Parte I Introducción

En esta era tecnológica en la cual vivimos, nuestras vidas están regidas de gran manera por las computadoras y por el software que las controlan. Las consecuencias del uso del software son muy importantes, a favor y en contra. Cuando todo funciona bien las computadoras son de gran ayuda pero cuando no, el resultado puede ser nefasto, algo que se ha visto a lo largo de los años en múltiples ocasiones.

En esta primera parte del libro se da la motivación al área de ingeniería de software orientada a objetos donde se discuten los siguientes temas: (1) el costo del software para la sociedad, (2) la razón para utilizar tecnología orientada a objetos y (3) el proceso de software necesario para desarrollar y mantener tales desarrollos.

1 El Costo del Software para la Sociedad

Se comienza haciendo una pregunta muy sencilla: ¿Cuánto le cuesta a la sociedad utilizar sistemas de software? De manera básica el costo del software puede calcularse en base al gasto mundial en comprar productos y servicios relacionado al software. Por ejemplo, en 1995 se calculó que el mercado mundial de software fue de alrededor de $400 billones de dólares y para el 2000 se estimó que sería mayor a $1 trillón de dólares. Según estadísticas del departamento de comercio americano, el mercado mundial de software empacado en 1994 fue de $77 billones de dólares (se calcula que en 1993 se perdieron $13 billones por piratería). Se calcula que para el año 2000 sería de $153 billones de dólares. Este software empacado incluye herramientas de aplicación, soluciones de aplicación, software de sistemas, y utilerías. El mercado de servicios de información mundial en 1995 en $324.7 billones de dólares con un incremento de 13% anualmente, lo que significaría un mercado de $600 billones de dólares para el año 2000. Sin embargo, estos costos no representan la realidad completa dada la dependencia que tenemos en el software. En este capítulo profundizaré un poco más en este tema para entender cuales son los gastos "ocultos" del software y que consecuencias pueden tener para la sociedad, desde costos económicos adicionales hasta, incluso, vidas humanas.

1.1 Costos Ocultos y Consecuencias del Software

Quizás el costo oculto (*externalidades*) más importante del software (el costo no oculto es el que se paga para adquirir o desarrollar más servicios adicionales) tiene que ver con su funcionamiento incorrecto. La pregunta que nos hacemos es, dada la dependencia sobre el software en el mundo, ¿cuáles son las consecuencias de su funcionamiento incorrecto?

Estas consecuencias se pueden agrupar de la siguiente forma:

?? **Consecuencias inmediatas y efectos directos**. Pueden significar horas de caída de los sistemas involucrados y horas de transacciones perdidas. A su vez, esto puede significar que la organización tenga que arreglárselas mientras tanto sin sus sistemas; y si los sistemas son centrales a los propósitos de la organización, una falla puede representar un costo inmenso. Estos costos corresponden a aplicaciones "criticas de negocios" o "críticas a la misión". Sin embargo, el costo total de una falla de computadora es más que las consecuencias inmediatas y/o efectos directos.

?? **Consecuencias a mediano y largo plazo y efectos indirectos**. Pueden significar productividad perdida, ventas perdidas, costos de servicios de emergencia, costos de restaurar datos, costos por propaganda negativa, costos por accidentes causados, incluyendo posibles juicios en su contra. Estos costos adicionales pueden volver insignificantes el costo básico del software inicial.

Estos puntos anteriores son indicativos de que es difícil predecir el costo real del software para la sociedad a mediano y largo plazo si consideramos los problemas que pudieran ocasionar por su utilización. Por otro lado el no utilizar software no sería una alternativa aceptable hoy en día ya que los efectos serían mucho mayores.

A lo largo de estos últimos años se han podido presenciar casos de fallas de software con consecuencias nefastas. Existen muchos casos donde errores en el software o su mala utilización han costado vidas humanas o perdidas económicas multimillonarias.

Peter G. Neumann, moderador del foro de la ACM sobre riesgos al público en el uso de computadoras y sistemas relacionados, ha mantenido una lista comprensiva de desastres ocasionados por fallas del software o su mala utilización. Otros grupos, como los Profesionales de la Computación para la Responsabilidad Social (CPSR por sus siglas en ingles) reportan sobre las implicaciones sociales de todo tipo de fallas en las computadoras.

Lamentablemente se aprende más de los errores que de la correcta ejecución de los sistemas. Las siguientes secciones ilustran algunos de estos casos que sirven para motivar y concientizar al lector en que el software no es algo que pueda o deba tomarse a la ligera en especial si se conoce poco del tema. Más adelante analizaremos qué se puede hacer para manejar su complejidad y evitar desastres.

1.1.1 Fallas en Sistemas de Software

El orden en la presentación de los siguientes ejemplos de desastres ocasionados directa o indirectamente por software y por las computadoras que lo operan, es exclusivamente cronológico sin relación al área de aplicación o las consecuencias que ocasionaron. ¡Por supuesto que sólo algunos ejemplos son mencionados, de lo contrario todo el libro pudiera haber sido dedicado al tema!

?? **Fracaso del Mariner 1 (1962).** Podemos remontarnos bastante tiempo atrás para descubrir errores ya causados por computadoras. La primera misión del programa Mariner (con costo total de la misión Mariner 1 hasta Mariner 10 de 554 millones de dólares) fracasó por culpa de un caracter incorrecto ('?') en la especificación del programa de control para el cohete de propulsión Atlas lo cual causó finalmente que el vehículo se saliera de curso. Ambos, el cohete y el vehículo espacial tuvieron que ser destruidos poco después del lanzamiento. Adicionalmente, se cree que un error de computadora también fue la causa del fracaso del Mariner 8 en 1971.

