# Las causas de las deficiencias de la Ingeniería de Software

Juan Francisco Giró,[i] Judith Disderi,[ii] y Belén Zarazaga[iii]

## Resumen

Desde el nacimiento de la *Ingeniería de Software* se ha instaurado un interrogante que no parece encontrar una respuesta definitiva: ¿Es posible "la aplicación de las prácticas habituales de las ingenierías tradicionales al desarrollo de productos de software"? Con esta premisa, la *Ingeniería de Software* debió resolver la llamada *Crisis de Software*, que sin embargo aún se mantiene vigente. La cuestión puede reformularse a "¿si es factible evitar deficiencias en los proyectos de sistemas aplicando dichas prácticas?". En este trabajo se identifican y desarrollan tres factores que lo impiden: *i)* no se aplican realmente las prácticas de las ingenierías tradicionales, ya sea porque la investigación no aporta las propuestas necesarias o bien porque la industria del software no las utiliza; *ii)* la *Ingeniería de Software* no alcanza a la totalidad de la industria, comprobándose que los sistemas técnicos permanecen en buena medida ajenos a ésta nueva ingeniería; y por último, *iii)* existe un gap semántico aún no superado entre el modelo funcional, de la fase de análisis, y el modelo de objetos requerido en la fase de diseño, afectando la calidad de los procesos de desarrollo. Por ser la sociedad cada vez más dependiente de los bienes y servicios informáticos que se le ofrecen, sin lugar a dudas el problema planteado representa una enorme responsabilidad ética y social, motivo por el cual se invita a reflexionar sobre los aportes que pueden esperarse de la *Ingeniería de Software* para superar la citada crisis.

***Palabras clave:*** ingeniería de software, crisis del software.

[i] Instituto Universitario Aeronáutico (IUA),Universidad Tecnológica Nacional (UTN, FRC) y Universidad Nacional de Córdoba (UNC, FCEFyN),Docente Investigador.

[ii] Universidad Católica de Córdoba (UCC, FI), Instituto Universitario Aeronáutico (IUA) y Universidad Nacional de Córdoba (UNC, FFyH),Docente Investigadora.

[iii] Universidad Católica de Córdoba (UCC, FI),Docente Investigadora.

## Abstract

Since *Software Engineering* was born, a question has been established that still nowadays happens to have no final answer: "is it possible the application of common traditional engineering practices to development of software products?". With this premise, *Software Engineering* had to resolve the *"Software Crisis"* that is still valid, nevertheless. The question can be reformulated as: "is it possible to avoid deficiencies in projects applying such practices?". The authors identify and develop the three factors that appear to prevent it: *i)* common traditional engineering practices are not applied, because investigation does not provide necessary proposals or because software industry does not use them; *ii) Software Engineering* does not cover the whole industry checking that technical systems remain largely oblivious to this new engineering; and last, *iii)* an unresolved semantic gap between the functional model in the analysis phase, and the object model required for the design phase affecting the quality of development processes. Since society is subject, as increasingly, to the quality of goods and computer services that are offered, no doubt the exposed problem represents and enormous ethical and social responsibility. That´s why this article suggests thinking on the contribution that is expected from *Software engineering* to overcome the well known *Software Crisis*.

***Keywords:*** software engineering, software crisis.

## Introducción

Para comenzar se invita al lector a un pequeño ejercicio mental: identificar el software que pudo haber estado involucrado en su actividad y bienestar en un día cualquiera. Se sugiere pensar unos minutos y proceder a su revisión: ¿tuvo energía eléctrica gracias a centrales y líneas de transmisión?, ¿encendió la cocina con gas proveniente de lugares lejanos?, ¿habló por teléfono?, ¿extrajo dinero de un cajero electrónico?, ¿hizo una reserva aérea?, ¿compró un bien con tarjeta de débito?, ¿se hizo un control médico con el auxilio de una ecografía y un electrocardiograma?, ¿hizo consultas por internet y accedió a redes sociales?, ¿utilizó un horno de microondas?, ¿se condujo en automóvil? y ¿vio televisión?, además de otras actividades. En nuestros días todo ello es posible gracias al software, donde en algunos casos se trata de productos que incluyen cientos de miles de líneas de código y en otros unas pocas decenas de ellas. A pesar de tratarse de una revisión seguramente incompleta, no quedan dudas sobre la importancia del "producto" software como soporte del mundo que hoy conocemos.

