SOFTWARE E INGENIERIA DE SOFTWARE

Carlos Barra P. *

Introducción.

Durante las tres primeras décadas de la informática, el principal desafío era el desarrollo del hardware de las computadoras, reduciendo el costo de procesamiento y almacenamiento de datos. A lo largo de la década de los ochenta, los avances en microelectrónica dieron como resultado una mayor potencia de cálculo a la vez que una reducción del costo. Hoy el problema es diferente, el principal desafío es mejorar la calidad (y reducir el costo) de las soluciones basadas en computadoras, soluciones que se implementan con el software.

La potencia de las grandes computadoras de la era de los ochenta está hoy disponible en una computadora personal. Las enormes capacidades de procesamiento y almacenamiento del hardware moderno representan un gran potencial de cálculo. El software es el mecanismo que nos facilita utilizar y explotar este potencial. Actualmente, el software ha superado al hardware como la clave del éxito de muchos sistemas basados en computadoras, en los cuales, sean éstos para llevar un negocio, controlar un producto o capacitar un sistema, el software es el factor que marca la diferencia. Lo que diferencia a una compañía de su competidora es la suficiencia y oportunidad de la información dada por el software (y bases de datos relacionadas). El diseño de un producto de software "amigable a los humanos" lo diferencia de los productos competidores que tengan funciones similares. La inteligencia y función que proporciona el software internamente integrado ("embeded") distingue normalmente dos productos industriales o de consumo similares. El software es el que marca la diferencia.

Evolución del Software.

El contexto en el que se ha desarrollado el software está fuertemente ligado a las casi cinco décadas de evolución de los sistemas informáticos. Un mejor rendimiento del hardware, una reducción del tamaño y un costo más bajo, han dado lugar a sistemas informáticos más complejos. Hemos pasado de los procesadores con válvulas de vacío a los dispositivos microelectrónicos que son capaces de procesar 200 millones de instrucciones por segundo y más.

La figura 1 describe la evolución del software dentro del contexto de las áreas de aplicación de los sistemas basados en computadoras. Durante los primeros años de desarrollo de las computadoras, el hardware sufrió continuos cambios, mientras que el software se contemplaba simplemente como un agregado.

La programación de computadoras era un arte para el que existían pocos métodos sistemáticos y el desarrollo del software se realizaba virtualmente sin ninguna planificación (los costos crecían y los planes eran un descalabro).

---

* Capitán de Corbeta. Ing. Nv. Eln. Magíster en Ingeniería Informática.

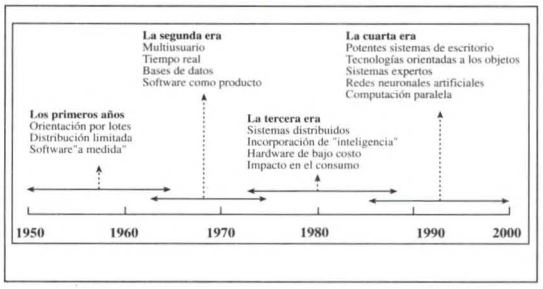


Figura 1: Evolución del software.

Durante este período, en la mayoría de los sistemas se utilizaba una orientación por lotes, siendo algunas excepciones notables varios sistemas interactivos tales como el sistema de reserva de pasajes de la American Airlines y los sistemas de tiempo real para la defensa. Sin embargo, la mayor parte del hardware se dedicaba a la ejecución de un único programa que, a su vez, se dedicaba a una aplicación específica.

Durante los primeros años, lo normal era que el hardware fuera de propósito general y, por otra parte, el software se diseñaba a medida para cada aplicación y tenía una distribución relativamente pequeña: el software como producto (es decir, programas desarrollados para ser vendidos a uno o más clientes) estaba en su infancia. La mayoría del software se desarrollaba y era utilizado por la misma persona u organización: la misma persona lo escribía, lo ejecutaba y, si fallaba, lo depuraba; debido a que la movilidad en el trabajo era baja, los ejecutivos estaban seguros de que esa persona estaría allí cuando se encontrara algún error. En este entorno personalizado del software, el diseño era un proceso implícito, realizado en la mente de alguien, y la documentación normalmente no existía. A lo largo de los primeros años se aprendió mucho sobre la implementación de sistemas informáticos, pero relativamente poco sobre la ingeniería de software. Sin embargo, es digno reconocer que durante esa era se desarrollaron muchos sistemas informáticos excepcionales, algunos de los cuales todavía se siguen usando hoy y, por sus características, siguen siendo admirados con toda justicia.

