciones o RAD, por su sigla en inglés. Este paradigma llegará rápidamente de la mano de las herramientas de desarrollo para Windows. Como explicación rápida de RAD, este se propone escuchar al usuario, generar en breve un trozo de código que cumpla con lo solicitado (no necesariamente un requerimiento pequeño, puede ser grande, pues de todos modos el programador hace desarrollo rápido de aplicaciones). Luego se prueba con el usuario, quien sugiere cambios, y se reinicia el ciclo.

Al final del camino (si es alcanzado) se debe tener un sistema robusto, realizado de acuerdo a las necesidades reales del usuario.

Bajo este paradigma, se configura TOP.

3. Desarrollo Rápido de Aplicaciones (RAD) y Programación Orientada a la Tabla (TOP)

El RAD viene a completar las ausencias de OOP y propone un modelo de desarrollo que es atractivo tanto para programadores como para usuarios, incluso para jefes de proyecto poco involucrados. Corresponde a un paradigma largamente esperado por todos los involucrados en el desarrollo de sistemas y acerca a la Informática los conceptos de "Just-in-time", muy en boga en casi todos los ámbitos de la administración. Ello, por supuesto, es mucho más deseable que lo que reza una de las "leyes de Murphy": un sistema se considera plenamente desarrollado cuando ha quedado obsoleto.

Las herramientas que siguen el paradigma RAD (por ejemplo, Visual Basic y Delphi), entregan un conjunto de objetos, ya sea de control o visuales, que facilitan el desarrollo de interfaces de ingreso y manipulación de datos. El conocido "Control-Data" de Visual Basic,[9] es una ayuda inestimable al momento de construir un mantenedor de una tabla de menor importancia para el usuario, pero de relevancia en el sistema (por ejemplo, una tabla que almacena información respecto de las AFP, en la que rara vez hay que hacer cambios, pero cuya información es absolutamente vital en el sistema de remuneraciones). Si a él se agregan controles visuales, como un DBGRid que automáticamente se enlaza con la tabla, obtiene los valores de todos los campos de varias tuplas de datos y los despliega ordenadamente como una planilla de cálculo, ofreciendo además la opción de editar interactivamente los datos de la tabla, se tiene una aplicación casi completa, en muy poco tiempo. Es evidente que lo que en COBOL tomaba casi una semana de trabajo ahora sólo requiere un par de minutos.

Adicionalmente a lo anterior, el uso de componentes previamente desarrollados, que han sido profusamente probados, da garantías al administrador

respecto de la calidad del producto que se está generando.

La filosofía RAD ayuda a desarrollar buenos sistemas, acerca al usuario a una mejor declaración de sus requerimientos, pues ahora ve casi inmediatamente qué es lo que está siendo entendido y puede —ante relativamente poco esfuerzo— indicar su conformidad o solicitar cambios. Es importante hacer notar que el usuario rara vez está conforme con la primera entrega; la diferencia es que ahora tiene un primer acercamiento muy rápido a la aplicación: antes podían pasar meses antes de que viera algo y pudiera opinar respecto de cómo se ven las cosas (más o menos es lo que una persona hace al probarse la ropa: ve si lo que le describieron como una chaqueta magnífica es adecuada a sus necesidades).

A la filosofía RAD también se le puede agregar el "copiar y pegar" de programas (muy común en Visual Basic). Es decir, se crea un formulario que tiene un objetivo general y luego se copia en otro, bastando cambiar las referencias a tablas y algunos de los textos mostrados en pantalla para tener un nuevo formulario, que permite editar interactivamente una tabla diferente... Y lo mismo se puede hacer con todas las tablas del sistema. Este esquema no tiene un nombre definido y por ahora se podría denominar "desarrollo muy rápido de aplicaciones".

El problema de este desarrollo muy rápido de aplicaciones es que lo que es razonable para algunas de las tablas de menor importancia en el sistema no lo es para las tablas más importantes. Es en este momento cuando se empieza a perfilar el programador TOP, y entre sus síntomas más importantes están:

a) Considera que todo elemento del sistema debe tener un dato asociado en las tablas. Lo contrario no es necesariamente cierto, puede poner datos en las tablas que sean "para uso posterior".

b) Rara vez tiene un modelo de estructura de bases de datos robusto o bien definido; lo común es que haga cambios, aun cuando ya casi todo el sistema está construido. Por lo mismo, considera que no es relevante trabajar demasiado en el modelo: basta con una aproximación inicial y "en el camino se arregla la carga".