Naturalmente, en 1968 (casi cincuenta años atrás) la dependencia del software era muchísimo menor; no obstante, ya cumplía un rol central en algunas actividades como las relacionadas con la defensa. Hasta ese entonces, los resultados obtenidos con los desarrollos de sistemas de computación eran pobres, normalmente de dudosa calidad y difícilmente terminados en los plazos y con los presupuestos previstos. A estos problemas debían agregarse el incumplimiento de las especificaciones y las serias dificultades que acarreaba el mantenimiento del producto final. Esto motivó que en ese año la OTAN (Organización del Tratado del Atlántico Norte) convocara a una conferencia que tuvo lugar en Garmisch, Alemania.

Para tener una referencia temporal sobre la época, cabe recordar que doce años antes John Backus había presentado (1956) el primer lenguaje de programación de alto nivel de propósito general (Fortran) y el correspondiente compilador. Esto significa que, al momento de la conferencia de Garmisch, ya se disponía de numerosos lenguajes de alto nivel, tales como: Algol (1958), Lisp (1958), Cobol (1959), PL/1 (1964), Simula (1964) y Basic (1964).

Volviendo a la citada conferencia, un grupo de estudios concluyó que estas reiteradas deficiencias de los proyectos de sistemas podrían superarse, adaptando y aplicando al software las prácticas que eran de uso común en las ramas tradicionales de la ingeniería. Uno de los asistentes (F. Bauer, 1968) caracterizó al problema como una *Crisis de Software* y las soluciones propuestas dieron nacimiento a la *Ingeniería de Software*. Un tiempo después, fue Edsger Dijkstra (1972) el que hizo referencia a la *Crisis de Software* en un artículo que alcanzó gran difusión, convirtiéndola en el rótulo que caracterizaba el problema recientemente descripto.

A partir de allí se viene realizando un gran esfuerzo para revertir esta situación, tanto mejorando los métodos, técnicas y herramientas orientadas a la construcción

del producto software, como así también trabajando sobre las actividades de soporte requeridas por el proceso de desarrollo. En la actualidad esto es puesto en evidencia por la numerosa bibliografía especializada que trata estos importantes temas, pudiéndose citar textos considerados clásicos, como son los de Roger Pressman (2010) e Ian Sommerville (2012), y el documento elaborado por IEEE denominado Swebok v3.0 (Bourque y Failey), entre muchos otros.

Retomando sobre la Crisis de Software, y a pesar de lo mucho realizado, no son pocos los que consideraron que la Ingeniería de Software no viene alcanzando plenamente sus objetivos; inclusive hay advertencias sobre que la Crisis de Software mantiene plena vigencia y en algunos casos se ha agudizado. Hacia fines de los '80 Frederick Brooks denunció en su célebre artículo "No Silver Bullet" (1987) que la Crisis de Software no había sido resuelta. Posteriormente, a mediados de los '90, Wayt Gibbs (1994) volvió sobre este tema reconociendo que la Crisis de Software no solo mantenía vigencia sino que se había hecho crónica. Mucho más recientemente Steven Fraser (2008) coordinó un panel de especialistas, que reunidos con motivo del 20 aniversario del célebre artículo de Brooks, discutieron el estado de situación y las perspectivas futuras. En esa ocasión no faltaron dudas con referencia a muchos de los progresos proclamados, sosteniéndose que los problemas principales no habían sido resueltos, más bien que habían quedado disimulados gracias a los progresos y favorable evolución del hardware. Desde entonces, y al día de la fecha, no ha habido novedades significativas como para revertir esta poco alentadora apreciación.