La segunda era en la evolución de los sistemas computacionales se extiende desde la mitad de la década de los sesenta hasta finales de los setenta. La multiprogramación y los sistemas multiusuario introdujeron nuevos conceptos de interacción hombre-máquina.

Las técnicas interactivas abrieron un nuevo mundo de aplicaciones y nuevos niveles de complejidad del hardware y del software. Los sistemas de tiempo real podían recoger, analizar y transformar datos de múltiples fuentes, controlando así los procesos y produciendo salidas en milisegundos en vez de en minutos. Los avances en los dispositivos de almacenamiento en línea condujeron a la primera generación de sistemas de gestión de bases de datos.

La segunda era se caracterizó también por el establecimiento del software como producto y la llegada de las "casas de software", donde el software ya se desarrollaba para tener una amplia distribución en un mercado multidisciplinario: los programas se distribuían para computadoras grandes y para minicomputadoras, a cientos e incluso a miles de usuarios; la industria, el gobierno y la universidad se aprestaban a "desarrollar el mejor paquete de software" y ganar así mucho dinero.

Conforme crecía el número de sistemas informáticos, comenzaron a extenderse las bibliotecas de software, las casas desarrollaban proyectos en los que se producían programas de decenas de miles de sentencias fuente y los productos de software comprados al exterior incorporaban cientos de miles de nuevas sentencias. Todos esos programas (todas esas sentencias) tenían que ser corregidos cuando se detectaban fallos, modificados cuando cambiaban los requisitos de los usuarios o adaptados a nuevos dispositivos de hardware que se hubieran adquirido; estas actividades se llamaron colectivamente mantenimiento del software. El esfuerzo gastado en el mantenimiento del

software comenzó a absorber recursos en una medida alarmante, aún peor, la naturaleza personalizada de muchos programas los hacía virtualmente imposibles de mantener. Había comenzado una "crisis del software".

La tercera era en la evolución de los sistemas computacionales, comenzó a mediados de los setenta y llega hasta los días de hoy. El procesamiento distribuido (múltiples computadoras, cada una ejecutando funciones concurrentemente y comunicándose con alguna otra) incrementó notablemente la complejidad de los sistemas informáticos. Las redes de área local y de área global, las comunicaciones digitales de gran ancho de banda y la creciente demanda de acceso "instantáneo" a los datos, supusieron una fuerte presión sobre los desarrolladores del software. Se produce la llegada y el amplio uso de los microprocesadores y las computadoras personales. El microprocesador es una parte integral de un amplio espectro de productos "inteligentes" que incluyen automóviles, hornos microondas, robots industriales y equipos de diagnóstico médico. En muchos casos, la tecnología del software es integrada en esos productos por equipos técnicos que conocen el hardware, pero que a menudo no tienen experiencia en desarrollo de software.

Las computadoras personales han sido el catalizador del gran crecimiento de muchas compañías de software. Mientras que las compañías de software de la segunda era vendían cientos o miles de copias de sus programas, las compañías de software de la tercera era venden decenas e incluso centenares de miles de copias.

El hardware de las computadoras personales se ha convertido rápidamente en un producto estándar, mientras que el software que se suministre con ese hardware, es lo que marca la diferencia. De hecho, mientras que las ventas de computadoras personales se estabilizaron hacia la mitad de los 80, las ventas de productos de software han continuado creciendo. Mucha gente en el campo industrial y muchos particulares han gastado más dinero en software que lo que se gastaron en la computadora sobre la que se ejecuta el software.

La cuarta era del software está empezando ahora. Las tecnologías orientadas a los objetos están desplazando rápidamente a enfoques de desarrollo de software más convencionales en muchas áreas de aplicación. Las técnicas de cuarta generación para el desarrollo de software ya están cambiando la forma en que algunos segmentos de la comunidad informática construyen los programas computacionales. Por fin, los sistemas expertos y el software de inteligencia artificial se han trasladado del laboratorio a las aplicaciones prácticas, para un amplio rango de problemas del mundo real. El software de redes neuronales artificiales ha abierto excitantes posibilidades para el reconocimiento de formas y habilidades de procesamiento de información al estilo de como lo hacen los humanos.

