Programación estructurada (Un intento de clarificación)

PERE BOTELLA
HORACIO RODRIGUEZ

1. INTRODUCCION

Bajo la etiqueta de Programación Estructurada (P.E.) se agrupan diferentes técnicas, métodos, procedimientos o simplemente enfoques que se aplican a una gama amplia de problemas informáticos.

Prácticamente todas las realizaciones que en los campos de la Tecnología o la Metodología de la programación se han producido en los últimos años, llevan esta etiqueta.

Hubo un tiempo en que la P.E. constituía el eje de toda una serie de polémicas, la más famosa de las cuales fue la controversia sobre el uso o no de sentencias Go To en los programas. Como suele suceder, la mayoría de estas polémicas se centraban en temas absolutamente marginales por lo que poco pudieron aportar al desarrollo de una conciencia clara del tema. Todo ello condujo a rodear el tema de una nube de incomprensión que aún hoy subsiste.

De cualquier forma, la P.E. está hoy plenamente aceptada, por lo menos a nivel de lo que se escribe o se publica sobre programación (la implantación real de la misma a nivel de centros de cálculo es un asunto más discutible) y, de alguna manera, constituye el marco donde se insertan casi todos los avances actuales en programación.

Parece pues adecuado, una vez superada esta etapa inicial, tratar de precisar y clarificar todo lo que la P.E. significa, cual es su incidencia en el campo de la programación y cuáles sus perspectivas de futuro. Este será el objetivo del presente trabajo.

2. UN POCO DE HISTORIA

Los orígenes de la programación estructurada cabe situarlos en los últimos años de la pasada década. Hasta entonces la tarea de los programadores estaba fuertemente limitada por el Hardware utilizado, fruto de una tecnología incipiente. Para *WIRTH* [WIRT-72], lo que caracterizaba la programación en esta época era una "optimización de la eficiencia de una máquina concreta al ejecutar un algoritmo concreto". Todo ello configuraba la tecnología de la programación como un conjunto de reglas y trucos, muchas veces personales, que se traducían en un desarrollo puramente artesanal de los programas. El uso de lenguajes de programación de bajo nivel venía a completar el panorama.

La creciente complejidad de los sistemas y aplicaciones, las necesidades cada vez más acuciantes de fiabilidad en los programas y al mismo tiempo la falta de adecuación de los métodos tradicionales a dichas necesidades llevó a la programación a una situación de práctica asfixia.

Esta situación, a la que muchos autores se refieren como "*La crisis del Software*" se manifiesta principalmente bajo dos aspectos: por un lado el coste del software y por otra parte la calidad del mismo.

En cuanto al coste del Software, la característica fundamental es el aumento gradual del peso del mismo como componente del coste total de un sistema informático. Otras características importantes son la disminución de la productividad de los programadores al abordar aplicaciones complejas que hacen imprescindible el trabajo en grupo y el alto porcentaje de tiempo dedicado a la labor de prueba y depuración de errores (que algunos estudios cifran en un 50 % o más).

Naturalmente, es imposible hablar del coste del software sin referirnos a su fiabilidad. *MILLS* [INFO-76] afirma al respecto "Cuando un equipo numeroso de programación construye programas a una velocidad de 3 instrucciones por día y hombre, o incluso de 6 ó 7, la primera reacción es preguntarse en qué invierten su tiempo los integrantes del equipo. Cuando los resultados que producen están plagados de errores la baja productividad resulta aún más inexplicable". *RAMAMOORTHY* [INFO-76], produce escalofríos cuando afirma que cada nueva versión del OS/360 contenía unos 1000 nuevos errores de software o que durante el vuelo del Apolo 14 se detectaron 18 errores en el software de seguimiento. Todo ello nos lleva al segundo punto de la crisis del software: la calidad del mismo.

Ya hemos indicado que hasta entonces todas las exigencias de calidad del software venían condicionadas por el hardware disponible. Veamos ahora las exigencias que la nueva situación plantea. *GILL* [INFO-76] define 8 características que configuran la calidad del software:

1. Ausencia de errores.
2. Conformidad con las especificaciones (el programa debe hacer todo lo que se le pide y nada más).
3. Eficiencia (es decir, óptima utilización de los recursos de máquina durante la operación).
4. Robustez (por así decirlo, resistencia a un medio hostil).
5. Portabilidad (mínimo esfuerzo de implementación en un entorno diferente. Por ejemplo en otra máquina, bajo diferente sistema operativo, etc.
6. Adaptabilidad (mínimo esfuerzo de adaptación a cambios que se le puedan introducir)
7. Claridad de diseño.
8. Documentación.

Naturalmente no deben entenderse estas 8 caracterísi-

cas como funciones a optimizar simultáneamente sino más bien como líneas que deben servir de guía para la construcción de los programas. El tipo de aplicación y otros factores determinarán en cada caso la o las características críticas.

Ya hemos indicado antes la impotencia de los métodos tradicionales para hacer frente a estos nuevos planteamientos.

En estas circunstancias, surgió en Europa una corriente*, cuya figura más representativa era *E.W. DIJKSTRA*, que proponía una nueva manera de enfocar los problemas de programación, potenciando las características de calidad del software a través de criterios que tuvieran siempre en cuenta las limitaciones del programador.