?? **Sobregiro del Banco de New York (1985).** En Noviembre de 1985, el Banco de New York (BoNY) tuvo accidentalmente un sobregiro de $32 billones de dólares (¡una buena suma si consideramos que esto fue hace 15 años!) por culpa de un contador de 16 bits (la mayoría de los demás contadores eran de 32 bits) que se activó ocasionando un "overflow" del contador que nunca fue verificado. BoNY no pudo procesar nuevos créditos de transferencias de "securities", mientras que la Reserva Federal de New York automáticamente hizo un traspaso de $24 billones de dólares al BoNY para cubrir sus gastos por un día, por lo cual el banco tuvo que pagar $5 millones de dólares por los intereses diarios, hasta que el software fue arreglado.

?? **Accidente de un F-18 (1986).** En Abril 1986 un avión de combate F-18 se estrelló por culpa de un giro descontrolado ("unrecoverable spin") atribuido a una expresión "IF-THEN" para la cual no había una instrucción "ELSE" porque se pensó que era innecesaria, resultando en una transferencia fuera de control del programa.

?? **Muertes por el Therac-25 (1985-1987).** El acelerador lineal médico, Therac-25, producido por AECL (Atomic Energy of Canada Limited), fue diseñado para tratamiento a pacientes por medio de radiación de Rayos X de dos tipos: (i) tratamiento de rayo directo de bajo poder, y (ii) tratamiento de rayo indirecto reflejado de alto poder. Entre 1985 y 1987 este sistema ocasionó la muerte de varios pacientes en diferentes hospitales de USA y Canadá por culpa de radiaciones de alto poder aplicadas de manera incontrolada. La probable causa de los accidentes consistía en que para ciertas secuencias de comandos del operador de la máquina, los controles de la computadora llevaban la máquina a un estado interno erróneo y muy peligroso, generando una sobredosis masiva de radiación al paciente. Después de amplia publicidad de estos accidentes se descubrió que la FDA (Federal Drug Agency) no especificaba requisitos, y no hacía revisiones sobre prácticas de desarrollo de software o control de calidad de software en dispositivos médicos. El FDA informó en Septiembre de 1987 que comenzaría a requerir controles de software integrados a ciertas clases de dispositivos médicos.

?? **Avión derribado por el USS Vincennes (1988).** En julio de 1988 la fragata USS Vincennes estaba asignada al golfo pérsico. Después de repetidos intentos de comunicación por radio, el USS Vincennes disparó un misil (por error) derribando un avión Airbus comercial Iraní matando a todos los 290 pasajeros y tripulantes. Esto ocurrió mientras el Airbus ganaba altura, bajo la suposición incorrecta de que era un avión de combate F-14 que descendería sobre el barco de manera hostil. Aunque la orden de disparo fue dada por el comandante del navío, se culpa como causa contribuyente del incidente al sistema de radar AEGIS, el cual con su sistema de interface de usuario mostraba únicamente un punto junto a un dato textual representando al avión, en lugar del eco real del radar sobre el avión. Posteriormente se supuso que en algún momento la aerolínea Iraní estuvo en la proximidad de un F-14, probablemente durante el despegue del aeropuerto, confundiendo al sistema AEGIS y asociando de manera incorrecta la información transmitida por los "transponders" aire-tierra del F-14 a la aerolínea. Cuando el avión despegó, éste quedó asociado con los datos del F-14 sobre la pantalla. Un despliegue inconveniente y posiblemente confuso de la información de altitud del avión posiblemente confundió aún más a los oficiales del barco, los cuales supusieron que el F-14 estaba descendiendo, aunque en realidad estaba ganando altura. La inclusión de un eco real del radar sobre la pantalla hubiera hecho posible determinar que el eco del radar del avión era del tamaño incorrecto para un avión de combate.

?? **Falla del software de AT&T (1990).** El 15 de enero de 1990, AT&T (American Telegraph and Telephone) la compañía que controla las redes del mayor sistema de comunicación en el mundo, experimentó una falla masiva que dejó fuera de servicio su sistema de comunicaciones de larga distancia. La falla duró alrededor de nueve horas e interrumpió millones de llamadas de larga distancia. Un error en el software de manejo de excepciones de un tipo particular de sistema de *switching* telefónico resultó en una falla de *switching*, que a su vez causó otras fallas de *switching* en un efecto cascada. Según Neuman, se reportó que la última causa del problema tuvo

origen en un programa en el lenguaje "C" que contenía una instrucción "BREAK" dentro de una cláusula "IF" dentro de una cláusula "SWITCH".

?? **Aberración esférica en el telescopio espacial Hubble (1990).** El 25 de Abril de 1990 se puso en órbita el famoso telescopio espacial Hubble desde el vehículo espacial Discovery. Al poco tiempo, NASA descubrió que el componente más crítico del telescopio de $4 billones de dólares, su espejo principal, tenía una gran falla, imposibilitando producir imágenes altamente enfocadas. El problema en su lente es técnicamente conocido como una "aberración esférica". Una investigación de la NASA reveló que el espejo se había construido con la forma incorrecta siendo 2 micrones (1 micrón = 10^{-6} metros) más plano a los lados de lo estipulado en el diseño original, un error bastante grande según los estándares de precisión de la óptica moderna. Este fue el error principal encontrado en el telescopio, considerando que hubo otros problemas adicionales, como en sus paneles solares, sus giroscopios, y contactos eléctricos. El problema del lente radicó en que nunca fue realmente probado antes de ser enviado al espacio. En su lugar, una simulación de computadora se usó como método de menor costo para validar el rendimiento del espejo. Por desgracia, malos datos de entrada se utilizaron en la simulación, significando resultados despreciables. Para corregir el error final en el espacio, se agregó al telescopio óptica correctiva a un costo muchas veces mayor que una prueba en tierra del espejo, significando además que el espejo nunca funcionaría tan bien como se planeó. Por lo pronto, la NASA no planea otro telescopio de la magnitud del Hubble, por lo cual los astrónomos tendrán que limitarse a las restricciones actuales del Hubble, con el cual sólo se pueden ver objetos aproximadamente 20 veces más grandes de lo original.