Pero aún en la cuarta era, continúan intensificándose los problemas asociados con el software:

- La tecnología del hardware ha dejado desfasada a la capacidad de construir software que pueda explotar el potencial del hardware.
- La capacidad de construir nuevos programas no puede dar abasto a la demanda de nuevos programas.
- La capacidad de mantener los programas existentes está amenazada por el mal diseño y el uso de recursos inadecuados.

Como respuesta a la crisis del software, muchas industrias están adoptando prácticas de ingeniería de software.

Características del Software.

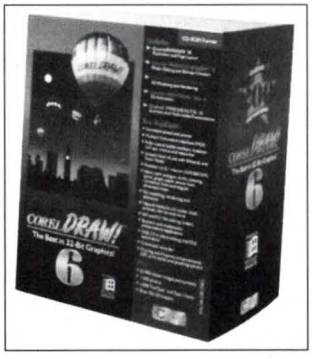
Hace veinte años, menos del uno por ciento de la gente podía describir de forma inteligente lo que significaba el "software de computadora". Hoy, la mayoría de los profesionales y muchas personas en general creen que entienden el software. Pero, ¿realmente lo entienden?

Una descripción del software de un libro de texto puede tener la siguiente forma: "Software: (1) instrucciones (programas de computadora) que cuando se ejecutan proporcionan la función y el comportamiento deseado, (2) estructuras de datos que facilitan a los programas manipular adecuadamente la información, y (3) documentos que describen la operación y el uso de los programas".

No hay duda de que podrían ofrecerse otras definiciones más completas, pero se necesita algo más que una definición formal para conocer los componentes del software.

Los componentes de software se crean mediante una serie de traducciones que hacen corresponder los requisitos del cliente con un código ejecutable en la máquina: se traduce un modelo (prototipo) de requisitos a un diseño; se traduce el diseño del software a una forma en un lenguaje que especifica las estructuras de datos, los atributos procedurales y los requisitos que atañen al software; la forma en lenguaje es procesada por un traductor que la convierte en instrucciones ejecutables en la máquina.

Los componentes de software se construyen mediante un lenguaje de programación que tiene un vocabulario limitado, una gramática definida explícitamente y reglas bien formadas de sintaxis y semántica (estos atributos son esenciales para la traducción por la máquina). Las clases de lenguajes que se utilizan actualmente son los lenguajes máquina, los lenguajes de alto nivel y los lenguajes no procedurales (orientados a objeto).



El Corel Draw 6. Software de Gráficos más potente disponible en el mercado.

En los sistemas modernos, es el software el que provee la mayor parte de la funcionalidad y, por tanto, expresa directamente la escala y complejidad de estos sistemas. La complejidad es una fuente de fallas de diseño, es decir, fallas en la construcción intelectual del sistema (las cuales serán causal de fallas de funcionamiento erróneo bajo ciertas circunstancias). Las fallas de diseño pueden ocurrir en cualquier sistema independientemente de la tecnología utilizada para su construcción, pero, debido a que éstas a menudo se originan por no anticipar ciertas interacciones sobre los componentes del sistema, o entre el sistema y su ambiente operacional, las fallas de diseño tienden a aumentar con el número y complejidad de los posibles comportamientos e interacciones. En los sistemas modernos, los componentes individuales de software realizan complejas funciones y, colectivamente, son el foco de interacción entre todas las partes del sistema y entre el sistema y sus ambiente (incluyendo operadores). Además, debido a la mutabilidad del software, éste también es el blanco de la mayoría de los cambios que se generan en los requerimientos y restricciones a medida que evoluciona el diseño global del sistema. De esta forma, el software se lleva el peso de la complejidad y volatilidad del sistema, por lo que es de esperar que las fallas de diseño normalmente se manifiesten en el software.

