Capítulo 1

El humilde programador

> "*If you carefully read its literature and analyse what its devotees actually do, you will discover that software engineering has accepted as its charter: How to program if you cannot.*"
>
> —Edsger W. Dijkstra
> *By way of introduction, 1989 [1]*

El título de este primer capítulo es un pequeño homenaje al texto homónimo que Edsger Dijkstra presentó ante la ACM[2], con motivo del Premio Turing con el que fue galardonado en 1972. En esta lectura Dijkstra describe cómo ya a mediados de la década de 1950 la profesión de programador era una actividad poco reconocida [2].

1.1 El desprecio por la programación

Se dice que no es labor de un ingeniero *picar código*, como tampoco es labor de un arquitecto colocar un ladrillo tras otro. El uso de la expresión *picar código* es síntoma del desprecio existente por la programación. La ingeniería de software tradicional presenta la programación como una actividad de construcción, mecánica, de mera traducción trivial desde lo que plasma una especificación al lenguaje de la computadora. Teóricamente, la verdadera actividad creativa o intelectual del desarrollo de software se encuentra en las fases de planificación estratégica, análisis de negocio y diseño técnico, encauzadas en un proceso formal cuyo resultado es una

[2] *Association for Computing Machinery*

especificación. La programación se considera por tanto como una actividad efectuada por recursos reemplazables, que no aportan valor añadido a la empresa.

Esta visión de la programación como una actividad de construcción ha sido consecuencia, entre otras causas, de la gran difusión que han tenido algunas obras relacionadas con los procesos formales de la ingeniería de software, como por ejemplo *Information Engineering* [3], escrita por James Martin y Clive Finkelstein en 1981. Hay otras muchas, pero las de James Martin destacan por ser pioneras y *best sellers* en el ámbito de las Tecnologías de la Información. El título de otra de sus obras, *Desarrollo de Aplicaciones sin Programadores* [4], es muestra de la escasa valoración que han recibido los programadores durante décadas. En este último libro, James Martin compara la demanda de programadores a comienzos de la década de 1980 con la necesidad de operadores en la industria telefónica de principios del siglo pasado, cuando cada llamada requería la conmutación manual del circuito por parte de un operador. El rápido aumento del número de usuarios de telefonía, y la imposibilidad de contratar operadores a un ritmo tan elevado, parecían imponer un límite al crecimiento de la industria telefónica. Gracias a los equipos de conmutación automática, los operadores dejaron de ser necesarios y la industria telefónica pudo seguir creciendo. De igual forma que la industria telefónica entonces, para Martin la industria del software requería en ese momento un mayor grado de automatismo, que permitiera a los usuarios finales crear sus propios programas y depender en menor medida de los programadores.

James Martin se encuentra entre los padres de conceptos como el de las herramientas CASE[3] o los lenguajes de cuarta generación (4GL). Estas herramientas y lenguajes prometieron simplificar drásticamente el proceso de desarrollo de software durante la década de

[3] *Computer Aided Software Engineering*.

1980. Muchos entonces vislumbraron un futuro de ciencia–ficción, donde los usuarios finales podrían crear sus propias aplicaciones sin necesidad de profesionales técnicamente cualificados. Con una mínima formación cualquier usuario final sin conocimientos de programación podría crear sus propios programas. Efectivamente, como afirmaba Edsger Dijkstra en 1989, el objetivo de la ingeniería de software se había reducido a la búsqueda de un mecanismo para producir programas a partir de personas que no saben programar.

Las herramientas CASE constituyeron durante los años siguientes una verdadera hipérbole tecnológica. Las hipérboles tecnológicas o *hypes* tecnológicos son fenómenos sociológicos que tienen lugar con determinados productos o conceptos, que nacen como la nueva panacea en algún ámbito de la tecnología, y que reciben después más difusión e importancia de la que realmente merecen. Con el tiempo se desinflan y caen en el olvido, aunque suelen cambiar de nombre y volver más tarde como algo completamente nuevo.