?? **Duplicación de solicitudes de transferencias bancarias (1990).** En 1990 un error de software ocasionó que un banco en el Reino Unido duplicara cada solicitud de transferencia de pago por un periodo de media hora, acumulando 2 billones de libras esterlinas adicionales. Aunque el banco expresó haber recuperado todos los fondos, se especuló que los posibles intereses perdidos pudieran haber llegado a medio millón de libras por día.

?? **Falla del software de los misiles Patriot (1991).** En las primeras etapas de la guerra del golfo pérsico de 1991, el sistema Patriot fue descrito como altamente exitoso. En análisis posteriores, los estimados de su efectividad fueron disminuidos seriamente de 95% a 13% o incluso menos. El sistema fue diseñado para trabajar en un ambiente mucho más limitado y menos hostil que el que había en Arabia Saudita. Según reportó posteriormente el New York Times, una falla en la computadora de tierra del misil Patriot fue responsable de evitar la peor baja americana durante la guerra. Esto resultó en su inoperabilidad, permitiendo que un misil "scud" destruyera unas barracas militares americanas en Dhahran, Arabia Saudita, causando 29 muertos y 97 heridos. Aparentemente el sistema de radar del Patriot nunca vio al misil Scud. Según oficiales del ejército "una combinación imprevista de *docenas* de variables - incluyendo la velocidad, altura y trayectoria del Scud - causaron la falla del sistema del radar…[este caso fue] una anomalía que nunca apareció durante las horas de pruebas." El error se atribuye a una acumulación de inexactitudes en el manejo interno del tiempo de la computadora del sistema. Aunque el sistema ejecutaba según las especificaciones, éste debía ser apagado y prendido con la suficiente frecuencia para que el error acumulado nunca fuera peligroso. Como el sistema se usó de manera no planeada, una pequeña inexactitud significó un serio error. Después de 8 horas de uso se detectó el problema del reloj acumulado. La corrección sólo se logró al día siguiente de la catástrofe (Mellor, 1994; Schach 1997).

?? **Error en el procesador Pentium de Intel (1994).** En 1994 un error de punto flotante en el procesador Pentium le costó US $475 millones a Intel. El error no fue reconocido públicamente por varios meses por Intel diciendo que el procesador era "suficientemente bueno" además de que sería muy difícil que el error ocurriera. Actualmente, Intel está sufriendo otros problemas similares con sus procesadores, como la unidad MTH (Memory Translator Hub) usado para transferir señales de la memoria a otra unidad de la computadora (Intel 820) que podría significarle un costo similar. Recientemente ha tenido problemas con la última generación del Pentium III de 1 Ghz, donde se ha visto obligada a retirarlo del mercado. ¡Al menos la compañía ya ha aprendido a reconocer sus errores!

?? **Error en sistema de autentificación de tarjeta de crédito (1995).** Según un artículo del 4 de Noviembre 4 de 1995 del periódico Guardian en UK se relata que los dos sistemas más grandes en UK para la autorización de crédito (Barclay's PDQ y NatWest's Streamline) fallaron el sábado 28 de octubre de 1995 dejando a los negocios sin poder verificar las tarjetas de crédito de sus clientes. En el caso de Barclay, más del 40% de las transacciones fallaron por un "error en el sistema de software". Para NatWest, el problema fue una gran cola de llamadas, por razones desconocidas, las cuales retrasaron la autentificación de tarjetas. Aunque ambos tenían sistemas de contingencia permitiendo a los negocios telefonear para autenticar solicitudes, por el volumen de ventas las líneas se saturaron rápidamente.

?? **Explosión del cohete Ariane 5 (1996).** El 6 de Junio de 1996 una computadora fue culpada por la explosión del primer vuelo, el 501, del cohete Ariane 5 con un costo de US$500 millones de dólares. El cohete, que parece que no estaba asegurado, llevaba cuatro satélites, ocasionando pérdidas totales de $1.8 billones de dólares. El Ariane-5 estaba funcionando perfectamente hasta los 40 segundos iniciales cuando de repente empezó a salirse de su curso y fue destruido remotamente por una explosión solo fracciones de segundo después, ocasionadas por una señal enviada por un controlador de tierra del Ariane. Según ESA (Agencia Espacial Eurpea), la desviación fuera de curso fue ocasionada por instrucciones de la computadora controlando los escapes de los dos poderosos impulsadores del cohete. Incluso se especuló que la instrucción fue generada por la computadora porque creyó que el cohete se estaba saliendo de su curso y de esta manera estaría corrigiendo el curso de vuelo. Según el reporte final, la causa de la falla fue una excepción de software ocurrida durante la conversión de un número flotante de 64-bits a un número entero de 16 bits. El número flotante siendo convertido tenía un valor mayor del que podía ser representado por un número entero de 16 bits (con signo). Esto resultó en un "error de operando". Las instrucciones de conversión de datos (código en Ada) no estaban protegidos de causar tal error de operando, aunque otras conversiones de variables similares en el mismo lugar, sí estaban protegidas. El origen del problema parece haber sido en que el Ariane 5 podía llevar un mayor número de satélites que el Arianne-4, incrementando así su peso. Sin embargo el Ariane-5 utilizaba una gran cantidad de software diseñado para el Ariane-4. Las conclusiones finales no oficiales fueron que ningún método formal hubiera detectado el problema, ya que la raíz de tal era a nivel de comunicación entre humanos, en relación a información física aparentemente no relacionada y no un problema de programación.

?? **Error del sistema de cobranza lleva a una compañía a la quiebra (1996).** Un artículo en la edición de Abril de 1996 de "TVRO Dealer" (una publicación del área de televisión por satélite) describe cómo el intento de un servicio de programación de una gran compañía de televisión por satélite de cambiar a un nuevo sistema de software de cobranza el 28 de Marzo anterior finalmente causó la quiebra de la compañía.

