La belleza del software

Dra. Alicia Pérez[32]
Facultad de Ingeniería e Informática
aperez@ucasal.net

Resumen

Cuando alguien aprende a programar, a menudo se encuentra con reflexiones sobre la "belleza" de algunas soluciones clásicas a problemas de programación. Aunque la elegancia del código pase a segundo plano cuando uno se embarque en proyectos más grandes, hay razones importantes, y no sólo académicas, para buscar la elegancia del software. Este artículo pretende fundamentar dichas razones.

1. Introducción

Cuando alguien aprende a programar, a menudo se encuentra con reflexiones sobre la "belleza" de algunas soluciones clásicas a problemas de programación. Este el caso, por ejemplo, de la elegancia de la secuencia de Fibonacci, uno de los ejemplos clásicos utilizados para aprender recursividad. Aunque la elegancia del código pase a segundo plano cuando uno se embarque en proyectos más grandes, o no sienta la presión de que alguien va a evaluar académicamente el trabajo, como profesora me gustaría que permaneciera como aprendido que a menudo las soluciones bellas, o elegantes, son soluciones buenas.

Mi motivación para escribir este artículo es variada. Me sorprendió leer un artículo reciente en la muy citada y respetada revista Communications de la asociación profesional ACM titulado "El software

[32] La autora es Licenciada en Informática por la Universidad Politécnica de Madrid y PhD in Computer Science por Carnegie Mellon University. Actualmente se desempeña en la Facultad de Ingeniería e Informática de la UCS como docente de Sistemas Expertos y de Compiladores y como Jefa del Departamento de Investigación y en la Universidad Carlos III de Madrid (España) como docente de Inteligencia Artificial (2004, 2006) de la carrera de Ingeniería Informática - grupo bilingüe.

como Arte" (Bond 05) y pensé que una reflexión sobre ese tema podría ser interesante en el entorno de nuestra carrera y nuestros estudiantes. Por otra parte, he intentado motivar a mis alumnos de Inteligencia Artificial, en el último año de su carrera, a que pensaran soluciones elegantes para un problema de programación relativamente complejo, prometiendo que si las encontraban su trabajo de programación y depuración de errores podría simplificarse enormemente. Me sorprendía enormemente su facilidad para usar código spaghetti, no poner comentarios y en general parchear sus soluciones sin saber exactamente al final qué es lo que estaban intentando hacer. Y todo esto después de haberles proporcionado en pseudocódigo una idea elegante como base para el algoritmo. Como practicante y estudiosa de la informática, y amante de la ciencia en esta disciplina, me cuesta aceptar este desinterés. Mi convicción, y la de muchos de mis compañeros en proyectos a lo largo de los años, he de admitir que en el campo de la investigación fundamentalmente, es que hay razones importantes, y no sólo académicas, para buscar la elegancia de nuestro software. En este artículo voy a intentar fundamentar esta convicción.

2. Belleza y elegancia del código

Un problema para este análisis es determinar cuál es el concepto de "bello" o "elegante" cuando se aplica al código. Es obvio que no siempre todas las opiniones van a coincidir. Cuánto y cómo comentar el código es un ejemplo clásico de esto, que nunca parece quedar obsoleto (Raskin 05). Una discusión reciente en Slashdot, un visitado sitio Web para programadores, mostraba con ejemplos y encendidas opiniones los distintos puntos de vista sobre qué es código bello. Por cierto que esta discusión apareció motivada por el siguiente mensaje, que puede ser relevante para nosotros como educadores de los nuevos informáticos:

> ¿Dónde puedo encontrar código bello? Posted by Michael on Wed Jan 24, '01 from the not-in-anything-I-write-that's-for-sure dept.
> eGabriel escribe "Uno de los beneficios del software libre que todavía no he visto explorar aquí [slashdot.com] es la oportunidad de estudiar código elegante, magistral. Además de poder compartir y disfrutar distintas aplicaciones, y reutilizar su código fuente, también es posible sencillamente descargar el código y verlo por placer, para aprender de los maestros de