Actualmente, en parte gracias a la influencia del artículo *No Silver Bullet* [5] de Frederick Brooks, se considera comúnmente aceptado que no existen ni existirán inventos milagrosos que vayan a tornar el desarrollo de software en algo trivial o automático, ya que la naturaleza de la complejidad del software no reside en las herramientas. Hasta los primeros años de la década del 2000, sin embargo, predominó la ilusión de que podía construirse software mediante fábricas, como churros, de igual forma que se construyen los coches. Mediante algún proceso repetible y predictivo, asistido por herramientas CASE, donde los usuarios modelen gráficamente el software para que después una máquina o un conjunto de obreros subcontratados a una ETT lo "piquen". Desde el punto de vista sociológico es digno de estudio cómo una temática que se encuentra a medio camino entre la charlatanería y la ciencia–ficción ha podido abrirse un hueco tan relevante en el ámbito académico y en la gran empresa, sembrando el prejuicio de que los programadores son peones

albañiles, y la programación una mera actividad de construcción. Las consecuencias de estos prejuicios, como será descrito, han sido catastróficas en el sector de las Tecnologías de la Información.

1.2 El desprecio por los perfiles técnicos

En la actualidad los perfiles técnicos están cada vez peor considerados. Para James Martin los programadores podían dividirse en dos personalidades: el *bit-twiddler* [4], o aquel técnicamente brillante pero despreocupado por los problemas de las personas, y el consultor, que es aquel con habilidades humanas y comunicativas, capaz de entender a los usuarios y las necesidades de negocio. Esta generalización categórica es inapropiada y esconde grandes prejuicios. Aquellos técnicamente brillantes no tienen por qué ser necesariamente una especie de sujetos socialmente inadaptados, que no desean involucrarse en los problemas de las personas, mientras que los técnicamente ineptos no tienen por qué ser grandes consultores, gestores o comerciales. A nadie se le ocurriría dividir a los cirujanos entre profesionales cualificados, pero socialmente inadaptados, y profesionales ineptos, pero con gran vida social. Tal vez es más seguro afirmar que las personas diligentes tienden a dominar tanto el aspecto técnico como humano de su trabajo, mientras que las personas no cualificadas tienden a ser ineptas en ambos aspectos. En cualquier caso, el encasillamiento de las personas técnicamente cualificadas como *frikis* o *nerds* se ha convertido en un topicazo y en un prejuicio de amplio calado social, que no contribuye en nada a la valoración de los profesionales del desarrollo de software.

1.3 La enseñanza de la programación

Una de las primeras consecuencias de que la programación sea despreciada, o considerada como impropia de un ingeniero, es que la

práctica de la programación, junto a los estándares de codificación y el dominio de las herramientas de desarrollo, han pasado a ser materias sin mucho interés dentro de los planes académicos. Las universidades pretenden formar ingenieros, no peones albañiles. Por otra parte, la enseñanza de la programación es compleja. Se requieren varios años de experiencia para empezar a adquirir buenos hábitos de programación. En término medio, las personas tardan tres años en empezar a ser productivas en un lenguaje como C. Tal vez en otros lenguajes como *Java* este tiempo sea menor, porque predisponen a hacer bien las cosas, pero de todas formas un verdadero profesional debería dominar ambos lenguajes, más muchos otros como *Python* o *Ruby*. No pueden dedicarse tres años de universidad a adquirir esta experiencia, porque la carrera duraría el doble, o habría que dejar de dar otras materias. En realidad, el verdadero problema tal vez no esté ligado tanto a la propia dificultad de la enseñanza de la programación, como a la enseñanza de ciertos contenidos que inculcan en los alumnos el desprecio por ella, el prejuicio de que la programación es una actividad impropia de un ingeniero, y el prejuicio de que puede programar cualquiera. Lo peor no es que los estudiantes terminen su formación con más o menos conocimientos técnicos de los que serían deseables, sino que terminen con una escasa predisposición por adquirirlos. Si a los estudiantes de cirugía se les dijera que la cirugía es una labor puramente técnica, y que la verdadera labor de un cirujano no es operar, sino dirigir o planificar las operaciones, las personas seguirían siendo operadas por los barberos como en el siglo XVIII. Afortunadamente, el desprecio por la técnica no ha llegado aún a la Medicina.