?? **Error del sistema de cobranza en MCI (1996).** En la edición del 29 de marzo de 1996 del Washington Post, MCI reportó que devolverían aproximadamente 40 millones de dólares a sus clientes por causa de un error de cómputo. El error de cobranza fue descubierto por un reportero investigador de una estación local de televisión en Richmond, VA. Los reporteros encontraron que fueron facturados por 4 minutos después de hacer una llamada de tan solo 2.5 minutos, dando lugar a una profunda investigación.

?? **Mayor falla de una computadora en la historia de bancos en USA (1996).** El 18 de mayo de 1996 la revista US & World Report, y al siguiente día el diario The Boston Globe, reportaron que aproximadamente 800 clientes del First National Bank of Chicago fueron sorprendidos al ver que sus saldos eran $924 millones de dólares más de lo que tenían la semana pasada. La causa fue el tradicional "cambio en el programa de la computadora". De acuerdo a la Asociación de Banqueros Americanos, el total de $763.9 billones fue la cantidad más grande para tal error en la historia bancaria de los Estados Unidos, más de seis veces el total de fondos ("assets") del First Chicago NBD Corp. El problema fue atribuido a un "error de la computadora".

?? **Falla de la computadora del centro de control de tráfico aéreo de NY (1996).** El 20 de Mayo 1996 falló la computadora del Centro de Control Aéreo de Nueva York (ARTCC - NY Air Route Traffic Control Center) que controla el tráfico aéreo sobre los estados de New York, Connecticut, New Jersey, Pennsylvania y parte del océano Atlántico. La computadora de 7 años de vigencia perdió capacidad de servicio efectivo ("falló") dos veces la tarde del lunes 20 de mayo; la primera vez por 23 minutos y la segunda por alrededor de una hora, una hora más tarde. Parece que 4 días antes se había instalado un nuevo software en el sistema. Se volvió al sistema anterior, con procedimientos de control de tráfico aéreo más ineficientes, ocasionando un límite más bajo de saturación de tráfico y retrasos en los despegues de alrededor de una hora en los aeropuertos principales en el área; junto con un incremento en la carga de trabajo de los controladores y menor seguridad, incluyendo la desactivación de la "alerta automática de conflictos".

?? **Mala planificación del nuevo sistema de una administradora de servicios de salud (1997).** Según reportó el Wall Street Journal el 11 de diciembre de 1997, Oxford Health Plans Inc., administradora de servicios de salud en USA, de gran crecimiento en los últimos tiempos, anunció que registraría una pérdida de US$120 millones o más durante ese trimestre, además de una pérdida adicional de US$78,2 millones, su primera pérdida desde que salió a la bolsa en 1991. La razón principal fue la larga lista de problemas ocurridos con un sistema informático que se puso en línea en 1996; desde el diseño del sistema y su instalación hasta cómo fue administrado por los ejecutivos del grupo Oxford. Los problemas ocasionaron que Oxford no pudiera enviar facturas mensuales a miles de cuentas de clientes además de incapacitarla para rastrear los pagos a cientos de médicos y hospitales. En menos de un año, los pagos no cobrados de sus clientes se triplicaron a más de US$400 millones, mientras que el monto que Oxford debía a los proveedores de servicios médicos aumentó en más del 50%, a una suma

superior a los US$650 millones. La administradora de servicios médicos comenzó a planear su nuevo sistema informático en 1993, cuando sólo tenía 217,000 miembros. El sistema, desarrollado por Oracle, no comenzó a utilizarse hasta Octubre de 1996, cuando el número de abonados a su seguro médico había llegado a 1,5 millones. En ese momento el sistema ya era obsoleto. En lugar de tomar 6 segundo inscribir a un nuevo miembro, tomaba 15 minutos. A pesar de esto y que la infraestructura administrativa de Oxford no daba abasto, los ejecutivos seguían inscribiendo nuevos clientes, en el último año se incorporó medio millón adicional. A finales de 1993, Oxford trató de ajustar el sistema, además de convertir de una sola vez la mayoría de su base de datos para facturación: unas 43,000 cuentas cubriendo a 1,9 millones de miembros. Esto significó la catástrofe final, ya que la transformación entre base de datos no funcionó, y mientras tanto se suspendió por unos meses la facturación ya que no se contaba con un sistema de seguridad, ni siquiera manual. A pesar de todo esto, Oxford siguió sus prácticas habituales de contabilidad, registrando aquellas facturas que no se habían cobrado como facturación trimestral. Los problemas finales surgieron cuando Oxford comenzó a poner al día las cuentas vencidas contactando a los clientes por primera vez en varios meses. Muchos se negaron a pagar y otros dijeron que hacía mucho que habían cancelado su cuenta. Por consiguiente, Oxford tuvo que registrar US$111 millones como deudas incobrables y reconoció que tenía 30,000 afiliados menos de lo que había calculado. El presidente reconoció que debía haber contratado un ejército de trabajadores temporales para que escribieran a máquina las facturas. Por otro lado, lo primero que perdieron fueron los clientes pequeños, pero como luego no resolvieron ningún problema comenzaron a perder a los clientes grandes.

?? **Error de un controlador de discos de Toshiba (1999).** En noviembre de 1999 Toshiba llegó a un arreglo fuera de corte que le costaría a la compañía más de $2 billones de dólares para cubrir errores que pudiesen haber significado la pérdida de información por culpa de fallas en los controladores de discos "floppy" producidos en sus computadoras portátiles a partir de 1980. Aunque los controladores fueron diseñados originalmente por NEC, Toshiba producía sus propios componentes y nunca incluyó la modificación hecha por NEC en 1987 lo cual hubiese evitado el problema. Lo mas interesante del caso es que realmente nunca se reportó falla alguna. Queda por ver que consecuencias traerá este caso al resto de los fabricantes de computadoras para los cuales este precedente los tiene extremadamente preocupados.