La primera manifestación de estas ideas la podemos situar en una carta abierta al director de C.A.C.M. del propio *DIJKSTRA* ("The Go To statement considered harmful", *La instrucción Go To considerada dañina*, 1968) donde el autor proponía, de forma marginal, la eliminación de la instrucción Go To de todos los lenguajes de alto nivel. Citamos la propuesta ya que, aunque el propio *DIJKSTRA* la hizo de manera informal y dentro de un determinado contexto, lo cierto es que supuso el comienzo de una gran polémica sobre el uso o no del Go To en los programas. El tema no era desde luego fundamental ya que como muy bien diría *MILLS* [MILL-72] "lo que caracteriza a un programa estructurado no es la ausencia de Go To sino la presencia de estructura", pero supuso un derroche de esfuerzos en una dirección equivocada, o sea, la sistemática eliminación del Go To a cualquier precio. La idea que aún hoy tienen muchos programadores de que programación estructurada es sinónimo de programación sin Go To es una triste secuela de la polémica. De todas formas, no todo lo publicado al respecto es desechable y el trabajo de *KNUTH* "Structured programming with Go To statements" (*Programación estructurada con instrucciones Go To*, 1974) es de lectura obligada para todos los interesados por el tema. Igualmente cabe recomendar como clarificadora la referencia [WIRT-74] cuyo segundo apartado expone magistralmente la idea de Mills que más arriba apuntábamos.

Pero realmente, el trabajo que supuso, de hecho, el lanzamiento de la P.E. se publicó un par de años más tarde, en 1970. Se trata de una nota interna de la universidad de Eindhoven, titulada "Notes on Structured Programming", que agrupaba una colección de reflexiones personales sobre la programación de Dijkstra. Estas notas, ampliadas con dos artículos de Dahl y Hoare se editaron conjuntamente con el libro "Structured Programming" [DIJK-72a]. El trabajo de Dijkstra dio precisamente nombre a la nueva filosofía (por cierto de manera un tanto fortuita: el propio *DIJKSTRA*, que sólo menciona el término en el título de la obra, afirmó posteriormente que le dio este nombre porque no se le ocurrió otro mejor, pero que igualmente podría haber hablado de programación sistemática o algún otro término similar).

A partir de aquí, el camino de la P.E. se clarifica. En 1971 *DIJKSTRA* publica su "A short introduction to the art of programming" (Una breve introducción al arte de programar) que viene a demostrar que la nueva filosofía no es para una élite sino que se puede aprender a programar estructuradamente y que de hecho, ello no supone una dificultad para el aprendiz a programador sino más bien al contrario. Las ideas de la P.E. entran con mayor facilidad y resultados sorprendentes en aquellas personas que no saben programar; en cambio el reciclaje y reconversión de programadores expertos es a veces más difícil. *WIRTH* publica el mismo año su trabajo "Program development by stepwise refinement" (Desarrollo de programas mediante el refinamiento por etapas) concretando las ideas anteriores y aplicándolas al campo de la construcción de programas. Ideas como la del diseño descendente (Top-Down) o el refinamiento por etapas de los programas (stepwise refinement) pasan a ser comúnmente aceptadas. El mismo año el propio *WIRTH* presenta el Pascal, primer lenguaje construido especialmente con las propuestas formuladas por la P.E. y que posteriormente ha obtenido una considerable difusión y aceptación.

La eclosión de las nuevas teorías del mundo científico y académico y su paso al campo de la producción fue causado por un número de la revista Datamation dedicado al tema y que, de hecho, supuso la aceptación americana de un movimiento hasta entonces mayoritariamente europeo. No es que a partir de este momento la P.E. tuviera ante sí un camino de rosas: la resistencia por parte de la vieja guardia de la programación clásica fue dura; sin embargo el progreso es continuo y cubre actualmente casi la totalidad del espectro de los que algo significan en programación (no podemos resistir la tentación de citar el caso de *KNUTH*, que ha interrumpido la publicación de su monumental "Art of computer programming" para adaptarla a unos criterios a los que en principio fue ajeno).

Algo más tarde, en 1974, la revista Computing Surveys de la A.C.M. [COSU-74] publicó un número bajo el epígrafe "Programming", conteniendo las referencias [WIRT-74], [KNUT-74], y [KERN-74]. Con él la aceptación y difusión de la P.E. es ya mayoritaria: cuando se hablaba de programación no cabía otro enfoque que el de la P.E.

3. ¿QUE ES LA PROGRAMACION ESTRUCTURADA

Hemos visto que la P.E. no es una metodología sino más bien una colección de ideas que preconizan una racionalización y sistematización de la función de programar.

Hemos visto, también, que su aparición y asentamiento supusieron una ruptura que afectó radicalmente a la forma de enfocar la programación.

Vamos a intentar profundizar más en las ideas básicas que configuran el concepto de P.E.

Buscar una definición formal, más o menos aceptada, es inútil. *GRIES* [GRIE-78] en un trabajo reciente cita hasta 14 definiciones diferentes que ha encontrado publicadas, muchas de ellas por autores de primera línea. Algunas de ellas no aportan gran cosa al tema y otras abordan sólo aspectos parciales del mismo. La más brillante, para *GRIES* y, modestamente, también para nosotros es la de *HOARE:* "Programación Estructurada es la tarea de organizar el propio pensamiento de forma que conduzca, en un tiempo razonable, a una expresión inteligible de una tarea informática".