c) Su modelo de base de datos está definido, en gran medida, a partir de conceptos propios de la interfaz. El caso más habitual es considerar que el tipo y tamaño de un campo que almacena el RUT debe contener número y dígito verificador y, por lo tanto, ser un texto que incluya los "." y el "-" en la base de datos (para que así, su despliegue sea más rápido).

d) Tiene una fuerte tendencia a discutir los requerimientos del cliente, en función de las herramientas con las que cuenta (o peor, de las que conoce).

e) Al momento de definir un formulario de interfaz visual sus referencias son respecto de los campos de la(s) tabla(s).

f) Es un gran defensor de los campos "auto-increment" para generar las llaves primarias de las tablas (de hecho, para un programador TOP un campo "id" es un campo llave, único y autoincrementable).

g) Tiene un manejo relativo de temas de normalización funcional: si bien tiende a conocerlos en la teoría, no acostumbra a manejarlos en forma práctica.

Por supuesto, el programador TOP tiene muy buenas razones para justificar todo lo anterior, siendo la más recurrente el que para él "siempre ha sido hecho así".

Tampoco es de extrañar que un programador TOP justifique su accionar; en rigor, su nivel de productividad es bastante alto y genera aplicaciones con una velocidad que es alabada por sus clientes (quienes además justifican pagar poco, pues el programador se demoró poco).

Un punto de importancia es la aparición de sistemas para internet o intranet. Utilizando lenguajes de script, como ASP o PHP, el programador TOP se ha movido de las herramientas RAD a editores de texto simple (como el conocido "block de notas") y, pese a no contar con componentes que lo avalen, la facilidad de copiar código entre aplicaciones web, le ha dado un nuevo espacio de trabajo. Por supuesto que no es el de grandes empresas ni son una fuente de innovación tecnológica, sin embargo, responden a la necesidad planteada por muchas empresas pequeñas (su nicho habitual de mercado) de contar con presencia de internet e incursionar en pequeñas aventuras de e-commerce.

4. Riesgos de la Programación Orientada a la Tabla

Si el programador TOP es eficiente y tiene a sus clientes contentos, la pregunta obvia es ¿de qué preocuparse? La respuesta tiene varios puntos que deben ser considerados.

En primer lugar, la preocupación se centra en los actuales alumnos de Informática. Estos se están acostumbrando a ser programadores TOP e incluso se sienten orgullosos de ello. Lo que no sería ningún problema si fueran capaces de, cuando las circunstancias lo ameriten, quebrar ese paradigma y concebir el sistema (o al menos una parte de él) en función de paradigmas diferentes. Mal que mal, el mundo de la Informática sigue evolucionando y la programación sigue sufriendo grandes cambios (es cosa de mirar el nuevo modelo de Microsoft, .NET, que quiebra muchas de las prácticas TOP).

Para un profesor, el explicar que la lógica de datos (es decir el almacenamiento) no necesariamente tiene que responder a la misma tecnología que se utiliza en la lógica de negocios y que esta, a su vez, no necesariamente responde a la misma tecnología que la lógica de presentación (por ejemplo, un sistema internet, con presentación en HTML que se comunica con una aplicación desarrollada usando PHP bajo Orientación a Objetos, y que almacena sus datos en una base de tecnología relacional), no es una tarea fácil. Peor aún, son muchos los alumnos que parecen no haber integrado ese dato a su base propia de conocimientos; a lo más lo consideran un dato "anecdótico", algo del estilo de "qué interesante que sea posible, pero eso a mí no me afecta".

En segundo lugar, aparece una legítima duda respecto de la calidad y originalidad del producto que se está generando. Mal que mal, la mayor parte de los programadores TOP están reciclando algoritmos antiguos, muchos de los cuales no han sido suficientemente entendidos por el programador.

Hay un riesgo en la calidad, dado que no se está completamente seguro de que todo el código sea necesario y suficiente para resolver los requerimientos planteados.

Hay una duda en la originalidad, pues se supone que se está pagando por un sistema expresamente desarrollado

para el cliente... Si el sistema en realidad es un refundido de otros, entonces no es original y puede haber problemas de propiedad intelectual.

Pero el inconveniente es más grave cuando el cliente se enfrenta a un problema, a una falla. El programador TOP, al no conocer en detalle el funcionamiento del software construido, tampoco es capaz de diagnosticar adecuadamente las fuentes de error.