> este arte. Desde luego hay criterios diversos para determinar qué hace que un fragmento de código sea excelente o bello, y no estoy interesado en discutirlo. Sin embargo, si alguien ha encontrado un fragmento de software libre que sirva como un caso de estudio excelente por cualidades valoradas por los programadores, me gustaría leer el código y así dejarme educar. Sería igualmente interesante ver código que sea realmente malo, siempre y cuando esto no se convierta en ataques directos contra los programadores involucrados. Cualquier código que demuestre un diseño elegante y magistral sería material de lectura excelente; no importa tanto en qué lenguaje esté escrito. Si se trata de código 'literato'[33], mucho mejor." (Michael 05)

Creo que el autor hace un comentario interesante, expresado también en las palabras de Grady Booch, autor de una conocida metodología para el diseño orientado a objetos y uno de los creadores de UML (Unified Modeling Language). En su blog Booch escribe:

> "Es un gusto estudiar el código fuente de MacPaint (escrito en Object Pascal). Don Knuth dice que es uno de los programas más bellos que ha leído. Saben, es curioso que en la mayoría de las disciplinas se aprende estudiando los artefactos de otros maestros, pero todavía no he visto un curso de informática que se titule *Lecturas en Software (código)*. Afortunadamente hay un nuevo libro que por fin trata de esto y que recomiendo encarecidamente, *Code Reading*, de Diomidis Spinellis. El trabajo de Don sobre programación literata también es una buena fuente." (Booch 04)

Booch habla también de su colaboración con el Museo de Historia de la Informática para conservar software clásico y está trabajando en un libro, el Handbook of Software Architecture, que pretende recoger las arquitecturas de una amplia colección de sistemas software, presentándolos de forma que puedan apreciarse los patrones esenciales y establecer comparaciones. Este libro está disponible en la Web (www.booch.com/architecture) y abierto a un enfoque colaborativo para su creación a medida que vaya avanzando. En la presentación de este libro, Booch insiste en que una señal de la madurez de una

[33] 'Literato' se refiere a código escrito siguiendo la metodología de 'literate programming' que describiremos más adelante. La traducción como 'programación literata' es la que aparece en Wikipedia (http://es.wikibooks.org/wiki/Agentes_de_chat#Literate_Programming, 21 Feb 2006)

disciplina de ingeniería es que se puedan nombrar, estudiar y aplicar los patrones relevantes a su dominio. Por ejemplo, en la ingeniería civil pueden estudiarse los elementos fundamentales de la arquitectura mirando obras que exponen y comparan distintos estilos. Sin embargo no existe un trabajo similar de referencia en la ingeniería del software.

La conversación provocada en Slashdot.com por el mensaje anterior fue archivada por este sitio después de 370 mensajes en diecisiete horas. Quedó claro que los desarrolladores que escribieron están apasionados por su trabajo y tienen opiniones claras sobre lo que es agradable y lo que es insoportable. Como Bond indica, “perciben una estética que subyace a cómo encaran el desarrollo de software y que les da realimentación instintiva en cada etapa del desarrollo”. Como consecuencia es obvio preguntar en qué se basan estos juicios estéticos, y si se basan simplemente en cuestiones propias de la informática o por el contrario el software puede llegar a cruzar la frontera utilitaria para convertirse en arte.

3. El software como arte

En los últimos años esta discusión se ha extendido más allá de los informáticos a los artistas que usan nuevos medios. Algunos festivales de arte moderno ya dan premios a “software artístico” y algunos museos dedican espacio a este tipo de obras. Por ejemplo, el festival Transmediale 2001 de Berlín (www.transmediale.de) fue el primero en ofrecer un premio (US$4.700) a una obra de software artístico, con la condición de que además de funcionar, tendría que proporcionar una experiencia que vaya más allá de su aplicación práctica. El festival READ_ME 1.2 de Moscú en 2002 estuvo dedicado únicamente a software y se buscaba “arte cuyo material sea código de instrucciones algorítmicas y/o que se refiera a conceptos culturales del software”. Este festival se ha repetido en distintos países en años posteriores (readme.runme.org). Se trataría según Bond de la primera generación de arte software, que además está siendo prácticamente ignorada por los informáticos, quienes tal vez serían los más capaces de apreciar este medio y su capacidad de expresión. Este artículo no pretende profundizar más estos aspectos.