Esta definición recoge, creemos, la esencia del concepto de P.E. Por un lado recalca que la programación es una actividad humana y debe tener en cuenta, por lo tanto, las limitaciones del programador. Un primer objetivo será, pues, la reducción de la complejidad. Por otra parte la definición de *HOARE* pone sobre el tapete el problema de la verificación (program proof o proof of correctness). En efecto, si el objeto de la P.E. es obtener una expresión inteligible de una tarea informática, debemos exigir que esa expresión sea "cierta", es decir, que responda a las especificaciones del planteo. Ello nunca podrá ser comprobado a posteriori mediante pruebas (el propio *DIJKSTRA* afirmó que una prueba sólo sirve para detectar la presencia de fallos, nunca su ausencia). Se hace, pues, necesario, certificar el programa en el momento de construirlo. Este será el segundo objetivo fundamental de la P.E.

Resumiendo, tenemos dos objetivos fundamentales que subyacen en el concepto de P.E.

* Una corriente que podemos situar dentro del TC2 (Technical Committe 2) de la I.F.I.P. (International Federation for Information Processing). Precisando más, fue a raíz de la presentación del ALGOL-68 por parte del WG2.1 del IFIP (Working Group 2.1) cuando un grupo minoritario de miembros del grupo se opuso a su publicación aduciendo que los programadores necesitaban algo más que lenguajes cada vez más complejos y potentes. Este grupo, del cual formaban parte Dijkstra, Hoare, Dahl, Turski y otros formó algo después, el WG2.3. Aquí podríamos situar el núcleo fundador de la P.E.

1. Reducción de la complejidad.
2. Verificación de los programas en el momento de construirlos.

Naturalmente, los logros conseguidos en el cumplimiento de estos objetivos son diferentes en las diferentes ramas de la aplicación de la P.E. En el próximo apartado los comentaremos. De todas maneras, si tuviéramos que encontrar una característica común a todas las líneas, esta debería ser el desarrollo del concepto de *abstracción* y concretamente la utilización generalizada de recursos abstractos. En efecto, las necesidades de reducción de la complejidad al abordar un problema deben pasar por técnicas de partición o desglose del problema en otros más sencillos. Ello trae consigo la necesidad de trabajar en algunas de las etapas de este proceso con objetos (datos o tratamientos) abstractos. El ejemplo más sencillo de lo que acabamos de decir lo encontramos en las primeras etapas de cualquier proceso de diseño descendente. La aplicación del concepto de abstracción da lugar a una serie de recursos: diseño descendente, pseudocódigo, tipos de datos, esquemas, etc. De todo ello hablaremos en el siguiente apartado.

4. AMBITOS, APORTACIONES Y PROPUESTAS DE LA P.E.

Volvamos, por un momento, al título y al primer párrafo del presente trabajo. En concreto, al subtítulo ("un intento de clarificación"). Una idea que era motivo de preocupación para los autores de este trabajo, y que motivó el subtítulo en cuestión, era la siguiente:

Sean dos programadores, a los que llamaremos A y B. Supongamos que A aprendió a programar COBOL en una academia, encontró trabajo, y después de poco tiempo le mandaron a un curso sobre el método Warnier de P.E. Supongamos que B se apuntó a un cursillo del Centro de Cálculo de la Universidad, cuando cursaba tercer año de carrera, en donde le explicaron cómo construir programas por sucesivos refinamientos y donde aprendió y practicó el lenguaje PASCAL. Preguntados ambos sobre su forma de trabajar, ambos afirmaban usar "la programación estructurada". Pero lo cierto era que sus conocimientos no tenían, aparentemente, ningún punto en común, y ante un único test de conocimientos, dieron resultados muy dispares. ¿Por qué? Pues porque como dice el primer párrafo de este trabajo, "bajo la etiqueta de P.E. se agrupan toda una serie de técnicas, métodos, procedimientos o simplemente enfoques que se aplican a una gama amplia de problemas informáticos".

Intentemos, pues, ordenar y sistematizar técnicas, métodos, etc. en forma de distintos ámbitos, bajo los que agruparemos ideas que fueron propuestas en los primeros textos y que ya son realizaciones, junto con aportaciones aún no implementadas. Los ámbitos que trataremos de describir son:

- Lenguajes (notación)
- Metodologías
- Verificación
- Estilo
- Tipos de datos
- Programación concurrente, o paralela
- Desarrollo de software
- Enseñanza

4.1 Lenguajes (notación)

El aspecto más difundido y conocido de la P.E. es la propuesta formulada por Dijkstra en [DIJK-72] y en [DIJK-71]. Con la idea de aproximar los lenguajes de programación a formas más ligadas a la manera natural de organizar el pensamiento humano, Dijkstra parte de lo que llama "ayudas mentales" utilizadas para construir programas [DIJK-72 a] o de las formas imperativas utilizadas en el habla habitual [DIJK-71] para deducir las estructuras que conviene contengan los lenguajes. Según los criterios expuestos en esas referencias, los programas se construirán con tres tipos de estructuras: sentencias compuestas, sentencias alternativas y sentencias repetitivas.

a) Sentencias compuestas

- Cuando una acción puede descomponerse en sub-acciones que se ejecutan secuencialmente (S_1, S_2 . . . S_n), las agrupamos en una única sentencia, mediante:

BEGIN S_1; S_2; . . . ; S_n END

b) Sentencias alternativas

- Cuando una acción S se ejecuta sólo si se cumple una condición C, escribimos

IF C THEN S

- Cuando se ejecuta una acción S_1 si es cierta una condición C, o bien S_2 si C es falsa,