La razón de que el software sea el foco de la mayor parte de la complejidad de diseño en los sistemas modernos, es su versatilidad: un sistema de software puede tener muchos comportamientos diferentes y puede ser programado para responder apropiadamente a muchas circunstancias también diferentes. La fuente de estos comportamientos y respuestas diferentes radica en las muchas decisiones discretas que se hacen durante la ejecución del software; cada decisión es discreta en el sentido de que el curso de ejecución subsecuente cambia de un patrón a otro según si alguna condición es verdadera o no.

Debido a que la relación entre las entradas y salidas de una parte del software es el efecto acumulativo de estas decisiones discretas, es que la relación global entrada/salida debe ser discontinua (pequeños cambios en las entradas pueden cambiar los resultados en ciertos puntos de decisión, dando origen a cambios radicales en los patrones de ejecución y, consecuentemente, grandes cambios en el comportamiento de salida). Usualmente en los sistemas físicos existe una relación continua entre entradas y salidas (cambios suaves en las entradas, producen los correspondientes cambios suaves en las salidas). Esto permite que el comportamiento completo del sistema físico sea extrapolado a partir de un número finito de pruebas: el carácter continuo del sistema asegura que las respuestas a configuraciones de entradas no probadas serán esencialmente similares a aquellos casos cercanos que han sido probados.

Pero con el software, este método de inferir propiedades de la totalidad de los posibles comportamientos a partir de pruebas sobre una selección de muestras, es mucho menos seguro: sin continuidad no es posible asumir que casos vecinos son esencialmente similares, de modo que hay poca justificación para extrapolar. Por otra parte, no es factible una cuantificación experimental de la confiabilidad de un software con restricciones de seguridad crítica (como los sistemas de control de tráfico aéreo), lo que obliga a un aseguramiento por otros medios.

Ingeniería de Software.

En los primeros días de la informática, los sistemas basados en computadora se desarrollaban usando técnicas de gestión orientadas al hardware. Los administradores del proyecto se centraban en el hardware, debido a que era el factor principal del presupuesto en el desarrollo del sistema. Para controlar los costos del hardware, los administradores instituyeron controles formales y estándares técnicos, exigían un análisis y diseño completo antes de que algo se construyera y medían el proceso para determinar dónde podían hacerse mejoras. Dicho sencillamente, aplicaban los controles, métodos y herramientas que reconocemos como ingeniería del hardware. Desgraciadamente, el software normalmente no era más que un agregado. En la primera era, la programación se veía como un "arte": existían pocos métodos formales y pocas personas los usaban; el programador aprendía normalmente su "oficio" mediante ensayo y error; la jerga y los desafíos de la construcción del software crearon una "orden" en la que pocos ejecutivos se preocuparon por entrar. El mundo del software era virtualmente indisciplinado. Hoy, la distribución de costos en el desarrollo de sistemas informáticos ha cambiado drásticamente: el software, en lugar del hardware, es normalmente el elemento principal del costo. Es por esto que aparecieron relevantes los siguientes cuestionamientos:

- ¿Por qué lleva tanto tiempo terminar los programas?
- ¿Por qué es tan elevado el costo?
- ¿Por qué no se pueden encontrar todos los errores antes de entregar el software?
- ¿Por qué resulta difícil constatar el progreso conforme se desarrolla el software?

Estas y muchas otras preguntas son una manifestación del carácter del software y de la forma en que se desarrolla; un problema que ha llevado a la adopción de la ingeniería de software como práctica.

Para poder comprender lo que es el software y, consecuentemente, la ingeniería de software, es importante examinar las características del software que lo diferencian de otras cosas que los hombres pueden construir.

Cuando se construye hardware, el proceso creativo humano (análisis, diseño, construcción, prueba) se traduce finalmente en una forma física. Al construir una nueva computadora, el boceto inicial, diagramas formales de diseño y prototipo de prueba, evolucionan hacia un producto físico (pas-

tillas de VLSI, tarjetas de circuitos impresos, fuentes de poder, etc.). El software es un elemento del sistema que es lógico, en lugar de físico, por tanto, el software tiene unas características considerablemente distintas a las del hardware.

En un sentido clásico, el software se desarrolla, no se fabrica.