?? **Actualización de software mal planificada paraliza Nasdaq (1999).** El 17 de noviembre de 1999 los corredores de la bolsa de valores Nasdaq no pudieron comprar ni vender acciones durante 17 minutos cruciales, después de que oficiales de Nasdaq intentaran actualizar sobre la marcha un sistema de software durante la última media hora de la sesión. Algo funcionó mal y los inversionistas tuvieron que pagar el precio.

?? **Error del milenio (2000).** Concluyo estos ejemplos con un pequeño pero nocivo error que se le conoce como el problema Y2K, “error del milenio”, o inclusive “el” problema de software del siglo 20. El problema se remonta a la década de los 60, y radica en que hace mucho tiempo los programadores adoptaron la convención de representar el año con dos dígitos en lugar de cuatro. Esta convención ocasionaría fallas en los sistemas al llegar al año 2000, ya que se *alambraba* el "19" (no se permitía utilizar un número que no fuera el “19”), para generar la fecha lo cual al llegar al año 2000 fallaría por saturar el registro de almacenamiento ("overflow"). Para empeorar las cosas a menudo los dígitos "99" o "00" eran valores reservados (“números mágicos”) significando "nunca borrar esto" o "esta es una cuenta de demostración". Además, esto resultaba en el uso de un algoritmo incorrecto para reconocer años bisiestos. No está claro si hay una razón única para haber hecho esto; porque la memoria de las computadoras en esa época era extremadamente cara, o porque no se esperaba que estos sistemas duraran tanto tiempo, o incluso quizás porque no reconocieron el problema. Aunque el problema se activó en este nuevo milenio, se tiene precedentes. Muy pocos se dieron cuenta que la IBM 360 no podía manejar fechas mayores al 31 de Diciembre de 1969, hasta que estas máquinas empezaron a fallar a la medianoche hora local. IBM recomendó a sus clientes en América y Asia mentirles a las computadoras cambiando a una fecha anterior, mientras IBM empezó a crear una solución al problema. El problema es muy extenso, afecta hardware (BIOS, relojes de tiempo real, "embedded firmware", etc.), lenguajes y compiladores, sistemas operativos, generadores de números aleatorios, servicios de seguridad, sistemas de manejo de bases de datos, sistemas de procesamiento de transacciones, sistemas financieros, hojas de cálculo, conmutadores telefónicos, sistemas telefónicos y más. No es solamente un problema de sistemas de información, todo aquel sistema que use fechas está expuesto: automóviles, elevadores, etc. (Por ejemplo, en cierto momento Visa y MasterCard pidieron a sus bancos asociados que dejen de dar tarjetas que expiren en el 2000 o después.) No solamente es un problema de aplicaciones antiguas: el año pasado el paquete Quicken para manejo de finanzas personales fue corregido para ir más allá de 1999. En enero se reportó que el Instituto Nacional de Salud (NIH) en USA recibió nuevas PCs con tres versiones diferentes de BIOS, dos de las cuales fallaron a la transición Y2K. Las soluciones fueron muchas, consistiendo de diferentes etapas para analizar el problema particular y decidiendo las medidas a tomar, incluyendo no hacer nada y dejar que el problema ocurriera para luego

arreglarlo. Existe aún toda una industria alrededor de este problema, con 1,800 compañías asociadas a la organización "year200" (year2000.com, year2000.org) y proporcionando certificaciones. Se dice que llegó a ser tanta la demanda por programadores del lenguaje Cobol (donde el problema fue más significativo) y tan desesperada la situación, que según se reportó, se fue a buscar programadores de Cobol ya retirados en asilos para ancianos Actualmente se desconoce el costo final al problema de Y2K. Es difícil estimar cual fue el costo total del problema Y2K. Según la compañía TMR (Technology Management Reports) de San Diego, los costos podrían haber sido superior a $1 trillón de dólares. Esto incluye reescribir programas existentes, adquisición e instalación de sistemas que los reemplacen, y productividad perdida por culpa de la interrupción de los sistemas para pruebas y las propias fallas por no ser funcionales en el año 2000. Y esto no incluye demandas por daños ocasionados. Otras compañías predijeron rangos de costos similares, como el grupo Gartner, que predijo un costo de $600 billones de dólares a nivel mundial. Varias compañías asignaron presupuestos para este problema: Chase Manhattan dijo que gastaría $250 millones de dólares, American Airlines y Hughes Electronics dijeron que gastarían $100 millones de dólares cada una. La oficina de administración de presupuesto de la Casa Blanca (OMB - Office of Management and Budget) calculó sus costos de reparación en 2.8 billones. Esto incluía 4,500 computadoras para defensa nacional, tráfico aéreo, pagos de impuestos y seguridad social. El Grupo Gartner estimó que los costos para el Departamento de Defensa de USA podrían ser superior a los US$30 billones.

1.1.2 Sobrecostos, Retrasos y Cancelaciones en los Sistemas de Software

Lamentablemente los costos de los sistemas de software no se restringen a fallas en el software o los sistemas de computadora. Según una encuesta hecha por el Standish Consulting Group en 1995 compañías y agencias gubernamentales americanas perdieron $81 billones de dólares por proyectos de software cancelados.
Según Rob Thomsett, las causas de esto pueden ser de dos niveles principales: (i) factores que casi garantizan la cancelación del proyecto, como la falta de un dueño del proyecto; y (ii) factores que no resultan en una cancelación inminente del proyecto, pero seguramente ocasionarán reducciones substanciales en su calidad.
En esta sección se muestran algunos ejemplos “clásicos” en orden cronológico.

?? **Sobrecosto y retraso en sistema de Allstate Insurance (1982).** En 1982, Allstate Insurance comenzó a construir un sistema para automatizar su negocio por $8 millones. El supuesto esfuerzo de 5 años continuó hasta al menos 1993 cuando terminó costando cerca de $100 millones.