Si el sistema es de relevancia al momento de tomar decisiones empresariales, entonces la calidad del software influirá directamente en la calidad de las decisiones (Cohen y Asín, 2000).

En tercer lugar, hay un riesgo en cuanto a la seguridad del sistema. Es posible que se estén heredando puertas traseras comunes entre sistemas. Si el presente sistema surge como resultado del reciclaje de otros, es posible que no todo lo que fuera necesario cambiar haya sido cambiado y, así, al detectar alguno en uno de los sistemas realizados por este programador TOP, se expongan las vulnerabilidades de los otros sistemas.

Se debe señalar que dichas puertas traseras no necesariamente son introducidas o controladas por el programador TOP. Es altamente posible que vengan de un código aún más antiguo, que aquel copiado inicialmente por el programador (o que se le entregó y que pasó deficientes controles de calidad, por ejemplo, muchas de las OCXs y DLLs de Windows).[10]

En cuarto lugar, hay un riesgo de datos. Los sistemas desarrollados según TOP tienen fuerte influencia de temas de despliegue. Muchos programadores TOP son incapaces de separar la lógica de datos de la lógica de presentación. Visto desde la perspectiva del desarrollo Cliente-Servidor multicapa, eso resulta sumamente conflictivo.

Por otra parte, está empezando a quedar en evidencia que los programadores TOP quiebran algunos de los conceptos más clásicos de la programación estructurada, en particular, de las bases de datos. El uso de "autoincrementables", aun en tablas donde hay una llave primaria evidente, rompe los tantas veces recomendados principios de normalización funcional (no sólo por la existencia de una llave primaria que no está dentro del conjunto definido de datos,

sino porque se dejan de establecer relaciones en función de la llave primaria y se mantienen sobre otros campos).

Los datos de un sistema elaborado bajo TOP son menos confiables, pues rara vez los modelos de datos cuentan con un adecuado desarrollo de las relaciones entre los datos, de manera que se corre el riesgo de que al momento de querer tener acceso a la información esta esté incompleta.

Además, la integridad de la información no se puede garantizar. Bajo un desarrollo TOP, sobre todo con errores de normalización funcional, no es posible asegurar que la base de datos contenga toda la información que deba y sólo aquella información que corresponda.

En quinto lugar, hay un riesgo para el programador. El programador TOP rara vez puede incorporarse exitosamente a grupos de trabajo, en lo fundamental, porque tiene la fuerte tendencia a reestructurar los datos constantemente. Un grupo de programadores TOP reestructurará tantas veces la base de datos que terminará con estructuras muy diferentes. Pero no sólo existe ese riesgo. El programador TOP, corre el riesgo de quedarse en la periferia casi obsoleta del mundo informático. Quizás hoy en día baste copiar algunos algoritmos para rescribir un sistema. Pero las tendencias emergentes, que incluyen un fuerte componente de orientación a objeto, no se pueden enfrentar exitosamente de esa manera. Para la OOP se requiere un adecuado diseño de objetos, antes de empezar a programar. Y ese es uno de los asuntos que más le cuesta al programador TOP, pues él siente la más profunda necesidad de poder hacer cambios al diseño, durante el proceso de construcción.

Finalmente, hay un riesgo en el usuario. Los programadores TOP van de la mano con los usuarios TOP. La generación de sistemas TOP ha llevado a un adiestramiento del usuario que espera y pide un proceso de desarrollo TOP. Este usuario (que normalmente no tiene preparación informática)[11] se ha acostumbrado a esperar y pedir desarrollos TOP. Discute los precios según su aproximación al sistema ("oye, pero si es cosa de que copies algo que ya hayas hecho..."), espera plazos extremos de desarrollo ("ahora que hemos tenido la primera reunión y tienes una visión general,

¿cuándo podemos ver una demostración?") y, por lo general, tiene un pobre concepto de cualquier asunto que suene a metodología ("¿pero eso no es viejo?" o "¿y para qué sirve?") y se manifiesta reacio a las explicaciones que le dan los "universitarios" (según el usuario TOP, los "universitarios" son aquellos que se las dan de sesudos, sólo para cobrar más; él no ve un valor agregado al título).