4. Donald Knuth: El Arte de la Programación

Al hablar de la programación como arte, inmediatamente viene a la mente el nombre de Donald Knuth, una figura histórica de la informática, a quien muchos consideran el creador del análisis de algoritmos, y quien ya en los años 60 había comenzado la publicación de su colección "The Art of Computer Programming" en siete volúmenes.[34] Knuth recibió el prestigioso Turing Award en 1974. En aquel momento eligió el tema "La programación como arte" para su conferencia al recibir el premio (Knuth 1974). A continuación revisamos algunos aspectos de aquél trabajo.

4.1 Obras de arte

Cuando Knuth habla de programación como arte, en primer lugar se refiere a una forma artística, en un sentido estético. Su meta principal como educador y autor es "ayudar a otros a aprender a escribir *programas bellos*. … Siento que cuando preparamos un programa puede ser como componer poesía y música". Para Knuth, citando a Andrei Ershov, poder dominar la complejidad y establecer un sistema de reglas consistentes son logros considerables capaces de dar satisfacciones intelectuales y emocionales. Knuth continúa:

> "Además cuando leemos los programas de otras personas podemos reconocer algunos como auténticas obras de arte. Todavía recuerdo la emoción de leer el listado del programa en ensamblador SOAP II de Stan Poley… y ver lo elegante que podía llegar a ser un programa de sistemas, especialmente al compararlo con el código pesado de otros listados de la época. La posibilidad de escribir programas bellos, hasta en ensamblador, fue lo que me hizo engancharme a la programación. Algunos programas son elegantes, algunos son exquisitos, algunos son brillantes."

[34] Para saber más de Don Knuth puede consultarse http://es.wikipedia.org/wiki/Donald_Knuth o visitar directamente sus páginas Web en la Universidad de Stanford, donde ha sido profesor y ha influenciado generaciones de informáticos e investigadores. Estas páginas http://www-cs-faculty.stanford.edu/~knuth/ describen sus variados proyectos, aportes e intereses. Un detalle pintoresco es que ofrecía recompensas de US$2.56, correspondiente a un $ hexadecimal, a quienes encontraran erratas en sus libros.

4.2 Gusto y estilo

Knuth indica que no hay un único “mejor estilo”, que cada uno tiene sus propias preferencias y que no se le puede obligar a algo que no le es natural y negarle un “placer completamente legítimo”. Aunque hay cosas que otros programadores hacen y él nunca haría, Knuth no se atreve a juzgarlas ya que “lo importante es que están creando algo que ellos piensan que es bello”. Otro pionero, Edsger Dijkstra, compara el enseñar programación con el enseñar composición musical. El profesor no debe enseñar a sus alumnos a componer una sinfonía particular, sino ayudarles a encontrar su propio estilo, explicándoles las consecuencias. Esa analogía llevó a Dijkstra a titular su libro “El Arte de la Programación” (Dijkstra 1971). Knuth, en sus famosas conferencias de 1999 en MIT sobre “Dios y las computadoras” (Knuth 1999) compara la belleza de un programa con la belleza en música o literatura: programas donde las tareas están elegantemente indicadas y cuyas partes se unen sinfónicamente. Así, unos programas pueden ser ingeniosos y otros causar “dolor” al que los lee (ciertamente un dolor de cabeza).

No obstante, Knuth propone algunos principios estéticos más importantes que otros, por ejemplo, la “utilidad” del resultado: “es especialmente agradable que lo que nosotros consideramos bello sea además considerado útil por otros”. Knuth indica además otros sentidos en que se puede identificar a un programa como “bueno”. Primero, que funcione correctamente. Segundo, que sea fácil de modificar cuando sea necesario. Ambos aspectos se consiguen si el programa es fácil de leer y de entender. Otro aspecto de un buen programa es que “interactúe bien con sus usuarios, especialmente al recuperarse de errores humanos en los datos de entrada” utilizando mensajes de error útiles o diseñando formatos flexibles para los datos de entrada.