?? **Sobrecosto, retraso y cancelación en sistema de London Stock Exchange (1983-1988).** El proyecto Taurus de la Bolsa de Valores de Londres estaba originalmente cotizado en 6 millones de libras. Varios años más tarde y más de 100 veces (13,200%) sobre presupuesto el proyecto fue cancelado, costando a la ciudad de Londres al momento de ser abandonado, 800 millones de libras.

?? **Sobrecosto y retraso en sistema del bombardero B-1 (1985).** El bombardero B-1 en servicio desde 1985 requirió US $1 billón adicional para mejorar su software de defensa aérea que era inefectivo, aunque problemas de software imposibilitaron alcanzar los objetivos originales.

?? **Sobrecosto, retraso y cancelación en sistema de Bank of America (1988).** En 1988, Bank of America gastó US $23 millones en una plan inicial de 5 años para desarrollar MasterNet, un sistema computarizado para contabilidad y reportes de "trust". Luego de abandonar el sistema viejo, gastaron $60 millones adicionales para lograr que el nuevo sistema funcionara y finalmente terminaron desistiendo. Las cuentas de los clientes perdidos pudieron haber excedido los billones de dólares.

?? **Sobrecosto y retraso en sistema de control de rastreo por satélite (1989).** El software para la modernización de la Facilidad de Control de Rastreo por Satélite tomó 7 años más de lo previsto y costó $300 millones adicionales ofreciendo menor capacidad de la requerida.

?? **Sobrecosto y retraso en sistema Airborne Self-Protection Jammer (1989).** El sistema ASJP (Airborne Self-Protection Jammer), un sistema electrónico de defensa aérea instalado en alrededor de 2,000 aviones de combate y ataque de la Marina Americana, costó US $1 billón adicional, tomó 4 años adicionales, y sólo fue "efectivo operacionalmente marginalmente y apropiado operacionalmente marginalmente".

?? **Sobrecosto en sistema del avión de carga C-17 (1989).** El avión de carga C-17 construido por Douglas Aircraft costó $500 millones adicionales por problemas del software aeronáutico. Un reporte de GAO notó que existieron 19 computadoras a bordo, 80 microprocesadores, y 6 lenguajes diferentes de programación.

1.1.3 Razón de los Problemas del Software

¿Cómo podemos justificar tantos problemas en el software y las computadoras? Una pequeña anécdota refleja la situación.

Se dice que hace unos años se juntaron un médico, un ingeniero civil y un ingeniero en computación para discutir cual era la profesión más antigua del universo. El médico explicó: Dios creó a Eva de la costilla de Adán; obviamente se requirió cirugía, por lo cual la medicina tiene que haber sido la profesión más antigua. A esto dijo el ingeniero civil: antes de Adán y Eva, Dios creó todo el universo, del caos puso orden en el cielo y en la tierra, siendo ésta la aplicación más espectacular de la ingeniería civil y también la más antigua. Finalmente, exclamó el ingeniero en computación, quién creen que creó el caos en primer lugar.
Una frase más actual dice que para hacer las cosas mal es suficiente una persona, pero para hacerlas verdaderamente desastrosas se requiere una computadora.

1.2 Complejidad del Software

El problema principal que hemos visto en la sección anterior radica en que cuanto más grandes son los sistemas de software mayor es su complejidad. Uno se pregunta, ¿de dónde proviene toda esta complejidad? Se puede hablar de dos aspectos que causan esta complejidad, uno estático y otro dinámico.
El aspecto estático del software tiene que ver con la funcionalidad que el software ofrece. Cuanto mayor es su funcionalidad mayor es el número de requisitos que debe satisfacer un sistema. Esto significa que los sistemas se vuelven más grandes y más difíciles de comprender por la cantidad de información y funciones que manejan. El nivel de complejidad radica en estos aspectos intrínsicos a la aplicación. Para reducir tal complejidad habría que simplificar la funcionalidad que el sistema ofrece. Obviamente, la complejidad puede fácilmente aumentar si la aplicación no está desarrollada de manera adecuada.
El aspecto dinámico del software tiene que ver con los cambios que pudieran hacerse en un sistema en el futuro. Según una “ley” de desarrollo de software (Lehman, 1985), “todo programa que se use se modificará”; y cuando un programa se modifica, su complejidad aumenta, siempre y cuando uno no trabaje activamente contra esto. Esto es similar al problema de la entropía, una medida de termodinámica sobre desorden. Según la segunda ley de termodinámica, la entropía de un sistema cerrado no puede ser reducida, solo puede aumentar o posiblemente mantenerse sin cambios. Una alternativa es aplicar reingeniería para reducir esta entropía, y así poder continuar con el mantenimiento del sistema. Por otro lado, cuando se llega a tal desorden, no es económicamente justificable continuar con el sistema, ya que es demasiado caro modificarlo. Lamentablemente, como se vió antes, la historia nos muestra que los sistemas raramente se desarrollan a tiempo, dentro del presupuesto y según las especificaciones originales. Más aún, los sistemas tienden a fallar. Sin embargo, según veremos en el Capítulo 3, no todo es negativo. Un aspecto importante para poder manejar la complejidad de los sistemas, es seguir un buen proceso de software.