## Factores que impactan en el éxito de la ingeniería de software

Llegado a este punto, cabe hacerse una pregunta: ¿Qué factores impiden, aún hoy, la plena confirmación de la premisa enunciada en Alemania en 1968, referida a que "la aplicación al software de las prácticas de uso común en las ramas tradicionales de la ingeniería evitarán las deficiencias en los proyectos de sistemas"? Se trata de una duda trascendental si se considera la fundamental importancia del software en el mundo actual y su indudable proyección creciente. Además, se trata también de una pregunta compleja, cuya respuesta seguramente admite muchos puntos de vista, presentándose a continuación los tres argumentos más importantes según la opinión de los autores de este artículo.

### 1) No se aplican las prácticas habituales de las ingenierías tradicionales

Conceptualmente, la ingeniería es una disciplina esencialmente objetiva, que se apoya en leyes naturales, resultados experimentales y fórmulas empíricas para proponer y respaldar soluciones a los problemas que se le presentan. Para ello, lo

hace inspirada en enfoques que podrían considerarse versiones tecnológicas del método científico, en el que las evidencias ocupan claramente un lugar central.

Entre otros muchos, Tom DeMarco (1982), prestigioso investigador de la Ingeniería de Software, manifiesta su adhesión a esta línea de pensamiento al proclamar su conocida *advertencia "No se puede controlar lo que no se puede medir"*. Por su parte la IEEE define la Ingeniería de Software como la *"aplicación de una aproximación sistemática, disciplinada y cuantificable al desarrollo del software..."*. Haciendo historia, Luis Pasteur decía que *"una ciencia tiene la misma madurez de sus herramientas de medición"*.

Por el contrario, es fácilmente comprobable que una parte considerable del esfuerzo de investigación en el campo de la Ingeniería de Software es esencialmente subjetivo, orientándose a modelos conceptuales, ontologías, marcos de trabajo (frameworks) y otros recursos abstractos, de muy difícil validación. Un amplio testimonio de esta apreciación la brindan los temas tratados en congresos y simposios de la especialidad. Cabe aclarar que aquí no se cuestionan estos enfoques, más bien se destaca su incongruencia con la hipótesis que respalda el éxito de la Ingeniería de Software. Es decir, no puede evaluarse una hipótesis si no se aseguran las condiciones que la sustentan. Planteado con otras palabras, de ser válida la hipótesis, los enfoques subjetivos no resolverán el problema de la Crisis de Software.

Siguiendo con esta línea de pensamiento y buscando las causas, cabe preguntarse si es el campo de la investigación la que no aporta propuestas y recursos innovadores, apropiados y suficientes, que permitan hacer más objetiva a la Ingeniería de Software, o es la industria del software la que no los utiliza.

Norman Fenton (1999), prestigioso investigador de esta actividad, ya entonces se inclinaba por esta última afirmación y al día de hoy (Fenton, 2014) aún sostiene que la brecha entre lo que se mide y lo que se podría medir permanece más grande de lo recomendable. Sin embargo, en el mismo trabajo también enuncia que *"muchas de las investigaciones realizadas sobre métricas son inherentemente irrelevantes a las necesidades de la industria, ya sea en alcance o en contenido"*, brindando también respaldo a la primera posición. Por su parte, la apreciación de Robert Glass (1994) es aún más rotunda, anticipando que *"Lo que la teoría y la práctica están haciendo con respecto a las mediciones en el desarrollo de software está en dos planos muy diferentes, que divergen en distintas direcciones"*. Todo lo expuesto deja en claro que este tema tiene todavía pendiente un amplio y profundo debate.