Otro aspecto importante de la calidad de un programa es que use eficientemente los recursos. Ya en 1974 Knuth indicaba “siento decir que hoy en día muchos condenan la eficiencia de un programa, diciéndonos que es de mal gusto. La razón de ello es que estamos experimentando una reacción a una época en que la eficiencia era el único criterio de calidad. … El problema real es cuando una preocupación excesiva por la eficiencia en los lugares equivocados en los momentos equivocados; la optimización prematura es la raíz de todos (o casi todos) los males en programación”.

Knuth encuentra satisfacción estética al conseguir algo con herramientas limitadas. “El programa del que estoy más orgulloso es un compilador que escribí para una minicomputadora primitiva que tenía sólo 4096 palabras de memoria, cada una de 16 bits”. Knuth observaba en 1974 que los mayores avances de programación de esa época surgieron de personas que no tenían acceso a computadoras muy grandes. Para él la moraleja es que deberíamos usar el concepto de recursos limitados en la educación, ya que el lujo tiende a hacernos letárgicos. “El arte de atacar miniproblemas con toda la energía agudizará nuestro talento para los problemas reales. … No deberíamos abandonar el arte por el arte; no deberíamos sentirnos culpables de hacer programas sólo por divertirnos”. Relata la experiencia del entusiasmo y el aprendizaje de un grupo de sus alumnos en Stanford cuando fueron capaces de escribir un programa en Fortran y otros lenguajes capaz de producirse a si mismo como salida.

4.3 Crear herramientas bellas para otros

Knuth termina su discurso pidiendo a los diseñadores que nos den herramientas bellas para trabajar con ellas: “por favor, dennos herramientas con las que trabajar sea un placer, especialmente para tareas rutinarias, en lugar de proporcionarnos cosas con las que tenemos que pelear. Por favor, dennos herramientas que nos animen a escribir mejores programas. Es muy difícil convencer a alumnos de primer año de que programar es hermoso cuando lo primero que les tenemos que decir es cómo perforar ‘barra barra JOB igual etc’ ”. Entre las herramientas con estas características están aquellas que permitan al programador usar su creatividad sin forzarle con una excesiva automatización; las que ayudan a medir la eficiencia de sus programas y los cuellos de botella reales (en lugar de sólo proporcionar compiladores que secretamente operan optimizando el código sin explicar cómo); y finalmente lenguajes de programación que fomentan buen estilo en la programación. Knuth escribía esto en 1974 en pleno surgimiento de la programación estructurada.

4.4 Programación literata

Merece la pena mencionar que Knuth, unos veinte años después de recibir el premio Turing, desarrolló lo que se dio en llamar “programación literata”.

> Creo que ha llegado el momento para que la documentación de los programas sea significativamente mejor, y la mejor forma para conseguir esto es considerando que los programas son obras literarias. Por eso mi título: “Programación literata”. (Knuth 1984)

Se trata de un enfoque para el desarrollo de programas con el objetivo de que escribir y leer software sea más agradable. La Wikipedia indica que es “cierta metodología para el desarrollo de programas que consiste en darle prelación a la documentación de un programa frente el código… Se concentra en describir el problema, las posibles soluciones, la descomposición del programa en fragmentos más sencillos, la descripción de cada fragmento y, finalmente, el código de cada fragmento” (Wikipedia 06). El primer paso es explicar a una persona (en lugar de a la máquina) qué es lo que se quiere que haga el programa y se va mezclando el lenguaje natural (usado en la documentación) con el lenguaje de programación en un solo archivo fuente. Knuth desarrolló el primer entorno para programación estructurada en 1981. Lo llamó WEB, según él porque era una de las pocas palabras en inglés de tres letras no utilizadas hasta entonces en informática, y lo utilizó para su popularísimo sistema TeX.

5. ¿Qué hace que un programa sea bello?

Ciertamente muchos programadores de hoy estarían de acuerdo con estos pensamientos de Don Knuth. El mismo Grady Booch propone algunos aspectos básicos en su blog ya mencionado.