Aunque existen algunas similitudes entre el desarrollo de software y la construcción del hardware, ambas actividades son fundamentalmente diferentes. En ambas actividades la buena calidad se adquiere mediante un buen diseño, pero la fase de construcción del hardware puede introducir problemas de calidad que no existen (o son fácilmente corregibles) en el software. Ambas actividades dependen de las personas, pero la relación entre la gente dedicada y el trabajo realizado es completamente diferente para el software. Ambas actividades requieren la construcción de un "producto", pero los métodos son diferentes. Los costos del desarrollo de software se encuentran en la ingeniería, esto significa que los proyectos de software no se pueden administrar como si fueran proyectos de fabricación.

El software no se "estropea".

La proporción de fallos como una función del tiempo, para el hardware, se describe mediante una relación denominada frecuentemente "tina de baño", como indica la figura 2, el hardware exhibe relativamente muchos fallos al principio de su vida (estos fallos son atribuibles normalmente a defectos del diseño o de la fabricación); una vez corregidos los defectos, la tasa de fallos cae hasta un nivel estacionario (bastante bajo, con un poco de optimismo) donde permanece durante un cierto período de tiempo. Sin embargo, conforme pasa el tiempo, los fallos vuelven a presentarse a medida que los componentes del hardware sufren los efectos acumulativos de la suciedad, la vibración, los malos tratos, las temperaturas extremas y muchos otros males externos. Sencillamente, el hardware comienza a estropearse.

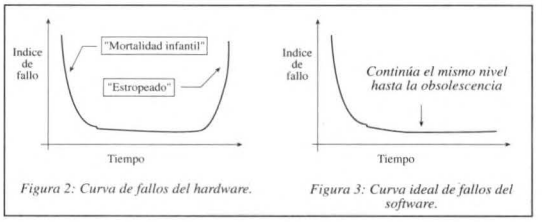


Figura 2: Curva de fallos del hardware. *Figura 3: Curva ideal de fallos del software.*

Figura 2: Curva de fallos del hardware.
Figura 3: Curva ideal de fallos del software.

El software no es susceptible a los males del entorno que hacen que el hardware se estropee, por lo que, en teoría, la curva de fallos para el software tendría la forma que muestra la figura 3. Los defectos no detectados harán que falle el programa durante las primeras etapas de su vida, sin embargo, una vez que se corrigen, suponiendo que no se introducen nuevos errores, la curva se aplana. Esa figura es una gran simplificación de los modelos reales de fallos del software, pero la implicación es clara: el software no se estropea, ¡pero se deteriora!

Esto, que parece una contradicción, puede comprenderse mejor considerando la figura 4. Durante su vida, el software sufre cambios (mantenimiento) y, conforme se hacen los cambios, es bastante probable que se introduzcan nuevos defectos, haciendo que la curva de fallos tenga picos (antes de que la curva pueda volver al estado estacionario original, se solicita otro cambio, haciendo que de nuevo se cree otro pico). Lentamente, el nivel mínimo de fallos comienza a crecer y el software se va deteriorando debido a los cambios. Por otra parte, existe otro aspecto de dicho deterioro que ilustra la diferencia entre el hardware y el software, ya que cuando un componente de hardware se estropea, se sustituye por una "pieza de repuesto", pero no hay piezas de repuesto para el software. Cada fallo en el software indica un error en el diseño o en el proceso mediante el que se tradujo el diseño a código máquina ejecutable, por lo

tanto, el mantenimiento del software tiene una complejidad considerablemente mayor que la del mantenimiento del hardware.

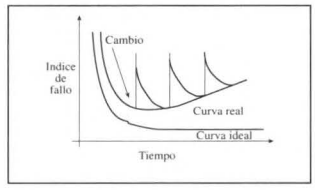


Figura 4: Curva real de fallos del software.

La mayoría del software se construye a medida, en vez de ensamblar componentes existentes.

Si se considera la forma en la que se diseña y se construye el hardware de control para un producto basado en microprocesador, el ingeniero de diseño construye un sencillo esquema de la circuitería digital, hace algún análisis fundamental para asegurar que se realiza la función adecuada y va al catálogo de muestras de componentes digitales existentes. Cada circuito integrado tiene un número de pieza, una función definida y válida, una interfase bien definida y un conjunto estándar de criterios de integración. Después de seleccionar cada componente, puede solicitar la compra. Por desgracia, los diseñadores de software no disponen de esa comodidad, ya que, con unas pocas excepciones, no existen catálogos de componentes de software. Se puede comprar software ya desarrollado, pero sólo como una unidad completa, no como componentes que puedan reensamblarse en nuevos programas. Aunque se ha escrito mucho sobre "reusabilidad del software", sólo estamos comenzando a ver las primeras implementaciones con éxito de este concepto.