## 2) La ingeniería de software no alcanza a la totalidad de la industria

En forma muy general, a los sistemas informáticos se los puede clasificar en *Sistemas de Gestión* y *Sistemas Técnicos*. Los primeros respaldan la actividad comercial, industrial y de servicios, mientras que los segundos son productos de software destinados

a resolver problemas en campos diversos como matemática, física, química, astronomía e ingeniería, entre otros. En el caso específico de la ingeniería, los sistemas técnicos dan respuesta a problemas de cálculo estructural, transferencia de calor, aerodinámica y control, por citar sólo algunas de las principales aplicaciones. Al mencionarse los sistemas de control debe recordarse que aseguran el buen funcionamiento de centrales nucleares, equipamiento médico, pilotos automáticos de aviones, transporte de energía eléctrica y combustible, coordinación de tráfico de aviones o trenes, etc.

Aquí es necesario anticipar dos consideraciones sobre los Sistemas Técnicos: *i)* el desarrollo e implementación de estos sistemas está normalmente a cargo de profesionales altamente especializados, que actúan en equipos multidisciplinarios relativamente pequeños y *ii)* muchos de estos sistemas son considerados críticos, pudiendo ser de tiempo real, estar embebidos o exhibir una combinación de ambas cualidades. Los Sistemas Críticos (Barbacci, 1995) son aquellos cuya respuesta incorrecta o tardía puede tener consecuencias irreparables, que van desde enormes perjuicios económicos hasta la pérdida de vidas humanas. Los sistemas de tiempo real y los embebidos presentan también particularidades especiales, ya que utilizan sistemas operativos específicos u operan directamente sobre los recursos de hardware. Todos ellos experimentan en la actualidad creciente difusión.

Resulta así sorprendente comprobar la frecuente distancia existente entre el desarrollo de Sistemas Técnicos y la Ingeniería de Software, presentándose la paradoja que el desarrollo de los sistemas de mayor complejidad y riesgo construidos por el hombre, es en buena medida ajeno a las prácticas, técnicas y recomendaciones que vienen siendo propuestas para superar la llamada crisis de software.

Dos de las justificaciones mas notorias para esta circunstancia son las siguientes: *i)* la escasa importancia que se asigna al análisis de requerimientos, lo que resultaría natural por tratarse de desarrollos evolutivos incrementales, a cargo de los mismos usuarios, profundos conocedores de los problemas a resolver y *ii)* la ausencia de capacitación a nivel de grado en las carreras clásicas de ingeniería con relación a la Ingeniería de Software y los recursos que ofrece.

Esto último conduce a que los ingenieros especialistas que ingresan al mundo del software desconozcan la existencia de la Ingeniería de Software, ya que normalmente, en los planes de estudios de sus carreras universitarias solo se les ofreció un curso de programación. Simultáneamente, la enorme cantidad de componentes y librerías disponibles en sus ámbitos laborales, escritos por muy reconocidos especialistas en lenguajes de programación tradicionales, habitualmente Fortran y en menor medida C y Pascal, respaldan toda una cultura de desarrollo y contribuye a desalentar la adopción de lenguajes y herramientas más actuales, favoreciendo la resistencia al cambio.

Por lo tanto, cabe destacar que muchas de las manifestaciones de la Crisis de Software que exhiben los sistemas técnicos estarían en este caso potenciadas

precisamente por la ausencia de la Ingeniería de Software en su desarrollo, y no necesariamente por sus deficiencias, lo que respalda la afirmación de que la Ingeniería de Software no alcanza a la totalidad de la industria.

Aquí es necesario reconocer que los usos y costumbres del desarrollo de software técnico son tan antiguos como lo son los lenguajes de programación, con muchísimos puntos de contacto con las prácticas que se presentaron treinta años después en la Ingeniería de Software, proclamadas como innovadoras, y reconocidas en la actualidad como "metodologías ágiles". Naturalmente, esto también debe conducir a la reflexión.