> ¿Qué hace que el software sea bello? La era *dot bomb* estaba llena de modelos de negocios estúpidos, pero al mirar hacia atrás nos dijo una cosa: la ingeniería sólida nunca pasa de moda. Para mí, esto significa construir sistemas software que sigan tres principios básicos: habilidad precisa y abstracciones flexibles, mantener los distintos objetivos separados, y crear una distribución de responsabilidades equilibrada (Booch 04)

¿Qué es entonces el código bello? Obviamente no es suficiente que funcione o tenga éxito comercial, que sea una de las llamadas “killer apps”. Mathew Heusser publicó en el leído sitio Dr. Hobbs (Heusser 05) algunas características que ha encontrado en código que

le parece bello. Las reproducimos a continuación complementadas con otros aportes.

1. El código bello es fácil de leer. Es terrible tener que mantener y modificar código que no se entiende. Una de los aspectos de la legibilidad del código es que las funciones no deberían ser demasiado largas ni tener demasiados argumentos. Para determinar cuál puede ser el número de variables razonable que se pueden manejar, Heusser indica que la ciencia militar e incluso el ajedrez nos enseñan que la mayoría de las personas sólo podemos mantener en la mente un máximo de 3 a 7 variables. A partir de ahí, les perdemos la pista, las olvidamos y por tanto es fácil que el software contenga errores. Por esta razón si el código es demasiado complejo, merece la pena descomponerlo en unidades más pequeñas. Aunque Heusser no lo indica, esta cifra máxima está sustentada en uno de los artículos fundamentales de la psicología cognitiva, escrito por George Miller en los años cincuenta y titulado “The Magical Number Seven, Plus or Minus Two: Some limits on our capacity for processing information” (Miller 56). Miller mostró que el número de elementos discretos de información que los seres humanos pueden mantener en la memoria a corto plazo es siete, más o menos dos. (De paso merece la pena mencionar que el trabajo de Miller es la razón por la que los números de teléfono locales en EEUU tienen siete dígitos.) (Raymond 04).

2. El código bello tiene un foco central. El código debería hacer solamente una cosa. Esta es la idea detrás de, por ejemplo, las arquitecturas modelo-vista-controlador que separan en componentes diferentes el modelo de datos, la interfaz de usuario y la lógica de control.[35] Así cualquier modificación a alguno de los componentes tiene un impacto mínimo en los otros. Si se mezclan estos componentes se dificultan la prueba (testing) del software y su reutilización. La sencillez y la generalidad (hacer una sola cosa pero hacerla bien) son características del código elegante.

Raymond, al describir el estilo de los programadores de Unix, especialmente sus creadores, habla de una propiedad relacionada, que llama ortogonalidad (Raymond 04, cap.4). “En un diseño puramente ortogonal, las operaciones no tienen efectos laterales; cada acción cambia una sola cosa sin afectar a otras. Hay una única manera de

[35] El MVC (Modelo-Vista-Controlador) surgió como parte de la interfaz de ventanas de Smalltalk en el laboratorio de investigación Xerox Parc en 1979.

modificar cada propiedad de cualquier sistema que se esté controlando". Por ejemplo, un monitor tiene controles ortogonales. Se puede modificar el brillo independientemente del contraste y del balance de color. Sería mucho más difícil ajustar el monitor si el control del brillo modificara el balance de color, ya que habría que ajustar el segundo cada vez que se modificara el primero. Raymond da un ejemplo frecuente de esta clase de error de diseño: código que lee y traduce datos de un formato (fuente) a otro (destino). Si el diseñador piensa que el formato fuente siempre se va a almacenar en un archivo de un disco puede que escriba la función de conversión que abra y lea de un archivo con nombre, mientras que la entrada podría provenir de otras fuentes. Con un diseño ortogonal la misma función de conversión podría utilizarse con diversas fuentes que no involucren la apertura de archivos. La ortogonalidad reduce el esfuerzo de testeo y desarrollo porque el código que no produce efectos laterales o depende de los efectos de otro código es más fácil de verificar o reemplazar en el sistema.

