Universidad Nacional
Autónoma de México

Unidad Académica de los Ciclos
Profesional y de Posgrado del Colegio
de Ciencias y Humanidades

Instituto de Investigaciones
en Matemáticas Aplicadas y
en Sistemas

Aplicación de la Ingeniería de Software en el Desarrollo Interactivo de Programas Estructurados

Tesis que presenta
Mónica Ardisson Pérez
Para obtener el grado
de Maestro en Ciencias
de la Computación

México, D.F. **Noviembre de 1989.**

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

CONTENIDO

Capítulo 1

Capítulo 2

Capítulo 4

El trabajo descrito en esta tesis está basado en el diseño e implementación de una herramienta (*PROGRE*) para ayudar a programar. Dicha herramienta sigue los lineamientos de la Ingeniería de Software e implementa teorías y técnicas de la programación estructurada.

PROGRE tiene dos objetivos:

1) ayudar al programador a, en una forma computarizada, desarrollar sus programas durante las diferentes fases del ciclo de vida de los mismos y,

2) ayudar a aprender a programar en forma estructurada, auxiliando al aprendiz en la algorítmica y dirigiéndole la sintaxis.

Estructura de la tesis

En el capítulo uno se proporciona una introducción a la Ingeniría de Software. Las dos primeras secciones de este capítulo intentan describir qué se entiende por esta disciplina y cómo surgió. La tercera y última sección se encarga de determinar los requisitos mínimos que debe cumplir el software para ser de alta calidad.

Dado que *PROGRE* es una ayuda para la creación de programas estructurados, en el segundo capítulo se plantea, brevemente, en qué consiste y algunas de las técnicas de programación que propone la teoría de la programación estructurada. También se da una reseña de lo que se entiende por ciclo de vida del desarrollo del software y en qué consiste cada una de las fases que lo componen.

El capítulo tres hace una revisión de los diferentes tipos de herramientas que le proporciona la Ingeniería de Software al programador, con el fin de ayudarle a alcanzar los objetivos que se buscan en dicha disciplina (mejorar la calidad y facilitar la producción de software).

El diseño, implementación y funcionamiento de *PROGRE* se describen en el capítulo cuatro. La primera sección introduce el lenguaje gráfico (GAL) en el cual se basa el sistema para el desarrollo de los programas del usuario. La segunda sección describe el funcionamiento de la herramienta, la sintaxis de la interface con el usuario y, por medio de un ejemplo, muestra cómo el programador puede utilizar *PROGRE* para

desarrollar su programa, o bien, cómo el novato de la programación puede aprender a programar en forma estructurada.

Como resultado del trabajo realizado para la elaboración del sistema que se reporta en la presente tesis, se logró obtener algunas conclusiones que se indican en el capítulo cinco. Este capítulo también propone una serie de actividades posibles para realizar en un futuro, con el fin de mejorar y ampliar a *PROGRE*.

El último capítulo reporta las referencias utilizadas, tanto para el desarrollo del sistema como para la elaboración de la tesis.

Finalmente, la tesis contiene dos anexos que son copias de los artículos publicados sobre el presente proyecto. El anexo A es el trabajo presentado en el "*Congreso Nacional sobre Informática*" organizado por la ANIEI, en Monterrey, Nuevo León, del 13 al 15 de octubre de 1988. El segundo anexo corresponde a la ponencia presentada en el "*Primer Foro de Avances de Investigación*" organizado por la Maestría en Ciencias de la Computación del CCH con sede en el IIMAS-UNAM, del 27 al 29 de mayo de 1989.

CAPITULO 1

Ingeniería de Software

Una computadora por sí misma, la "máquina" física, es de poca utilidad. La idea de construir una computadora nació de la necesidad humana de ser ayudado.

Al principio los programas se escribían directamente en el código de la máquina que se iba a utilizar. Era un proceso muy complicado y tardado y, obviamente, los programas eran difíciles de depurar y casi imposibles de mantener. Se podría decir que los problemas del software se originaron, principalmente, por las severas restricciones impuestas por el hardware con que se contaba en esa época. Al transcurrir el tiempo, el hardware sufrió grandes avances tecnológicos; sin embargo, aunque éste se volvió mucho más flexible, los problemas con el software no disminuyeron; por el contrario, eran más grandes que nunca. La complejidad de los sistemas de software se volvió inmanejable y las consecuencias naturales de ello fueron, principalmente, la falta de confiabilidad y la calidad pobre, retardos en los tiempos de entrega, costos excesivos en la producción y en el mantenimiento del producto de software. La situación realmente empezó a cambiar a mediados de los sesentas con el surgimiento de la **Ingeniería de Software**.

1.1 El Desarrollo de la Ingeniería de Software

> ***"Mientras no hubo computadoras, la programación no fue un problema; cuando hubo algunas computadoras sencillas la programación constituyó un problema ligero, y ahora que tenemos computadoras gigantescas la programación se ha convertido a su vez en un problema gigantesco."***
>
> **E. W. Dijkstra [Dijkstra 72b]**

La necesidad de aproximaciones sistemáticas para el desarrollo y mantenimiento de productos de software surgió en los sesentas. Durante esa década, se crearon tres generaciones de computadoras y al mismo tiempo se desarrollaron técnicas de multiprogramación y de tiempo compartido. Estas capacidades proporcionaron la tecnología para la creación de sistemas de cómputo interactivos, multi-usuario, en-línea y de tiempo real. Entre las nuevas aplicaciones de las computadoras basadas en esta nueva tecnología se incluyeron sistemas para reservaciones en aerolíneas, información médica, tiempo compartido de propósito general, control de procesos, guía navegacional y control militar.

Las primeras computadoras se usaron, principalmente, en el campo de la ciencia aplicada. El programador no requería de un conocimiento informático especial, únicamente necesitaba conocer un lenguaje de programación. El programador era también, generalmente, el usuario. Los programas se utilizaban sólo ocasionalmente y las tareas rara vez eran especificadas. La única dificultad era garantizar que el programa fuera correcto y eficiente. Los problemas que tenían que ser resueltos en esa época eran relativamente sencillos comparados con los de hoy en día y, por lo mismo, los programas eran relativamente pequeños. Los errores de programación que se cometían a fines de los años cuarentas y a principios de la década de los cincuentas no representaban un gran problema. Por lo general, estos errores eran producto de la inexperiencia en el uso de las nuevas máquinas. No obstante, esto se volvió un problema más grande al utilizar los sistemas de programación en la solución de aplicaciones comerciales y científicas complejas. Con el paso del tiempo, grupos completos de programadores tuvieron que trabajar en la producción de sistemas que eran utilizados por varios usuarios.

La especificación del problema y las demandas sobre el sistema cambiaban frecuentemente durante la fase de diseño y posteriormente, cuando el programa ya estaba en uso. Estos cambios en la programación provocaron que ya no sólo era importante que un programa fuera correcto y eficiente, sino que el grado de complejidad debido a la descomposición de un problema en subproblemas, la especificación de las interfaces, la seguridad y la confiabilidad, la flexibilidad, la documentación, el mantenimiento y la organización del proyecto se convirtieron en los principales problemas de la producción de grandes sistemas de programación. Esto llevó a tal extremo las dificultades en el diseño y la producción de software que en 1965 surgió el término "*crisis del software*".

Las posibilidades que emergieron con las nuevas generaciones de computadoras excedieron por mucho a las técnicas de programación que habían sido desarrolladas hasta ese momento. Sin embargo, la importancia del crecimiento económico de la producción de software y la enorme expansión de la industria del procesamiento de datos, que implicó el desarrollo de numerosos sistemas de programación grandes, forzó más y más la demanda de una mejora en la tecnología de programación, fundamentada en investigaciones en el campo de la ciencia de la computación.

Los intentos para lograr una investigación de técnicas de programación aceptables dieron lugar a dos conferencias sobre Ingeniería de Software, organizadas por la OTAN en *Garmish, Alemania Occidental* en 1968, y en *Roma, Italia* en 1969 para

considerar el crecimiento de los problemas de la tecnología de software. Esas reuniones estimularon el interés en los procesos técnicos y de administración utilizados para desarrollar y mantener software de cómputo [NAU 76].

Esencialmente, la Ingeniería de Software busca el desarrollo de un software menos costoso y más confiable. Los trabajos principales han consistido en analizar, científicamente, la producción de software, considerándolo como un proceso coherente y, sobre todo, centrando el interés de la investigación en resolver los problemas de especificación, diseño métodico de programas, requerimientos de los lenguaje de programación, organización de los proyectos, control de la calidad, documentación, mantenimiento y automatización de la producción del software.

1.2 Concepto de la Ingeniería de Software

Los objetivos primarios de la Ingeniería de Software son mejorar la calidad de los productos de software e incrementar la productividad y la satisfacción del trabajo de los ingenieros de software. Uno de sus principios fundamentales consiste en diseñar productos de software que minimicen la distancia intelectual entre el problema y la solución. La variedad de aproximaciones a los desarrollos de software está limitada únicamente por la creatividad del programador.

La Ingeniería de Software es interdisciplinaria. Se auxilia de diversas disciplinas tales como: matemáticas, para analizar y certificar algoritmos; ingeniería, para estimar costos y establecer compromisos; administración, para definir los requerimientos, valorar los riesgos, supervisar al personal y monitorear los progresos del proyecto.

La Ingeniería de Software difiere de la programación tradicional en el sentido de que la Ingeniería, como técnica, se utiliza para especificar, diseñar, implementar, validar y mantener los productos del software dentro del tiempo y de las restricciones de presupuesto establecidas por el proyecto. Además, está relacionada con los eventos administrativos que caen fuera del dominio de la programación tradicional.

La Ingeniería de Software es una disciplina pragmática que se apoya en la ciencia de la computación para proporcionar principios científicos en la misma forma en que las disciplinas de la Ingeniería tradicional, tales como la ingeniería eléctrica y la ingeniería química, se apoyan en la física y en la química. La Ingeniería de Software, siendo una actividad de labor intensiva, requiere tanto de las habilidades técnicas como del control administrativo. La ciencia de la administración proporciona las bases para la administración de los proyectos de software. Los sistemas de cómputo deben desarrollarse y mantenerse dentro de los tiempos y los costos estimados; la economía, a su vez, proporciona la estimación de los recursos y el control de los costos. Las actividades de la Ingeniería de Software ocurren dentro de un contexto organizacional y se requiere, por lo tanto, de un alto grado de comunicación entre usuarios, directores, ingenieros de software, ingenieros de hardware y otros tecnólogos. Una buena

comunicación interpersonal oral y escrita es crucial para el ingeniero de software.

Como puede observarse, la Ingeniería de Software es una nueva disciplina tecnológica, diferente de ciencias tales como: computación, administración, economía, comunicación e ingeniería; pero basada en los principios de estas ciencias para resolver sus propios problemas.

El término de Ingeniería de Software obviamente intenta ser una provocación, e indica que la producción de programas comerciales es una disciplina de la ingeniería; sin embargo, no existe una definición del concepto que sea aceptada totalmente. Entre otras podemos encontrar la de Boehm [**Boehm 79**], quien define la Ingeniería de Software como: "*La aplicación práctica del conocimiento científico al diseño y a la elaboración de programas de cómputo y de la documentación asociada requerida para desarrollarlos, operarlos y mantenerlos*". Como Boehm señala, el término "diseño" debe ser interpretado ampliamente para incluir actividades tales como análisis de requerimientos de software y de rediseño durante las modificaciones del software.

En Dennis [**Dennis 75**] encontramos la siguiente definición: "*La Ingeniería de Software es la aplicación de principios, habilidades y arte para el diseño y la elaboración de programas y sistemas de programación.*"

D.L. Parnas [**Parnas 74**] escribe: "... *Ingeniería de Software es programar al menos bajo una de las siguientes dos condiciones:*

(1) Hay más de una persona involucrada en la elaboración y/o el uso del programa y

(2) se llegará a producir más de una versión del programa."

Para Pomberger [**Pomberger 84**] la Ingeniería de Software puede definirse como: "*La aplicación práctica del entendimiento científico a la producción comercial y al uso de software confiable y eficiente.*"

Para Fairley [**Fairley 85**] Ingeniería de Software es: "*La disciplina tecnológica y administrativa relacionada con la producción sistemática y el mantenimiento de los productos de software que son desarrollados y modificados en el tiempo y dentro de los costos estimados.*"

El Glosario estándar de la IEEE de Ingeniería de Software [**IEEE 83**] por su parte define Ingeniería de Software como: "*La aproximación sistemática al desarrollo, operación y mantenimiento del software,*" *donde* '**software**' *se define como "los*

programas de cómputo, procedimientos, reglas y posiblemente documentación asociada, así como los datos pertenecientes a la operación de un sistema de cómputo."

Zelkowitz, Sharo y Gannon [**Zelkowitz 79**] opinan que la Ingeniería de Software: "*Es la especificación, diseño, implementación, prueba y operación de los programas.*"

Finalmente, F.L. Bauer [**Bauer 75**] escribe: [El objetivo de la Ingeniería de Software es:] "*Obtener software comercial que sea confiable y que trabaje eficientemente en máquinas reales.*"

Estas definiciones no son más que un pequeño subconjunto de las que se pueden encontrar en la literatura. Casi podríamos decir que cada autor tiene la suya propia. Por otro lado, se puede observar que al analizar las diferentes opiniones o sugerencias sobre lo que es la Ingeniería de Software existe poca diferencia entre ellas, teniendo todas un común denominador: "*producir programas de buena calidad*". Por programas de buena calidad se pueden entender muchas cosas y regresar al problema anterior donde cada autor puede definirlos de diferente forma. En la siguiente sección intentaremos establecer cuáles deberían ser las características de calidad del software.

La definición de Ingeniería de Software que, posiblemente, es aceptada por la mayoría de los autores y sobre la que basaremos esta tesis es: "*La Ingeniería de Software es el estudio, creación y aplicación práctica de teorías y técnicas para la elaboración de software de buena calidad durante todo su ciclo de vida de desarrollo (especificación, diseño, codificación, prueba e instalación, documentación y mantenimiento).*"

La producción de programas grandes genera nuevos problemas, de naturaleza distinta a la producción de programas pequeños y muestra muchas semejanzas con la producción de otros productos técnicos. Los problemas principales son:

a) dominio de la complejidad,

b) descomposición de un problema grande en partes, las cuales serán resueltas por varios grupos,

c) especificación de las interfaces entre estas partes,

d) organización del proyecto,

e) eficiencia del producto de software,

f) documentación y mantenimiento de los sistemas y

g) portabilidad y adaptabilidad.

Para resolver estos problemas se requiere hacer un examen científico, con el fin de establecer los prerrequisitos para el desarrollo de métodos y herramientas que apoyen el desarrollo y la producción del software. Por lo tanto, se espera de la Ingeniería de Software que proporcione métodos, herramientas, normas y ayudas que hagan posible manipular los problemas técnicos (*especificación, diseño, codificación, prueba, eficiencia, documentación y mantenimiento*) y los problemas organizacionales (*organización de proyectos y especificación de interfaces*), que surgen en la producción de software y en el proceso de producir y aplicar el software en forma comercial.

1.3 Características de la calidad del software

Hasta el momento no existe una definición precisa de lo que significa calidad del software. No obstante, existe un concenso entre los profesionales de la materia de que esta calidad implica mucho más que el hecho de que el software sea correcto. A continuación se definirán algunas de las características primarias que debería tener un software de calidad:

Funcionamiento correcto

Un programa es correcto si concuerda con su especificación; de otro modo resulta incorrecto. Es decir, por funcionamiento correcto de un programa se entenderá que el programa satisface las especificaciones funcionales que son la base del desarrollo de programas. De tal forma que esta característica se relaciona sólo con la conformidad de las especificaciones funcionales y con el texto del programa y por lo tanto no está relacionado con el uso actual del programa. En otras palabras, si p es la probabilidad de que un determinado programa sea correcto, entonces la probabilidad S de que un sistema de programación, que está constituído por n programas, sea correcto, está dado por:

S = p ** n. **Si n es grande, p debe ser muy cercano a 1 para que S sea significativamente diferente de 0 [Dijkstra 72b].**

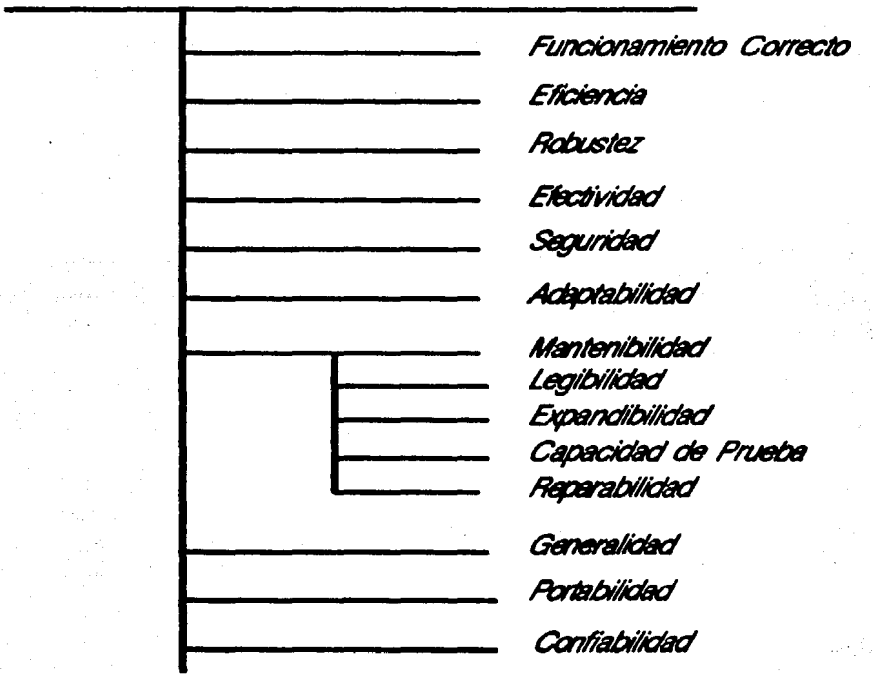


Figura 1.1 Características de calidad del software

Un sistema debe estar diseñado de tal forma que resulte correcto en todos los estados del desarrollo. En cada uno de los niveles de diseño, codificación o prueba, es necesario mostrar que el funcionamiento correcto es preservado por cualquiera de las nuevas adiciones hechas al sistema.

Eficiencia

Por eficiencia de un programa se entenderá la habilidad del programa para ejecutar la tarea con el uso óptimo de todos los recursos. Por recursos podemos entender espacio de memoria, tiempo de CPU, canales de entrada y salida, etc.

Robustez

Se dice que un sistema de software es robusto cuando las consecuencias de errores en los datos de entrada o en el hardware, relacionados con una determinada aplicación, son inversamente proporcionales a la probabilidad de que ocurra un error en esa aplicación [Kopetz 79].

Efectividad

Un sistema es efectivo cuando no sólo realiza las tareas que tiene asignadas sino que también funciona durante un periodo largo. La efectividad como una función del tiempo depende no únicamente de la confiabilidad, sino también de la mantenibilidad.

Seguridad

La seguridad es una medida de la probabilidad de que el sistema desarrollado por un usuario pueda destruir o hacer referencia, en forma accidental o intencional, a datos que son propiedad de otro usuario o interferir con la operación del sistema. Las medidas de seguridad comprenden el cuidadoso aislamiento de los datos del usuario; el aislamiento de los programas del usuario con respecto a los de otros usuarios; y el aislamiento de los programas del usuario con respecto a los programas del sistema operativo.

Adaptabilidad

La adaptabilidad es la facilidad con la que se puede añadir al producto otras funciones.

Mantenibilidad (capacidad de hacer mantenimiento)

Por mantenibilidad de un programa se entenderá la facilidad con la que los errores pueden localizarse y corregirse, y la facilidad con la que las funciones del programa pueden modificarse o expanderse. La mantenibilidad y la confiabilidad son compatibles debido a que la primera está muy relacionada con la adaptabilidad. Además, las técnicas de confiabilidad, tales como detección de fallas y aislamiento de errores, tienen un efecto positivo en ella. Esta definición indica también que la mantenibilidad de un programa depende de que sea legible, expandible y posible de probar.

La *legibilidad* de un programa depende de la forma en que ha sido representado, del estilo de la programación y su consistencia, de la legibilidad del lenguaje de programación, de la estructura del sistema y, más decisivamente, de la calidad de la documentación.

La *expandibilidad* de un programa depende de si es o no posible insertar los cambios deseados en los puntos lógicamente apropiados sin que se produzcan efectos no deseados. Esto depende especialmente de la modularidad y de la estructura del programa, así como de la legibilidad y de la disponibilidad de una buena documentación.

Por la *capacidad de prueba* de un programa se entenderá la facilidad con que un programa permite la prueba en su ejecución y la localización de los errores que haya en él. La facilidad con que un programa puede probarse depende principalmente de la modularidad y de la estructura del programa. Los programas modulares que están bien estructurados permiten una prueba sistemática mejor, paso por paso.

Kopetz [**Kopetz 79**] define la mantenibilidad de un sistema como la probabilidad de que después de la aparición de un error el sistema sea regresado a una condición operacional en un tiempo dado. El tiempo promedio que se tarda para corregir un error en el sistema, se conoce como el "*Tiempo Medio de Reparación*", abreviado como *MTR*. Este intervalo comienza con la aparición de un error y termina cuando el sistema regresa a su modo operacional nuevamente. La mantenibilidad depende de la disponibilidad y de la competencia del personal de mantenimiento, de la disponibilidad de las partes de reserva (discos, circuitos integrados, cables, etc.) y de la facilidad con la que el sistema puede ser reparado, es decir, de la reparabilidad.