La Ingeniería de Software frente al problema del Software.

El desarrollo de software se ha caracterizado por estar fuera de plazo, ser caro y contener errores.

¿Qué hace que la ingeniería de software sea tan diferente?

¿Por qué no se puede realizar correctamente?

La implicación tácita en estas preguntas es que las disciplinas de ingeniería tradicionales, hacen las cosas mejor. Esta comparación poco feliz del software con las materias de otras ingenierías, tiene alguna justificación; en particular, las disciplinas tradicionales se fundamentan en la ciencia o las matemáticas y son capaces de modelar y predecir las características y propiedades de sus diseños con bastante exactitud, mientras que la ingeniería de software es más una actividad de artificio basada en ensayo y error en vez de cálculo y predicción. Sin embargo, la comparación es muy liviana, al no reconocer que el software es diferente en dos aspectos importantes: la complejidad del comportamiento del software y su falta de continuidad.

La evaluación experimental no es el único medio de aseguramiento de los comportamientos de los sistemas físicos de ingeniería. Normalmente, en el campo de la ingeniería existen modelos matemáticos validados que permiten predecir el comportamiento y propiedades de un diseño dado, mediante cálculo (por ejemplo, la ingeniería electrónica puede calcular el comportamiento de una fuente de poder antes de que sea construida). Una de las distinciones entre las actividades de artificio y de ingeniería, es que esta última utiliza el modelamiento matemático para predecir comportamiento, mientras que en las actividades de artificio se utiliza el ensayo y error.

Todas las definiciones de ingeniería de software coinciden en que ésta trata de la construcción de sistemas de software mediante grupos o equipos de trabajo, más que individuos, utilizando principios de ingeniería en el desarrollo de estos sistemas, incluyendo tanto los aspectos técnicos como los no técnicos (el ingeniero de software, aparte del conocimiento sobre técnicas de computación, debe ser capaz de comunicar en forma escrita y oral, conocer la importancia de la administración de proyectos y apreciar los problemas que los usuarios del sistema pueden tener al

interactuar con software cuyo funcionamiento pueden no entender).

Software no sólo implica programas computacionales asociados con alguna aplicación o producto, además, incluye la documentación necesaria para instalar, operar o usar, desarrollar y mantener dichos programas. Para sistemas grandes, a menudo el esfuerzo requerido para escribir esta documentación, es mayor que el requerido para el desarrollo de programas.

Como en todas las ingenierías, en la ingeniería de software no se trata sólo de producir productos si no que producir productos de un modo costo-efectivo. Con recursos ilimitados, podrían resolverse la mayoría de los problemas de software, pero el reto para el ingeniero de software es producir software de alta calidad con una cantidad finita de recursos y dentro de una programación preestablecida (el recurso tiempo también es finito).

Suponiendo que el software entrega la funcionalidad requerida, existen cuatro atributos clave que un sistema de software de buena ingeniería debe poseer:

- *El software debe ser mantenible.* Dado que un software de larga vida útil está sujeto a cambios, debe ser escrito y documentado de modo que los cambios puedan ser efectuados sin incurrir en costos indebidos.
- *El software debe ser confiable.* Esto significa que debe comportarse como lo esperan los usuarios y su frecuencia de fallas no debe ser mayor que lo indicado en su especificación.
- *El software debe ser eficiente.* Esto no necesariamente significa que haya que lograr el grado más alto de rendimiento del sistema de hardware, ya que la maximización de rendimiento puede resultar en un software en que los cambios son dificultosos. Eficiencia significa el sistema no debe malgastar el uso de los recursos del sistema tales como memoria y ciclos de procesador.
- *El software debe ofrecer una apropiada interfase de usuario.* Muchos software no son explotados en todo su potencial debido a que su interfase dificulta su uso. El diseño de la interfase de usuario debe ser adecuada a las capacidades y experiencias de los usuarios.