Un concepto muy relacionado al de ortogonalidad es el de refactorización (del inglés *refactoring*). Se trata del proceso de reescribir un programa para mejorar su estructura o legibilidad, sin cambiar su significado o comportamiento externo (Wikipedia 06). Aunque el concepto no es nuevo, darle nombre e identificar técnicas para hacerlo ha contribuido a un interés por el tema. Fowler, al introducir este concepto (Fowler 99), proponía eliminar las repeticiones de código y otros "malos olores", muchos de los cuales estaban relacionados con violaciones del principio de ortogonalidad.

Otra regla relacionada con la ortogonalidad ha dado en llamarse "no te repitas a ti mismo" o SPOT (único punto de verdad, o *Single Point of Truth* en inglés).[36] Cada elemento de conocimiento debe tener una representación única, autoritaria, en el sistema. Las repeticiones pueden generar inconsistencias, ya que hay que cualquier cambio ha de realizarse en todas las instancias de la repetición. A menudo las repeticiones indican que el código o las estructuras de datos no están bien pensados.

Una distinción útil en este aspecto se da entre código que es producto de un proceso de formalización y código que es fruto de

[36] La primera denominación proviene del libro *The Pragmatic Programmer: From Journeyman to Master*, de Hunt y Thomas, Addison-Wesley, 2000. La segunda se atribuye a Brian Kernighan, uno de los creadores del lenguaje C.

heurísticas y sucesivos parches para casos especiales. Un diseño es compacto (Raymond) cuando está organizado en torno a un algoritmo que responde a una definición clara del problema. Lo opuesto a un enfoque formal es el uso de heurísticas, que llevan a una solución que es correcta con cierta probabilidad pero no con certeza. Estas reglas son necesarias cuando el problema o su solución incluyen incertidumbre y no determinismo. Por ejemplo, un algoritmo demostrablemente correcto para filtrar *spam* sería demasiado complejo, precisando por ejemplo la resolución del problema de la comprensión del lenguaje natural. El objetivo de las heurísticas es mejorar la eficiencia o la sencillez conceptual, aunque sea a costo de precisión o exactitud de la solución. En general tienden a aumentar el número de casos especiales a considerar. Al aumentar el tamaño del problema, aumenta la complejidad y puede que se llegue a perder la ventaja en eficiencia por la complejidad del código.

3. El código bello puede probarse. Una función que esté bien definida debería tener una salida definida y clara. Sin embargo, muchas funciones están escritas de manera tal que su comportamiento no está bien definido en situaciones inesperadas.

4. El código bello es elegante. A veces se dice que un charlatán hace que lo sencillo parezca difícil de entender, mientras que un genio hace que lo complejo sea fácil de entender. Para Heusser, si un problema difícil tiene una solución sencilla, por ejemplo mediante el uso de recursión, estamos ante código elegante. El clásico algoritmo de exclusión mutua de Dijkstra para serializar el acceso a datos compartidos por procesos concurrentes, es un ejemplo de elegancia (Bond 05).[37] Lo opuesto es código desorganizado, lleno de errores, largo y liado como “espagueti”. Heusser cita a Antoine de Saint Exupery:

> Parece que todo el esfuerzo industrial del hombre, todos sus cálculos, todas las noches en vela encima de sus planos, sólo conduzcan de modo visible a la sencillez. Parece que se necesite toda la experiencia de varias generaciones para perfilar lentamente la curva de una columna, de un casco de barco, de un fuselaje de avión, para lograr la pureza primigenia de un seno o de un hombro. Parece que el trabajo de los

[37] El algoritmo original apareció en Dijkstra, E.W. “Solution to a problem in concurrent control”, *Communications of the ACM* 8, 9 (Sept. 1965).

> ingenieros, de los delineantes, de los analistas del centro de estudios, consiste, aparentemente, en borrar y pulir, en aligerar aquel empalme, equilibrar esta ala hasta que ya no se la note, hasta que ya no sea un ala incrustada en un fuselaje, sino una sola forma que, perfectamente lograda, se ha desprendido de su ganga, una forma que sea como un conjunto misteriosamente ensamblado, espontáneo como un poema. *Parece que la perfección se alcanza no ya cuando no quede nada por añadir, sino cuando no queda nada por suprimir. (Tierra de Hombres,* énfasis mío*)*

Heusser acaba sugiriendo que en lugar de buscar medidas artificiales de la eficiencia del código, deberíamos buscar la belleza en el código, "porque creemos que las cosas bellas son mejores… El código bello tiende a ser más corto, por lo que se podría escribir software más potente en menos tiempo. Tiende a ser testeable, por lo que podemos mejorar su calidad y mantenerla con mayor confianza. Porque hace una sola cosa y está bien dividido, se presta a ser reutilizado."