IF C THEN S_1 ELSE S_2

- Cuando seleccionamos una acción de entre unas ciertas posibles S_1 . . . S_n, según el valor V_1 . . . V_n que tome una variable V, escribiremos

CASE V OF V_1 : S_1; V_2 : S_2; . . . ; V_n : S_n END

c) Sentencias repetitivas

- Cuando queramos ejecutar repetidamente una acción S hasta que se cumpla C,

REPEAT S UNTIL C

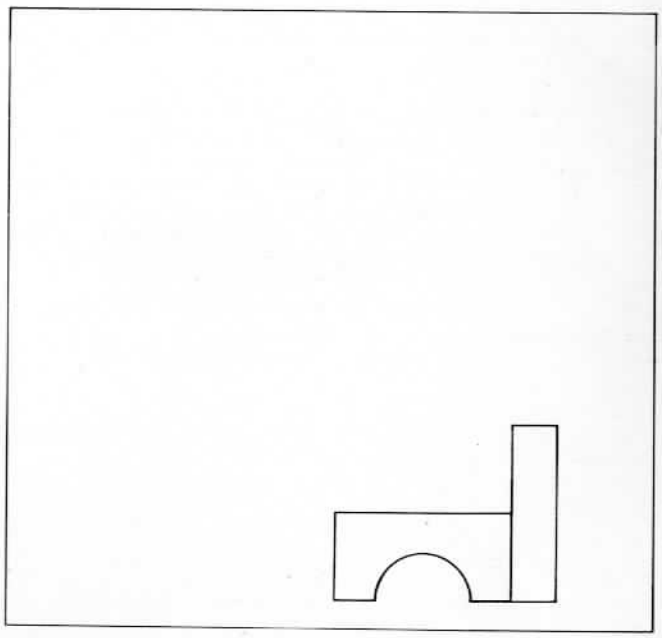
(S se ejecuta al menos 1 vez)

La propuesta de Dijkstra consistía en incorporar este tipo de cláusulas (igual, o con variaciones sintácticas) en los nuevos lenguajes de alto nivel. Es obvio que en esta propuesta, el símbolo de "acción" S, representa cualquier sentencia (compuesta, repetitiva o alternativa). Por combinación de estas formas se puede obtener cualquier programa (afirmación que desde el punto de vista teórico, se demuestra en [BOHM-66].

La propuesta de Dijkstra se recoge íntegra en el lenguaje PASCAL [WIRT-74 b]. Posteriormente, y manteniendo los tres tipos de sentencia citados, ha habido gran cantidad de nuevas propuestas, algunas de las cuales son meras variaciones sintácticas, y otras, como la posterior del propio Dijkstra [DIJK-76], representan nuevas alternativas o aportaciones.

Las razones de dedicar esfuerzos en esta línea, son, según expresa acertadamente Gries [GRIE-78], las siguientes:

a) Buscar una mejor notación y mejores características de los lenguajes que puedan ayudar a simplificar el proceso de programar.



b) Determinar qué estructuras de control son más útiles para describir algoritmos con corrección y claridad.

c) Aprender cómo describir estructuras de datos de una manera clara. (Veremos este aspecto en 4.5). En los lenguajes actuales que no dispongan de estas estructuras, pueden (y deben) simularse mediante las instrucciones existentes (incluido el GO TO).

4.2 Metodologías

El objetivo principal de esta línea de estudio consiste en definir métodos ordenados y eficientes que nos permitan obtener programas correctos, manejables y claros. En sus "Notes on structured programming", Dijkstra, basándose en la abstracción, componía sus programas paso a paso, deduciendo primero un esquema, para ir refinando posteriormente las acciones que contenía. ¿Qué significa esto? Veamos un rápido ejemplo: un programa para calcular unos datos agrupados (totales, medias) de los registros de un fichero.

En primera aproximación, tenemos

```
leer-primer-registro;
WHILE hay-registros DO
      BEGIN
            tratar-registro;
            leer-registro
      END;
calcular totales;
imprimir resultados
```

Nuestro programa no está totalmente escrito en un lenguaje de programación: quedan acciones y condiciones abstractas por definir. Acciones como "leer-registro" pueden traducirse directamente en forma de una sentencia READ. En cambio, al refinar, por ejemplo, la acción "tratar-registro", podríamos obtener algo como:

```
tratar-registro
   validar-registro;
IF registro-correcto
      THEN     acumular-contadores
      ELSE     mensaje-de-error
```

en donde nos aparecen tres acciones y una condición en forma abstracta (en lenguaje natural). Una vez refinadas todas las acciones del primer esquema, procederíamos, en un segundo paso, al refinamiento de aquellas que quedan por refinar (validar-registro, acumular-contadores, etc.). Repetiríamos el proceso hasta tener todas las acciones redactadas, en lenguaje de programación, es decir, hasta tener el programa.

En el ya citado artículo (incluido en [GRIE-78]) de Wirth, "Program development by stepwise refinement", se formaliza este concepto, el de *refinamiento por etapas* (stepwise refinement) utilizando el clásico ejemplo de las 8 reinas (situar 8 reinas en un tablero de ajedrez sin que se maten entre ellas; ver NOVATICA n.º 22). El mismo autor en [WIRT-72] y [WIRT-76] insiste en el mismo concepto.

La propuesta de Dijkstra, formalizada por Wirth, se generaliza dando paso al concepto de programación "*top-down*", de arriba a abajo, de lo general a lo particular, de un esquema abstracto a un programa concreto.