Los usuarios TOP se han acostumbrado a leer mensajes crípticos (Pressman, 2002; Cohen, 2000). Considerando que los componentes han sido desarrollados normalmente en países de habla inglesa y que han sido traducidos bajo paradigmas comunicacionales, diferentes de los nuestros, los mensajes predefinidos del sistema son extraños y poco claros (por ejemplo, el muy común: "Error de tiempo de ejecución", o el "pantallazo azul" de Windows que, salvo indicar que ya no hay nada que hacer, los datos que entrega no aportan nada). Pero el usuario TOP aprende a vivir con esos mensajes, le provocan una sensación de familiaridad y cierto sentido de control sobre el sistema ("otro habría quedado atónito, yo he aprendido a soportarlo", diría alguno).

Los usuarios TOP se aglutinan principalmente en las pequeñas y medianas empresas. Volviendo al concepto de calidad, aquellos no están dispuestos a pagar mucho por el sistema, sólo les interesa que funcione y que sea rápido (si el desarrollo se retrasa respecto de sus expectativas se sienten defraudados y desconfían del desarrollador). El que funcione adecuadamente no es algo que haya que probar, para el usuario TOP se da por descontado ("¿para eso no es que copian algo que ya se sabe que funciona? Si nosotros no vamos a reinventar la rueda", señalarán).

Pero los riesgos son sólo eso, riesgos. Luego de toda esta lectura se podría creer que TOP es lo más malo o lo peor que existe. No es así. Los riesgos son sólo eso, algo que puede ocurrir, una probabilidad de daño. Pero tal vez nunca ocurran; mal que mal, parte del orgullo del programador TOP estriba en evitarlos y, cuanto antes corrija un error, más rápido será capaz de preverlo en todos sus nuevos desarrollos, pues reutilizará el código ya corregido.

El programador TOP se encuentra conforme en su nicho de mercado, no aspira a proyectos mayores, su

ámbito son sistemas donde realmente puede sacar provecho al código antes generado. Por otra parte, es un mercado que no dispone de recursos para contratar grandes desarrollos ni le causa mayor ventaja competitiva el contar con un sistema muy distinto del de su vecino.

5. Conclusiones

El presente documento surge de un análisis preliminar; sería bastante ambicioso (e injusto) hacer mayores conclusiones de lo ya expuesto. Sin embargo, la investigación respecto de esta tendencia seguirá siendo elaborada para una posterior publicación.

El proceso de identificación de beneficios y riesgos de este tipo de programación sigue su curso, con una investigación que se está llevando a cabo parcialmente en la Universidad de Ciencias de la Informática y en otros ámbitos, no necesariamente universitarios.

Durante mucho tiempo, han sido diversos los conflictos experimentados por la Informática. Con el advenimiento de las normas de calidad total se esperaba que fueran corregidos muchos de los vicios detectados y fueran establecidos mecanismos que impidieran su reiteración.

No obstante, olvidamos que "calidad", no es ni más ni menos que lo que está dispuesto a pagar el cliente, y una de las cosas que no está dispuesto a pagar es el tiempo de desarrollo. Es por ello, entonces, que no es justo culpar sólo a los programadores, dado que quien les "da el afrecho" es el cliente.

TOP es un paradigma peligroso; aparte de los riesgos ya detectados se intuyen otros, de mayor cuantía en el largo plazo...

Tampoco se trata de desautorizar todo lo hecho por estos programadores (ello podría inferirse del tenor del artículo); lo que se ha intentado es poner sobre la mesa académica un tema que hoy no está siendo analizado apropiadamente.

La brusca separación entre la teoría de programación (con estándares como OOP, CORBA, etc.) y la práctica, debe ser estudiada por todos aquellos que hoy tienen responsabilidad en la educación informática de

los futuros ingenieros, analistas y programadores.

También es relevante analizar los sentimientos que despiertan las metodologías en los programadores. Se ha sostenido antes, y nunca va a ser suficiente, que para que una metodología sea seguida DEBE ser de utilidad para el programador. Si la metodología corresponde únicamente a una definición formal, sin utilidad real para aquel, sólo será un problema adicional.

La conclusión final de este análisis preliminar tiene que ver con el tema más analizado en el presente artículo. La historia muestra cómo los programadores son los primeros en tomar por asalto un nicho tecnológico. Primero con los grandes servidores, luego con los computadores personales y hoy en día gracias a RAD y TOP. Sin embargo, esa misma historia demuestra que detrás de los programadores vienen los administradores. Quizá se demorarán un poco aún, pero llegará un momento en que los programadores TOP vean controlado su accionar.