La *reparabilidad* de un sistema es la probabilidad de que un error sea reparado en un tiempo dado por personal de servicio de habilidad promedio, suponiendo que las partes de reserva están disponibles. Mientras que la mantenibilidad es también

una función de la organización del servicio, la reparabilidad es una característica del sistema, esto es, depende de la documentación, del desarrollo del sistema, etcétera.

Generalidad

La generalidad es una medida del número, potencia y alcance de las funciones desarrolladas por el usuario.

Portabilidad

Por *portabilidad* de un programa se entenderá la facilidad con la cual el programa puede ser implementado en diferentes sistemas de hardware. La portabilidad de un programa es, por lo tanto, una función de su independencia del hardware. La independencia del hardware se determina, por ejemplo, por la elección del lenguaje de programación y por el grado de uso de las funciones especiales del sistema operativo y de las propiedades del hardware. De tal manera que la portabilidad depende en gran medida, de si el programa está o no organizado de tal forma que las partes dependientes del sistema estén agrupadas en unidades fácilmente intercambiables. Un programa se considera portable si el esfuerzo requerido para adaptarlo es mucho menor que el esfuerzo requerido para reprogramarlo.

Debido a los altos costos del desarrollo de los productos de software y del rápido desarrollo del hardware, la portabilidad es una característica de importancia creciente. La portabilidad de un producto de software se fija durante la implementación. Es claro que un sistema que va a ejecutarse en una determinada máquina debe respetar las convenciones que esa máquina impone. El programador debe, sin embargo, asegurarse de que las partes adaptadas a las peculiaridades de esa máquina sean fáciles de localizar y de modificar y de que la estructura básica del sistema de programación se preserve.

Este proceso comienza con la elección del lenguaje de programación. En ninguna computadora existe un compilador para cada uno de los lenguajes de programación existentes, por ello, para que un programa sea portable, debe elegirse un lenguaje cuyo compilador sea considerado como un estándar en la mayoría de las computadoras.

Para obtener un alto grado de portabilidad, el programador debe asegurarse de que las unidades que dependen del sistema no estén dispersas por todo el sistema, sino que se localicen en pocos módulos. En caso de que se requiera transportar el sistema de una máquina a otra, sólo esos módulos tendrán que reescribirse.

Existen técnicas para incrementar el grado de portabilidad para los lenguajes de programación convencionales. Una discusión detallada sobre ello se encuentra en los trabajos de Tanenbaum [**Tanenbaum 78**] y de Brown [**Brown 77**].

Confiabilidad

El concepto de *confiabilidad* no debe confundirse con el hecho de que un programa funcione correctamente. Un programa correcto es aquél que cumple sus especificaciones. En contraste, un programa confiable no necesita ser correcto, pero sí dar respuestas aceptables aun cuando los datos o el medio no cumplan con los supuestos acerca de ellos. Parnas [**Parnas 75**] define un sistema como correcto si carece de fallas y sus datos internos no contienen errores; por otro lado, es confiable si las fallas no evitan que su operación sea satisfactoria.

Los sistemas operativos con procedimientos de *fallas suaves* ("fail-soft") ilustran la diferencia entre un programa confiable y uno correcto. Un error detectado ocasiona que el sistema se interrumpa, posiblemente sin perder información, restableciéndose después de que ocurre la recuperación de errores. Tal sistema no puede ser correcto debido a que está sujeto a errores; pero es confiable porque su operación es consistente.

La confiabilidad de un programa se determina, por un lado, por su funcionamiento correcto y, por el otro, por su disponibilidad. El funcionamiento correcto de un programa se define sin hacer mención del intervalo de tiempo durante el cual debe satisfacer una especificación dada. Este tiempo determina la confiabilidad del programa.

La confiabilidad de un programa se puede definir, según Kopetz [**Kopetz 79**], como la probabilidad de que un programa pueda proporcionar una función (determinada por la especificación) para un cierto número de casos, fijando las condiciones de los datos de entrada durante un intervalo de tiempo dado (bajo la suposición de que tanto el hardware y los datos estén libres de errores). De esta forma, la confiabilidad de un programa es su probabilidad de supervivencia. Puede describirse la función de confiabilidad **R(t)**, donde R cumple las siguientes propiedades:

$R(0) = 1$ Indica que el programa funciona bien al comenzar el intervalo

$R(\infty) = 0$ Indica que el programa no funciona cuando $t=\infty$ (∞=infinito)

Donde en el intervalo $(0, \infty)$ la función decrece monótonamente.

Si Q(t) es la falta de confiabilidad de un programa, entonces se tiene que para toda t:

$$Q(t) + R(t) = 1.$$

Existen considerables implicaciones económicas relacionadas con la falta de confiabilidad del software. Del esfuerzo total requerido para desarrollar y mantener un sistema se calcula que la tercera parte o, en ocasiones, hasta la mitad de éste es empleado en probarlo y depurarlo. Desde que se inicia el desarrollo de un sistema de cómputo grande, generalmente, la mayor parte de los recursos se asignan al software y los costos directos de la falta de confiabilidad representan una fracción substanciosa de los mismos. Además, si se consideran los costos indirectos que ocasionan los errores, la importancia económica de la confiabilidad del software es aún más pronunciada.

Todos los principios y prácticas para obtener confiabilidad pueden agruparse, esencialmente, en cuatro módulos: aquéllos que se aplican para evitar las fallas; los que se encargan de detectar las fallas; los que se ocupan de corregirlas y finalmente, los que se encargan de su tolerancia.

Si se tuviese la suerte de encontrar una metodología de diseño que llevara a crear un producto de software altamente confiable, se podrían eliminar completamente, al menos en teoría, todas las actividades que están relacionadas con probar, depurar y manipular los errores que ocurren al tiempo de ejecución. Se han creado varios métodos para ayudar a la obtención de sistemas de software más confiables, los cuales pueden clasificarse fundamentalmente en dos:

1) Probar y depurar.

2) Incluir redundancia con el fin de detectar y corregir errores que se muestren durante el uso de los sistemas de software.

La Ingeniería de Software pretende producir un software de buena calidad, en el que estén presentes las características descritas anteriormente.

CAPITULO 2

Programación Estructurada

Con el transcurso del tiempo, las bases ideológicas de la programación han sufrido un fuerte cambio. Antes se pensaba que un buen programa debía ser correcto, eficiente y creativo (lo que significaba "oscuro"). Posteriormente, estas ideas cambiaron aunque permaneció de importancia primaria el hecho de que fuera correcto. Se pedía también que fuese mantenible (facilidad de encontrar y corregir errores) y modificable (claridad del programa) lo que reemplazó la eficiencia y la oscuridad de los programas. Estas nuevas metas crearon la necesidad de desarrollar un nuevo estilo de programación llamado "**Programación Estructurada**". Los principios de esta teoría se deben principalmente a Bohm y Jacopini [**Bohm 66**], Floyd [**Floyd 67**], Dijkstra [**Dijkstra 69**], Hoare [**Hoare 69**], Dahl [**Dahl 72**] y Wirth [**Wirth 73**].

La idea principal de esta teoría es asegurar que existe una correspondencia entre la notación estática de un algoritmo y su comportamiento dinámico en la ejecución, para mantener claro el control del flujo, reduciendo así la probabilidad de errores durante el desarrollo del sistema y haciendo más simple la verificación del algoritmo ([**Kopetz 79**] y [**Pomberger 84**]). El punto más importante es tratar de evitar las estructuras de datos no acotadas que resultan del uso indisciplinado de algunas estructuras de control, tales como el GOTO.

El término "*Programación Estructurada*" fue introducido por Dijkstra en 1972 [**Dijkstra 72b**] y definido más exactamente por Mills ese mismo año en sus "*Fundamentos Matemáticos de la Programación Estructurada*" [**Mills 72**]. Subsecuentemente, sin embargo, el término ha sido usado en tantos contextos que el significado cada vez se ha ido volviendo más general y más oscuro [**Kopetz 79**].

Mills [**Mills 72**] escribe: "*La Programación Estructurada... identifica al proceso de programación de funciones matemáticas, con una expansión por pasos sucesivos en estructuras de conectivas lógicas y subrutinas. Esto se lleva a cabo hasta que las subrutinas derivadas puedan ser ejecutadas directamente en el lenguaje de programación que está siendo utilizado*".

Para Paul Oliver [**Oliver 75**] la *Programación Estructurada* puede verse como un conjunto de reglas diseñadas para mejorar la legibilidad de un programa, reduciendo así las diferencias de estilo entre los programas escritos por varios individuos y mejorando la habilidad del programador para entender y modificar los programas existentes.

Oliver determina como reglas que debe cumplir la *Programación Estructurada*:

a) el uso de convenciones de formateo (v.g. indentación);

b) el tamaño de las subrutinas debe ser limitado;

c) cada subprograma debe tener una única entrada y una única salida;

d) el control de flujo debe estar limitado al uso de las tres estructuras básicas: secuencial, condicional e iterativa.

Por otro lado Myers [**Myers 76**] dice que su definición favorita de *Programación Estructurada* es: *"la actitud de escribir código con la intención de comunicarse con las personas en lugar de con las máquinas: y para alcanzar esto, los requerimientos mínimos que debe cumplir un programa estructurado son:*

1. *El código debe estar construído de secuencias de las tres proposiciones básicas: secuencial, condicional e iterativa.*

2. *El uso de proposiciones GOTO debe evitarse siempre que sea posible. En particular, aquél que es el peor tipo de GOTO, el que regresa el control a una proposición anterior en el texto del programa.*

3. *El código debe estar escrito siguiendo un estilo aceptable.*

4. *El código debe estar indentado correctamente en el texto, de tal forma que las interrupciones que aparezcan en la secuencia de ejecución puedan seguirse fácilmente.*

5. *Debe haber un único punto de entrada y un único punto de salida para cada módulo.*

6. *El código debe estar físicamente segmentado en el texto del programa para mejorar la legibilidad. Las proposiciones ejecutables de un módulo deben aparecer en una página del listado únicamente.*

7. *El código debe representar una solución simple e íntegra del problema".*

Aunque los siete puntos anteriores tratan de ilustrar los objetivos de la *Programación Estructurada* (complejidad mínima, claridad del pensamiento del programador y programas legibles), quedan algunos aspectos poco definidos. Por ejemplo, el punto 3 habla de un estilo aceptable de programación pero no determina qué significa "aceptable". En cuanto a la identación de un programa (punto 4), podríamos decir que depende mucho del sentido estético del programador, así como del tipo de estructuras y anidamientos que se estén utilizando. En el punto 6 se dice que las proposiciones ejecutables de un módulo deben aparecer en una sola página del listado, pero no aclara de que tamaño debe ser la página (líneas por página). A nuestro modo de ver Myers deja mucho a juicio del programador.

Tanto Bohm y Jacopini [**Bohm 66**] como Mills [**Mills 72**] han demostrado que la solución algorítmica de cualquier problema arbitrario puede describirse como una combinación de las estructuras de secuencia, selección y repetición.

Las proposiciones disponibles en los lenguajes de programación para soportar esos conceptos son:

- proposición sencilla
- llamada a subrutina
- decisión binaria
- decisión generalizada
- iterativa condicional
- repetitiva condicional
- iterativa con índice

Sin embargo, las proposiciones de decisión binaria, iterativa condicional y la proposición sencilla constituyen el conjunto de estructuras de control sugerido con mayor frecuencia para programar en esta disciplina; no obstante, no hay nada sagrado acerca de ello [**Zelkowitz 79**].

Se pueden identificar como ventajas de la *Programación Estructurada* el incremento en la productividad del programador y claridad y legibilidad de los programas [**Yourdon 74**]. Por otro lado, la principal crítica que se hace a la teoría de la *Programación Estructurada* es que, en esencia, no es más que programación que evita cuidadosamente el uso de transferencias de control (proposiciones GOTO por ejemplo). Se puede, sin duda, llegar a esta conclusión observando el resultado total, pero en la dirección inversa, ya que, en efecto, el método de refinamiento por pasos sucesivos (*step-wise refinement*) de la tarea de programación nos lleva, automáticamente, a programas libres de proposiciones GOTO. La discusión acerca de que la *Programación Estructurada* surgió probando que todos los programas pueden ser formulados sin proposiciones GOTO está basada, por lo tanto, en este malentendido [**Wirth 74**].

Uno de los argumentos más comunes contra la *Programación Estructurada* es que conduce a escribir programas menos eficientes. Es decir, el incremento en el énfasis que se pone en las llamadas a subrutinas, como una alternativa a las proposición GOTO, aumenta el tiempo de procesamiento del programa y puede, tal vez, añadir cantidades significativas de requerimientos de memoria.

Aunque los elementos de la *Programación Estructurada* evolucionaron lentamente desde los inicios de la década de los sesentas, no fue sino hasta 1968 cuando el concepto recibió una atención amplia con la publicación de la famosa nota de Dijkstra "*Go to statements considered harmful*" [**Dijkstra 68b**]. En ella, Dijkstra señala que debería haber una correspondencia cercana entre el texto del programa y su flujo de

ejecución (un programa debe ser capaz de ser leído de arriba hacia abajo) y el uso excesivo y descuidado de las proposición GOTO interfiere con esta correspondencia. Es por esto, que la *Programación Estructurada* es llamada frecuentemente una programación "*libre de proposiciones GOTO*". Sin embargo, la presencia o ausencia de estas proposiciones es una medida pobre de lo que es un buen programa [Myers 75]. Knuth ilustra muchos ejemplos de ésto en su artículo "*Structured programming with go to statements*" [Knuth 74].

2.1 Ciclo de Vida del Desarrollo del Software

> "*El software es intangible: no tiene masa, volumen, color ni olor. El software no se degrada en el tiempo como el hardware. Las fallas del software son causadas por los errores de implementación y destino, no por la degradación. Dado que el software es intangible, deben tomarse medidas extraordinarias para determinar el estado del producto de software en desarrollo.*"
>
> Fairley [Fairley 85]

Generalmente, un sistema cuando es muy grande sobrepasa la habilidad de cualquier individuo para entenderlo y construirlo. Para tener un mejor control de su desarrollo, es posible dividir el ciclo de vida del desarrollo de un sistema en las siguientes seis fases:

1. Análisis de requerimientos
2. Definición de requerimientos (especificación del sistema)
3. Diseño
4. Codificación
5. Instalación y Pruebas
6. Documentación y mantenimiento

En la siguiente figura (figura 2.1) se muestra gráficamente el proceso que debería seguir el desarrollo del ciclo de vida del software.

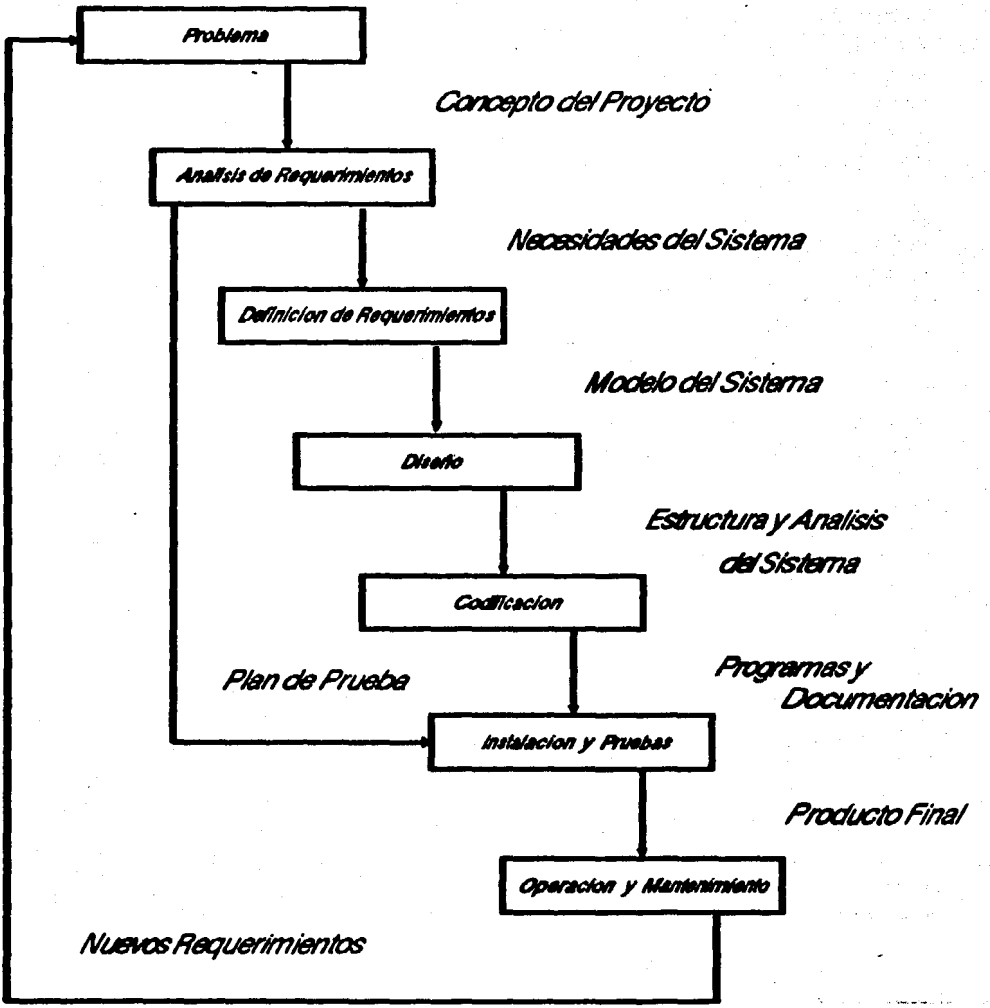


Figura 2.1 Ciclo de vida del software.

El propósito de la fase de análisis es determinar y documentar las funciones y pasos que se van a ejecutar y la naturaleza de las interacciones entre esas funciones. La fase de definición de requerimientos tiene como finalidad llegar a un contrato entre el usuario y el programador para determinar, con precisión, qué es lo que debe proporcionar el sistema. En la siguiente fase, es decir, la fase de diseño, se determina la forma en que serán implementados los requerimientos dados por la especificación del sistema. La fase de codificación tiene como tarea traducir el concepto, determinado en la fase de diseño, a un programa en un lenguaje de programación. La realización de la prueba del sistema intenta encontrar tantos errores en el producto de software como sea posible y garantizar que la implementación cumple con la especificación. Finalmente, al terminar la fase de prueba, el sistema de software es instalado y puesto en uso. Posteriormente, durante la fase de mantenimiento se busca eliminar los errores que aparecieran durante la operación de implementar los cambios y expansiones al sistema. A estas seis fases en conjunto se les conoce como el *"ciclo de vida del desarrollo del software"*.

Con el fin de direccionar varios aspectos del desarrollo del software y de su evolución, se han desarrollado diversos métodos, técnicas y herramientas. Al principio, esas aproximaciones se enfocaron sólo a las actividades de codificación; sin embargo, esfuerzos más recientes han permitido cubrir todas las fases del ciclo de vida del software, desde el concepto inicial del sistema hasta las fases de prueba y mantenimiento. Es importante poder moverse de una fase del ciclo a otra en ambas direcciones y examinar el progreso del trabajo en varios puntos intermedios. Esto permite identificar los problemas que surgen al inicio del proyecto con el fin de tomar las acciones correctivas necesarias [Wasserman 81].

El desarrollo de sistemas de software constituye un proceso iterativo, no lineal. La experiencia muestra que frecuentemente cada una de las fases acarrea consecuencias en los resultados de las fases siguientes. En ocasiones, no queda claro sino hasta que se llega a la fase de diseño que la definición de requerimientos está incompleta; o es durante la fase de implementación o de prueba cuando se descubre dónde se cometieron errores. La secuencia de las fases en el ciclo de vida del software se interrumpe con frecuencia y, en ocasiones, el proceso de desarrollo debe ser reiniciado en una de las fases tempranas o, en el peor de los casos, desde la fase inicial.

La figura 2.2 muestra las diferentes fases que conforman el ciclo de vida del desarrollo del software, así como la cantidad de tiempo promedio que se utiliza en cada una de ellas:

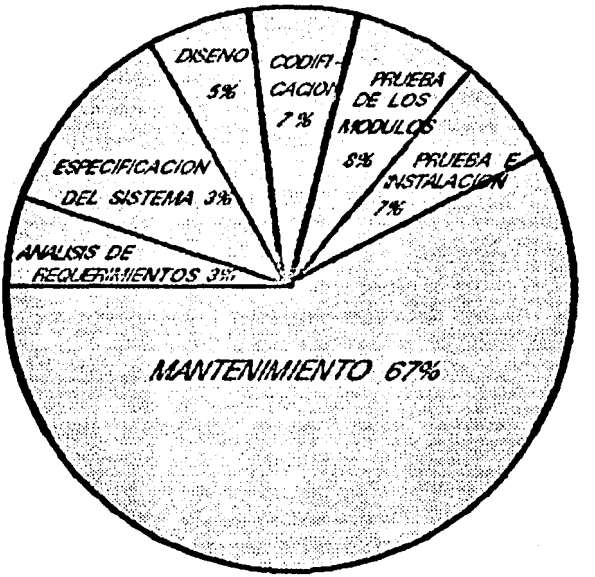


Figura 2.2 Tiempo promedio utilizado para el desarrollo de las etapas del ciclo de vida del software.