Finalmente, debe notarse que si bien la escasa presencia de la Ingeniería de Software en muchos campos de desarrollo intensivo de software técnico no tuvo repercusión durante mucho tiempo, está siendo motivo de creciente atención. Como referencia pueden citarse los trabajos de Segal (2008) y de Naguib y Li (2010), entre otros muchos. Además, en las sucesivas nuevas ediciones de los ya citados textos de Robert Pressman e Ian Sommerville se comprueba que el tratamiento de los sistemas críticos viene experimentando creciente importancia, lo que refleja una tendencia muy alentadora con referencia a la atención que merece este importante tema.

### 3) La existencia de un "gap semántico" todavía no superado

Los progresos en la computación siempre se originaron en la etapa de programación, para luego extenderse hacia arriba en el ciclo de vida, al diseño y análisis. Esto nunca fue una excepción y cuando el entonces Capitán de la USAF Grady Booch dictaba sus primeros cursos de programación orientada a objetos, las etapas previas respondían a un paradigma anterior, lo que él reconoció como una incongruencia. Esto estimuló en Booch la necesidad de renovarlas, y para ello inició un proceso que culminó en una alianza con James Rumbaugh e Ivar Jacobson, que dieron lugar al Lenguaje Unificado de Modelado o ***UML*** (Booch, Rumbaugh y Jacobson, 2014).

A pesar de no tratarse de ningún lenguaje unificado, sino más bien de una recopilación heterogénea de "artefactos" de modelado, muchos de ellos con objetivos superpuestos, el ***UML*** cubrió un vacío existente y se convirtió rápidamente en un incuestionable estándar del mercado, que hoy no se discute. El aporte de Jacobson fueron sus Casos de Uso, en los que venía trabajando hace ya mucho tiempo, y que pasaron a formar parte del "artefacto" central del análisis funcional de ***UML***.

Hasta aquí un resumen de los hechos, ahora las causas del problema: ***UML*** induce a la confusión al incorporar una técnica ajena al paradigma de objetos, que son los *Casos de Uso*, para la definición del modelo de análisis. El propio Jacobson (1994) deja este punto en claro al decir que "*El modelo de Casos de Uso define los requerimientos del sistema, pero no se ocupa de su estructura interna. En teoría, esto significa que, basado en el modelo de Casos de Uso, cualquier método apropiado de diseño (estructurado u orientado a objetos) puede ser utilizado para*

*construir el sistema, siempre que el producto pueda desempeñar correctamente todos los Casos de Uso"*.

Sobre el tema Craig Larman dice (2004): *"No hay nada orientado a objetos en los Casos de Uso, no estamos haciendo análisis **OO** al escribirlos. Eso no es un problema, los Casos de Uso son ampliamente aplicables, lo que incrementa su utilidad. Dicho esto, los Casos de Uso son clave para el ingreso de los requerimientos al diseño **OO** clásico"* (p.120).

En el mismo artículo ya citado Jacobson (1994) hace un comentario aún mas esclarecedor: *"El diseño puede ser descrito como un proceso de generación y validación de una hipótesis. Es decir, dado un Caso de Uso: i) se genera una hipótesis utilizando los principios de diseño de **OO** con uno o más objetos posibles y ii) de alguna manera se valida, con referencia a algún criterio, la hipótesis contra el Caso de Uso original para determinar si se trata de una buena interpretación. Queda establecido un "gap" (brecha) entre un Caso de Uso y su diseño que está planteado en términos de objetos que interactúan"*.

Esto delata el verdadero problema, frecuentemente no advertido: que la arquitectura de objetos diferirá significativamente de su esquema de descomposición funcional, como consecuencia del "gap semántico", entre el modelo funcional desarrollado hasta el momento y el modelo de objetos requerido para ingresar en la fase de diseño.

João Fernandez y Johan Lilius (2004) son muy categóricos al reconocer que *"el salto desde los Casos de Uso y escenarios a las Clases es, en nuestra opinión, muy grande... Este paso requiere demasiado ingenio y experiencia, ya que no hay ninguna relación directa evidente entre Casos de Uso y Clases"*.