Basándose en estas ideas, como líneas maestras, varios autores han propuesto metodologías de estricta formulación y definición, aplicables a diferentes áreas, aunque quizás el área de la gestión ha sido en la que han florecido con mayor ímpetu diversos métodos, y ello, a nuestro entender, por varias razones:

a) Fuerte implantación del lenguaje COBOL. Las propuestas de la P.E. se orientan a la definición de nuevos lenguajes, pero el coste de sustituir el COBOL es un precio muy alto. La forma de aprovechar las nuevas ideas sobre programación consiste en crear métodos en los que se pueda utilizar el COBOL.

b) Existe, en esta área, un problema muy grave de ausencia metodológica.

c) La mayor parte de los programas tienen una característica común: Transforman unos datos (ficheros) de entrada en datos (ficheros) de salida, lo que requiere métodos especializados.

d) La tarea de programar es la última fase de un proceso de análisis y diseño: el programador recibe unas especificaciones hechas por otras personas (división no tan clara en las áreas científico-técnica, o de sistemas), lo cual aconseja unificar el método desde el inicio del proceso.

En esta línea se han definido métodos como los propuestos por Warnier [WARN-73] y [WARN-75] Jackson [JAK-75], que deducen la estructura del programa partiendo de la estructura de los datos. También la metodología definida por Constantine, Myers y Yourdon [YOUR-75] se orienta al área de la gestión. La metodología que ha desarrollado IBM [BAKE-75] apunta a objetivos más generales.

Dado, por una parte, que la P.E. no contempla el uso del ordinograma como elemento de diseño, y por otra la fuerte implantación de este instrumento gráfico, han surgido propuestas de ordinogramas alternativos para P.E., tales como el llamado (erróneamente) método Bertini [BERT-73] o los diagramas de Nassi-Schneiderman [NASS-73].

4.3 Verificación

En el punto 3. de este trabajo hemos afirmado que "verificar los programas en el momento de construirlos" es uno de los objetivos de la P.E. El mismo objetivo, en otras palabras, podría formularse así: "producir programas correctos, sin errores, como producto final del proceso de programar". El tema tiene gran importancia y está plagado de dificultades. Por ello lo tocamos aquí de pasada y dirigimos al lector al artículo sobre validación y puesta a punto de programas que se incluye en este mismo número.

Los aspectos que han sido (y son) estudiados por la P.E. son:

- Verificar (probar o demostrar) que un programa es correcto.
- Construir, al mismo tiempo, el programa y la prueba de su corrección.
- Definir lenguajes de programación que incluyan pruebas de corrección, o bien automatizar el proceso de prueba, mediante programas "verificadores".

Las pruebas de verificación se basan en una parte de las matemáticas: la lógica, y, en concreto, en la lógica proposicional. El concepto elemental es la *aserción* (assertion, en inglés), que consiste en la expresión de una condición lógica que se cumple en un punto del programa. Dado un programa, podemos tratar de definir las aserciones de entrada y salida (condiciones de partida y condiciones que deban cumplirse al terminar). La prueba consistirá en deducir, por medio de unas reglas de inferencia, paso a paso, todas las aserciones intermedias, y si hay coherencia, el programa es correcto. P. ej., sea

```
z: = x + y;
IF z > 0        THEN acción1
                ELSE acción2
```

Supongamos que la aserción de partida es que la suma de x e y es menor que 10 en valor absoluto. Expresamos la aserción:

$$-10 < x + y < 10$$

Tras efectuar: $z: = x + y$, será: $-10 < z < 10$
y tras la condición $z > 0$
al llegar a acción$_1$: $0 < z < 10$
y si llega a acción$_2$: $-10 < z \leqslant 0$

Las investigaciones realizadas en esta línea, en su estado actual, ponen en evidencia que el nivel de dificultad de verificar un programa es mucho mayor que el de construirlo; parece ser además (como se afirma en [YEH-77], Vol. 2) que hasta el momento sólo se han verificado comple-

tamente programas pequeños y de ciertas características muy restringidas. Ello hace pensar en que los *métodos inductivos* de verificación (verificar un programa ya construido), sin poner en duda su interés teórico, no tienen perspectivas de aplicación práctica. Por el contrario, la vía que día a día adquiere mayor relevancia es la de los *métodos constructivos,* es decir, aquellos que, o bien parten de las aserciones para deducir el programa, o bien construyen programa y verificación en paralelo. El último trabajo del prolífico Dijkstra [DIJK-76] es una aportación magistral en esta línea (aunque, valga como aviso, de difícil lectura para los no familiarizados con la formalización matemática).

Otros trabajos se han desarrollado en la línea de automatizar el proceso de verificación, mediante programas que verifican (por ejecución simbólica) o incluyendo sentencias de verificación en los lenguajes de programación.

Dos últimas anotaciones:

– Es evidente que hablamos del aspecto de la P.E. que aún no ha salido de los ámbitos académicos y universitarios. No sólo eso, sino que ha dado origen a trabajos absolutamente teóricos como la *teoría de esquemas* de los soviéticos Ianov y Ershov, los trabajos de Zohar Manna sobre *teoría de la computabilidad,* el estudio de Mills [MILL-72] basado en el anterior de Böhm y Jacopini [BOHM-66], etc. Pero no por ello ha de ser un aspecto desechado desde el punto de vista práctico. No hay que perderlo de vista, ya que representa claramente el futuro de la programación, y no está muy lejos el momento en que empezará a producir resultados prácticos. Recordemos que los lenguajes de alto nivel (FORTRAN, COBOL) se consideraban estudios teóricos y académicos en los años 50.