2.1.1. Análisis de requerimientos

Durante esta primera fase, que curiosamente está ausente en la mayoría de los proyectos, se definen los requerimientos para llegar a una solución aceptable del problema. Su propósito es el de establecer cuáles son las necesidades del usuario con respecto de un producto en particular y, por supuesto, es deseable involucrar profundamente en la definición de los requerimientos tanto al usuario potencial del producto como al grupo que se encargará del desarrollo del software.

En términos de confiabilidad, su objetivo es asegurar que los requerimientos del usuario se especifiquen de la manera más correcta y más precisa que sea posible y que el grupo de trabajo translade esos requerimientos a un diseño del sistema con un mínimo de errores.

2.1.2. Definición de requerimientos (especificación del sistema)

Mientras que en la fase de análisis de requerimientos se trata de determinar si se debe o no usar una computadora, es en la fase de definición de requerimientos (también llamada especificación del sistema), dónde se busca establecer qué es lo que la computadora va a hacer.

Debido a que en esta fase se describe el ámbito de la solución, este documento debe dar una estimación inicial del tiempo de realización, del personal requerido y de los otros recursos que son necesarios para el proyecto [**Zelkowitz 79**].

La especificación describe el sistema en términos del problema que se va a resolver, indicando las funciones que lo conforman y las políticas que deben gobernar su comportamiento. La definición de requerimientos también debe imponer restricciones a la ejecución del sistema o restricciones de tipo económico y sugerir los atributos que son deseables para los sistemas propuestos. Por lo tanto, una especificación puede ser caracterizada como una descripción orientada hacia el dominio del problema de un sistema de software [**Riddle 78**].

La especificación define la función de un sistema desde el punto de vista del usuario y, por lo tanto, proporciona una liga entre éste y el diseñador del sistema. Los principales requerimientos del usuario sobre la especificación son la completez y la consistencia. Todas las especificaciones subsecuentes proporcionan una base para el diseño y para la codificación.

La preparación de una definición de requerimientos que sea completa y que al mismo tiempo carezca de ambiguedad y que además, esté de acuerdo con las necesidades del usuario, es una de las tareas más difíciles de resolver en el desarrollo de un sistema. El análisis de los errores de software muestra que del 30 al 50 por ciento de todos los errores que surgen pueden atribuirse a una especificación incompleta, inconsistente o falsa [**Kopetz 79**].

Las partes del sistema que se especifican de una manera rápida y superficial, en general, son las que contienen más errores lógicos. De acuerdo con Boehm [**Boehm 74**], cualquier esfuerzo adicional que se realice durante la fase de especificación implicará un ahorro en el trabajo que se haga durante las fases de codificación, prueba e integración. La definición de requerimientos es de gran importancia, no sólo desde el punto de vista de la confiabilidad, sino también desde el punto de vista de la administración del proyecto. Como el primer documento importante en el desarrollo del sistema, éste representa el punto de partida desde el que se puede monitorear su progreso.

La descripción de las funciones del sistema es la parte esencial de la fase de especificación, comenzando con la descripción de las salidas deseadas, las entradas requeridas y el acoplamiento entre ellas. Liskov y Zilles [**Liskov 75**] propugnan porque siempre debería usarse una representación formal, permitiendo así un procesamiento posiblemente mecanizado de la especificación. En algunos casos, el uso de un programa se basa solamente en la especificación funcional, por lo que en tales circunstancias es

irrelevante si esta especificación está de acuerdo con los requerimientos originales del usuario o no.

Una solución a este problema es elaborar dos especificaciones: una que sea apropiada para el usuario, la cual, en su mayor parte, es verbal; y la otra, una descripción formal que pueda ser el punto de inicio para el desarrollo del sistema [Rault 73]. Sin embargo, en tal caso, es esencial que esas dos especificaciones sean consistentes.

Es de particular importancia que la definición de requerimientos contenga una descripción del ambiente en el que el nuevo sistema va a operar y una definición de la interface entre este sistema y su ambiente. También debe contener una descripción de los límites superior e inferior para los valores de los datos de entrada y de salida. Sólo entonces será posible detectar los errores que aparezcan en los datos de entrada y verificar la verosimilitud de los resultados.

2.1.3 Diseño

"El arte de la programación comienza con el diseño. La calidad de un producto de software está especialmente influenciada por la calidad del diseño. Esta fase, por lo tanto, tiene una posición importante en el ciclo de vida del desarrollo del software, ya que su propósito es determinar la arquitectura de un sistema con el fin de obtener la implementación menos costosa que satisfaga, al mismo tiempo, los requerimientos de calidad."

Pomberger [Pomberger 84]

En contraste con la fase de especificación, durante esta fase se describe el sistema en términos de la solución propuesta al problema y no en términos del problema al cuál está dirigido. El diseño presenta esta solución como una colección de unidades de procesamiento conceptual o módulos, especificando los lineamientos para cada una de sus actividades individuales e indicando las interacciones que se realizan entre ellos. Aunque, tanto el diseño como las especificaciones pueden imponer restricciones y sugerir atributos deseables para el sistema propuesto, esas restricciones y atributos normalmente están relacionados directamente con las propiedades de la actividad e interacción de los módulos.

Se usará la frase "*proceso de diseño*" para denotar la actividad de crear el diseño de un sistema basado en su especificación. Con el fin de minimizar el número de errores y de facilitar la producción de un diseño, el proceso deberá desarrollarse como una serie de pasos ordenados y verificables. La transición de la fase de especificación a la de diseño es aún muy abrupta y propensa a errores [Riddle 78].

Cada metodología de diseño de software puede dividirse en dos partes: por un lado, un conjunto de las características deseadas de la solución y, por el otro, los lineamientos para el proceso de solución mismo. Al desarrollar una metodología de diseño, normalmente se comienza definiendo las características deseadas de la solución y, posteriormente se desarrolla el proceso de pensamiento requerido para llegar a la solución deseada [Myers 76].

Los programas diseñados de manera tradicional, en ocasiones, presentan exceso de complejidad, lo cual resulta en una falta de transparencia o propósito. Algunas de las causas más comunes de ésto son las siguientes [Coleman 79]:

- proliferación de instrucciones de transferencia de control;
- modificación del código, lo que puede hacer que el significado de un pedazo de texto cambie dinámicamente al tiempo de ejecución;
- existencia de banderas globales que se prenden y apagan en varios lugares sin explicación;
- ausencia u olvido de comentarios u otra documentación;
- nombres de identificadores que no concuerdan con el uso de la variable que están nombrando;
- inconsistencia al ejecutar subtareas (v.g. el paso de parámetros a procedimientos);
- uso de trucos inexplicables para optimizar la ejecución y,
- programas con una estructura monolítica o arbitraria.

Estas malas prácticas pueden eliminarse desarrollando programas con una estructura simple y consistente. La consistencia y la simplicidad de la estructura pueden obtenerse abordando la producción de programas de una manera sistemática y ordenada. El diseño sistemático tiene una ventaja adicional, proporciona una defensa contra el error humano. Al finalizar cada fase en el diseño de un programa, el programador puede revisar y eliminar cualquier error. Para que los programas sean simples de entender deben expresarse de tal forma que se encuentren relacionados con el problema que intentan resolver y no con la máquina en que se van a ejecutar.

Por lo tanto, el diseño de programas se debe llevar a cabo siguiendo tres fases:

1) diseño de un algoritmo;

2) diseño de la forma en que se van a asociar los datos con los algoritmos que los van a representar y

3) traducción de la representación al lenguaje de programación que va a utilizarse.

Estas fases no son completamente independientes una de la otra y, por lo tanto, la separación no puede ser nunca completa en la práctica. El diseño del algoritmo y su representación constituyen lo que llamaremos "*diseño del programa*".

Idealmente, cualquier proceso de diseño debería ser capaz de garantizar:

a) Funcionamiento correcto.- El programa debería cumplir con sus especificaciones con exactitud, es decir, para todos los posibles conjuntos de datos válidos el programa debería obtener la respuesta correcta.

b) Flexibilidad.- En apariencia, los cambios menores y razonables en el ambiente deberían realizarse sin mayor problema.

c) Completez.- El programa debería estar preparado para recibir todas las entradas inválidas o inesperadas y producir los mensajes de error apropiados antes de que se generen daños irreparables. No debe confiarse en salidas no válidas que se detecten después de la ejecución.

d) Eficiencia.- La ejecución debería completarse dentro de límites de tiempo y espacio aceptables.

e) Transparencia de propósito.- El programa debería ser sencillo de entenderse.

Los conceptos fundamentales del diseño de software incluyen: abstracción, estructura, ocultamiento de información, modularidad, concurrencia, verificación y diseño estético [Fairley 85].

2.1.4 Codificación

Generalmente, esta fase es la más fácil. Durante la fase de codificación de un sistema se realiza la traducción del diseño al programa. El programador debe tratar de que el diseño del sistema sea independiente del lenguaje en el que el sistema va a programarse posteriormente.

En esta fase, la elección del lenguaje de programación resulta de gran importancia para el programador. No todos los lenguajes de programación son igualmente apropiados para la traducción del diseño al programa. Algunos, como ADA y MODULA2, ofrecen facilidades que permiten la modularidad y la abstracción de los datos y son más apropiados para el proceso de transformacion que otros lenguajes, sin estas características, como COBOL y ensamblador entre otros.

2.1.5 Pruebas e Instalación

"La prueba de programas puede usarse para mostrar la presencia de errores, pero nunca para mostrar su ausencia".

[Dijkstra 68b]

La calidad de un producto de software se distingue por la medida en la cual se satisface el funcionamiento correcto y la confiabilidad. Esto significa, encontrar cuántos errores hay en el sistema total y cuántos errores graves ocurren durante el uso del producto de software.

La confiabilidad y el funcionamiento correcto del software, o por el contrario, la existencia de errores, puede confirmarse solamente hasta que los resultados de las pruebas realizadas al programa a ser verificado, o el programa mismo, son comparados con los criterios de aceptación que se han preparado a partir de los requerimientos del sistema.

El propósito principal de la fase de prueba es, primero asegurar que el sistema satisface las demandas, es decir, la definición de los requerimientos y segundo, descubrir tantos errores como sea posible. La experiencia muestra que la producción de software que esté libre totalmente de errores en general no es posible. Por ello, las actividades de prueba que señalan la mayoría de los errores que pueden ocurrir son las más útiles.

Por error se entenderá, una desviación del comportamiento estipulado en la definición de los requerimientos. La causa de un error puede estar oculta en la especificación, en el diseño o en la codificación y no puede, si es robusto, encontrarse en la elección errónea de los valores de entrada al sistema. Está relacionada con la definición

de requerimientos, con el diseño, con la codificación y con la fase de mantenimiento y debe, también, forzar la verificación de la robustez. En general, la fase de prueba se realiza después de que han concluído las fases precedentes.

Los errores ocurren cuando alguno de los aspectos del desarrollo del producto de software está incompleto, inconsistente o incorrecto. Las tres categorías principales de errores son: errores en los requerimientos , errores en el diseño y errores en la codificación. Los errores en los requerimientos son causados por la definición incorrecta de las necesidades del usuario, por fallas al especificar completamente los requerimientos funcionales y de ejecución, por inconsistencias entre los requerimientos y por requerimientos no factibles.

Los errores en el diseño se generan por fallas al traducir los requerimientos en estructuras de solución completas y correctas, por inconsistencias dentro de las especificaciones del diseño o por inconsistencias entre las especificaciones del diseño y los requerimientos.

Los errores en la codificación son aquéllos que resultan al traducir las especificaciones del diseño a código fuente. Estos errores pueden ocurrir en la declaración de los datos o al hacer referencia a ellos, en la lógica del control de flujo, en las expresiones computacionales, en las interfaces con los subprogramas y en las operaciones de entrada/salida.

Existen muchas razones para que ocurran errores. La causa principal es, probablemente, que las especificaciones estén incompletas o sean inadecuadas. Otra causa de errores es el diseño algorítmico incorrecto. El refinamiento por pasos sucesivos es un intento para minimizar esos problemas. Una tercera causa menor, aunque no insignificante, son los errores accidentales o burocráticos.

Relacionadas muy cercanamente con la fase de prueba están la verificación, la validación, la certificación y la depuración. Un sistema se valida probándolo para mostrar que actúa de acuerdo con sus especificaciones. Mientras que la prueba es una actividad para descubrir errores, la depuración es una actividad para encontrar y anular fuentes de error. Los objetivos de la verificación y de la validación son asegurar y mejorar la calidad de los productos del trabajo generados durante el desarrollo y la modificación del software. Los atributos de calidad de interés incluyen: funcionamiento correcto, completez, consistencia, confiabilidad, utilidad, usabilidad, eficiencia, conformancia a estándares y efectividad del costo total. La certificación no resolverá los problemas del software, aun cuando es una herramienta importante. Gerhart y Yelowitz [Gerhart 76] han mostrado que hay muchos programas "*certificados*" que contienen errores.

La fase de prueba debe abarcar la especificación del sistema, los módulos individuales, las interacciones entre los módulos, la integración de los módulos en el sistema total y la aceptabilidad del producto de software.

La instalación real de un producto de software tiene lugar después de concluída la fase de prueba. Después de la instalación, el usuario debería realizar una prueba

de ejecución. El objetivo de la prueba de ejecución es verificar los requerimientos no funcionales y la confiabilidad del producto de software, es decir, su comportamiento durante un periodo de tiempo largo.

2.1.6 Documentación

La documentación del software puede dividirse de la siguiente forma:

1) Documentación para el usuario.- Debe contener toda la información necesaria para aprender el software y su uso, sin necesidad de información adicional. Esta documentacion incluye:

a. la descripción general del sistema,

b. el manual de instalación y el manual de usuario,

c. el manual de operación.

2) Documentación del sistema.- Debe contener todos los detalles necesarios para el entendimiento de la estructura y de la prueba del sistema. Debe servir de comunicación entre los programadores y de apoyo en el mantenimiento del sistema.

Debe describir todos los detalles de la elaboración del sistema de software, la estructura de los componentes individuales y las actividades de prueba. Debe contener toda la información necesaria para el entendimiento de la implementación total, la comunicación entre los programadores, la detección de errores y realizar las alteraciones y expansiones necesarias al sistema. Por lo tanto, dicho documento debe estar compuesto de todos los escritos elaborados en cada fase.

3) Documentación del proyecto.- Debe contener todos los detalles del desarrollo del sistema desde un punto de vista organizacional y contable. Debe servir para supervisar el progreso del proyecto y para calcular los costos relacionados con éste.

En conclusión, es necesario señalar que, la documentación es una parte integral del sistema de software. La elaboración de este documento no constituye una fase independiente dentro del ciclo de vida del software; por el contrario, es parte de cada fase. Además, al llegar a su término las fases de implementación y de prueba del sistema, esta documentación debe ser revisada y ampliada. Cada cambio en el sistema implica un cambio en la documentación. Una administración fácil y eficiente de este material es un prerrequisito para tener una documentación ordenada y por ello es aconsejable utilizar herramientas de software que la apoyen.

2.1.7 Mantenimiento

El mantenimiento del software incluye todos los cambios que se le hacen al software después de la terminación de los estados de desarrollo. Prácticamente, es imposible desarrollar un producto de software que no requiera mantenimiento. Ya se ha señalado con anterioridad que no es posible demostrar, sin ambiguedades, el funcionamiento correcto de un producto de software por medio de pruebas. Muchos errores se reconocen solamente hasta el momento en que se usa realmente el sistema y por ello estos errores sólo pueden eliminarse después de las fases de desarrollo. También, durante la operación del sistema surgen nuevas demandas del usuario, lo cual implica nuevos cambios en el sistema.

Para Pomberger [**Pomberger 84**] el mantenimiento de software incluye:

1) Optimizar la interface con el usuario y realizar mejoras al sistema,

2) corregir los errores que no se detectaron en las pruebas, y

3) realizar las modificaciones requeridas por el usuario.

Lientz y Swanson [**Lientz 80**] llaman a esos tres tipos de mantenimiento "*perfectivo, correctivo y adaptativo*" e indican que aproximadamente el 65% de los costos de mantenimiento se emplean en el primer tipo, 17% en el segundo y 18% en el tercero.

Dijkstra [**Dijkstra 72b**] establece que cada programa, que sea grande y resulte éxitoso, será suministrado en diferentes versiones durante su tiempo de vida y que cada nuevo programa representa únicamente el punto de partida de una familia de programas que se usarán en la práctica.

Hay que considerar que durante el tiempo de vida de un producto de software éxitoso, con frecuencia se ha empleado más esfuerzo en la fase de mantenimiento que en cualquier otra de las fases del ciclo de desarrollo. Por ello, los costos de mantenimiento pueden ser el factor más importante del ciclo de vida del software [**Sloughter 74**]. La investigación de Lientz y Swanson [**Lientz 80**] muestra que los costos de mantenimiento significan aproximadamente el 50% del costo total y Pressman [**Pressman 82**] calcula que los costos de mantenimiento abarcan del 40 al 60 por ciento del costo total del producto de software. Por lo tanto, es conveniente poner particular atención en la mantenibilidad del sistema y en el diseño del software, de tal forma que éste sea fácil de mantener.

Por lo anterior, el principal objetivo del desarrollo del software debería ser la producción de sistemas mantenibles, entendiéndose como tales aquéllos en que están presentes atributos de alta calidad, los cuales contribuyen a la mantenibilidad del software. Estos atributos son: claridad, modularidad y buena documentación del código fuente, así como, documentos de soporte apropiados.

2.2 Técnicas de programación

Tal vez, el cambio más grande que ha ocurrido en el diseño de los lenguajes de programación es en el papel que juegan los datos en el desarrollo de programas. Todo lo contrario sucede con las estructuras de control que casi no han sufrido variaciones desde los inicios de la programación de alto nivel. Estos cambios considerables han vuelto muy complejas las estructuras de datos, lo que ha dado lugar a problemas al probar y mantener los sistemas que las usan.

Al hacer pequeños cambios a una estructura de datos es posible causar daños a todo un sistema y hacer del mantenimiento un proceso complejo y costoso.

Por lo anterior, la característica más deseable en un sistema es la "*simplicidad*". La simplificación de una tarea compleja puede obtenerse dividiéndiendola en tareas más pequeñas e independientes, donde cada una sea discreta, visible y esté autocontenida (i. e. contiene las suposiciones que la hacen considerar la codificación de otras tareas) [**Oliver 75**].

Dos conceptos de la teoría general de sistemas pueden adaptarse para combatir la complejidad del software: el primero es la independencia, es decir, para minimizar la complejidad se debe maximizar la independencia de cada uno de los componentes del sistema; el segundo es la estructura jerárquica, ya que las jerarquías permiten la estratificación de un sistema en niveles de entendimiento, donde cada nivel representa un conjunto de relaciones agregadas entre las partes de los niveles más bajos.

Aunque el método tradicional para controlar la complejidad es la idea de "*divide y vencerás*", llamada **modularización**, en la práctica esta idea no ha resultado muy efectiva. Liskov [**Liskov 72**] señala tres razones para esta falla:

1. los módulos tienen que ejecutar muchas funciones diferentes pero relacionadas. Esto oscurece su lógica;

2. las funciones comunes no son identificadas en el diseño, lo que genera que se encuentren distribuídas entre muchos módulos distintos y,

2. los módulos interactúan en formas inesperadas sobre datos comunes o compartidos.

2.2.1 Modularización

El primer paso para hacer un programa menos complejo es descomponerlo en un conjunto grande compuesto de unidades pequeñas altamente independientes, con interfaces bien definidas entre ellas y más fáciles de manejar. Esta aproximación de "*divide y conquistarás*" se usa con frecuencia en disciplinas tales como Ingeniería, Arquitectura y otras que involucran el análisis y la síntesis de objetos complejos. En la Ingeniería de Software a estas unidades se les conoce como *módulos*.

La complejidad del diseño de un programa es una función de las relaciones que existen entre los módulos. La complejidad de un solo módulo es una función de las conexiones que existen entre las instrucciones del programa dentro del módulo.

Por un módulo se entenderá, según Goos [**Goos 73**], un segmento de programa con las siguientes propiedades: se puede comunicar con el mundo exterior sólo a través de una interface bien definida; su integración dentro de un programa más grande puede llevarse a cabo sin conocer sus mecanismos internos y su funcionamiento correcto puede determinarse sin considerar su acomodo en sistemas más grandes [veáse también **Dennis 73**]. Un módulo es una subrutina cerrada que puede ser llamada desde cualquier otro módulo en el programa, o desde el programa principal y que puede compilarse independientemente.

Para Fairley [**Fairley 85**] un módulo presenta las siguientes características:

1. contiene instrucciones, lógica de procesamiento y estructuras de datos;

2. puede compilarse independientemente y almacenarse en una biblioteca;

2. puede incluírse en un programa;

4. sus segmentos pueden usarse invocando un nombre y algunos parámetros y,

5. puede usar otros módulos.

Algunos ejemplos de módulos son: procedimientos, subrutinas y funciones; grupos funcionales de procedimientos que están relacionados, subrutinas y funciones; grupos de abtracciones de datos; grupos de rutinas de utilería y procesos concurrentes.

La modularización permite que el diseñador descomponga un sistema en unidades funcionales, imponga un ordenamiento jerárquico sobre el uso de cada función, implante abstracciones de datos y desarrolle subsistemas útiles independientes. Además, la modularización puede usarse para aislar las dependencias de la máquina, para mejorar la ejecución del producto de software o para facilitar la depuración, prueba, integración, verificación y modificación del sistema [**Fairley 85**].