Las posibles soluciones a este problema no parecen muy concretas y ello lo demuestra Ian Sommerville (2011), quién reconoce que *"En la práctica, diversas fuentes de conocimiento se utilizan para descubrir objetos y clases"* (p. 183), citando el análisis gramatical, el uso de entidades tangibles y el análisis basado en escenarios, sin brindar mayores detalles. Por su parte, Bruce Douglass en su metodología Ropes (1999) postula que *"hay alrededor de una docena de estrategias diferentes que permiten definir los objetos con suficiente grado de aproximación"*.

Kenneth Kendall y Julie Kendall (2011) tampoco son muy específicos en la forma de superar el "gap" entre los Casos de Uso y el modelo de clases. Enuncian que "*Los diagramas de interacción (secuencia o comunicación), junto con los diagramas de clases, se utilizan para la realización de un Caso de Uso…"* (p. 294) y que *"Hay varias formas de determinar las clases… También se debe examinar los Casos de Uso en busca de sustantivos, ya que cada sustantivo puede generar un candidato a una clase potencial…"* (p. 308). Tanto Jim Arlow e Ila Neustadt (2005) como el ya citado Roger Pressman reconocen también la existencia de varias técnicas para la identificación de clases, similares a las propuestas anteriores, pero no es explícito el vínculo con los Casos de Uso.

Es necesario observar que la mayoría de estas estrategias están inspiradas en un trabajo de Russell Abbott (1983), denominado Análisis Textual, según el cual: a) los nombres son candidatos a quedar representados por clases, b) los verbos representan operaciones y c) los adjetivos representan atributos.

Ahora bien, al margen de las opiniones que merezcan la rigurosidad de las técnicas propuestas para la identificación de objetos, está claro que se apoyan en una especificación escrita de los requerimientos y no está tan claro cómo se las vincula a los Casos de Uso definidos en el proceso de análisis. Al igual que en otras técnicas, como el "brainstorming" o el "role playing" propuestos por David Bellin y Susan Suchman (1997), todas ellas procuran identificar primero los objetos, para luego asignarles sus atributos y sus métodos, en un proceso que podría denominarse de explosión de propiedades. Finalmente, definidos los objetos, se procederá a identificar las clases.

Aquí cabe destacar que la mayoría de los libros de texto citados en este artículo, al igual que muchos otros, son actualizados regularmente con nuevas ediciones, algunos de los cuales lo vienen haciendo desde hace más de diez años. Sin embargo, una vez propuesta la construcción del modelo funcional con Casos de Uso y el modelo de diseño con clases y objetos, se mantienen exactamente las mismas recomendaciones ya comentadas y poco específicas para establecer un vínculo entre ambos. Es decir, al no ponerse el foco en este tema parece haber una resignación a aceptar el "gap" semántico entre ambos modelos.

Lo expuesto haría pensar que el problema no tiene solución, y en realidad no es así, ya que hay numerosas propuestas para superar racionalmente el "gap" semántico, pero con muy escasa repercusión en la industria del software. Muchas de éstas recurren a un enfoque implosivo, donde a partir de un modelo funcional detallado se identifican las responsabilidades, permitiendo éstas reconocer los métodos a ser provistos, que son los que conducen finalmente a la definición de los objetos y las clases. Entre muchas otras, pueden citarse las propuestas de Biddle, Noble y Tempero (2002) y la de uno de los autores de este artículo (Giró, 2007).

El interrogante es obvio: ¿habiendo otras opciones, por qué se persiste en el enfoque explosivo tradicional, que inevitablemente conduce a un gap que impacta desfavorablemente en el desarrollo del sistema? La respuesta también es obvia: el costo de un diseño implosivo, que necesariamente debe ser muy detallista, es mucho mayor que el explosivo, que comienza identificando los objetos, en muchos casos a partir de los sustantivos presentes en el texto de las especificaciones. La consecuencia es inevitable: buscando reducir costos se sacrifica la posibilidad objetiva de responder a las especificaciones a través del modelo de diseño óptimo para atender cada problema.