1.4 El desprecio por la calidad

Del desprecio por la programación y del mal empleo de la técnica se deriva una escasa calidad de los programas. En el ámbito de los

servicios de software abundan empleados que escriben código sin respetar el sangrado, usando una nomenclatura inconsistente, con vicios que delatan muy poco rodaje. No es cuestión de una mera falta de experiencia, sino desde no saber cómo compilar un programa, hasta ni tan siquiera saber manejar el editor, pasando por un código envilecido, del que se intuyen muchas lagunas acerca de conceptos como los punteros y la memoria dinámica, una persistente tendencia a reinventar la rueda, a abusar de las macros, a escribir funciones de dos mil líneas si se pueden hacer en cuatro, y todo tipo de malas artes que acaban por arruinar los proyectos. Cuando el código presenta estos problemas resulta inmantenible, porque no es posible depurarlo ni modificarlo sin reescribirlo por completo. Además, la falta de respeto por los estándares de codificación imposibilita que estas personas puedan trabajar en equipo.

El problema no es de rodaje sino de mentalidad. Muchos piensan que no han estudiado una ingeniería para acabar picando código. Que cualquiera puede hacerlo. A falta de la predisposición necesaria, jamás aprenden. En muchos casos no importa, porque hacerlo bien o mal es indiferente. Nadie va a valorárselo, ni nadie espera de ellos nada más salvo que el código funcione. Es como el trabajo de un peón albañil en la construcción, donde no sirve de nada ser un verdadero artista en la colocación de ladrillos. Los ladrillos o están bien puestos, o están mal, pero no existen grados, o por lo menos a nadie le importan. Esta situación conduce a un código de baja calidad, mal documentado, indepurable, inmantenible, y repleto de errores. Acaba reinando la desidia, la ley del mínimo esfuerzo, el principio de no tocar lo que ya estaba y el principio de que lo arregle quien lo hizo. La ausencia de calidad no es problema de falta de metodología, ni se resuelve enfatizando en procesos formales o en la exhaustiva documentación de cada aspecto del programa. Es sencillamente un problema de despreocupación por el código, desinterés por la excelencia técnica y tecnoanalfabetismo. La no–calidad tiene

un coste que no es meramente estético, es económico. Es el coste de los proyectos que no cumplen las expectativas del cliente, que se retrasan de forma indefinida, que se entregan medio probados tras apretones de horas extras, y que traen después por el camino de la amargura de los parches urgentes y las quejas.

1.5 El bodyshopping

Si la programación no es una actividad creativa, sino una labor mecánica, trivial, intensiva en mano de obra, y que no aporta valor añadido a la empresa, entonces podrá ser cubierta por recursos reemplazables, que no necesitan tomar decisiones. El perfil de estos recursos responderá a personas poco valoradas, sin suficiente formación, sin experiencia o sin aspiraciones. La escasa valoración del puesto de programador impulsa a las personas con más talento a abandonarlo en favor de puestos mejor remunerados, como suelen ser los relacionados con el área comercial o la gestión. Los puestos de programador quedan relegados a recursos contratados en función de la demanda de trabajo, con muy alta rotación. La demanda de este tipo de mano de obra ha fomentado la proliferación de ETTs[4], bajo la piel de empresas de consultoría y servicios informáticos, cuyo negocio es el *outsourcing* o la subcontratación de personas, lo que coloquialmente se denomina como *bodyshopping*.

La palabra *bodyshopping* suele traducirse al castellano como venta de carne. Consiste en la práctica de subcontratar mano de obra a terceras empresas con objeto de ahorrar costes estructurales, riesgos de contratación, y aprovechar ciertas ventajas fiscales, como la imputación del sueldo de los empleados como inversión y no como gasto. Pese a estas supuestas ventajas, la subcontratación de personas tiene diversos inconvenientes:

[4] ETT significa Empresa de Trabajo Temporal.