Un módulo tiene tres atributos básicos: ejecuta una o más funciones, contiene alguna lógica y se usa en uno o más contextos. La función es una descripción externa del módulo; describe qué es lo que hace el módulo cuando es llamado, pero no cómo lo hace. La lógica describe el algoritmo interno del módulo, es decir, cómo ejecuta la función. El contexto describe un uso particular de un módulo.

La modularidad ayuda a mejorar la claridad del diseño, lo que a la vez facilita la codificación, la depuración, la prueba, la documentación y el mantenimiento del sistema.

Durante el proceso del diseño del sistema se crea la estructura del programa que incluye la definición de todos los módulos del programa, se establece jerarquía de los módulos y las interfaces entre ellos. Al diseño de las interfaces entre los módulos se le llama "*diseño modular externo*" y al conjunto de pasos que incluyen la definición de los datos, la selección de los algoritmos, la lógica de diseño y la codificación de los módulos se le conoce como "*diseño lógico modular*".

La forma en la que un sistema se descompone en segmentos tiene efecto sobre todos los criterios que afectan su calidad y, por lo tanto, es crucial para la calidad del producto final de software. Las condiciones que deben considerarse para tal proceso de descomposición se ennumeran a continuación:

a) Cerradura de un módulo. Cada módulo debería realizar una tarea constituyendo una entidad cerrada, es decir, las funciones de un módulo deberían componer una unidad lógica. Es peligroso y, por lo tanto indeseable, que una decisión particular de diseño se divida en varios módulos.

b) Interface mínima y visible. La interface entre dos módulos es el conjunto de todas las suposiciones que hace uno acerca del otro [**Parnas 72**], por ejemplo, nombre, significado y tipo de datos. Cada sistema de programación debería descomponerse de tal forma que las interfaces entre los módulos fueran tan simples como sea posible y que, además, puedan ser especificadas explícitamente.

c) Facilidad de prueba. Cada módulo debería estar construído de tal forma que su funcionamiento correcto pueda determinarse considerando simplemente las interfaces, sin necesidad de conocer su ubicación en el sistema completo.

d) Libertad de interferencia. La descomposición modular debería garantizar que no se ejerce alguna influencia interna entre ellos y que cada uno puede reemplazarse por otro módulo con respecto a la interface original, sin afectar al sistema completo.

e) Tamaño del módulo. Cada módulo debería ser pequeño. Esto no puede definirse en términos del número de líneas o páginas de código fuente, obviamente. La regla "*no mayor que una página*" lleva al malentendido de que la complejidad de un programa está determinada por su longitud. Los programas bien estructurados pueden, ciertamente, tener muchas páginas y sin embargo, ser concisos y entendibles.

f) Importación de un módulo. Es el número de módulos a que hace referencia el mismo. Un número importado grande posiblemente indica que el módulo tiene que realizar muchas tareas de coordinación y de decisión. Sin embargo, un número pequeño implica, tal vez, la necesidad de verificar si el módulo debe descomponerse más detalladamente.

g) Utilización de un módulo. Un módulo es referenciado o "llamado" por otros, lo cual determina el grado de utilización del primero.

Los beneficios esperados de la programación modular son:

1) Administrativo - el tiempo de desarrollo se minimiza, ya que grupos independientes trabajan en un módulo sin requerir mucha comunicación.

2) Aumento en la flexibilidad del producto - Es posible hacer cambios drásticos a un módulo sin necesidad de cambiar otros módulos.

3) Incremento en la comprensibilidad - Es posible estudiar un módulo del sistema a la vez. El sistema completo puede diseñarse mejor debido a que se tiene un mayor entendimiento de él.

La descomposición de un sistema en módulos puede realizarse siguiendo la técnica de refinamiento por pasos sucesivos (*step-wise refinement*), originalmente concebida por Dijkstra[Dijkstra 68b] y posteriormente mejorada por Wirth [Wirth 71 y 74].

Dicha técnica se utiliza para descomponer un sistema a partir de las especificaciones de alto nivel en niveles más elementales.

Derek Coleman [Coleman 79] describe este proceso de la siguiente manera:

"*En esencia, se trata de descomponer un problema en una serie de subproblemas. Dado un problema* **P**, *se trata de descubrir un conjunto de problemas más pequeños* **P1,P2,P2...,Pn** *de tal forma que resolviendo primero* **P1**, *después* **P2** *y así sucesivamente se obtiene una solución al problema original P. Esta aproximación se aplica entonces a cada uno de los subproblemas* **P1, P2, P3, ..., Pn** *produciendo así un conjunto de sub-subproblemas cuyas soluciones juntas son una solución de* **Pi**. *Se continúa este proceso hasta que los problemas son lo suficientemente pequeños para solucionarlos directamente*". Este análisis, y por lo tanto la estructura de la solución, puede representarse claramente guiados por un desarrollo de árbol, como puede observarse en la siguiente figura (figura 2.3):

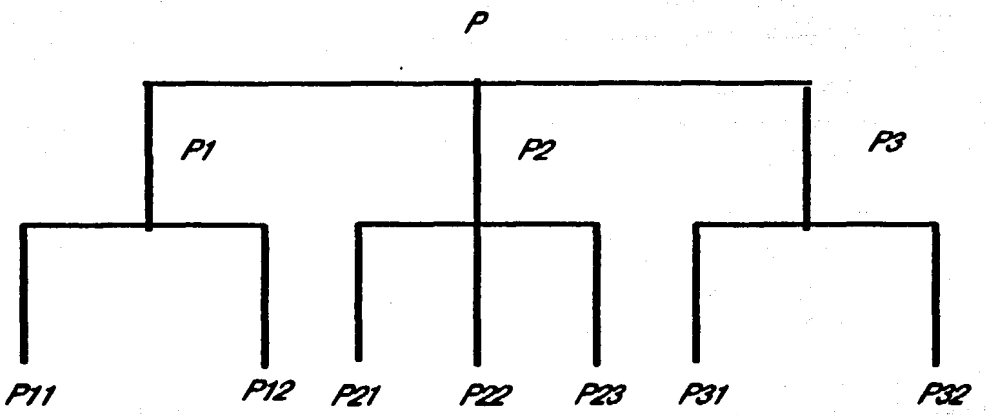
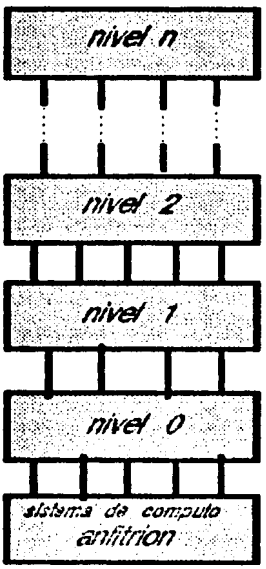


Figura 2.3 Estructura de árbol mostrando el proceso de refinamiento por pasos sucesivos.

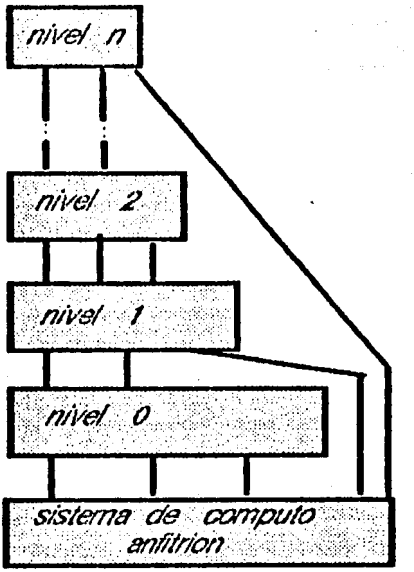
Cada nivel del árbol ignora los detalles del nivel inferior. Es posible referirse a "*niveles de abstracción*" cuando se utiliza la técnica de refinamiento por pasos sucesivos en la solución de un problema.

La idea de niveles de abstracción se debe a Dijkstra [**Dijkstra 68 y 68b**]. Un sistema se divide en distintas piezas jerárquicas llamadas niveles que contienen un diseño interior determinado. Cada nivel consiste de un grupo de módulos que están muy relacionados. El propósito principal de los niveles de abstracción es minimizar la complejidad del sistema definiendo los módulos de tal forma que sean altamente independientes uno del otro. Esto se lleva a cabo ocultando las propiedades de ciertos elementos del sistema (v.g. *recursos y representación de datos dentro de cada nivel*), lo que permite que cada nivel represente una "*abstracción*" de esos objetos. Existen dos estructuras generales de esos niveles que se muestran en las figuras 2.4.a y 2.4.b. En la primera el problema se ve como el desarrollo de una "*máquina de usuario*" comenzando con una máquina a nivel muy bajo. Una progresión de los niveles, llamada "*máquina abstracta*", se define de tal forma que cada máquina de mayor nivel se construye sobre las máquinas que están en niveles más bajos, extendiendo así sus capacidades. Cada nivel sólo puede invocar a otro nivel: el nivel inmediatamente subordinado a él.

En la estructura de la figura 2.4.b los niveles no son abstracciones completas de los niveles más bajos, lo que permite que cualquiera de los niveles haga referencia a cualquiera de los niveles inferiores.



(a). Una primera aproximación a los niveles de abstracción.



(b). Una segunda aproximación a los niveles de abstracción.

Figura 2.4 Niveles de abstracción.

Aunque la definición actual del método de niveles de abstracción es en sí mismo bastante abstracto, se le han comenzando a reconocer ciertas propiedades:

1. Cada nivel desconoce absolutamente todo acerca de las propiedades, o aún de la existencia, de niveles más altos [**Goos 73**]. Esta es la propiedad fundamental de los niveles de abstracción.

2. Cada nivel desconoce todo acerca del interior de cualquier otro nivel. Toda la comunicación entre los niveles se realiza a través de interfaces rígidas, definidas anteriormente.

2. Cada nivel es un grupo de módulos (subrutinas compiladas separadamente). Algunos de esos módulos son internos al nivel, es decir, no pueden llamarse desde otros niveles. Los nombres del resto de los módulos son conocidos potencialmente por un

nivel más alto. Esos nombres representan la interface a ese nivel.

4. Cada nivel contiene ciertos recursos y puede actuar de dos formas: ocultando esos recursos a los otros niveles o proporcionando alguna abstracción de esos recursos a los otros niveles.

5. Cada nivel puede tambien soportar una abstracción de los datos dentro del sistema. En un sistema de administración de datos un nivel puede representar cualquier archivo como una estructura de árbol, aislando así la estructura física actual del archivo desde los otros niveles.

6. Las suposiciones que cada nivel hace acerca de los otros niveles deben minimizarse [**Parnas 71**]. Estas suposiciones pueden tomar la forma de relaciones que deben cumplirse previamente a la ejecución de una función, de una representación de datos y de factores ambientales.

7. La conexión entre los niveles debe limitarse a los argumentos pasados explícitamente de un nivel a otro. Los niveles no pueden compartir datos globales [**Liskov 72**]. Además, es deseable eliminar completamente todos los datos globales (aún dentro de los niveles) en un sistema [**Myers 75**].

8. Cada nivel debe tener resistencia alta y acoplamiento bajo. Eso significa que cada función que es ejecutada por un nivel de abstracción debe estar representada por un único punto de entrada. Los argumentos pasados entre niveles deben ser tipos de datos simples y no estructuras complejas.

La aproximación de niveles de abstracción para el diseño de sistemas no ha sido muy utilizada a la fecha. Los dos usos más conocidos son en el diseño de sistemas operativos a pequeña escala, el sistema *THE* [**Dijkstra 68 y 68b**] y el sistema *Venus* [**Liskov 72b**].

Es claro, que el proceso de diseño es un proceso creativo. El principio de refinamiento por pasos sucesivos no libera al ingeniero de software de la necesidad de asirse intuitivamente al problema, ni de encontrar la descomposición correcta; por el contrario, lo ayuda en el diseño de un sistema de programación grande que va a distinguirse por su buena estructura y por haber sido pensado sólidamente hasta en el último detalle.

2.2.2 Abstracción

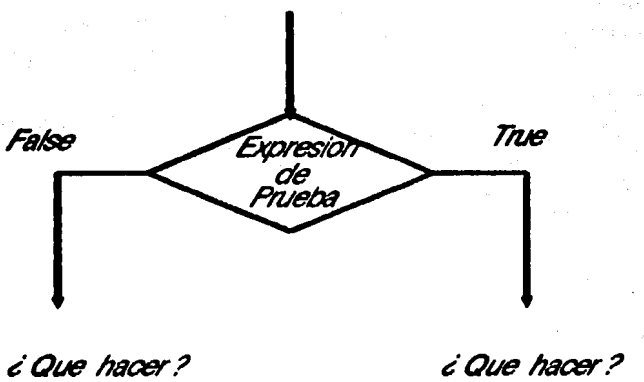
El concepto de abstracción de datos también es un medio de implantar el ocultamiento de la información. La abstracción juega un papel extremadamente importante en la programación debido a que es la herramienta principal disponible para controlar la complejidad de los programas [**Liskov 75**]. Es la herramienta intelectual que permite tratar con conceptos más allá de las instancias particulares de los mismos. Durante la definición de requerimientos y del diseño, la abstracción permite separar los aspectos

conceptuales del sistema de los detalles de la implantación.

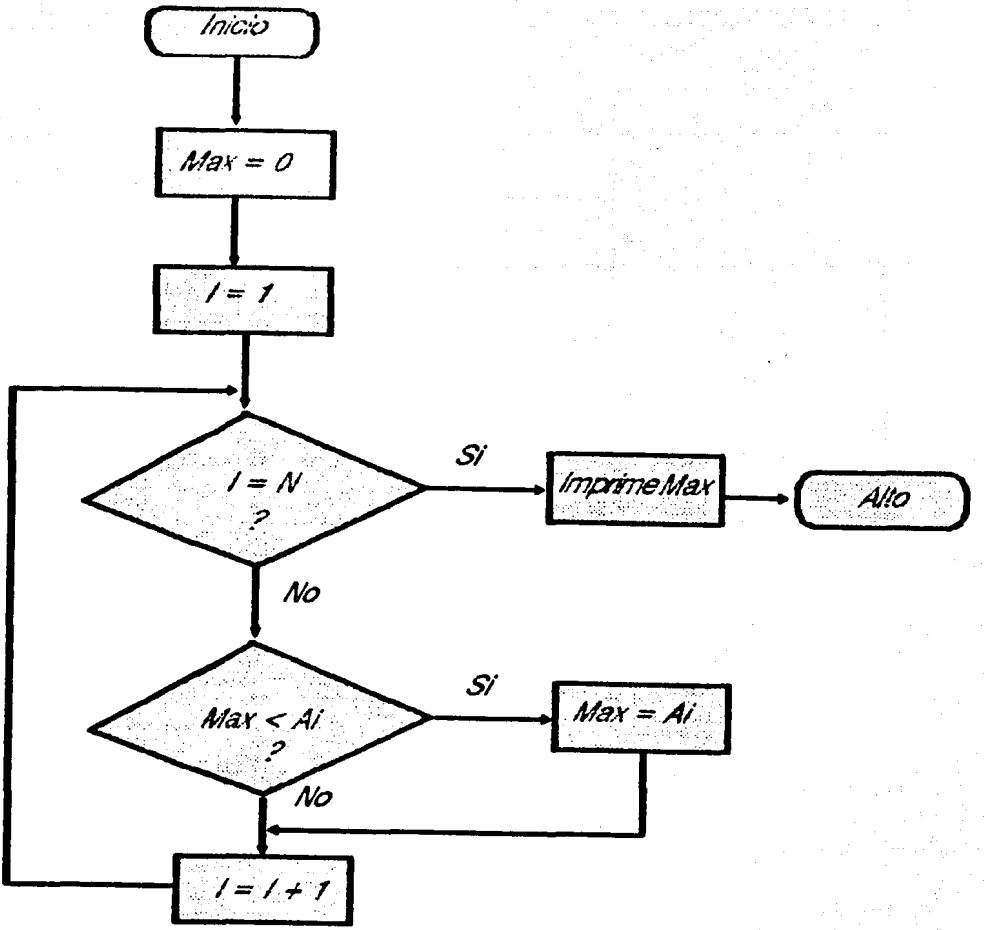
Durante el diseño del software, la abstracción permite organizar y conducir el proceso del pensamiento postponiendo consideraciones estructurales y detalles algorítmicos hasta que se han establecido las características funcionales, las estructuras de datos y el almacenamiento de los mismos. Las consideraciones estructurales se direccionan antes que la consideración de los detalles algorítmicos, lo que reduce la cantidad de complejidad que debe existir en cualquier punto particular del proceso de diseño.

Para hacer una descripción más amplia de la abstracción de datos se discutirán dos formas de diseño de programas: *programación orientada hacia el control* y *programación orientada hacia los objetos.*

En la programación orientada hacia el control, el objetivo básico es decidir "*qué hacer después*". Los programas desarrollados vía los diagramas de flujo están, generalmente, orientados hacia el control. El programador dibuja un símbolo de decisión en un diagrama de flujo y es entonces cuando decide qué camino tomar (figura 2.5.b). En la figura 2.5.a se muestra un ejemplo donde el flujo de control es bastante aparente pero los efectos sobre los datos no son claros.



(a). Símbolo de decisión.



(b). Ejemplo de flujo de control sin determinar el efecto sobre los datos.

Figura 2.5 Programación orientada hacia el control.

En la programación orientada hacia los objetos, el programador crea objetos y los usa en un conjunto restringido de operaciones. Al declarar un objeto sólo se conoce su nombre y las funciones que pueden manipularlo. En la figura 2.6 se muestra un objeto conocido como pila ("stack"). La forma de cómo operan las tres funciones y qué estructuras de datos se usan para crear la pila son inmateriales para el usuario. Todas las referencias a la pila deben filtrarse a través de abtracciones y los cambios son relativamente fáciles. Esto refuerza los aspectos de información oculta del diseño.

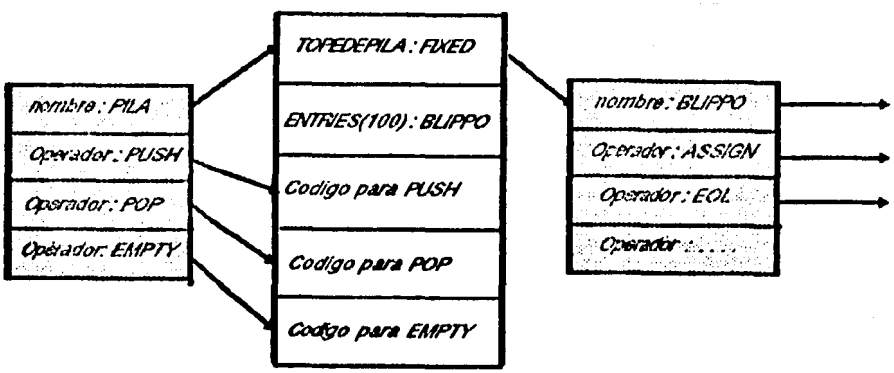
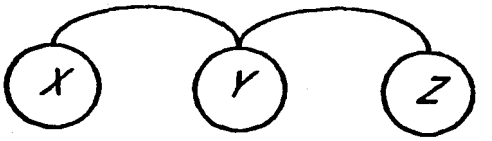


Figura 2.6 Representación gráfica de la utilización de una pila en la programación orientada hacia objetos.

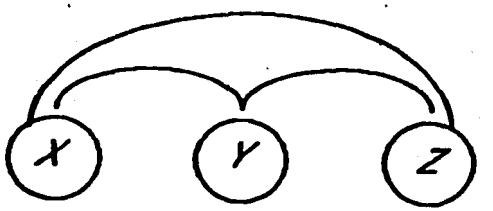
Con el fin de implantar las abstracciones se requiere que el lenguaje de programación tenga las siguientes dos características: la capacidad de crear estructuras de datos para representar tipos de datos abstractos y la posibilidad de definir procedimientos que accesen esos datos.

Al elegir una abstracción apropiada de los datos se pueden representar sólo los aspectos funcionalmente necesarios para su representación y suprimir los detalles de su implantación.

Ya que la abstracción de datos incluye tanto a los objetos de datos como a las operaciones aplicables sobre ellos, una modularización apropiada debe incluir los procedimientos para cada una de esas operaciones. Este módulo multiprocedural encápsula la información representacional y la hace inaccesible fuera del módulo. Por lo tanto, las dependencias se conservan dentro del módulo, como se desea. Lo importante es reconocer y considerar las abstracciones de los datos cuando se elige la modularización, limitando así el acceso a los datos y a las operaciones relevantes para manipularlos ([**Zilles 75**] y [**Pomberger 84**]). En la figura 2.7 se muestra un ejemplo de violación del atributo de información oculta.



X sabe acerca de Y
Y sabe acerca de Z
X no sabe acerca de Z



X llama a Z directamente

Figura 2.7 Violación del atributo de información oculta

Para Dahal, Dijkstra y Hoare [Dahl 72] el proceso de abstracción puede sumarizarse en las siguientes cuatro fases:

1.- Abstracción: Es la decisión de concentrarse en las propiedades que comparten muchos objetos o situaciones en el mundo real, ignorando las diferencias entre ellos.

2.- Representación: Consiste en la elección de un conjunto de símbolos que utilizados en la abstracción y que pueden ser el medio de comunicación. Es la forma en la cuál se va a representar la abstracción en la computadora.

2.- Axiomatización: Comprende las proposiciones rigurosas de las propiedades que se han abstraído del mundo real y que son compartidas por manipulaciones del mismo y de los símbolos que representan.

4.- Manipulación: Son las reglas para realizar la transformación de las representaciones simbólicas como un medio de predicción de los efectos de manipulaciones similares en el mundo real.