Se justifica aquí un breve análisis comparativo. El costo del diseño forma parte del costo de desarrollo de cada sistema, y al optarse por diseños implosivos se alcanzan montos más elevados. Como contrapartida, se reducen los costos de las pruebas (testing), especialmente las de integración, y los costos correspondientes a los esfuerzos

de depuración. Por su parte, los diseños explosivos tendrán un costo muy inferior. Sin embargo, sus consecuencias se traducirán en importantes costos operativos, que reconocerán tres vertientes: *i)* mayores costos de funcionamiento, *ii)* necesidad de superar las desfavorables consecuencias de las fallas del sistema y *iii)* mayores dificultades de mantenimiento. En estos casos los costos operativos merecerían la denominación de "pasivos técnicos", ya que sus montos son difícilmente previsibles y los tendrán que afrontar los usuarios a lo largo de toda la vida útil de los sistemas. Se trata de un cabal testimonio de que tarde o temprano los reales costos se terminan pagando.

Para completar este breve análisis comparativo, hay que destacar que, al no superarse en el diseño explosivo el gap semántico entre el modelo funcional proveniente de las especificaciones y el modelo de objetos, se carece de toda posibilidad de establecer un vínculo natural y fluido entre ambos, lo que como mínimo impedirá que el sistema exhiba una de las propiedades consideradas esenciales, que es la trazabilidad. Tampoco es posible asegurar que el sistema alcanzará las restantes propiedades esenciales: *ii) consistencia, iii) completitud, iv) corrección, v) precisión y vi) no ambigüedad*. Todas ellas deberían ser posibles de validar desde las tres dimensiones ortogonales (independientes) de un sistema: funcional, estática o estructural y dinámica.

Lamentablemente, el tiempo diluye los conceptos y afianza la idea que el problema es inevitable. Peor aún, lo que nunca encontró una respuesta clara es disimulado mediante herramientas "CASE" (Computer Aided Software Engineering) que se "hacen cargo" a través de heurísticas, "wizards" y artificios de suplir las carencias originadas por las limitaciones ya expuestas.

## Actualidad y expectativas futuras

Los tres argumentos presentados demuestran que el tema tratado no admite respuestas definitivas, pero no cabe dudas que su análisis es muy necesario. En efecto, se trata de un disparador para reflexionar sobre alguna de las posibles razones para que, aún hoy, haya dudas sobre si la Crisis de Software verdaderamente se ha hecho crónica y convivimos con ella, relativizando la efectividad y los beneficios que ha demostrado hasta ahora la Ingeniería de Software.

La actividad humana se desarrolla en condiciones en el que el software ocupa un lugar cada vez más destacado, que está caracterizado por: *a)* Creciente participación en todos los ámbitos del quehacer humano, *b)* Ininterrumpido crecimiento del volumen, diversidad y complejidad de los productos en los que el software está involucrado, *c)* Mayores exigencias de calidad como consecuencia de aplicaciones cada vez más críticas, *d)* Pronunciada escasez de personal calificado en un mercado laboral de alta demanda, *e)* Ciclos de vida cada vez más breves debido a la creciente volatilidad del mercado, *f)* Permanente evolución de la tecnología, ofreciendo

nuevos recursos y renovados escenarios para la operación de los sistemas y *g)* Mayor flexibilidad de los productos de software, buscando dar respuestas a un mercado dirigido por la demanda en vez de la oferta. Mientras tanto, la comunidad se apoya y confía su actividad y bienestar en los bienes y servicios que se le ofrecen, lo que representa una enorme responsabilidad ética y social para todos los que estamos involucrados en el desarrollo de este producto intangible llamado software. Todo lo expuesto representa motivos por demás suficientes para atender prioritariamente este problema y estimular la búsqueda de soluciones, tanto perfeccionando las líneas de investigación conocidas como también aportando ideas innovadoras.