1.2.1 Robustez del Software

Tomemos el caso de los sistemas de control de tráfico aéreo de Estados Unidos. El gobierno requiere que sus nuevos sistemas no dejen de funcionar por más de 3 segundos al año. Además requiere que en los sistemas de las aerolíneas civiles la probabilidad de ciertas fallas catastróficas no sean mayor a 10^{-9} por hora. La problemática de esto es cómo comprobar que dichas fallas nunca ocurran dado que de por sí ocurren muy raras veces. Por ejemplo, el requisito anterior significa que se tendría que ejecutar un programa varios múltiplos de 10^9 (100,000 años) para asegurarse que el sistema funcione bien y que tales fallas no ocurran. Según Edward Adams de IBM Thomas Watson Research Center, un tercio de todas los errores (bugs) son defectos de "5,000 años" (MTBF - mean time between failures): cada una de ellas produce un error una vez cada 5,000 años. Además de la dificultad para encontrar la falla, remover una de ellas significaría una mejora insignificante en la robustez del sistema. Según Caper Jones (1995), se puede estimar el número de defectos "latentes" en un sistema típico de acuerdo al tamaño del sistema, medido en puntos de funciones, subido a la 1.2 potencia.
Se calculó que cuando Microsoft Windows 3.1 se envió al mercado en 1992 contaba con 5,000 errores (“bugs”) conocidos. Considerando que Windows 95 consistía aproximadamente de 80,000 puntos de función, esto sugiere que Windows 95 tenía aproximadamente 765,000 errores latentes (o sea, errores que en el proceso de desarrollo debían componerse durante las etapas de pruebas). Si todas estas pruebas removieran el 99% de los errores, aun quedarían 7,650 para ser encontrados después de haberse enviado el software al mercado. La primera versión de Word tuvo 27,000 líneas mientras que Word 6.0 tenía 2 millones. Word 6.0 tenía 10 veces más funcionalidad que Word 5.1, y 3 veces la cantidad de código, significando obviamente una gran lentitud en el sistema. ¡Los números para Office 2000 y Windows 2000 aterrarían a cualquiera!
Por supuesto que existen métodos formales que son pruebas matemáticas para garantizar si un programa funciona de acuerdo a sus especificaciones. Sin embargo, esto tampoco ofrece una solución completa, aunque siempre ayuda. ¿Entonces, que alternativa tenemos para mejorar la robustez del software? Analizaremos esto y otros temas más adelante.

1.2.2 Software Suficientemente Bueno

En general no existe una sola medida que nos diga que tan bueno es un sistema de software. Por un lado, un sistema de software se puede considerar exitoso cuando satisface y posiblemente excede las expectativas de los clientes y/o usuarios en el momento de utilizarse. A nivel de negocios, esto también implica que se desarrolle a tiempo, de manera económica, y que se ajuste a modificaciones y extensiones posteriores.
De manera general se pueden caracterizar aspectos externos e internos al sistema. Como factores externos, los usuarios esperan resultados rápidos, que el software sea fácil de aprender, sea confiable, etc. Como factores internos los administradores del software esperan que el sistema sea fácil de modificar y extender, al igual que sea fácil de comprender, verificar, migrar (a diferentes ambientes de cómputo), etc. Quizás de todos estos aspectos, lo que más se puede medir cuantitativamente es la cantidad de errores o defectos que resultan.
Aunque en la práctica no se puede garantizar el software perfecto, o sea cero defectos, la pregunta es cuándo el software es *suficientemente bueno*, y cuanto esfuerzo amerita invertir para eliminar defectos adicionales.

Según Yourdon los tres elementos más importantes del software "suficientemente bueno" son funcionalidad ("feature richness"), calidad y tiempo (“schedule”) como se muestra en la Figura 1.1. Cualquier cambio en uno de estos aspectos afecta a los otros.

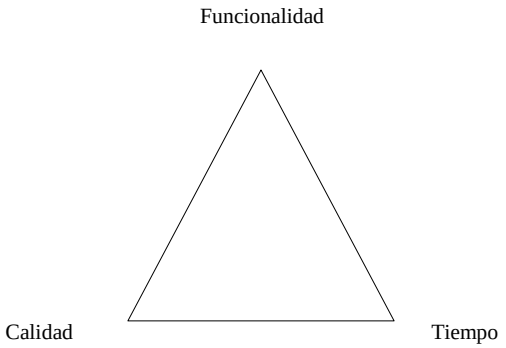

Figura 1.1 Diagrama de calidad versus funcionalidad versus horario del software.

Actualmente la situación es tan extrema que en el apogeo de la guerra de "browsers" entre Netscape y Microsoft se competía por quien liberaba más rápido su siguiente browser, agregando cada vez mayor funcionalidad, con ciclos de desarrollos de sólo unos pocos meses. Esto obviamente afectó la calidad del producto significando muchos errores en los nuevos "browsers" que no fueron depurados de manera adecuada, volviéndose el usuario el encargado de probar realmente el software y encontrar sus errores.
En 1997 errores de seguridad en Netscape y Explorer 4.0 hicieron que las compañías revisaran sus programas y los quitaran temporalmente del mercado. Situaciones similares son comunes en la actualidad. Lo peor del caso es que ante la opción de escoger entre un software perfecto, con cero defectos o una versión más nueva con todo lo novedoso, pero que pudiera tener algunos errores, la gente siempre quiere la nueva. En cierta manera nosotros mismos impulsamos el deterioro en la calidad del software comercial. La famosa frase "más rápido, más barato, mejor" realmente significa en la actualidad "suficientemente rápido, suficientemente barato, suficientemente bueno".

1.2.3 La Bala de Plata ("Silver Bullet")

En 1975 Fred Brooks, "el padre del Sistema 360 de IBM", escribió su famoso libro "The Mythical Man-Month" en el cual resaltaba la complejidad en el desarrollo de sistemas de software; un clásico aún hoy en día vuelto a publicar en 1995 para su vigésimo aniversario. El sistema operativo OS/360 de IBM escrito en la década de los 60 tuvo en su apogeo 1000 personas trabajando al mismo tiempo en el proyecto, incluyendo programadores, documentadores, operadores, secretarias, administradores y demás. Entre 1963 y 1966 se calculó que se utilizaron para el diseño, construcción y documentación del sistema 5000 años-persona. Calculado a 100 líneas/persona/mes esto sería equivalente a 5,000x100x12 = 6 millones de líneas. Este sistema es el precursor de MVS/370 y MVS/390 aún utilizado por los mainframes de IBM. Uno de sus más conocidos legados es la famosa "Ley de Brooks" que resalta que cuanto más gente se agregue a un proyecto de software ya retrasado más se retrasa el proyecto, como se muestra en la gráfica de la Figura 1.2.