La idea básica detrás de la abstracción es el diseño de arriba hacia abajo (*top-down*), el cual tuvo su origen con Dijkstra [**Dijkstra 69**] y Wirth [**Wirth 71**].

En el diseño de arriba hacia abajo el programador escribe primero una subrutina como una instrucción sencilla. A su vez esa instrucción se expande a mayor detalle. En cada nivel, la función se va expandiendo con más y más detalle hasta que la descripción resultante es el programa fuente en algún lenguaje de programación.

Hay que hacer notar que Wirth [**Wirth 71 y Wirth 74**] y Yourdon [**Yourdon 75**] consideran la técnica de diseño top-down y la de refinamiento por pasos sucesivos como si fuesen iguales.

En el diseño de arriba hacia abajo, el programa se estructura jerárquicamente y se describe por refinamientos sucesivos. Cada refinamiento describe sus acciones refiriéndose a otros refinamientos en una manera de arriba hacia abajo.

El desarrollo de arriba hacia abajo no es tan fácil como parece ya que existen tres posibles "*puntos críticos*" para un sistema:

1. el inicio de la ejecución,

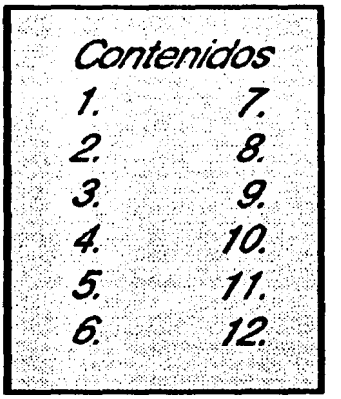
2. el foco de control y,

2. la interface con el usuario.

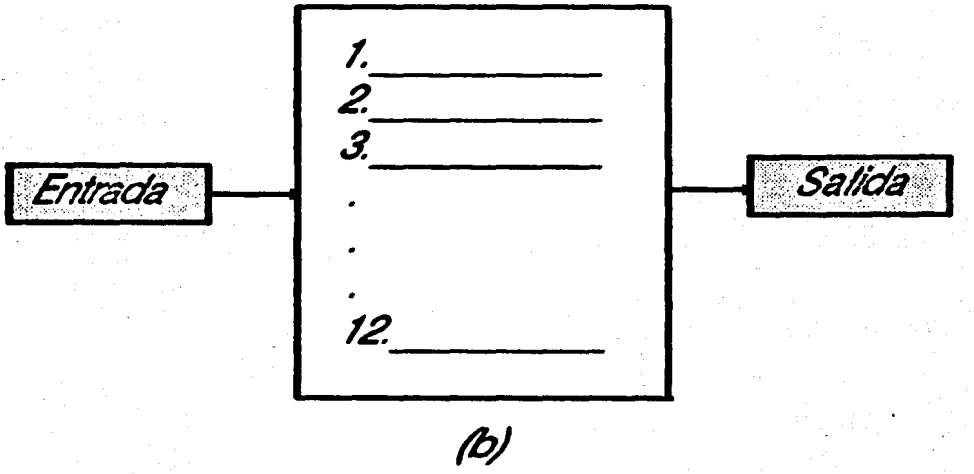
Al utilizar la técnica de arriba hacia abajo, el usuario conoce en una fase temprana las interfaces de nivel alto en el sistema. Los cambios pueden realizarse en el ciclo del desarrollo rápidamente y de forma relativamente fácil.

La principal ventaja de la estrategia de arriba hacia abajo es que la atención se dirige primero a las necesidades del usuario, a las interfaces con él y a la naturaleza total del problema a ser resuelto.

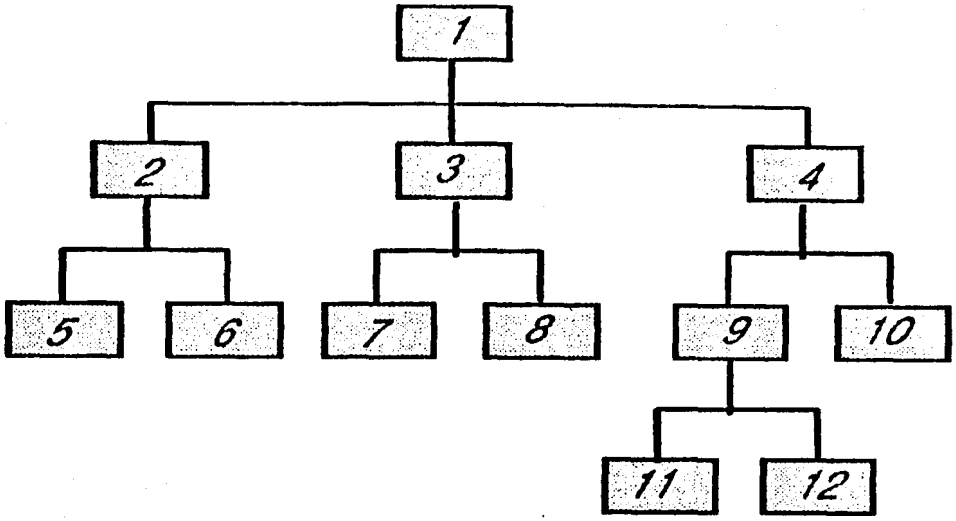
Una de la técnicas más interesantes y más conocidas de la aproximación de arriba hacia abajo (*top-down*) es el sistema HIPO (*Hierarchy Input Process Output*) desarrollado por IBM en 1975. HIPO requiere que la estructura total del sistema se muestre como un diagrama total (figura 2.8.a) y se enlista el contenido de la estructura del sistema (figura 2.8.b).



(a). Estructura general de un sistema en HIPO.



(b). Visión global del sistema.



(c). Diagrama detallado de un módulo del sistema.

Figura 2.8 Ejemplo de un diagrama en HIPO.

Posteriormente, cada módulo del diagrama total se describe más detalladamente en un diagrama (figura 2.8.c). Pueden realizarse diagramas con más detalles de otros niveles si fuese necesario. El método HIPO puede emplearse también como un auxiliar de la especificación. Se caracteriza por hacer una distinción clara entre la notación de datos y la notación de funciones y por tener un medio de representación jerárquica orientada a los árboles. Una aproximación aún más formal de la representación gráfica de la estructura del sistema ha sido sugerida por Stevens, Myers y Constantine [**Constantine 74**]. Esto no implica que HIPO sea la única herramienta a ser usada o que la representación gráfica de Constantine sea la única aproximación. El punto esencial consiste en alguna aproximación formal y estándarizada a la documentación de un diseño top-down [**Yourdon 75**].

2.2.3 Diseño Estructurado

El diseño estructurado se considera que fue desarrollado por Larry Constantine [**Constantine 74**] de 1965 a 1968, como una técnica de arriba hacia abajo para el diseño

arquitectónico de sistemas de software. Principalmente, se concentra en las relaciones entre los módulos para lo cual su función principal consiste en convertir diagramas de flujo de datos a gráficas estructuradas. Introduce los conceptos de acoplamiento y cohesión para guiar el proceso de diseño. Un sistema bien diseñado muestra un bajo grado de acoplamiento entre los módulos y un alto grado de cohesión entre los elementos en cada módulo.

El primer paso en el diseño estructurado es la revisión y el refinamiento del (los) diagrama(s) del flujo de los datos desarrollado(s) durante la definición de los requerimientos y el diseño externo. El segundo paso lo determina el hecho de si el sistema está centrado en transformaciones o está dirigido a transacciones. En un sistema centrado en transformaciones, el diagrama de flujo de datos contiene segmentos de entrada, procesamiento y salida que se convierten en subsistemas de entrada, procesamiento y salida, respectivamente, en el mapa de estructura.

El tercer paso en el diseño estructurado es la descomposición de cada subsistema usando lineamientos tales como: acoplamiento, cohesión, información oculta, niveles de abstracción, abstracción de datos, etc.

La descomposición de funciones en módulos debe llevarse a cabo iterativamente hasta que cada módulo sea lo bastante pequeño, de tal forma que su implantación total pueda realizarse de una sola vez. En la fase inicial del diseño, el sistema debería poder subdividirse tan finamente como fuese posible ya que de esta forma los módulos pequeños pueden recombinarse de manera más fácil posteriormente.

Además del acoplamiento, la cohesión, la abstracción de los datos, el ocultamiento de información y otros criterios de descomposición, los conceptos de "*campo de efecto*" y "*campo de control*" pueden usarse para determinar las posiciones relativas de los módulos en una estructura jerárquica. El "*campo de control*" incluye al módulo y todos los módulos que están subordinados a él. El "*campo de efecto*" de una decisión es el conjunto de todos los módulos que contienen código ejecutable basado en el resultado de la decisión. En general, los sistemas están más débilmente acoplados cuando el campo de efecto de una decisión está dentro del campo de control del módulo que contiene la decisión.

El diseño detallado de los módulos individuales en un sistema puede llevarse a cabo usando la técnica de refinamiento por pasos sucesivos y utilizando notaciones de diseño detallado, tales como diagramas HIPO, seudocódigo, y/o lenguaje natural estructurado.

CAPITULO 3

Ayudas Computarizadas para la Ingeniería de Software

Una industria que está emergiendo con rapidez es la del desarrollo de *"ayudas computarizadas para la ingeniería de Software"(CASE)*. Estos programas ayudan en la automatización de un sistema de software desde su creación hasta su terminación y durante la etapa de mantenimiento. *CASE* (**Computer-aided Software Engineering**) consiste en la aplicación de tecnologías o herramientas automáticas para el funcionamiento y procedimientos de la Ingeniería de Software.

La industria del *CASE* se apoya en el concepto de la programación estructurada, surgida en los inicios de la década de los setentas. En esa época, las herramientas existentes eran mínimas. Se contaba con editores, compiladores, intérpretes y diccionarios de datos; sin embargo, se carecía de las que podían ayudar al desarrollo de sistemas estructurados. La carencia de este tipo de herramientas afecta en gran medida el ciclo de vida del desarrollo de los sistemas en muchos aspectos: eficiencia, ahorro de tiempo, pérdidas económicas, aumento en la dificultad y en el tiempo para accesar información y en la complejidad para crear y mantener el sistema.

Con el transcurso de los años, aumentó la demanda de sistemas de tipo diverso (cuya complejidad en tamaño y forma era a su vez creciente) y se generó una necesidad urgente en cuanto a optimización y ahorro de todo tipo de recursos. Es por esto, que surgió la necesidad de contar con herramientas autómaticas (*CASE*) que auxiliaran en el análisis, diseño, implementación y mantenimiento de los sistema desde su inicio hasta su terminación.

Los elementos esenciales de un *CASE* son:

- La metodología del ciclo de vida del desarrollo del software.
- Métodos estándares y técnicas de diseño para producir proyectos de calidad.
- Integración automática de sus herramientas.

Para un sistema *CASE* los principales requisitos y capacidades son: mejorar la productividad y la calidad del software, incrementar el control del sistema y garantizar que la herramienta sea flexible y fácil de usar.

Al mejorar la productividad se ayuda a reducir el costo en el desarrollo del sistema, al mejorar la calidad del software se reduce el trabajo en la fase de mantenimiento del sistema y al incrementar el control del sistema, se contribuye a la buena administración del software.

Otras capacidades adicionales deseables en un *CASE* son: detección de errores en tiempo real y corrección de éstos en forma rápida y fácil; facilidad de validar automática y completamente cualquier sistema de información y relacionar los métodos y técnicas estándares para el análisis y diseño de los sistemas con su ciclo de vida.

Una herramienta *CASE* reúne, en el producto final, todos los subsistemas que integran el sistema total. Otros ingredientes necesarios, como son los manuales de usuario, de operación y de mantenimiento, también están contenidos en un *CASE*.

Desafortunadamente, el amplio uso de redes de computadoras personales para el desarrollo de grandes paquetes da a la industria del *CASE* algunos dolores de cabeza. Sin un estricto control, una infinidad de versiones y módulos pueden generar que un desarrollo de software se convierta fácilmente en un caos. Por esto mismo se han desarrollado a su vez versiones distintas de herramientas de control.

El número y tipo de herramientas utilizadas por un *CASE* es muy diverso, lo cual se debe principalmente a los nuevos desarrollos y tecnologías de la Ingeniería de Software. Dentro de los sistemas *CASE* existentes también es posible encontrar paquetes integrados. La esencia de un paquete integrado *CASE* radica en su capacidad para verificar la consistencia y la completez entre varios medios gráficos y a través de todos los niveles jerárquicos. Solamente un sistema de este tipo tiene el potencial para generar un código ejecutable correcto.

Ultimamente, se han desarrollado muchas ayudas computarizadas para apoyar y facilitar al programador en el desarrollo del ciclo de vida del software. Algunos de estos esfuerzos son:

* Probablemente, el primer sistema *CASE* es **Pride/ASDM** (*Automated System Development Methodology*) desarrollado por Bryce y Asociados. Este sistema proporciona una base de datos para administrar y controlar los proyectos y definir información en un diccionario de datos.

* **USE.IT Systems** es una técnica para construir sistemas que emplea gráficas y texto.

* **Excelator/RTS**, desarrollada por Index Technologies Corporation (Intech) en Cambridge Mass., contiene capacidades integradas en el diccionario de datos con un editor de línea para la creación de flujos y cartas gráficas.

• **TAGS/IORL** de Teledyne Brown Enginnering contiene un soporte gráfico automático muy completo para los requerimientos de entrada y salida.

• **NASTEC CASE 2000** de Nastec Corporation es un ejemplo de un sistema *CASE* integrado. Soporta multiples ciclos de vida desarrollando diferentes técnicas. El sistema CASE 2000 contiene herramientas de ayuda de diseño, las cuales integran texto y gráficas en el editor de la pantalla, además de contener diseño de diccionarios que catalogan la información encontrada en el diccionario de datos y permiten accesar archivos u objetos para ser asignados al tipo de objetos y almacenados en el diccionario. Las ayudas de diseño incluyen a los depuradores, los cuales validan todo lo relacionado con el sistema, actualizando automáticamente el diccionario de datos.

• **SA/SD** (*Structured Analysis and Design*) cuyo objetivo es, simplemente, el ordenamiento y la planeación e implementación sistemática del software.

Entre las gráficas que dominan la metodología SA/SD, el diagrama de flujo de datos (DFD) es, probablemente, el más importante. A este diagrama se le conoce también como el diagrama Yourdon-DeMarco como reconocimiento a dos de los pioneros en SA/SD. La carta de estructura, una parte esencial de SA/SD, muestra la ruptura jerárquica del sistema en módulos, complementada con información acerca del flujo de datos (y, en ocasiones, del flujo de control).

• **SADT** (*Structured Analysis and Design Technique*) método desarrollado en SofTech, Inc. de Waltham, Mass. en 1977, por D. T. Ross y Schoman [**Schoman 77**], combina la descomposición jerárquica con flujos de control y datos combinados. Es apropiado para las aplicaciones en tiempo real.

Algunas de las principales características que distinguen a **PROGRE** de los sistemas *CASE* antes mencionados son:

a) Ayuda a aprender a programar (la computación es al mismo tiempo el objetivo y la herramienta).

b) Utiliza un lenguaje gráfico de programación estructurada por lo que el programador aprende a programar estructuradamente.

c) No hace un análisis sintáctico del texto escrito por el usuario, por lo que se puede utilizar desde el diseño del programa en lenguaje natural o adaptarse a otros lenguajes de programación del tipo de Pascal.

d) La documentación de un programa es un elemento importante para el futuro mantenimiento del mismo. Con **PROGRE** se obtiene el diagrama de flujo actualizado que corresponde exactamente

al programa fuente, sin esfuerzo por parte del programador.

Una tarea importante de investigación, en el campo de la Ingeniería de Software, es el desarrollo de herramientas que apoyen la producción del software y que proporcionen técnicas para implementar tales herramientas.

3.1 Herramientas de Software

Cuando el procesamiento de datos comenzaba a ser una disciplina había pocas herramientas disponibles para la programación, entre éstas se encontraban: los ensambladores, editores, ligadores y cargadores. Posteriormente, con la introducción de los sistemas de tiempo compartido se desarrollaron gran cantidad de herramientas de software: sencillas, relacionadas a fases específicas del proyecto y para la dirección del proyecto. Las herramientas actuales que se ofrecen pretenden adaptarse al desarrollo de todas las fases del ciclo de vida del software.

Dichas herramientas, a las cuales frecuentemente se les llaman "ambientes de programación", son, generalmente, difíciles de manejar, diseñadas solamente para aplicaciones especiales y rara vez portables, por lo cual, se puede concluir que no existe un ambiente de software universal. A continuación se da una lista de herramientas aisladas que ayudan a la racionalización y que son necesarias para asegurar la calidad de la producción de software.

3.1.1 Herramientas sencillas

Son herramientas elementales y fáciles de producir cuyo principal objetivo es el de simplificar la producción del software. Principalmente, se pueden dividir en tres grandes grupos, según sus propósitos:

a) Las que ayudan a la administración de archivos, como manejo de catálogos, protección de archivos, comparación entre dos o más archivos, compresión y expansión de archivos.

b) Las que asisten en tareas de documentación, como formateadores de programas, generadores de índices, diseñadores de diagramas de flujo, etc.

c) Las que permiten realizar un análisis del perfil estático de los programas, o sea, buscar errores en los programas, obtener medidas de complejidad, analizar anidamientos de instrucciones, detectar identificadores declarados pero no usados, reconocer objetos no inicializados, etc.

3.1.2 Herramientas relacionadas a las fases específicas del proyecto

La meta principal de estas herramientas es la de facilitar el reconocimiento y la corrección de errores desde que el proyecto se encuentra en una fase temprana de su desarrollo. También, tienen como objetivo auxiliar en la creación de la documentación y tratar de garantizar la calidad de los resultados obtenidos en cada una de las fases del ciclo de vida del software, individualmente.

Como apoyo en el análisis de requerimientos de un sistema se pueden obtener los diccionarios de datos, los cuales intentan aclarar la comunicación entre el analista y el diseñador y establecer una consistencia de términos para el usuario en relación con el análisis del sistema. Dichos diccionarios requieren de herramientas para introducir, suprimir, buscar y modificar descripciones de datos. Los diccionarios aumentan en forma poderosa la capacidad del diseñador para verificar la existencia de redundancia (de procesos, archivos o flujos), con lo que se evita la duplicidad de módulos que hayan sido creados anteriormente. Si un proceso, archivo o flujo de datos se elimina o modifica, la herramienta elimina o modifica en todo el sistema la primitiva correspondiente con todos sus atributos relacionados, accesando al diccionario de datos en forma directa y rápida.

Son varios los métodos que se han diseñado para asistir al analista en esta fase del desarrollo de programas, aunque ninguno de ellos ha sido aceptado universalmente. Algunos de estos métodos son: *SADT* [**Schoman 77**], *SA* [**Halsteadt 72**], *HIPO* [**IBM 75**] y *PSL/PSA* [**Teichrow 77**].

La aplicación de herramientas en la definición de requerimientos está limitada, principalmente, al procesamiento de texto, debido a que no existe aún un lenguaje de especificación aceptado en su totalidad. Jones [**Jones 79**] propone tópicos importantes para el desarrollo de un lenguaje, mientras que Budde [**Budde 84**] sugiere ideas de proptotipos.

Las herramientas para el diseño de programas apoyan en el desarrollo interactivo de programas siguiendo el principio de refinamiento por pasos sucesivos, de tal forma que el diseño puede ser formulado rigurosamente o en detalle, completa o incompletamente, lo cual facilita las posibles modificaciones, correcciones o refinamientos al diseño en cualquier momento.

La estructura de los sistemas y de los algoritmos puede ilustrarse en forma más clara a través de la notación gráfica, por lo que son de gran valor para el ingeniero de software las herramientas para representación gráfica de documentos de diseño (v.g. generadores de diagramas de flujo, generadores de diagramas Nassi-Shneiderman). Con el apoyo de estas herramientas se puede realizar el diseño completo de los sistemas ayudándose de la computadora, sin necesidad de utilizar pluma o papel. Herramientas de este tipo son descritas por Frei y otros [**Frei 78**], Pomberger [**Pomberger 82**], Truol [**Truol 81**] y Willis [**Willis 81**], entre otros.

En la fase de implementación se requieren herramientas que apoyen la edición,

compilación, carga y ejecución de los programas.

Para facilitar la codificación son útiles los editores de texto eficientes o, mejor aún, editores orientados hacia la estructura. Generalmente, estas herramientas presentan la desventaja de auxiliar en la edición de los programas tan sólo en un determinado lenguaje de programación. Ejemplos de este tipo de herramientas pueden encontrarse en Teitelbaum y Reps [**Teitelbaum 81**] y Donzeau-Gonge y otros [**Donzeau-Gonge 80**].

Los programas generadores de diagramas o planillas son también herramientas útiles para el programador. Estos ayudan en la generación automática del diagrama del programa en el cual estén contenidos encabezados del programa, descripciones de las interfaces e indicadores de lugar para los comentarios. Esto hace que la observancia de los estándares del proyecto sea más simple para el programador.

La mayoría de las técnicas de desarrollo de sistemas usan representaciones gáficas para su análisis y diseño. Virtualmente, todas estas representaciones unen texto y gráficas, tales como: flujos de datos, diagramas de flujos de datos, cartas de estructuras, etcétera.

Al crearse algún proceso, archivo o flujo de datos en forma gráfica, la interacción con el editor de textos es automática y podrá definirse, en ese momento, alguna de estas primitivas funcionales.