– Hay que advertir que el definir una aserción es una actividad humana, y por tanto sujeta a error. Puede ocurrir que una verificación sea convincente, pero falsa, si parte de aserciones aparentemente correctas (pero sólo aparentemente).

4.4 Estilo

Nos hallamos ante un tema, digamos "menor" de la P.E., pero no por ello de poca importancia. Recordemos el objetivo, citado en 3., de reducir la complejidad. Se trata de producir programas claros, legibles, elegantes, además de robustos, eficientes y correctos. La presencia de estructura fortalece la claridad, legibilidad y elegancia pero no lo es todo: hay muchas más normas y criterios a tener en cuenta. Los autores que han abordado este tema, lo que han hecho ha sido agrupar toda una serie de normas, consejos, criterios y advertencias destinadas a obtener programas bien escritos. Por analogía con los textos literarios, se habla de estilo. Las referencias [KERN-74 a], [KERN-74 b] y [LED-75] son ejemplos de este tipo. Es curioso el hecho de que en la segunda referencia, todos los programas puestos como ejemplo de "lo que no debe hacer el buen programador" se han sacado de textos de enseñanza de la programación.

En algunos textos actuales de enseñanza (como [SCHN-78]) se incluyen normas de estilo en forma de "cuñas" al texto.

4.5 Tipos de datos

El artículo "Notes on Data Structuring" de C.A.R. Hoare, incluido en [DIJK-72 a] contiene una serie de ideas y propuestas que, en síntesis y resumen, son las siguientes:

– Los programas son acciones que se aplican sobre unos objetos, que llamamos datos.

– A cada problema concreto se le puede asociar una representación concreta de estos datos, que simula o modela la realidad.

– Al representar el problema en un lenguaje de programación, nos vemos obligados a representar los datos ciñéndonos a unas estructuras fijas: las que tiene ese lenguaje.

– A cada dato, pues, se le asocia una representación de la clase de valores que puede adoptar: el *tipo.* P. ej., decimos REAL ALFA en FORTRAN o DCL X CHAR (3) en PL/I, en donde ALFA y X son *datos,* y REAL y CHAR (3), tipos, o formas de representación.

– Es restrictivo y limitado el verse obligado a representar todos los objetos del mundo real posibles con un repertorio finito de tipos (p. ej., en FORTRAN un caracter debe representarse en un campo INTEGER o REAL).

– Se propone la inclusión de una cláusula TYPE (tipo) en los lenguajes de alto nivel, de forma que se puedan definir *nuevos tipos* de variables, para utilizarlos en nuestros programas, acercando de esta forma el objeto representado a su representación.

Este concepto ha sido recogido por varios lenguajes, entre ellos, el ya citado PASCAL. P. ej., podemos definir el siguiente tipo:

TYPE PALO = (oros, sotas, espadas, bastos): con lo cual, de la misma forma que diríamos que la variable TIEMPO es entera: VAR TIEMPO: INTEGER; podemos afirmar que la variable NAIPE es del tipo PALO:

VAR NAIPE: PALO;

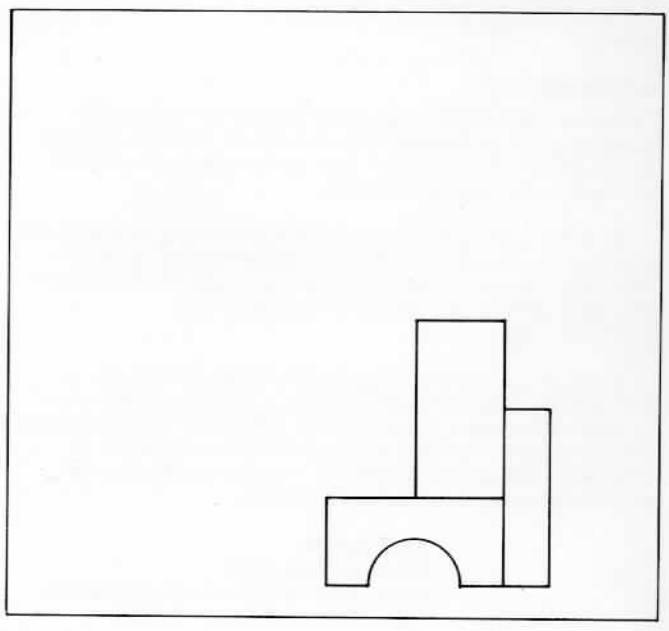
y en un momento dado, preguntar si el valor actual de NAIPE es espadas: IF NAIPE = espadas THEN...

Este pequeño ejemplo sólo manipula una de las posibilidades de la declaración TYPE, que es la enumeración de los posibles valores. Evidentemente, las posibilidades son mucho mayores. En la obra de Wirth "Algorithms + Data = Programs" (*Algoritmos + Datos = Programas*), ref. [WIRT-76] se revisan los conceptos clásicos de estructuras de datos (listas, árboles, etc., "clásicos" por su sistematización en la obra "The Art of Computer Programming", de D.E. Knuth) utilizando declaraciones TYPE para representarlos.