Hay cientos de aplicaciones basadas en software en una situación crítica y que necesitan ser renovadas:

- Las aplicaciones de sistemas de información escritas hace veinte años, que han sufrido cuarenta generaciones de cambios y que ahora son virtualmente imposibles de mantener. Incluso la más pequeña modificación puede hacer que falle todo el sistema.
- Las aplicaciones de ingeniería que se utilizan para generar datos críticos de diseño y que, sin embargo, a pesar de su edad y estado de conservación, realmente no se entienden. Nadie tiene un conocimiento detallado sobre la estructura interna de esos programas.
- Sistemas empotrados (usados para controlar plantas de potencia, tráfico aéreo y fábricas, entre sus cientos de aplicaciones) que parecen extraños y a veces tienen un comportamiento inexplicable, pero que no se pueden poner fuera de servicio porque no hay nada disponible para reemplazarlas.

No es suficiente "reparar" lo que está mal y dar una imagen moderna a las aplicaciones desarrolladas bajo las prácticas de desarrollo no metódicas. Muchos componentes de software requieren una reingeniería o reestructuración importante o, de lo contrario, no serán competitivos durante los años noventa. Desafortunadamente, pocos parecen dispuestos a comprometer los recursos necesarios para emprender este esfuerzo de reestructuración. "Las aplicaciones todavía funcionan", dicen, o "no es económico comprometer recursos para mejorarlos".

Algunas empresas han empezado a "subcontratar", reduciendo al mínimo su personal dedicado a los sistemas de información y contratando o pactando con una tercera parte la gestión de todo el desarrollo del nuevo software, gran parte del mantenimiento de sistemas en marcha y todas sus

operaciones computacionales. Pero, cabe recordar que en los años sesenta, empresas como RCA y Motorola intentaron reducir los costos de los televisores, mediante la subcontratación de tan sólo unos pocos componentes electrónicos con el fin de poder desplazar a otros fabricantes, permaneciendo la industria central en Estados Unidos: hoy en día no quedan fabricantes importantes de televisores en Estados Unidos.

La Armada de Chile, específicamente a bordo de las unidades que componen nuestra Escuadra Nacional, cuenta con sistemas de alta tecnología tanto de hardware como de software, dentro de los cuales se incluyen sistemas de armas, de ingeniería, de comunicaciones y de mando y control. Por esto, claramente nuestra Institución no está exenta del desafío que significa enfrentar las demandas que el desarrollo de software presenta en su ciclo de vida, estos es, desde la elaboración de requerimientos hasta la fase de mantenimiento, pasando por la administración de los procesos de desarrollo (análisis de requerimientos, diseño, implementación y pruebas), administración de la configuración y aseguramiento de calidad.

Todas estas actividades requieren de la aplicación de métodos de ingeniería de software que, además, sean adecuados para las características especiales de los sistemas antes indicados, ya que normalmente presentan requerimientos de tiempo real y niveles de seguridad de funcionamiento críticos. Esto significa que la aplicación de estándares militares debe ser metódicamente incluida en cada uno de los productos de las fases del ciclo de vida de un software, lo que en la práctica se traduce en una apropiada documentación de los procesos de desarrollo de software (modelos de requerimientos, modelos de análisis, modelos de diseño, modelos de implementación y pruebas) mediante una metodología que permita asegurar tanto una exitosa administración del proyecto de desarrollo, como el nivel de calidad requerido.

BIBLIOGRAFIA

- S. Pressman, Roger: "Ingeniería del Software, un enfoque práctico"; tercera edición.
- Sommerville, Ian: "Software Engineering"; fourth edition, Addison- Wesley, 1992.
- Bowen, Jonathan - Stavridou, Victoria: "Safety-Critical Systems, Formal Methods and Standards"; submitted to the Software Engineering Journal, may 1992.
- Rusby, John: "Formal Methods an their Role in the Certification of Critical Systems"; Technical Report CSL-95-1, Computer Science Laboratory SRI International, march 1995.
- Jacobson, Ivar: "Object-Oriented Software Engineering", Addison- Wesley, 1994.
- S. Humphery, Watts: "Managing the Software Process", Addison-Wesley, 1989.