Según Raymond, la elegancia de la que hemos hablado ha estado presente en el diseño de Unix desde sus comienzos. Kernigham y Ritchie lo expresan así en el primer artículo sobre Unix, que apareció en 1974 en *Communications of the ACM (*"The UNIX Time-Sharing System", vol. 17, No. 7 pp. 365-375): "… las restricciones han fomentado no sólo economía sino también una cierta elegancia en el diseño". ¿Será que al no haber tales restricciones en los sistemas actuales se ha perdido el interés por la elegancia en el diseño? En la portada del libro *La Práctica de la Programación* Kernigham y Pike (Prentice Hall, 1999) listan "sencillez, claridad y generalidad" como virtudes deseables.

6. Conclusión

El prestigioso premio Turing 2006 ha sido recientemente otorgado a Peter Naur, uno de los diseñadores principales del lenguaje ALGOL 60. En la nota que describe las razones de esta elección, se da énfasis a “la elegancia y la sencillez” del diseño de este lenguaje pionero y que sirvió de modelo para tantos lenguajes de programación posteriores. El jurado, presidido por el Presidente de la compañía Intel, indicó que “este premio debería animar a futuros diseñadores de lenguajes que enfrentan los mayores desafíos de programación actuales, tales como la computación multi-threaded de propósito general, a conseguir el mismo nivel de elegancia y sencillez que fueron la marca de ALGOL 60” (ACM 06). Sirvan estas anécdotas, ejemplos y opiniones de los maestros, como dice Brooch, para animar, y hasta entusiasmar, a nuestros futuros informáticos.

Bibliografía

ACM (Association for Computing Machinery), “Software pioneer Peter Naur wins ACM's Turing Award”, Nota de prensa, http://campus.acm.org/public/pressroom/index.cfm

Bond, Gregory W., “Software as Art”, en *Communications of ACM*, August 2005/Vol. 48, No. 8, pp. 118-124

Booch, Grady, Blog del 7 Junio 2004, http://www-128.ibm.com/developerworks/blogs/dw_blog_comments.jspa?blog=317&entry=51348&roll=-5.

Dijkstra, Edsger, *A short introduction to the art of programming*, Prefacio, EWD-316, T.H. Eindhoven, 1971.

Fowler, Martin, *Refactoring,* Addison-Wesley. 1999

Heusser, Matthew, “Beautiful code”, www.ddj.com/184407802 Agosto 2005.

Knuth, Donald E., “Computer Programming as an Art”, *Communications of the ACM*, 17 12, December 1974, pp. 667-673.

Knuth, Donald E. “Literate Programming”, *The Computer Journal* 27 (1984), pp. 97-111. También en www.literateprogramming.com.

Knuth, Donald E., “Things a Computer Scientist Rarely Talks About” http://www-cs-faculty.stanford.edu/~knuth/things.html

Michael (anon)
http://ask.slashdot.org/article.pl?sid=01/01/25/0230208&tid=156 (visitado 1 Abril 2005)

Miller, George, “The Magical Number Seven, Plus or Minus Two: Some limits on our capacity for processing information”, *The Psychological Review*, 1956. 63. pp. 81-97.

Raskin, Jef, “Comments are more important than code”, *ACM Queue*, vol. 3, no. 2, Marzo 2005.

Raymond, Eric S., *The Art of UNIX Programming,* Pearson Education, Boston, 2004.

Wikipedia colaboradores, "Literate programming," Wikipedia, The Free Encyclopedia, http://en.wikipedia.org/w/index.php?title=Literate_programming&oldid=54477834 (visitado Mayo 23, 2006).

Wikipedia colaboradores, "Refactoring," *Wikipedia, The Free Encyclopedia,* http://en.wikipedia.org/wiki/Refactoring (visitado Mayo 23, 2006).