- El cliente paga un precio que duplica o triplica el coste real de los servicios contratados. No sólo paga por becarios presentados como expertos consultores, sino por una larga cadena de intermediarios e interesados que encarecen el precio total.
- El cliente asume un coste continuo en formación y una alta rotación de personal. Durante los primeros años las personas no rinden al cien por cien. Una vez formadas empiezan a ser productivas, pero la ley suele establecer un periodo máximo de subcontratación que obliga a contratarlas directamente o a sustituirlas.
- Las empresas intermediarias, pretendiendo presentarse como empresas de consultoría o ingeniería tecnológica, acaban convirtiéndose en meras ETTs encubiertas, con un personal de alta rotación que no aporta experiencia ni conocimiento a la empresa.
- El *bodyshopping* representa un modelo de negocio agotado, en la frontera de la legalidad, dentro de un mercado donde cada vez resulta más absurdo tratar de competir en mano de obra barata frente a otros países, en vez de competir en calidad y productividad.
- Desde el punto de vista de los empleados, el *bodyshopping* es sinónimo de precariedad laboral, con sueldos muy bajos, alta rotación y escasas perspectivas de progresión profesional.

1.6 El offshoring

El *offshoring* o deslocalización es la práctica de trasladar determinados procesos de negocio, típicamente intensivos en mano de obra, a países donde los costes productivos son más bajos, como aquellos que están aún en vías de desarrollo. Generalmente las empresas trasladan actividades como la manufacturación o la producción, mientras que las actividades de diseño, que no requieren mucha mano de

obra sino alta tecnología y personal especializado, se mantienen en los lugares de origen. Vista la programación como una tarea trivial, no estratégica para la empresa pero intensiva en mano de obra, es práctica cada vez más frecuente trasladarla a lugares con mano de obra barata. Estos locales situados en zonas menos desarrolladas son denominados como fábricas de software o *software factories*, expresión que no trata de disimular en absoluto el prejuicio de que el software se *fabrica* como rosquillas.

Uno de los problemas de las fábricas de software y del *offshoring* es la distancia. Si trasladar el trabajo es costoso, otra opción es traer aquí la mano de obra. Con objeto de salvar los obstáculos legales que en materia de legislación laboral existen, hay quien alberga a trabajadores de países como la India dentro de barcos cercanos a la costa o en aguas internacionales fuera de toda jurisdicción[5]. La idea de tener personas traídas del Asia meridional, trabajando en barcos sin el amparo de ninguna ley laboral, en condiciones precarias o denigrantes, parece más propia de la época colonial que del siglo XXI. Es un caso extremo pero sintomático de un problema real, que es el profundo desprecio existente por las personas.

La productividad de las fábricas de software es muy baja, ya que no es posible crear software por el método de la fuerza bruta. La programación no es una actividad intensiva en mano de obra, ni tampoco una actividad de construcción o fabricación. La práctica del *offshoring* proviene en realidad de otros campos de la industria, donde existen actividades que sí son intensivas en mano de obra, como por ejemplo, la industria del automóvil o la industria textil. Lo que se traslada a los países en vías de desarrollo suelen ser actividades de producción. Las actividades de diseño, que no requieren mucha mano de obra poco cualificada, sino alta tecnología y personal especializado, no se trasladan, ya que dichos países no se caracterizan

[5] Ver por ejemplo `http://www.sea-code.com`.

por un entorno altamente tecnológico, ni por un nivel educativo medio elevado. Por ilustrarlo con un ejemplo, algunos *ipod* incluyen una inscripción que afirma lo siguiente:

"*Designed by Apple in California. Assembled in China.*"[6]

Si Apple decidiese trasladar su actividad de diseño desde California a China, o si Nokia trasladase sus oficinas de investigación y desarrollo desde Finlandia a su fábrica de Chennai en la India, no obtendrían ninguna ventaja. La mayor parte de las personas entiende que el diseño de un circuito integrado es una actividad de diseño, no intensiva en mano de obra, que requiere alta tecnología y personal especializado, no fuerza bruta. No es que dicho personal especializado no pueda existir en la India. Existe, pero no supone ningún ahorro trasladar allí las actividades de diseño, ya que al tratarse de poca mano de obra el ahorro es insignificante o inferior a los costes que la distancia ocasiona.