La exploración de los distintos niveles del sistema (inferiores y superiores) también se realiza en forma automática y la facilidad con que la gráfica permite navegar dentro del sistema hace posible el ahorro de tiempo, complejidad y esfuerzo en la creación, implementación, corrección y mantenimiento del sistema.

Otra herramienta muy importante para el programador es un compilador que permita traducir el código fuente a código objeto.

Generalmente, todos los sistemas contienen errores, aún cuando son analizados cuidadosamente y diseñados por ingenieros de software profesionales. Eliminar la fuente de un error, sin producir nuevos errores en el proceso, es tan difícil como frecuente. Las herramientas que ayudan a localizar el origen y a eliminar las fuentes de los errores son, principalmente: los analizadores sintácticos, los depuradores y las herramientas de comparación de archivos.

Los analizadores sintácticos ayudan a obtener información acerca de la composición y de la estructura del sistema que va a probarse, por lo que también son de gran importancia para la documentación y el mantenimiento de los sistemas.

Otra herramienta importante para los especialistas que efectúan las pruebas es un depurador eficiente, cuyo objetivo principal es el de respaldar las pruebas sucesivas de los programas por medio del análisis del estado del programa en cualquier momento y en cualquier punto dado de un sistema. Otra de sus funciones es la de permitir la alteración del contenido del área de almacenamiento para relizar pruebas dinámicas.

Los analizadores y depuradores aceptan gráficas y textos como entradas, para

luego validar la sintaxis de las gráficas de acuerdo con las reglas de las técnicas de diseño utilizadas tomando en cuenta a todos los elementos que se encuentran en el diccionario de datos.

Con estas herramientas, es posible detectar algunos errores automáticamente por lo que el usuario puede corregirlos en tiempo real. Esto implica un ahorro substancial de tiempo en el desarrollo del sistema.

La realización de una prueba requiere de la comparación de los resultados de una ejecución con los resultados obtenidos previamente; pero la manipulación manual de estos resultados, para volúmenes grandes de información, resulta muy difícil. Por lo anterior, es conveniente que los resultados sean escritos sobre archivos para que puedan ser analizados usando herramientas de comparación de archivos. Jacobi [**Jacobi 82**] presenta un ejemplo de un buen depurador.

Las herramientas para el mantenimiento se utilizan en muchas, usualmente en todas, las fases del ciclo de vida del desarrollo del software. Son diversas las necesidades que incitan a su utilización, por lo que su diseño debe estar especialmente bien pensado y su operación debe ser conceptualmente simple y compatible con el de otras herramientas.

En proyectos grandes de software los documentos, así como las bibliotecas de módulos, son usados y alterados por muchos usuarios. Los objetos, en esas bibliotecas están presentes, generalmente, en diferentes versiones. Conservar actualizadas estas bibliotecas es una tarea delicada y esencial. Las herramientas de prueba, descritas anteriormente, son de importancia primaria para la fase de mantenimiento, ya que, es de esperar, que una prueba revele un error, lo cual implica que el programa requiere de mantenimiento.

3.1.3 Herramientas para la administración de proyectos

Las herramientas para la administración de proyectos proporcionan las funciones que permiten regir todos los objetos generados durante el desarrollo del proyecto, esto es: estándares del proyecto, documentos, tablas, planos, programas, etcetera y, además, son capaces de manipular las redes para controlar los progresos del proyecto.

El prerrequisito para este tipo de herramientas es una base de datos común, la cual debe garantizar que todos los miembros del proyecto sólo pueden manipular los productos a los cuales tienen acceso directo y, también, que todos los miembros trabajan con la versión actualizada del producto.

Ciertamente, estas herramientas son las más costosas y difíciles de implementar. La especificación de un ambiente de programación que contiene este tipo de herramientas puede encontrarse en el Stoneman Report [**Fisher 80**].

3.2 Implementación de las herramientas

Virtualmente, todas las herramientas descritas anteriormente, deben implementarse como programas interactivos para que resulten efectivas. El diseño correcto de la interface con el usuario es de primordial importancia para la aceptación de esas herramientas. Durante el proceso, es importante decidir qué debe ser comunicado, a quién y en qué forma. Para la pregunta "en qué forma", el programador de las herramientas debe tener siempre presente las propiedades del medio de comunicación empleado.

Para las herramientas de software, el medio principal de comunicación es la pantalla. Los métodos gráficos interactivos pueden ser de gran utilidad para expresar volúmenes extensos de información en forma rápida y clara, en contraste con la utilización de descripción textual. Por esta razón, es aconsejable utilizar técnicas de procesamiento gráfico de datos en el desarrollo de las herramientas.

Las herramientas de software deben modelarse de tal forma que puedan ser "aceptadas" por los ingenieros de software. Esto, de nuevo, presupone que éstas son simples y que no modifican el método de trabajo al que están acostumbrados al requerir una adaptación grande de su parte. Lo cual, a la vez, implica que las herramientas deben ser confiables, que deben operar rápidamente, sin obstaculizar el trabajo del ingeniero de software, y producir los resultados - ya sea por impresora o por pantalla- en una forma ordenada y medida.

Al respecto, es necesario señalar los requerimientos mínimos aconsejables de los componentes de hardware para una comunicación eficiente hombre-máquina. Estos son:

* una pantalla con capacidad gráfica y una impresora lasser como dispositivos de salida;
* una rejilla de localización (ratón o palanca de mando "joystick") como dispositivo de entrada, además del teclado;
* un procesador eficiente;
* un controlador de despliegue con una razón de transferencia de datos eficiente entre el almacenamiento principal y la pantalla.

La carencia de esos requerimientos mínimos significa intentar implementar las herramientas del presente con la tecnología del pasado.

Al implementar las herramientas se debería comenzar con las de tipo sencillo. Cuando se ha probado que éstas son confiables, entonces es factible añadir herramientas más complejas.

CAPITULO 4

PROGRE: Herramienta Práctica para la Programación Gráfica Estructurada

4.1 GAL- Graphical Abstract Language

GAL es un lenguaje abstracto gráfico para desarrollar programas estructurados. GAL representa las estructuras del lenguaje en vez de cadenas de caracteres. El lenguaje abstracto gráfico intenta acercarse a los lenguajes sometidos a la programación y diseño estructurado.

GAL está compuesto por nueve símbolos cuyos cuerpos, excepto el de proceso, son estructuras de complejidad arbitraria. No hay flechas para conectar los símbolos sino que se colocan adyacentemente, con lo que se evitan las transferencias de control arbitrarias.

En GAL se adoptaron cuatro símbolos de Nassi- Shneiderman charts [Nassi 73] (figura 4.1), se diseñaron cuatro más (figura 4.2) y se modificó la forma del símbolo del CASE (figura 4.3).

El símbolo de proceso se usa para representar asignaciones, instrucciones de entrada/salida y llamadas a subrutinas (figura 4.1.1).

Hay dos tipos de instrucciones de loop, aquéllas que tienen la condición para terminar el loop arriba y aquéllas que la tienen abajo. Las instrucciones de loop que tienen la condición arriba se dividen en dos clases: las instrucciones para las cuales el fin del loop depende de una condición booleana, como el WHILE-DO de Pascal (figura 4.1.2) y aquéllas en las cuales la condición es una variable indexada, como el FOR-TO de Pascal (figura 4.2.2). Las instrucciones con la condición del loop abajo, como el REPEAT-UNTIL (figura 4.1.3)

El símbolo de decisión (figura 4.1.4) se utiliza para representar instrucciones donde la acción a llevarse a cabo depende de una expresión booleana, como el IF-THEN-ELSE de Pascal.

Un grupo de instrucciones relacionadas en un módulo se incluyen en un marco (figura 4.2.3). Este símbolo le permite al programador reconocer fácilmente los módulos y la estructura modular del programa; una aplicación de esta estructura es el PROCEDURE o FUNCTION de Pascal.

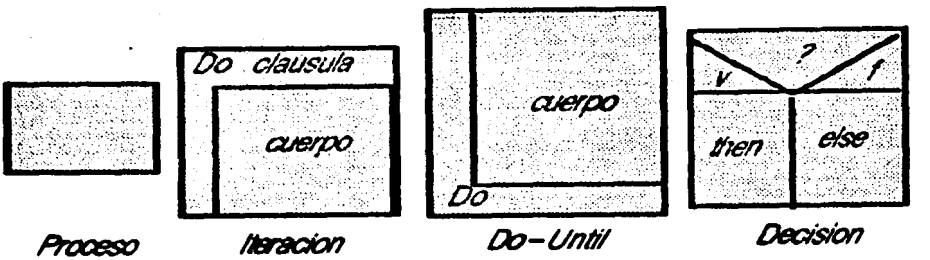


Figura 4.1 Símbolos de N-S charts adoptados por GAL.

El símbolo de declaración (figura 4.2.4) le permite al programador establecer la definición de los identificadores locales de cada módulo y determinar fácilmente su alcance.

Con el símbolo de la figura 4.3 se obtiene la generalización del símbolo de decisión permitiendo tener n valores para la condición, tal como sucede con el CASE de Pascal.

Finalmente, la instrucción compuesta, la cual es un grupo de instrucciones, se encuentra claramente representada por el símbolo compuesto (figura 4.2.1), por ejemplo el WITH-DO de Pascal.

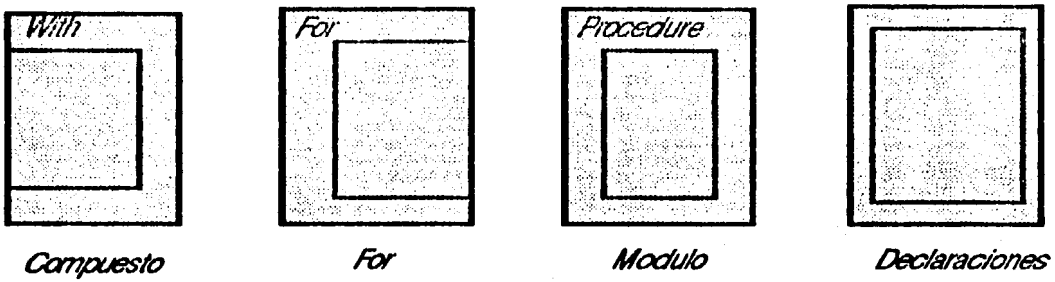


Figura 4.2 Símbolos de **GAL** diseñados.

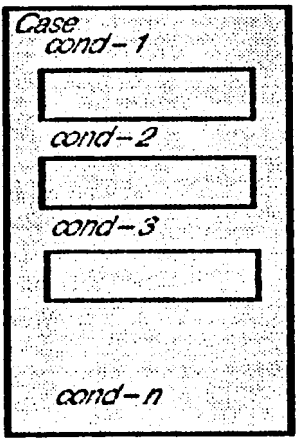


Figura 4.3 Símbolo CASE.

4.2 Usos de *PROGRE*

PROGRE (PROgramación GRáfica Estructurada) ayuda a desarrollar gráficamente programas estructuradamente en forma interactiva. Consiste de un traductor de programas escritos en un lenguaje de programación estructurado (Pascal) a su representación gráfica en un lenguaje de programación gráfico abstracto (GAL) y viceversa, es decir, la traducción de un programa desarrollado utilizando un lenguaje de programación gráfico abstracto a su correspondiente versión en un lenguaje de programación estructurado. En *PROGRE*, la interacción entre el programador y el programa es en términos de la estructura gráfica del programa y no a través de su representación textual.

Al representar gráficamente un sistema, las herramientas *CASE*, como *PROGRE*, establecen que la estructura interna del sistema debe ser correcta desde el principio. Algunas herramientas ya permiten al diseñador simular al sistema desde esta etapa, liberándolo de la fatiga de escribir miles de líneas de código antes de poder someter a prueba sus conceptos.

El *CASE* implementado en este trabajo de tesis es una herramienta de Software que resulta útil a través de los estados de especificación, diseño, implementación y mantenimiento del ciclo de vida del desarrollo de un programa; proporcionando soluciones legibles, entendibles y modificables.

Generalmente, cuando se habla de computación aplicada a la enseñanza se piensa en programas que enseñen matemáticas, geografía, gramática, etcétera, es decir, se utiliza la computación como herramienta pero no como el objetivo mismo del aprendizaje. En *PROGRE* la computación es el objetivo y la herramienta.

En *PROGRE* el programador aprende primero a diseñar algoritmos estructurados y posteriormente se ocupa de aprender los detalles sintácticos del lenguaje. Para ayudar a quien aprende a programar se diseñó e implementó una herramienta con la cual el nuevo programador desarrolla su programa en forma gráfica (diagrama de flujo estructurado) con la peculiaridad de que la herramienta lo dirige en la sintaxis del lenguaje, es decir, sólo le permite introducir estructuras donde el lenguaje así lo indica, le pide la condición en una estructura condicional en el lugar preciso, le indica cuando hay bifurcación de control, etcétera. Los detalles sintácticos, tales como signos de puntuación [; , . :], así como ciertas cláusulas [begin, end, until, do, etc.] son responsabilidad de *PROGRE*. Una vez que el programador ha terminado de crear el programa obtiene la versión correspondiente en forma textual indentada para mostrar claramente la modularidad y la estructura del programa. Además, con *PROGRE*, el diseño de los programas se facilita ya que existe la posibilidad de trabajar en una forma estructurada sin la necesidad de un lenguaje de programación específico. Por lo tanto, el diseño de un programa puede llevarse a cabo en lenguaje natural.

La documentación de un programa es necesaria y fundamental para el futuro mantenimiento del mismo. El diagrama de flujo estructurado, tanto del diseño como de

la codificación del programa, puede ser de gran utilidad cuando se requiere cualquier tipo de modificación. *PROGRE* le proporciona al usuario, en forma automática, dichos diagramas sin mayor esfuerzo por su parte, basta solicitárselo.

4.2.1 Sintaxis de *PROGRE*

La interface entre *PROGRE* y el usuario se realiza a través de menús, seleccionando la opción deseada por tres medios distintos: 1) con un "mouse", 2) utilizando las flechas para posicionarse y oprimiendo la tecla de "return" u 3) oprimiendo el número que aparece a la izquierda de la opción.

Algunas palabras claves de Pascal (IF, THEN, ELSE, FOR, CASE, etcétera) son comandos que invocan estructuras. La sintaxis se presenta en la figura 4.4 en forma de seis diagramas. Después de los diagramas se muestra una lista (figura 4.5) de todas las operaciones posibles de realizarse con *PROGRE*.

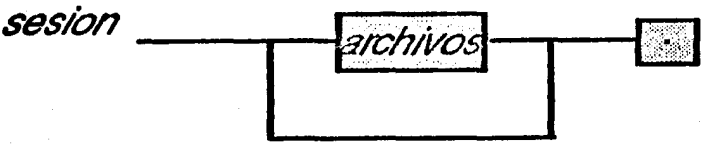


Figura 4.4.1 Sesión.

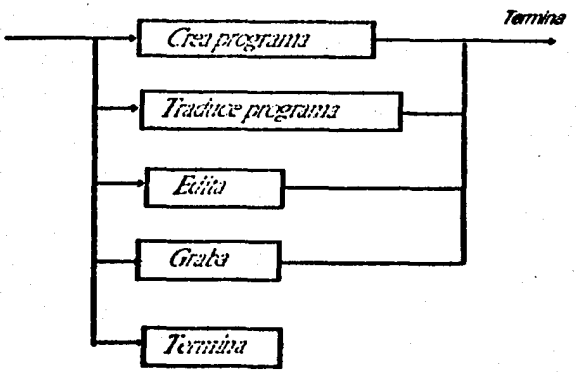


Figura 4.4.2 Archivos.

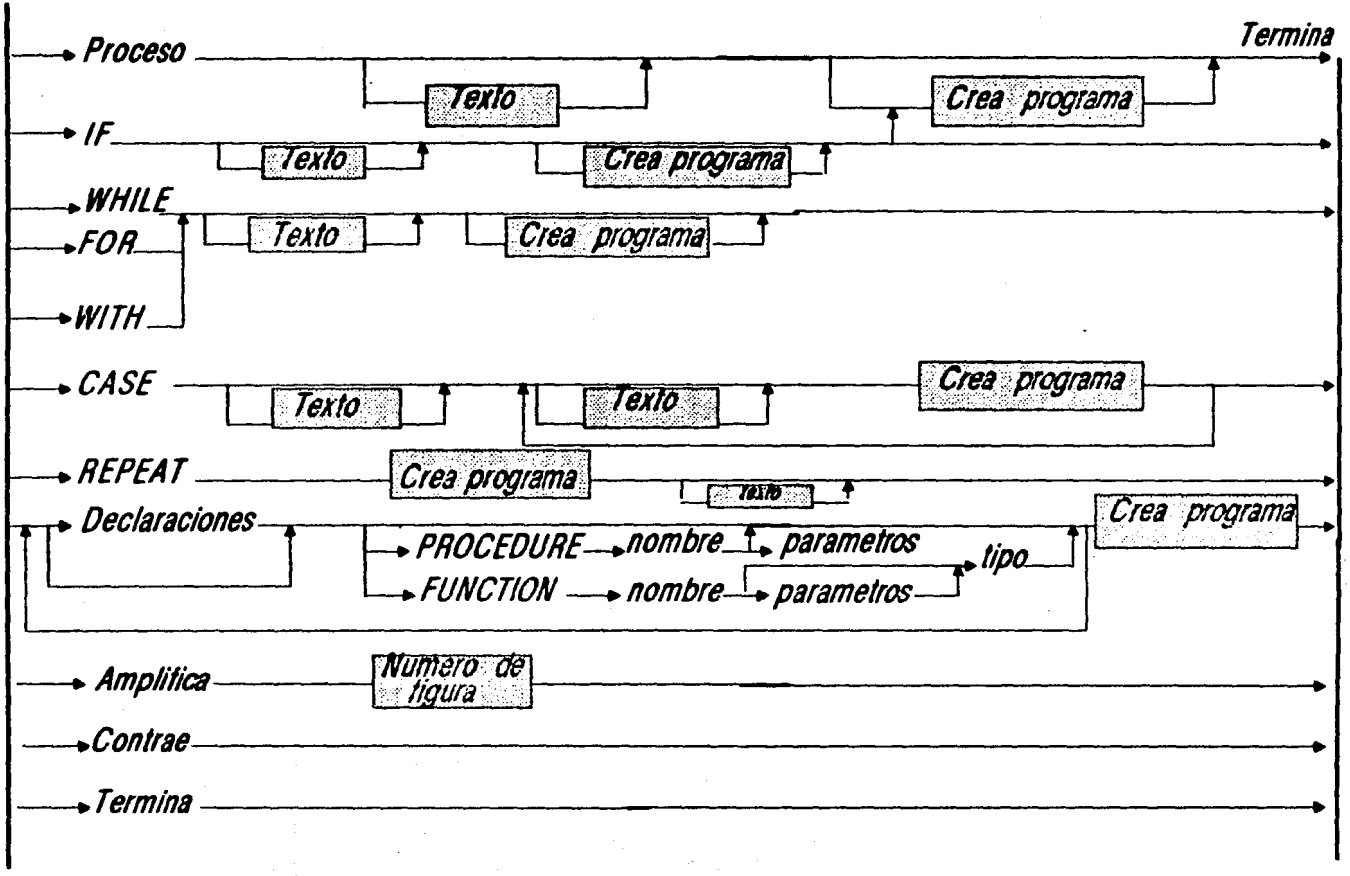


Figura 4.4.3 Crea programa.

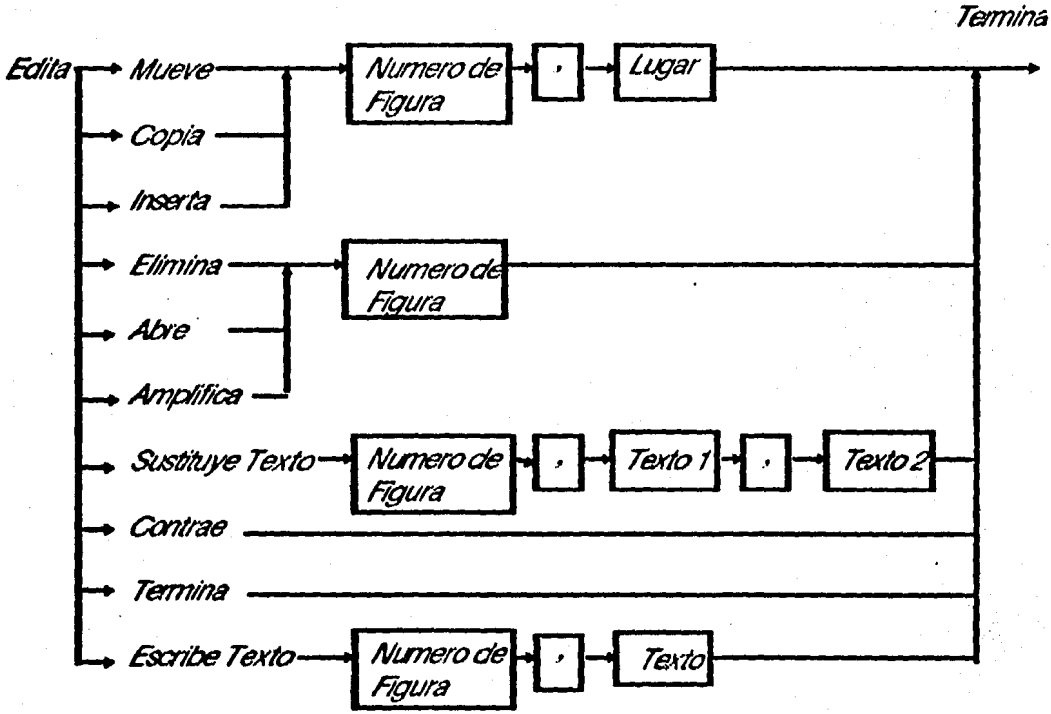


Figura 4.4.4 Edita.