4.6 Programación concurrente, o paralela

La necesidad de programar sistemas operativos presenta un tipo de problemas específicos, que con el advenimiento de la multiprogramación, procesos interactivos, redes, etc., se han visto aumentados. En este tipo de trabajos es frecuente el que varias tareas o procesos tengan que solaparse en el tiempo, se ejecuten a la vez. Cuando esto ocurre, se dice que son procesos *concurrentes* o *paralelos.*

Evidentemente, programación concurrente o paralela es el nombre con el que se agrupan las técnicas que abordan esta problemática, que puede desglosarse en problemas de sincronización, determinismo (no se conoce el orden real en que se solapan las instrucciones de los procesos), exclusión mutua (dos o más procesos no puede utilizar al



mismo tiempo un recurso determinado) y "deadlock" (bloqueo de procesos entre sí). Aún cuando estos problemas se resolvían con lenguajes de bajo nivel, la creciente complejidad de los sistemas operativos aconsejaba su programación en lenguaje de alto nivel, los cuales podrían tener aplicación en otras áreas (tiempo real, simulación, etc.).

En esta línea surgieron propuestas de Dijkstra, Hoare, Brinch Hansen, entre otros, aportando conceptos como los semáforos, regiones críticas, procesos disjuntos, procesos que cooperan, regiones críticas condicionadas, monitores, etc.

Quizás sirva, para ilustrar estas ideas, la sugerencia realizada por Dijkstra en 1968, como sentencia compuesta para un grupo de procesos $S_1 \ldots S_n$ que se ejecutaran concurrentemente:

COBEGIN S_1 ; S_2 ; . . . ; S_n COEND

Los artículos de C.A.R. Hoare y Per Brinch Hansen, contenidos en la 3.ª parte de la referencia [GRIE-78] constituyen una excelente introducción al tema.

Sobre estas bases se han implementado los lenguajes CONCURRENT PASCAL, de Brinch Hansen y MODULA, de Niklaus Wirth: una vez más lo que era "propuesta" hace unos años, es ya realidad.

4.7 Desarrollo de software

Si se observa el diseño y creación de software, sobre todo de grandes proyectos, desde el punto de vista productivo o industrial, el producto final es lento de producción, caro, poco fiable y no acorde con las especificaciones, en la mayoría de los casos. Con la idea de trasladar todas las aportaciones de los teóricos de la programación al campo productivo, se habla actualmente de Software Engineering (*Ingeniería del Software*). Según define Turski en [GRIE-78] "software engineering es un término de relativamente nuevo cuño, para designar el rápidamente creciente cuerpo de conocimientos que conciernen al diseño de programas para computador, su composición y producción".

De acuerdo pues con esta definición, todo lo dicho en los puntos 4.1 a 4.6 cae dentro del ámbito del "software engineering". Hay que advertir que no todos los creadores de la P.E. están de acuerdo con esta idea (a Hoare, p. ej., le parece pedante; ver [GRIE-78]). El objetivo final de la ingeniería de software es producir programas correctos (que se hagan rápido y sean baratos no sirve de nada si son incorrectos). Pero dado que el estado actual de las técnicas de *verificación* no permite aplicarlas a grandes proyectos, el "software engineering" engloba técnicas de *validación por prueba* ("program testing"). Y dado que una prueba nunca permite asegurar que un programa es correcto, el objetivo se modifica: obtener programas *fiables*. De ahí el término Reliable Software (*software fiable*), que aparece con tanta frecuencia últimamente.

4.8 Enseñanza

Con el advenimiento de todas las nuevas técnicas de programación, se impone revisar a fondo muchos criterios de enseñanza. En concreto, hay que *enseñar a programar*, en lugar de *enseñar un lenguaje*.

Es obvio que aprender en la forma habitual para después conocer un conjunto de técnicas, para muchos "pintorescas", no conduce a nada. Hay que formarse desde el principio. (Ver [DIJK-71], [WIRT-72], [GRIE-73] o [SCHN-78]).

Decíamos que Dijkstra agrupó diversas reflexiones sobre la programación en un manual, y puso en la portada el nombre Structured Programming. De hecho, sólo una etiqueta. Pero por suerte las aguas están volviendo a su cauce: se empieza a hablar únicamente de *programación*, sin más, pero distinguiendo la buena de la mala.

Pere Botella
Horacio Rodríguez
Dept. Programació de Computadors
Fac. d'Informàtica de Barcelona

BIBLIOGRAFIA COMENTADA

[BAKE-75] Baker F.T. Structured Programming in a production programming environment. Proc. ICRS (IEEE), 1975. Presenta la metodología propuesta por la IBM Federal Systems Division, en forma de una jerarquía de técnicas.

[BERT-73] Bertini, Tallynneau. Le Cobol structuré. Ed. d'Informatique. Paris, 1973. Se presenta una propuesta de ordinogramas estructurados, orientados a COBOL. Muy sencillo.

[BOHM-66] Bohm y Jacopini. Flow-diagrams, Turing machines and languages with only two formation Rules. Comm. A.C.M., Vol. 9, p. 366-371. Mayo, 1966. Fundamento teórico de la P.E. Se demuestra su viabilidad.

[COSU-74] Varios. Special Issue: Programming. ACM Computing Surveys V. 6, n.º 4. Dic. 1974. Artículos de Denning, Brown, Yohe, Wirth, Knuth y Kernigham Plauger sobre programación.

[DIJK-68] E.W. Dijkstra. Go To Statement considered Harmful. Comm. ACM, vol. 11, Mayo, 1968. Carta al director. Primera aparición pública de la P.E.