TOP existe porque aún tenemos una baja cultura de exigir calidad. Los conceptos de inversión aplicados en nuestro país son —por lo general y particularmente en las PyMEs— relacionados con la obtención de utilidades en el corto plazo. La Informática rara vez es vista como inversión; por el contrario, se la considera una fuente de gasto. Si hoy no asombra al gerente el tener que comprar un computador cada tres años, no es porque haya asumido su importancia real, sino porque se acostumbró a gastar en computación.

Una reflexión final: ninguna tecnología puede ser dejada al arbitrio de los técnicos, por muy especializados que estos sean. Debe cumplir un objetivo superior al nivel técnico y ser controlada (al menos en cuanto a las directrices principales) por quienes requieren de sus servicios. Esa es una lección amargamente aprendida en la ex Unión Soviética, en Chernobyl.[12]

Notas

[1] El concepto de "puerta trasera" puede ser descrito, en forma simple, como la idea de dejar ciertos trozos de código que se ejecutan sólo bajo condiciones muy estrictas (que únicamente el programador conoce) y que producen efectos no documentados, como tener acceso a información

que, de una manera tradicional, no debe estar disponible al programador. Por otra parte, cuando los administradores "olvidaban" las claves de acceso más relevantes de sus sistemas, agradecían la previsión del programador de crear accesos alternativos.

[2] En rigor, Clipper surgió como un simple compilador de otro lenguaje/plataforma de datos: DBase; sin embargo, es Clipper el que realmente permite la masificación de las máquinas personales, al generar código ejecutable.

[3] La técnica de *overlay*, hoy obsoleta, permitía enfrentar la limitación de memoria de DOS (640Kb) mediante el reemplazo de código no necesario en un momento por superposición de trozos de memoria.

[4] Siendo más precisos, las interfases visuales tienen mucho más tiempo de desarrollo que Windows, sin embargo, será con Windows que se van a popularizar y salir de ambientes bastante cerrados.

[5] En la práctica, en Chile no se usaron las versiones anteriores a la 3.0, al menos no más allá de ámbitos muy reducidos de experimentación.

[6] Nuevamente, es justo destacar que la orientación a objeto tiene muchos más años que Windows; en 1984, MacIntosh ya lo aplicaba. Sin embargo, el uso de este paradigma en los sistemas de información vendrá sólo después de Win95.

[7] En programación estructurada es la "modularidad", es decir, el que cada módulo sea construido sin tener presente cómo se construyen los demás. Así se pueden generar bibliotecas de software con mucha mayor facilidad.

[8] Visual Basic es el software de desarrollo más utilizado para la plataforma Windows; más usado en Chile, tanto por su parentesco evidente con Windows (ambos, productos Microsoft) como por su disponibilidad.

[9] En Delphi son los componentes TTable, TDataSource y un DBNavigator los que reemplazan al "Control Data" de Visual Basic.

[10] OCX (OLE Custom Control) y DLL (Dinamic Linked Library) son medios provistos por Windows para compartir código. Normalmente, las aplicaciones desarrolladas en Visual Basic son una larga suma de OCXs y DLLs, no necesariamente bien probadas o libres de virus.

[11] Y tampoco tendría por qué tenerla: él es usuario del sistema, nada más... Aunque hay usuarios TOP que se sienten capacitados en la lógica de desarrollo de software.

[12] Más antecedentes en: http://www.chernobyl.co.uk/ (información confirmada en otras fuentes).

Bibliografía

El autor se excusa de no poder presentar más que bibliografía de referencias secundarias. El presente artículo se fundamenta más en sus observaciones, producto de su actividad académica y profesional, que del aporte de otros autores. La

versión corregida (donde se deben presentar los resultados de la investigación) incluirá más amplias referencias.

Cohen, Daniel y Enrique Asín. *Sistemas de información para los negocios: un enfoque de toma de decisiones*. 3ra. ed. México DF: McGraw-Hill/Interamericana Editores SA de CV, 2000.

Davek, Frank, Nickolai Zeldovich, Frans Kaashoeck, Davis Mazières y Robert Morris. "Event-driven Programming for Robust Software". Proceedings of the 10th ACM SIGOPS European Workshop. September (2002): 186-189. Disponible en: http://www.pdos.lcs.mit.edu/~rtm/papers/dabek:event.pdf

Pressman, Roger S. *Ingeniería del software, un enfoque práctico*. 5ta. ed. Madrid: McGraw Hill/Interamericana de España S.A.U., 2002.

Santibáñez, José Miguel. "Fundamentos de las metodologías en la ingeniería de software". *Akadèmeia* Vol. 1, Año 2 (2002): 43-53.