1.7 Qué es la programación

La consideración de la programación como una actividad de construcción es un prejuicio que ha tenido consecuencias funestas en el mundo de los servicios de software. Durante décadas ha existido un acalorado debate acerca de la naturaleza de la programación. ¿Es un arte?, ¿una actividad de construcción?, ¿una actividad de diseño? Para diversas personas de contrastada autoridad en materia de software, como Donald Knuth, la programación es efectivamente un arte, aunque arte y técnica, como él mismo señala en su discurso *Computer Programming as an Art* [6], no son términos enfrentados, sino términos que tienen bastante en común. Las palabras como *técnica* o *tecnología* provienen del prefijo griego $\tau\epsilon\chi$, que significa

[6] "*Diseñado por Apple en California. Ensamblado en China.*"

arte. En general todas las actividades de diseño, desde la arquitectura al diseño industrial, tienen un componente artístico. Para Edsger Dijkstra, con quien hemos comenzado este capítulo, la programación no es sólo un arte [7], sino una de las ramas más difíciles de las matemáticas aplicadas [8].

No obstante, ha prevalecido la visión de la programación no como una actividad de diseño, ni como un arte, sino como una actividad de construcción. Este punto de vista siempre ha ido acompañado de la expectativa por automatizar la programación, y por obtener los programas a partir de diseños expresados en notaciones gráficas como UML, como si la actividad de diseño o la esencia de la ingeniería estuviera intrínsecamente ligada al uso de notaciones gráficas, como la empleada en el plano de un edificio, o como en el esquema del circuito eléctrico mostrado en la **figura 1.1**.

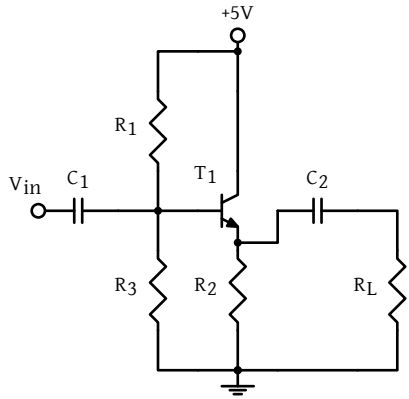

Figura 1.1 Diagrama de un circuito electrónico.

El **listado 1.1** no corresponde a ningún lenguaje de programación. Es un lenguaje de descripción de *hardware* denominado Verilog, como el que pueden usar los ingenieros de Nokia en Finlandia, hasta

```
module counter (clk, clr, q);
  input  clk, clr;
  output [3:0] q;
  reg    [3:0] tmp;
  always @(posedge clk or posedge clr)
  begin
  if (clr)
    tmp <= 4'b0000;
  else
    tmp <= tmp + 1'b1;
  end
  assign q = tmp;
endmodule
```

Listado 1.1 Circuito descrito en Verilog.

ahora felices y ajenos a la existencia de diversos gurus de la ingeniería de software empeñados en mandarles como peones albañiles a Chennai. El motivo por el que se usan este tipo de lenguajes es porque a medida que el diseño de un circuito presenta mayor escala de integración de componentes, se torna inviable dibujarlos mediante los diagramas tradicionales y es necesario definirlos mediante lenguajes de mayor nivel como Verilog o VHDL[7]. Una vez definidos en dichos lenguajes, es posible simularlos, verificarlos, y generar automáticamente el esquema físico del circuito mediante herramientas de síntesis. Está claro, por tanto, que el uso de notaciones gráficas o no gráficas es puramente accidental. Las notaciones gráficas no tienen por qué ser de mayor nivel que las notaciones textuales. De hecho, dibujar manualmente el esquema gráfico de un circuito integrado, que puede tener millones de transistores, puede ser materialmente imposible para una persona. De igual forma que los ingenieros emplean lenguajes textuales de descripción de *hardware* para diseñar circuitos, y no dejan por ello de ser ingenieros, los programadores emplean lenguajes textuales de programación para

[7] *VHSIC Hardware Description Language*.

diseñar programas, y no por ello son peones albañiles. La programación es por tanto una actividad de diseño, no una actividad de construcción.