Figura 4.4.5 Traduce.

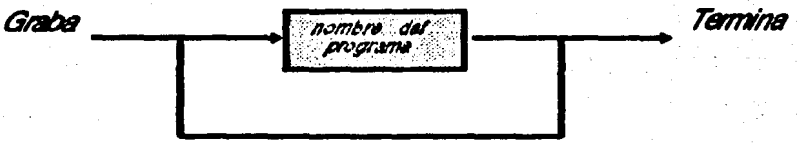


Figura 4.4.6 Graba.

Figura 4.4 Representación gráfica de la sintaxis de *PROGRE*.

1.. Abre estructura.

2. Amplifica estructura.

3. CASE.

4. Contrae estructura.

5. Copia estructura.

6. Crea programa.

7. Declaraciones.

8. Edita Programa.

9. Elimina estructura.

10. Escribe texto.

11. FOR

12. FUNCTION.

13. Graba archivos.

14. IF.

15. Inserta estructura.

16. Localiza estructura.

17. Mueve estructura.

18. PROCEDURE.

19. Proceso.

20. REPEAT.

21. Sustituye texto.

22. Termina.

23. Traduce programa.

24. WHILE.

25. WHIT.

Figura 4.5 Lista de las posibles operaciones a realizarse con *PROGRE*.

La idea básica de *PROGRE* es ayudar al usuario en el desarrollo de programas, por lo cual, esencialmente, se trabaja con archivos. De tal forma que el primer menú de operaciones que le ofrece al usuario es el de archivos (figura 4.6). Las dos primeras

opciones (Crea programa y Edita) le permiten al usuario tener presente el programa en memoria y trabajar con él. Tanto la creación de un programa como su edición se realizan siempre en forma gráfica utilizando GAL.

MENU DE ARCHIVOS

1. Crea programa

2. Edita

3. Traduce programa

4. Graba

0. Termina

Seleccione una opción

Figura 4.6. Menú de archivos.

El usuario utiliza la opción 1 (Crea programa) cuando va a comenzar a desarrollar el mismo. Dado que *PROGRE* no hace un análisis sintáctico del texto proporcionado por el usuario, se puede utilizar lenguaje natural o seudo código para "rellenar" las estructuras del programa, es decir, se puede comenzar desde el análisis, especificación y diseño del programa sin tener que preocuparse de la sintaxis estricta de un lenguaje de programación determinado.

Para la creación del programa el usuario va eligiendo las estructuras deseadas y rellenándolas o dejando "hoyos" para posteriormente cubrirlos.

El usuario elige la instrucción que desea incorporar en su programa y en ese momento *PROGRE* dibuja, en la pantalla, la estructura que representa la instrucción solicitada y dirige la sintaxis de la misma.

Dado que en cualquier programa puede haber varios niveles de anidamiento o profundidad, tanto en las instrucciones como en los procedimientos, pueden surgir dificultades, por límites de espacio de la pantalla, en la visibilidad del diagrama. Para resolver ésto, *PROGRE* tiene la facilidad de amplificar el tamaño de las estructuras internas como si estuviesen a primer nivel de definición (opción amplifica), es decir, hace un "zoom" de la figura. También cuenta con la operación inversa, que consiste en contraer el diagrama a su estado original (opción contrae).

CREA PROGRAMA

1. Proceso
2. IF
3. WHILE
4. FOR
5. WITH
6. CASE
7. REPEAT
8. Declaraciones
9. PROCEDURE
10. FUNCTION
11. Amplifica
12. Contrae
13. Termina

Figura 4.7 Crea programa.

Cuando el programador ha terminado de desarrollar su programa, o una sección de él, y selecciona la opción 13 (termina), *PROGRE* le muestra el *Menú de archivos*, donde se puede optar por traducir el programa (opción 3) a su versión textual en Pascal indentada siguiendo la sintaxis de este lenguaje de programación, o puede, en la misma sesión o en otra, editar el programa seleccionando la opción Edita del *Menú de archivos* (figura 4.8).

Con esta opción, el programador cuenta con un grupo de once operaciones. La opción 1 mueve de lugar una estructura, eliminándola de su lugar original. La opción 2 crea una copia de la estructura en el lugar indicado por el usuario. La tercera opción anula la estructura o conjunto de estructuras. Con la opción 4 se inserta la estructura existente en una nueva, la cual se define al seleccionar esta opción. Las opciones 5 y 6 permiten modificar el texto de las estructuras. La opción 7, Abre estructura, permite abrir una estructura ya existente e insertar en ella nuevas estructuras. Las operaciones de amplificación y contracción producen los mismos resultados que en la creación de programas. La opción 10 localiza estructuras por su número, permitiendo así realizar cualquiera de las operaciones del presente menú. Por último, con la opción 11 se termina la operación seleccionada del menú así como también la edición del programa.

y entonces regresa al *Menú de archivos.*

EDITA

1. Mueve

2. Copia

3. Elimina

4. Inserta

5. Sustituye texto

6. Escribe texto

7. Abre estructura

8. Amplifica

9. Contrae

10. Localiza

11. Termina

Figura 4.8 Edita programa.

Por último, la cuarta operación que se puede realizar con archivos, utilizando PROGRE, es grabar (opción 4, Graba del Menú de archivos) los programas gráficos y/o textuales en disco, para un posible futuro uso.

4.2.2 Ejemplo del desarrollo de un programa con *PROGRE*

Por medio de un ejemplo sencillo se intentará mostrar el uso de ***PROGRE*** en la creación de un programa durante el ciclo de vida de su desarrollo.

El problema a resolver consiste en encriptar un mensaje. La criptografía tiene por objetivo la transmisión, o almacenamiento, de mensajes "indescifrables"para todo receptor que no disponga de la "clave" o algoritmo descifrado. Son muy diversos y variados los sistemas criptográficos utilizados. Entre éstos se puede mencionar uno muy antiguo que consiste en substituciones y transposiciones alfabéticas (incluso utilizando varios alfabetos simultáneamente). Para la encriptación de los mensajes, en el presente ejemplo,se utilizará este sistema, de tal forma que el problema se reduce a desarrollar

un programa que reciba un mensaje, lo codifique sustituyendo unas letras por otras (la sustitución se debe realizar uniformemente a lo largo de todo el mensaje original; los signos de puntuación, blancos y letras minúsculas permanecen iguales) y, por último, guarde el texto modificado (encriptado).

El usuario invoca al sistema tecleando la palabra *PROGRE*. En ese momento se le ofrece el primer menú (*Menú de archivos*) del cual va a elegir la operación que desea ejecutar, proporcionando el número asociado a esa opción en el menú (en este caso, dado que va a comenzar a crear un programa, elige la opción 1, Crea programa). Para hacer una distinción entre lo que *PROGRE* escribe y lo que el usuario proporciona como texto de las estructuras en las figuras, aparecerán en mayúscula las cláusulas introducidas por el sistema y en minúscula el texto, que no es analizado sintácticamente, dado por el usuario. Con esta convención se podrá mostrar claramente la característica importante de *PROGRE* de auxiliar al programador en fijar su atención en la algorítmica y no detenerse en la sintaxis específica del lenguaje de programación (Pascal, en este caso).

Hasta el momento, en primera instancia, se han detectado tres funciones principales que el programa debe realizar: LeeCódigo, Encripta y EscTexto. Cada una de estas funciones será realizada por un módulo, es decir, el programa estará constituído por tres módulos. La interface entre ellos se llevará a cabo en el programa principal. Este primer diseño y la sesión utilizando *PROGRE*, con las opciones que el usuario ha solicitado, se muestra en la figura 4.9

Ejemplo 1

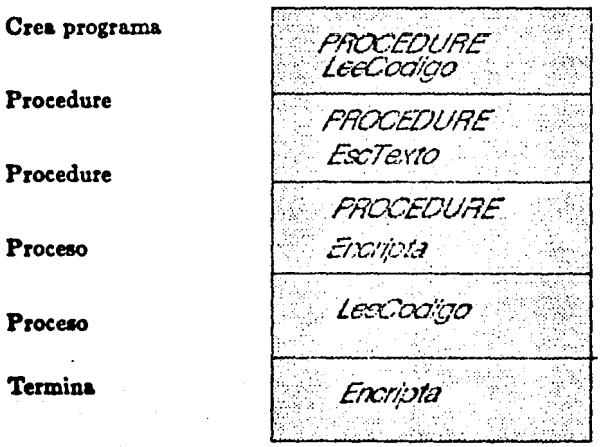


Figura 4.9 Primer estado del diseño de un programa.

Se continuará el desarrollo del diseño, siguiendo la técnica de *refinamiento por pasos sucesivos* (i.e. se definirán con más detalle los procesos de estos módulos). Para ello, se utilizará el arreglo "codigo" (cuyos índices van a variar de la 'A' a la 'Z') para almacenar el símbolo del código correspondiente a cada letra mayúscula (v.g. CODIGO[A] es el símbolo del código para la letra A). Los símbolos del código se leerán por medio de una proposición FOR en el procedimiento LEECODIGO, comenzando con el símbolo del código para la letra A.

El procedimiento ENCRIPTA leerá el mensaje y lo almacenará en el arreglo "mensaje" para examinar cada carácter por medio de una iteración FOR. Si un carácter representa una letra mayúscula su símbolo de código se guardará encriptado; de otra forma, se graba el carácter correspondiente en el mensaje, por medio del procedimiento ESCTEXTO. La salida muestra el criptograma desplegado bajo el mensaje original.

En el segundo estado del diseño se pueden comenzar a definir los módulos que compondrán el programa (figura 4.10).

Ejemplo 2

Sesión
opción
1 (Crea programa)

Menú de diseño
opción
8 (Declaraciones)

9 (PROCEDURE)

4 (FOR)

1 (Proceso)

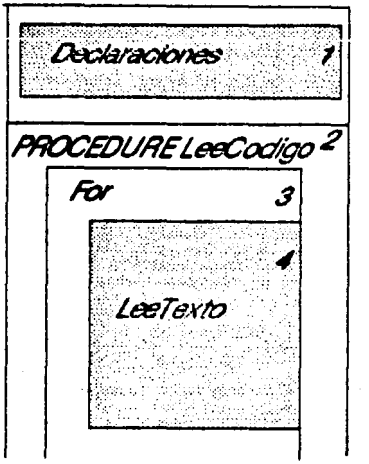


Figura 4.10 Segundo estado del diseño de un programa.

9 (PROCEDURE)

9 (PROCEDURE)

4 (FOR)

2 (IF)

1 (Proceso)

1 (Proceso)

3 (WHILE)

1 (Proceso)

1 (Proceso)

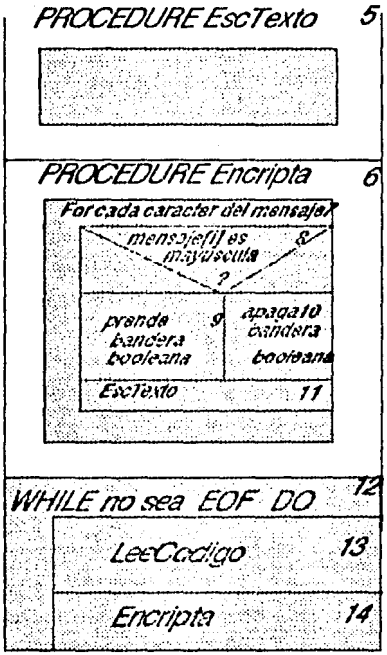


Figura 4.10 Continuación.

Hasta el momento se conoce la función de cada uno de los módulos y sus interfaces, pero el diseño aún es demasiado abstracto para poder codificarse en algún lenguaje de programación.Para continuar trabajando en este diseño de programa se tienen tres opciones:

a) Edita el archivo (ejemplo 1) y después de realizar los cambios, Graba el archivo modificado con otro nombre (ejemplo 2).

b) Edita el archivo (ejemplo 1) y después de realizar los cambios, Graba el archivo modificado con el mismo nombre (ejemplo 1).

c) Graba programa utilizando un nuevo archivo (ejemplo 2) con la nueva versión del diseño.

Con las opciones a) y c) se mantienen los dos archivos (lo cual es muy útil en la depuración y mantenimiento), mientras que con la opción b) se conserva únicamente la última versión del programa.

La diferencia entre las opciones a) y c) es que con la primera se tiene presente (en la pantalla) la última versión y no se requiere una copia del diseño en papel.

En la figura 4.11 se muestra la sesión donde se transforma "ejemplo 1" a "ejemplo 2" siguiendo la opción a). Puesto que el programa que se desea desarrollar es muy sencillo, no requiere de muchos estados de diseño, por lo que esta transformación produce el último estado del diseño del programa.

Ejemplo 2

Sesión
Menú de archivos

opción
2 Edita

opción
6, 1 (Escribe texto en estructura 1)

7, 2 (Abre estructura 2 e inserta)

1 Crea programa

opción
1 (Proceso)

1 (Proceso)

13 (Termina)

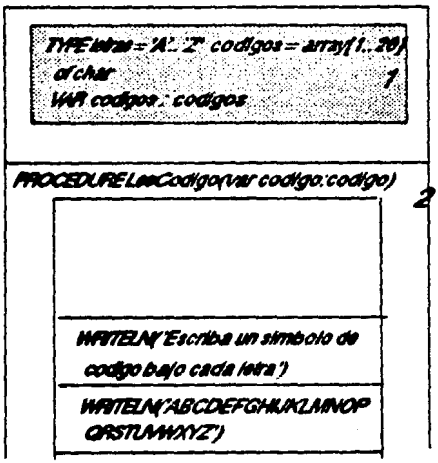


Figura 4.11 Ultimo estado de diseño de un programa.

Menú de Archivos

opción
2 (Edita)

opción
6, 3 (Escribe texto
estructura 3)

5, 5 (Sustituye texto
en estructura 4)

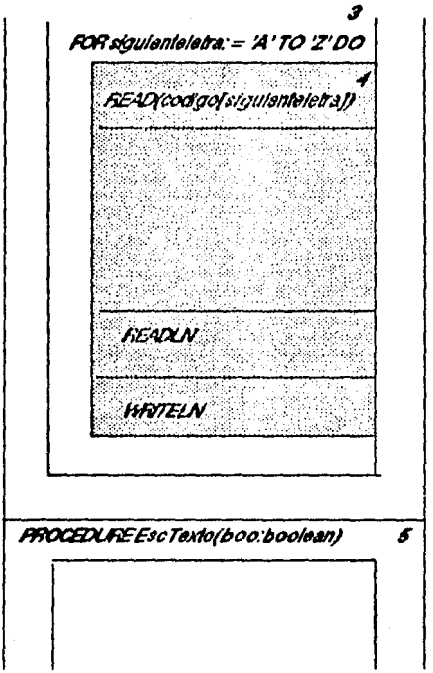
7, 5 (Abre estructura 5
e inserta)

Menú de archivos

opción
1 (Crea programa)

opción
1 (Proceso)

1 (Proceso)
13 (Termina)



Menú de archivos

opción
2 (Edita)

opción
7, 5 (Abre estructura 5
e inserta)

Figura 4.11 Continuación.

Menú de archivos

opción
1 (Crea programa)

opción
2 (IF)

1 (Proceso)
13 (Termina)

13 (Termina)
1 (Proceso)

Menú de archivos

opción
2 (Edita)

opción
5, 6 (Sustituye texto
en estructura 6)

7, 7 (Abre estructura
e inserta)
Menú de archivos

opción
1 (Crea programa)

opción
8 (Declaraciones)

1 (Proceso)

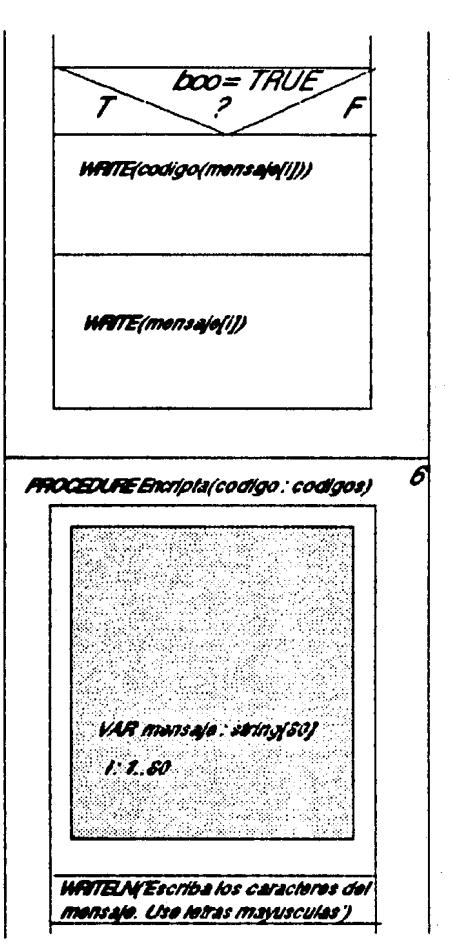


Figura 4.11 Continuación.

1 (Proceso)
13 (Termina)

5, 7 (Sustituye texto en estructura 7)

5, 8 (Sustituye texto en estructura 8)

5, 9 (Sustituye texto en estructura 9)

5, 10 (Sustituye texto en estructura 10)

5, 11 (Sustituye texto en estructura 11)

5, 12 (Sustituye texto en estructura 12)

11 (Termina)

Menú de archivos

opción
4 (Graba)

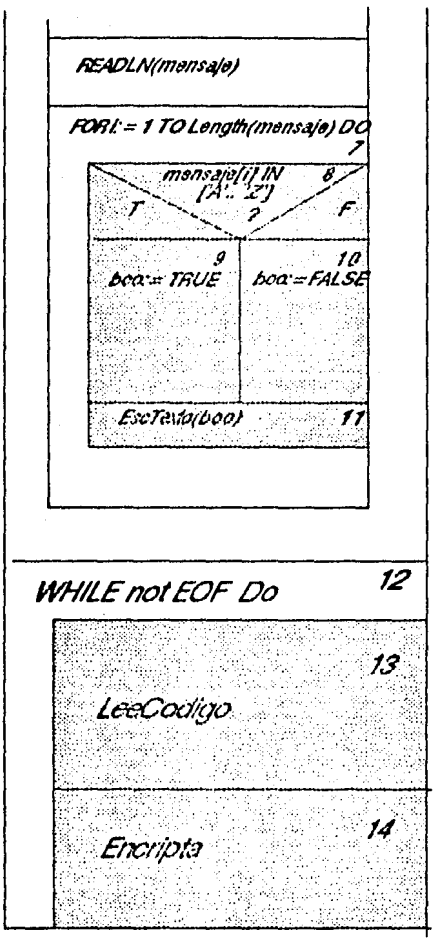


Figura 4.11 Continuación.

Un programa puede requerir de muchas modificaciones durante su vida. Estas pueden generarse porque hay cambios en las especificaciones, su entorno puede ser alterado o porque el programa no trabaja como debería hacerlo.

Si se codifica el diseño de la figura 4.11 y se ejecuta en una computadora van a surgir problemas: el arreglo "codigos" está declarado con una dimensión de 1 a 20; sin embargo, por un lado, se requieren 26 localidades en memoria para alojar los símbolos de los códigos que corresponden a las letras de la 'A' hasta la 'Z' y, por el otro, el tipo de índice no es el indicado; faltó declarar la variable "siguienteletra" en el procedimiento LEECODIGO y, finalmente, la instrucción que escribe el carácter correspondiente al mensaje va a ejecutarse siempre, sin importar si el carácter está escrito con letra minúscula o mayúscula, lo cual no satisface las especificaciones del problema. En la figura 4.12 se muestra la sesión en la cual se edita el programa para realizar dichos cambios y así depurar el programa (en la estructura 1 sustituye el rango indicado de 1..20 por la palabra LETRAS: inserta una estructura de declaraciones para declarar la variable SIGUIENTELETRA, después de la estructura 2; mueve la estructura 17 donde aparece la proposición WRITE(MENSAJE[i]) a la estructura del IF (estructura 15), en el lugar correspondiente al ELSE. Esta misma modificación puede llevarse a cabo copiando la estructura de proceso WRITE(MENSAJE[i]) al lugar indicado en la estructura del IF y eliminándola posteriormente de su lugar original.

Se podría continuar trabajando sobre esta última versión del programa, posiblemente en otra sesión, para, por ejemplo, estructurarlo un poco más y de esta forma hacerlo más autodocumentado.

```
Edita

Sustituye texto: "1..20","letras"

Inserta 2, Declaraciones

var siguienteletra:letra

Mueve 13,12 o,

(Copia 13,12

Elimina 13)
```

Figura 4.12 Ejemplo de una sesión de *PROGRE*.