## Bibliografía

Abbott, R. (1983). Program design by Informal English Descriptions. *Communications of the ACM, 26(11)*, 882-894. Arlow, J. y Neustadt, I. (2005). *UML 2 and the Unified Process*. México: Anaya.

Barbacci, M. et al. (1995). Quality Attributes, *Technical Report CMU/SEI-95-TR-021*, 56. Pittsburgh, USA: Software Engineering Institute of Carnegie Mellon University.

Bauer, F. (1968). Report of a conference sponsored by the NATO Science Committee. *NATO Software Engineering Conference*. Garmisch, Germany: P. Naur and B. Randell.

Biddle, R., Noble, J. y Tempero, E. (2002). Essential Use Cases and Responsability in Object-Oriented Development*, Proc. of the Australasian Computer Science Conference*. Melbourne, Australia.

Bellin D. y Suchman, S. (1997). *The CRC Card Book*. USA: Addison-Wesley.

Booch, G., Rumbaugh J. y Jacobson I. (2014). *UML: El Lenguaje Unificado de Modelado*. España: Pearson - Addison Wesley.

Bourque P. y Failey R. (2014). Swebok: Guide to the Software Engineering Body of Knowledge v3. *IEEE Computer Society*.

Brooks, F. (1987). No Silver Bullet, Essence and Accidents of Software Engineering. *Computer Magazine, 20(4)*, 10-19.

DeMarco, T. (1982). *Controlling Software Projects: Management, Measurement, and Estimation*. USA: Prentice Hall - Yourdon Press.

Dijkstra, E. (1972). The Humble Programmer, ACM Turing Award Lecture (EWD340). *Communications of the ACM, 15(10)*, 859-866.

Douglass, B. (1999). *Ropes: Rapid Object-Oriented Process for Embedded Systems*. USA: I-Logic.

Fenton N. y Bieman J. (2014). *Software Metrics: A Rigorous and Practical Approach*. CRC Press. USA: Chapman & Hall.

Fenton N. y Neil M. (1999). Software metrics: successes, failures and new directions. *The Journal of Systems and Software*, 47, 149-157.

Fernandez, J. y Lilius, J. (2004). Functional and Object-Oriented Views in Embedded Software Modeling, *Proc. of the 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, 378-387.

Fraser S. y Mancl D. (2008). No Silver Bullet: Software Engineering Reloaded. IEEE Software, 25(1), 91-94.

Gibbs W. (1994). Software's Chronic Crisis. *Trends In Computing. Scientific American, 86*.

Giró, J. (2007). Definición de modelos de objetos a partir de sus responsabilidades. *JIDIS: Jornadas de Investigación y Desarrollo en Ingeniería de Software*. Córdoba, Argentina.

Glass, R. (1994). A tabulation of topics where software practice leads software theory. Journal of Systems Software, 25, 219-222.

Jacobson, I. (1994). Use Cases and Objects, *Beyond Object Orientation, 1(4)*.

Kendall, K. y Kendall, J. (2014). *Análisis y Diseño de Sistemas*. México: Pearson.

Larman, C. (2004). *Applying UML and Patterns: an introduction to Object Oriented Analysis and Design and Interactive Development*. USA: Addison Wesley Professional.

Naguib H. y Li Y. (2010). Applying Software Engineering Methods and Tools to Computational Science and Engineering Research Projects. *International Conference on Computational Science*, ICCS, 1505-1509.

Pressman R. (2010). *Ingeniería del Software, un enfoque práctico*. México: Mc. Graw Hill.

Segal, J. (2008). Scientists and software engineers: a tale of two cultures, *Proceedings of the 20th Annual Meeting of the Psychology of Programming Interest Group*. Lancaster University, Lancaster, UK.

Sommerville I. (2012). *Ingeniería de Software*. México: Pearson - Addison Wesley.