1.8 La reducción de costes

El objetivo de la búsqueda de mano de obra barata es presuntamente la reducción de costes. Sin embargo, pensar que la contratación de mano de obra barata reduce los costes es una falacia. Las industrias que logran reducir sus costes productivos suelen hacerlo mediante la inversión en maquinaria más moderna y trabajadores más cualificados y polivalentes, porque aunque en principio estos recursos son más caros, a medio y largo plazo el coste productivo es menor, ya que puede producirse más con menos, y además puede producirse mejor, o con mayor calidad. Los retrasos, los desperdicios, los defectos y en general la no–calidad, tienen un fuerte impacto en los costes productivos. La renuncia a la calidad y a la eficiencia, con el objetivo de una reducción de costes efímera, es una apuesta de fracaso, propia de empresas decadentes o agonizantes, que toman esas medidas como forma temporal de mantenerse a flote o de ganar tiempo. Una estrategia de este tipo mantenida durante demasiado tiempo conduce inevitablemente al hundimiento y colapso de cualquier empresa, al menos en una situación de libre mercado. Las prácticas del *bodyshopping* y el *offshoring* incrementan los costes del desarrollo de software, ya que reemplazan unos pocos desarrolladores altamente cualificados por grandes cantidades de mano de obra inexperta, poco valorada y poco comprometida, en condiciones laborales precarias.

La visión del proceso de desarrollo de software como una actividad de construcción está estrechamente relacionada con el prejuicio de pensar que la programación no es una cuestión de talento, sino de fuerza bruta, o de contar con mucha mano de obra barata. Pero

no es así, porque hay actividades que no son cuestión del número de personas que sean asignadas, sino de saber hacerlas. Existen diversos estudios que constatan en la práctica el hecho de que la productividad de un desarrollador experto y con talento es varias veces mayor que la de un programador sin experiencia ni talento. Según Robert L. Glass, los mejores programadores son hasta 28 veces más productivos que los peores [9]. Hay estudios al respecto por Sackman [10] y DeMarco [11] con resultados de una variabilidad de hasta diez veces en la productividad.

Una consecuencia de todas estas consideraciones es que existen muchas empresas en el mundo de los servicios de software trabajando con una productividad muy baja, ya que derrochan el presupuesto contratando mucho más personal del que sería necesario. Aunque parezca paradójico, esta improductividad desencadena una destrucción neta de empleo en el sector, ya que la improductividad siempre destruye riqueza y por tanto empleo[8]. Es destacable que desde el punto de vista de una empresa de servicios o una ETT, cuanto más baja sea la productividad, más beneficios se obtienen, ya que facturan al cliente más personas durante más tiempo. Los grandes perjudicados de este modelo improductivo y extremadamente caro son los clientes, o en concreto, sus accionistas. La teoría de la agencia, desarrollada en 1977 por Michael Jensen y William Meckling [13], puede explicar en algunos casos el conflicto de intereses entre accionistas y directivos cuando existe separación entre propiedad y control dentro de una empresa. Los fenómenos del *bodyshopping* y del *offshoring*, en el caso del desarrollo de software, suelen darse precisamente en empresas como la banca, las grandes operadoras de telecomunicaciones y la administración pública, cuyo denominador común es la existencia de esta separación entre propiedad y control.

[8] Una explicación muy clara en *Economía en una lección* [12], de Henry Hazlitt.

Resumen

Algunas de las ideas generales de este capítulo son las siguientes:

- La ingeniería de software tradicional ha difundido la visión de la programación como una actividad de construcción, realizable por fuerza bruta e impropia de un ingeniero.
- Los programadores han sido considerados como peones albañiles o recursos reemplazables, que no aportan valor añadido a la empresa.
- La ingeniería de software ha promovido el *amateurismo* o la falta de cualificación para la programación, proponiendo que sean los usuarios finales los que creen sus propios programas mediante las míticas herramientas CASE, los llamados lenguajes de cuarta generación, y otras muchas hipérboles tecnológicas.
- Los prejuicios anteriores han motivado la aparición de prácticas como el *bodyshopping* o el *offshoring*, con objeto de producir software mediante mano de obra barata y por el método de la fuerza bruta, lo que ha conducido a un mercado caracterizado por la precariedad laboral, la falta de profesionalidad, y la ausencia de calidad.
- La programación no es una actividad de construcción, sino una actividad de diseño. Las actividades de diseño no requieren mucha mano de obra, sino poca mano de obra pero altamente cualificada.
- Aquello en lo que están fallando las empresas de consultoría, como presuntas empresas de consultoría y no como presuntas ETTs, es en resaltar el hecho de que lo que realmente requiere el desarrollo de software no es simple fuerza bruta contratada a mansalva allí donde resulte más barata, sino mano de obra cualificada. Esta mano de obra cualificada también es más barata, porque es infinitamente más productiva.

Lecturas recomendadas

A propósito de este capítulo es más que recomendable la lectura de los siguientes artículos, que pueden ser encontrados fácilmente en Internet:

- *The humble programmer*, de Edsger Dijkstra en 1972 [2].
- *Written in anger* de Edsger Dijkstra en 1978 [14].
- *Computer Programming as an Art*, de Donald Knuth en 1974 [6].
- *No Silver Bullet:Essence and Accidents of Software Engineering*, de Frederick Brooks en 1987 [15].
- *What should we teach new software developers? Why?*, de Bjarne Stroustrup en 2010 [16].

Referencias

[1] Edsger Dijkstra. *By way of introduction*. Austin, 1989.
http://www.cs.utexas.edu/users/EWD/ewd10xx/EWD1041.PDF

[2] Edsger Dijkstra. The humble programmer (Turing Award Lecture). *Communications of the ACM*, 15(10):859-866, 1972.
http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD340.PDF

[3] James Martin y Clive Finkelstein. *Information Engineering*. Savant Institute, 1981.

[4] James Martin. *Application Development Without Programmers*. Prentice Hall, 1982.

[5] Frederick P. Brooks. *The Mythical Man Month*. Addison Wesley, 1975.

[6] Donald Knuth. Computer Programming as an Art. *Communications of the ACM*, 17(12):667–673, 1974.
http://doi.acm.org/10.1145/361604.361612

[7] Edsger Dijkstra. *A Short Introduction to the Art of Programming*. Austin, 1971.

http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD316.PDF

[8] Edsger Dijkstra. *How do we tell truths that might hurt?*. Austin, 1975.

http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD498.PDF

[9] Robert Glass. *Facts And Fallacies Of Software Engineering*. Addison-Wesley Professional, 2002.

[10] H. Sackman, W. Erikson and E. Grant. Exploratory experimental studies comparing online and offline programming performance. *Communications of the ACM*, 11(1):3–11, 1968.

http://doi.acm.org/10.1145/362851.362858

[11] Tom DeMarco y Tim Lister. Programmer performance and the effects of the workplace. *ICSE '85: Proceedings of the 8th international conference on Software engineering. IEEE Computer Society Press*, páginas 268–272, 1985.

[12] Henry Hazlitt. *Economía en una lección*. Ciudadela Libros, S.L., 2008 (original en 1946).

[13] Michael Jensen y William Meckling. Theory of the Firm: Managerial Behavior, Agency Costs and Ownership Structure. *Journal of Financial Economics*, 3(4):305–360, 1976.

[14] Edsger Dijkstra. *Written in anger*. Austin, 1978.

http://www.cs.utexas.edu/users/EWD/ewd06xx/EWD696.PDF

[15] Frederick P. Brooks. No Silver Bullet: Essence and Accidents of Software Engineering. *Information Processing: Proceedings of the IFIP Tenth World Computing Conference*, 1986. También publicado en [17].

[16] Bjarne Stroustrup. What should we teach new software developers? Why?. *Communications of the ACM*, 53(1):40–42, 2010.
http://doi.acm.org/10.1145/1629175.1629192

[17] Frederick P. Brooks. *The Mythical Man Month (Anniversary Edition with four new chapters)*. Addison Wesley, 1995.