[DIJK-71] E.W. Dijkstra. A short introduction to the art of programming. Report EWD316 Tech. Univ. Eindhoven. 1971. Texto de introducción a la programación usado por el profesor Dijkstra en sus clases.

[DIJK-72 a] Dahl, Dijkstra, Hoare. Structured Programming. Academic Press. London, 1972. Libro que agrupa tres monografías: Notes on Structured Programming – E.W. Dijkstra. Notes on data structuring – C.A.R. Hoare. Hierarchical program structures – O.J. Dahl y C.A.R. Hoare.

[DIJK-72 b] E.W. Dijkstra. The humble programmer. Comm. A.C.M., Vol. 15, Oct. 1972. Texto de la lectura del prof. Dijkstra al recibir el precio ACM Turing Award de 1971. De lectura fácil. Muy recomendable.

[DIJK-76] E.W. Dijkstra. A discipline of programming. Prentice Hall, 1976. Presentación de un método para construir programas y verificarlos en paralelo, junto con una propuesta de lenguaje. Se da a la programación categoría de disciplina científica.

[GRIE-73] R. Conway, D. Griess. An introduction to Programming: a Structured Approach using PL/I and PL/C. Winthrop, Cambridge, Mass., 1973. El título se explica por sí solo.

[GRIE-78] D. Gries (Ed.). Programming Methodology. Springer-Verlag. Nueva York, 1978. Colección de artículos de miembros del WG 2.3 de la I.F.I.P. Contiene 26 artículos agrupados en 5 partes:

1 – Viewpoints on programming
2 – The concern for program correctness
3 – Harnessing parallelism
4 – Data types
5 – Software development

[INFO-76] Infotech State of the Art Report. Structured Programming. Infotech, Maidenhead, 1976. Completo informe, conteniendo un análisis de la P.E. (Overview, Complexity in prog., Reliability and correctness), muchos artículos originales (invited papers) y una bibliografía comentada. Muy interesante, pero de precio inasequible.

[JACK-75] M.A. Jakson. Principles of program design. Academic Press, 1975. Descripción de la Metodología Jackson de programación. Contiene muchos ejemplos en COBOL.

[KERN-74 a] Kernigham and Plauger. Programming Style: Examples and Counterexamples. Incluido en [COSU-74]. Introducción al tema del estilo.

[KERN-74 b] Kernigham and Plauger. The elements of programming style. McGraw-Hill, 1974. Manual de estilo.

[KERN-76] Kernigham and Plauger. Software Tools. Addison Wesley, 1976. Revisión de los conceptos de programación (sort, merge, tablas, etc.) desde la óptica de la P.E.

[KNUT-74] D.E. Knuth. Structured programming with go to statements. Incluido en [COSU-74] y en [YEM-77] Vol. I. Conceptos e ideas sobre programación. Uso de la sentencia go to para simular estructuras.

[LEDG-75] Henry Ledgard. Programming Proverbs. Hayden Books, 1975. Informal colección de normas ("proverbios") de programación.

[MILL-72] Harlan D. Mills. Mathematical Foundations for Structured Programming. Report FSC-72-6012 IBM Corp. Nueva aproximación y aportaciones al teorema de Bohm y Jacopini [BOHM-66].

[NASS-73] Nassi y Schneidermann. Flowchart Techniques for Structured programming. A.C.M. SIGPLAN Vol-8-8. Agosto 1973. Propuesta de ordinogramas para P.E.

[SCHN-78] Schneider, Weingart y Perlman. An Introduction to programming and problem solving with PASCAL. Wiley & Sons, 1978. Introducción a la programación, con el criterio de la P.E. y usando PASCAL como vehículo o soporte de enseñanza.

[WARN-73] J.D. Warnier. Programación lógica (2 tomos). Edit. Tecn. Asociados. Barcelona, 1973. Contiene la metodología de Warnier de construcción de programas (L.C.P.). Muy extenso, y poco denso.

[WARN-75] J.D. Warnier. Síntesis de programación lógica. Edit. Tecn. Asoc. Barcelona, 1975. Contiene una exposición no muy sistemática, de la metodología Warnier. Resumen de [WARN-73] Abundantes ejemplos.

[WIRT-72] Niklaus Wirth. Systematic Programming. An introduction. Prentice Hall 1972. Texto de introducción a la programación. Excelente. Incluye verificación, refinamiento por pasos, etc., así como el lenguaje PASCAL.

[WIRT-74 a] N. Wirth. On the composition of well-structured programs. Incluido en [COSU-74]. Descripción de los conceptos básicos de la P.E., en forma de introducción.

[WIRT-74 b] K. Jensen, N. Wirth. PASCAL User Manual and Report. Manual del lenguaje PASCAL. Definición del standard para cualquier implementación.

[WIRT-76] N. Wirth. Algorithms + Data = Programs. Prentice Hall, 1976. Revisión, estructurada y con PASCAL, de todos los tópicos de programación (listas, árboles, matrices, sort, búsqueda, recursividad, compilación, etc.).

[YEH-77] Yeh R.T. (Ed.). Current Trends in Programming Methodology. Prentice-Hall Inc.
Colección de artículos, agrupados en 4 volúmenes:

I — Software specification and design
II — Program validation
III — Modelling
IV — Data structuring.

[YOUR-75] Yourdon E., Constantine L. Structured Design. Yourdon Inc. New York, 1975. Desarrollo de la metodología definida por Myers, Yourdon y Constantine en diversos trabajos. Util para programación modular y orientada a gestión.

P. Botella
H. Rodríguez