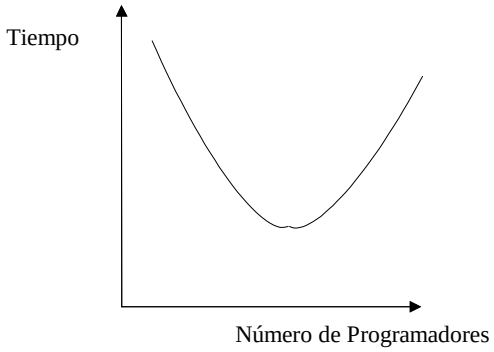

Figura 1.2 Ley de Brooks: cuanto más se aumente el número de trabajadores mayor el tiempo de desarrollo.

La razón para esto se basa en que las necesidades de personal se calculan inicialmente según una simple medida de líneas de código producidas por una persona al mes (el estándar actual es aproximadamente de 100 a 1,000 líneas por personas al mes), lo cual significaría que si un proyecto requiere 10 millones de líneas, simplemente se puede dividir por los meses y personas que se requieren para lograr esa cantidad y si llegase a haber un retraso, simplemente se pueden agregar más personas en base a un cálculo similar. Sin embargo, esto no funciona así en la realidad. La razón principal de esto es que a las nuevas personas hay que entrenarlas y explicarles el proyecto. Significa que se quita temporalmente personal ya involucrado en el proyecto para dedicarle a las nuevas contrataciones, causando que el proyecto se retrase aún más. Pero esta ley no es el legado más importante de Fred Brooks para la computación. Brooks ha contribuido con muchas ideas, incluso algunas controversiales. En 1987 (IEEE Computer, Abril 1987) publicó su ya célebre artículo "No Silver Bullet" en el cual menciona: "..según miramos al horizonte de una década, no vemos ninguna bala de plata. No existe un solo desarrollo, en la tecnología o técnica de administración, que por si sólo prometa incluso una mejora de un orden de magnitud en productividad, seguridad (“reliability”), simplicidad, dentro de una década". En otras palabras, no hay nada que permita mejorar la calidad del software de manera radical.

1.2.4 Ciclo de Vida del Software

Para apreciar esto es necesario comprender un poco más en qué consiste desarrollar software.
Un sistema de software tiene un ciclo de vida que comienza con la formulación de un problema, seguido por la especificación de requisitos, análisis, diseño, implementación, verificación, validación, integración y pruebas del software, continuado de una fase operacional durante la cual se mantiene y extiende el sistema.
Todo desarrollo de software incluye aspectos esenciales, como la creación de las estructuras que resuelvan el problema, junto con aspectos secundarios ("accidentales"), como la codificación y las pruebas. Según Brooks, existe una regla empírica ("thumb rule") que dice que para el desarrollo de un proyecto de software se debe asignar, 1/3 del tiempo a la planeación, 1/6 a codificación, 1/4 a pruebas de componentes, y 1/4 a pruebas del sistema, como se muestra en la Figure 1.3. O sea, la mitad del esfuerzo (2/4) son dedicados a pruebas lo cual también incluye la depuración y aspectos secundarios del software.

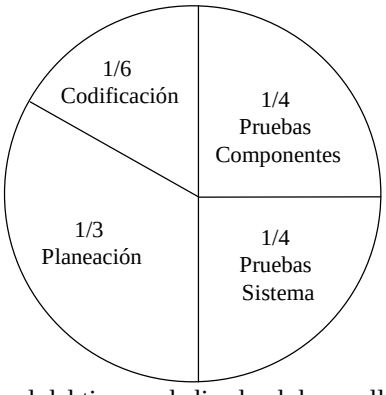

Figura 1.3 Estimado general del tiempo dedicado al desarrollo de un proyecto de software.

La mayoría de las mejoras en la productividad del sofware se han dado históricamente simplificando las tareas secundarias como las herramientas, ambientes y lenguajes de programación. Según la premisa de Brooks, a menos

que lo secundario fuese más de 9/10 del esfuerzo total, reduciendo estas actividades a cero no resultaría en un orden de magnitud de mejoría. Ya que éste no es el caso, sería necesario también reducir el tiempo dedicado a lo esencial. ¿Entonces, como hacer para mejorar tan radicalmente la productividad del software? El autor muestra cierto pesimismo, y lamentablemente bastante realismo. Todo esto es un reflejo de que los sistemas de software son muy complejos, pudiendo contar con muchos millones de líneas de código. Esta complejidad requiere de un proceso de desarrollo de software eficiente y sistemático, con base a buenas metodologías y herramientas de apoyo. Como no se puede eliminar la complejidad, por lo menos se podrá reducirla a un nivel manejable.
Otro famoso autor y tecnólogo, Ed Yourdon discute en su libro "Rise and Resurrection of the American Programmer" en 1996 (una revisión a su libro anterior titulado "Decline and Fall of the American Programmer, 1993), que aunque no hay un sólo desarrollo que sea la "bala de plata", sí se pueden ver varios aspectos que juntos pueden dar ese incremento en orden de magnitud. En particular, él da énfasis en la cuestión humana ("peopleware"), proceso de software, tecnología de objetos, reuso y métricas de software. Estos temas han sido tratados por múltiples autores, algunos más optimistas que otros. Considerando que es inevitable seguir desarrollando software, veamos qué se puede hacer.

Analicemos en el resto de la introducción de este libro los aspectos más relevantes en la actualidad, del software orientado a objetos (Capítulo 2) y del proceso de software (Capítulo 3).