Ejemplo 2

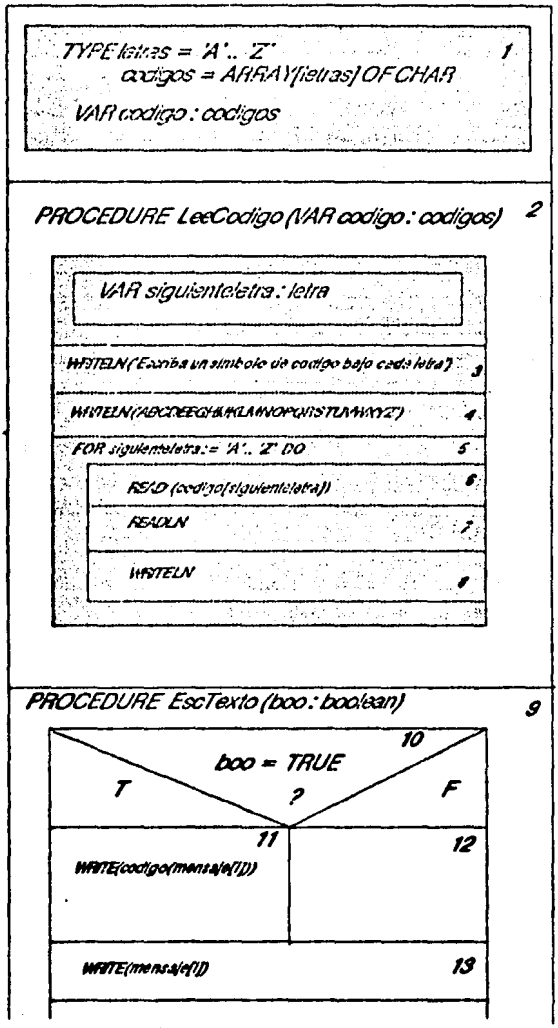


Figura 4.12 Continuación.

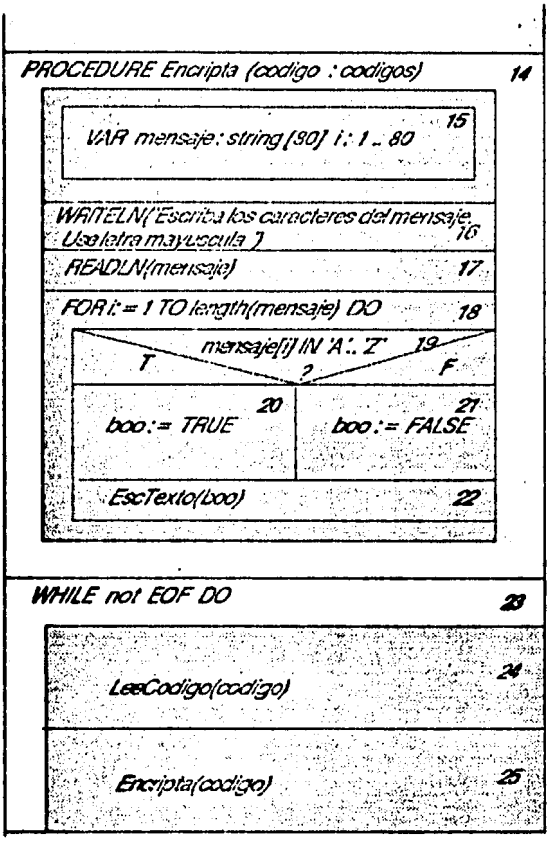


Figura 4.12 Continuación.

En la siguiente figura (figura 4.13) se muestra el programa en su forma textual (programa fuente) producido por *PROGRE* a partir de su representación gráfica, el cual está listo para ser compilado y posteriormente ejecutado. Este forma textual indentada, se obtuvo al invocar la opción Traduce (que pertenece al Menú de Archivos), con lo cual se activó el analizador semántico.

```
PROGRAM criptograma;
TYPE
    letras = 'A'..'Z';
    codigos = ARRAY [letras] OF CHAR;
VAR
    codigo : codigos;
    boo:boolean;

PROCEDURE leecodigo(VAR codigo : codigos);
VAR
    siguienteletra : letras;

BEGIN
    WRITELN (' Escriba un símbolo de código bajo cada letra.');
    WRITELN ('ABCDEFGHIJKLMNOPQRSTUVWXYZ');

    FOR siguienteletra := 'A' TO 'Z' DO
      READ (codigo[siguienteletra]);
    READLN;
    WRITELN
END; (* leecodigo *)

PROCEDURE esctexto(boo:boolean);
BEGIN
    IF boo THEN
      WRITE(codigo[mensaje[i]])
ELSE
      WRITE(mensaje[i])
END; (* esctexto *)
```

Figura 4.13 Texto del programa en PASCAL.

```
PROCEDURE encripta (codigo : codigos);
VAR
    mensaje : STRING[80];
    i: 1..80;
BEGIN
    WRITELN ('Escriba los caracteres del mensaje. Use letras mayúsculas.');
    READLN(mensaje);
    FOR i := 1 TO length(mensaje) DO
      IF mensaje[i] IN ['A'..'Z'] THEN
        boo:=TRUE
      ELSE
        boo:=FALSE;
      esctexto(boo)
END; (* encripta *)

BEGIN
    WHILE NOT(EOF) DO
    BEGIN
      leecodigo(codigo);
      encripta(codigo)
    END
END.
```

Figura 4.13 Continuación.

4.3 Implementación de PROGRE

Como ya se mencionó, al utilizar PROGRE, el programador únicamente puede desarrollar programas estructurados. Esto se debe, principalmente, a que los programas están diseñados e implementados a través de un diagrama de flujo estructurado (GAL), mediante el cual se obtiene la representación gráfica del programa y cuya descripción se encuentra en la sección 4.1. Esta técnica obliga al programador a seguir el método de modularidad, por lo que, esencialmente, sus programas están constituídos de procedimientos y funciones, los que, al mismo tiempo, le permiten observar las abstracciones funcionales [**Cheatman 81**]. Además, el hecho de utilizar un lenguaje de programación estructurado como Pascal, le facilita la implementación del concepto de abstracción.

Para llevar a cabo esta implementación, *PROGRE* representa un programa internamente como un árbol de sintaxis, cuyo concepto se describe a continuación:

Un programa es bastante más que únicamente una cuerda de caracteres; es una estructura jerárquica de estructuras sintácticas. El programador, al construir un programa, lo hace como un conjunto de estructuras y no como una secuencia de líneas de caracteres y, al escribir el programa en un lenguaje de programación, éste puede dividirse en componentes sintácticos, los que a su vez están relacionados por las reglas sintácticas que gobiernan el lenguaje. Por lo mismo, la representación interna de un programa es un árbol de sintaxis, donde cada nodo representa una estructura del lenguaje y donde el nivel de profundidad del nodo es el nivel de la estructura dentro del programa. Este árbol sintáctico, que apoya la estructura en niveles del programa, está de acuerdo con la técnica de la programación estructurada para el diseño de programas conocida como "*refinamiento por pasos sucesivos*" [**Coleman 78**]. Los elementos de la representación concreta del lenguaje, tales como: palabras reservadas, signos de puntuación, separadores, etcétera, no forman parte del árbol.

Como *PROGRE* representa internamente un programa como un árbol de sintaxis, cada estructura corresponde a un nodo de cierto tipo en el árbol. El número de hijos de cada nodo y el número de ramificaciones de la estructura correspondiente son iguales. Por ejemplo, un nodo que represente una proposición IF tiene dos hijos, uno le corresponde a la parte del THEN y el otro a la parte del ELSE y esto siempre es cierto. Es decir, una proposición IF se define y representa en el árbol como una estructura compuesta de dos partes (THEN y ELSE), aunque una de ellas o ambas puedan estar vacías. En *PROGRE*, las proposiciones siempre se definen de la misma forma, por lo que ésta no depende de su uso específico en una determinada construcción. La ventaja de utilizar esta estructura de árbol es que facilita añadir y suprimir estructuras (subárboles) y, al realizar modificaciones, no es necesario recorrer de nuevo todo el árbol.

La forma en que *PROGRE* representa internamente un programa de usuario es utilizando dos estructuras de árbol: el primero de esos árboles está formado por el identificador del programa (que representa la raíz del árbol) y por las declaraciones y definiciones de los módulos, donde el cuerpo de estos últimos se ve como un todo, sin considerar detalles de su funcionamiento. Cada nodo que representa un módulo puede tener dos hijos, uno está constituído por sus declaraciones y el otro por su cuerpo. Si el módulo tiene módulos definidos dentro de él, entonces el nodo de declaraciones tendrá un hijo (que es el siguiente módulo); en caso contrario, el nodo de declaraciones será un nodo terminal. Los módulos que estén definidos en el mismo nivel serán hermanos.

El segundo árbol construído por el sistema sirve para la representación de los módulos. En este caso, únicamente, se representan los módulos sin incluir sus declaraciones, ya que éstas se encuentran representadas en el primer árbol. Las proposiciones en un mismo nivel de definición son hermanas. Los hijos de cada proposición son las proposiciones que están definidas dentro de ella, en el siguiente nivel de definición, y así sucesivamente.

La proposición secuencial o de proceso no tiene hijos. La proposición IF tiene dos hijos (que corresponden a la parte del THEN y a la parte del ELSE y de las cuales una de ellas o ambas pueden ser vacías). Las proposiciones WHILE, WITH, FOR y REPEAT pueden tener un solo hijo, cada una. Finalmente, la proposición CASE tiene n hijos donde n es el número de condiciones que aparecen en el CASE.

Knuth [**Knuth 68**] definió un método para examinar los nodos del árbol. Por medio de este método, sistemáticamente, cada nodo del árbol es visitado exactamente una vez al caminar a lo largo del mismo. Son tres los recorridos de un árbol definidos por Knuth: preorden, postorden y enorden.

PROGRE recorre el árbol sintáctico en preorden, para obtener la representación concreta de las estructuras, así como la gráfica del programa. El recorrido en preorden del árbol binario consiste en un algoritmo recursivo en el cual, primero se visita la raíz, después el subárbol izquierdo y, por último, el subárbol derecho.

La representación de un programa como dos árboles, en la forma descrita anteriormente, da al usuario la facilidad de diseñar su programa utilizando los métodos de modularidad o abstracción, sin perder la idea completa de lo que esta haciendo. La técnica de refinamiento por pasos sucesivos da lugar, únicamente, a un incremento en la profundidad del árbol.

Para implementar los dos árboles del programa, *PROGRE* requiere de dos archivos. Uno de ellos almacena la información que relaciona la representación de la forma gráfica y los dos árboles como una lista ligada. El otro archivo contiene el texto del programa, sin las palabras reservadas y los signos de puntuación que corresponden a Pascal.

El archivo, cuyos registros almacenan la información referente a la representación gráfica del programa, en disco se llama *ESTRUCTURAS* y se guarda en memoria en un arreglo de registros, que es sobre el que trabajará directamente *PROGRE*. A este

arreglo se le llamó *FORMAGRAFICA*.

Una segunda estructura utilizada por *PROGRE* es un arreglo de caracteres que contiene el texto de cada una de las estructuras. Además, contiene un apuntador al arreglo *FORMAGRAFICA*, al registro que contiene la información de la estructura. A este arreglo se le llamó *TEXTOEST*.

PROGRE utiliza un tercer archivo, cuyo nombre es *TEXTOS*, para guardar la forma textual indentada, siguiendo la sintaxis de Pascal, generada al invocar la operación para traducir un programa que se encuentre en su representación gráfica a su forma textual. Este archivo se almacena en memoria en un arreglo, cuyos registros son arreglos de caracteres.

Cada registro de *FORMAGRAFICA* representa una estructura del programa y está compuesto por diez campos. Cada campo sirve para almacenar la información descrita a continuación.

1.- El primer campo, llamado **código**, contiene el código de la estructura representada por el registro. Cada estructura tiene asociada un número entero, o código, el cual la identifica. Los códigos se muestran en la figura 4.14.

1. Proceso
2. PROCEDURE
3. FUNCTION
4. WHILE
5. FOR
6. REPEAT
7. IF
8. CASE
9. WITH

Figura 4.14 Códigos para las diferentes estructuras de Pascal.

2.- El nivel de profundidad a que se encuentra la estructura se almacena en el campo **nivprof** y representa la profundidad del nodo en el árbol del programa. Esta información es necesaria para recuperar o construir la forma gráfica de un programa al desplegarlo en la pantalla. Este campo es un número entero mayor o igual a uno. El

nivel uno representa la raíz del árbol.

3.- El siguiente campo, **numpant**, indica el número de pantalla en que se dibujó la estructura al construir el programa, ya que debido a las limitaciones del tamaño de la pantalla, es imposible representar un programa completo, íntegramente, en una sola pantalla (a menos, por supuesto, de que éste sea muy pequeño).

4.- Cada estructura y, consecuentemente el registro que la representa, debe conectarse de alguna manera con el identificador del programa al cual pertenece, por lo que se usará el campo **approg** como apuntador al archivo de programas.

5.- Como ya fue mencionado, a excepción de la estructura secuencial o de proceso, las demás estructuras pueden contener otras estructuras, por lo que una estructura puede tener tantos hijos como estructuras contenga. El número de hijos de una estructura se almacenará en el campo **numestint**.

6.- En cada registro, también existe un campo, llamado **apestint**, que contiene el apuntador al registro de la primera estructura contenida en esa estructura.

7.- De la misma forma en que una estructura puede contener otras estructuras dentro de ella, también puede estar contenida dentro de una estructura. Los árboles, en ***PROGRE***, están implementados como listas ligadas y las ligas son en ambas direcciones, esto es, listas doblemente ligadas, por lo que también existe un campo, para cada estructura, llamado **appadre** para guardar el apuntador al registro de la estructura que la contiene. En los registros de las estructuras que se encuentren en un nivel 1 de profundidad, este campo toma el valor de NIL.

8.- El campo **sigest** es un apuntador a la siguiente estructura en el mismo nivel, es decir, es una liga a una estructura hermana.

9.- Las ramificaciones de las estructuras se conectan por este campo que es un apuntador llamado **apramif**. Por ejemplo, el campo apramif de un registro IF apunta al primer registro del THEN, cuyo apuntador apramif, apunta a su vez al primer registro del ELSE.

10.- Por último, se tiene el campo **aptexto** que es un apuntador al arreglo de texto, al registro en el cual se guarda el texto de la estructura dado por el usuario. Este texto no contendrá las palabras reservadas de Pascal, ni signos de puntuación.

Adicionalmente, se utilizaron otros arreglos de registros para almacenar la información referente a los usuarios y a los programas relacionados con cada uno de los usuarios. La estructura de datos *USUARIOS* tiene tres campos: **idusuario, apprograma y contprogramas**. El primer campo es una cuerda de caracteres donde se almacenará el identificador de cada usuario del sistema; el segundo, es un apuntador al

primer programa de ese usuario en el arreglo *PROGRAMAS* y, el tercer campo guarda el número de programas que han sido desarrollados por este usuario. Este arreglo únicamente podrá ser accesado y modificado por el administrador del programa ya que solicita, para su acceso, una *clave secreta* ("password").

PROGRAMAS es otro arreglo que utiliza *PROGRE* para almacenar la información general de los programas. Cada registro tiene seis campos: **idprograma, apusuarlo, apsigprog, apformgraf, apdec** y **approfun**. En **idprograma** se almacena el identificador del programa; el siguiente campo es un apuntador al arreglo *USUARIOS* y sirve para indicarle al sistema a qué usuario pertenece, ya que existe la posibilidad de que dos o más usuarios utilicen un mismo nombre para identificar programas diferentes. El campo **apsigprog**, apuntará al registro, en el arreglo *PROGRAMAS*, donde se localice la información del siguiente programa, que pertenezca al mismo usuario. Este campo valdrá *NIL* cuando el programa sea el último de la lista. El cuarto campo, llamado **apdec**, liga este programa con el arreglo en el cual se guardarán las declaraciones globales del programa y las declaraciones locales de los procedimientos y funciones que en él se definan. El campo **apformgraf** apunta al arreglo de estructuras, conectando de esta forma al programa con las proposiciones que constituyen su cuerpo. Por último, el campo llamado **approcfun** apunta al arreglo para información de procedimientos y funciones, en el cual se almacenan los identificadores de los procedimientos y de las funciones que conforman el programa.

El arreglo *DECLARA* se utiliza para almacenar tanto las declaraciones globales del programa principal, como las declaraciones locales de los procedimientos y funciones. Cada registro consta de los siguientes campos: **tipo, dec, apnext, appadre** y **apbody**. En **tipo** se guarda un número entero entre 0 y 2 que indica si se trata del programa principal (0), de un procedimiento (1) o de una función (2). El siguiente campo, llamado **dec**, es un arreglo de 80 caracteres donde se almacenarán las declaraciones. En caso de que la longitud del registro sea insuficiente, para almacenar todas las declaraciones, se cuenta con **apnext** que es un apuntador al registro donde continúan las declaraciones. Para indicar que se trata del último registro se guarda NIL en este campo. Para conectar este registro con el programa, procedimiento o función en que están contenidas las declaraciones, se cuenta con el campo **appadre**, el cual apuntará, en caso de que se trate del programa principal, al registro en que se encuentre el identificador del programa correspondiente del arreglo *PROGRAMAS*. Por otro lado, si se trata de un procedimiento o función, apuntará al registro respectivo del arreglo *PROCFUN* . Finalmente, **apbody** apunta al registro, en *FORMAGRAFICA*, en que se encuentra la información de la primera estructura del cuerpo del módulo.

En el arreglo *PROCFUN* se almacena la información que identifica a los procedimientos y funciones. Cada registro cuenta con un campo para almacenar el identificador o nombre, otro para diferenciar si se trata de un procedimiento o de una función, y dos apuntadores, uno que apunta al arreglo de declaraciones y otro al de estructuras.

En *TEXTOEST* se guardará el texto de las estructuras, sin almacenar las palabras reservadas, ni los signos de puntuación. Además, cada registro cuenta con un

apuntador al registro de la estructura correspondiente.

El último arreglo que utiliza *PROGRE* es *TEXTOS*, en el cual se almacenará la forma textual indentada del programa. El primer registro del programa tiene un campo que apunta al identificador del programa. Otro campo sirve para apuntar al siguiente programa del mismo usuario.

4.3.1 Descripción del programa

PROGRE esencialmente se compone de dos conjuntos de operaciones, las que se utilizan para crear el programa (*Crea programa*), desarrollándolo interactivamente en su forma gráfica y, las que se utilizan para realizar las modificaciones en el programa (*Edita programa*). Las operaciones de creación y edición de programas siempre se realizan a través de su versión gráfica. Además, el usuario puede optar por la opción *Traduce* la cual traduce su programa en dos formas: permitiéndole obtener a partir de la forma gráfica el programa textual en Pascal o, a partir de la forma textual en Pascal obtener la representación gráfica o diagrama de flujo del mismo. Por último, el programa se puede grabar en disco para usos futuros. Estas cuatro operaciones con los programas constituyen el primer nivel de menús o *Menú de archivos*.

La idea básica de *PROGRE* es ayudar al usuario en la manipulación de archivos, es decir, crearlos, modificarlos, traducirlos y protegerlos; por lo tanto, una sesión consistirá de un conjunto de operaciones con archivos. Esas operaciones permitirán al usuario tener su programa presente en memoria. El programa puede traerse a memoria usando los comandos *Crea, Edita y Traduce*. El comando de edición solamente puede utilizarse cuando ya existe la representación gráfica del programa. El árbol del programa se genera al utilizar los comandos *Crea y Traduce*. El primero al crear la representación gráfica del programa utilizando *PROGRE* y el segundo, al traducir un programa que esté escrito en Pascal, aunque su forma textual no haya sido generada utilizando *PROGRE*, al invocar a *PROGRE* para obtener su representación gráfica.

La selección de las opciones se puede llevar a cabo por dos medios distintos: utilizando el teclado o con la ayuda de un "mouse". Si se elige el teclado, la elección, a su vez, puede realizarse utilizando las flechas o proporcionando el número de la operación elegida. El usuario debe indicarle al sistema al inicio, la forma de elección y el medio que va a utilizar.

Al iniciarse una sesión de *PROGRE*, el sistema solicita al usuario su identificador. Si éste no existe, despliega en la pantalla un mensaje de error y finaliza la sesión; en caso contrario, el sistema llama al procedimiento *INICIALIZA*, el cual inicializará todas las variables, apuntadores y arreglos necesarios. Este procedimiento invoca a la rutina *CARGAARCHIVOS*, la cual se encarga de copiar la información de los archivos *ESTRUCTURAS, FILEUSUARIOS, FILEPROG* y *FILETEXTOS* en disco, a los arreglos correspondientes en memoria.

A continuación, el sistema limpia la pantalla y despliega el *Menú de archivos*, ya descrito en la sección 4.2.1. para que el usuario haga su elección.

El programa principal de *PROGRE* implementa las operaciones del *Menú de archivos*: Crea programa, Edita, Traduce y Graba. El usuario puede elegir las operaciones que quiere llevar a cabo y el sistema va a invocar los procedimientos correspondientes, hasta que el usuario elija la opción que indica "*fin de sesión*" (opción 5 del *Menú de archivos*). Si durante la sesión un archivo es modificado, el usuario debe proteger la versión actualizada mediante la opción 4 (Graba). Sin embargo, esta protección se realizará automáticamente, aún cuando el usuario olvide hacerlo, ya que existe una variable booleana que controla este hecho y que se vuelve cierta si algún archivo es modificado durante la sesión.

El procedimiento *MENUPRIN* llama al procedimiento correspondiente dependiendo del comando elegido por el usuario. Las acciones que van a ejecutar los diversos procedimientos se describen a continuación:

4.3.1.1 Crea programa

Al elegir el usuario esta opción del *Menú de archivos*, el sistema, en una forma interactiva, desarrolla el programa dibujando estructuras, las cuales representan las diferentes construcciones de Pascal, y llenándolas con otras estructuras o con el texto correspondiente. El procedimiento *CREAPROG*, inicialmente, solicita el identificador del programa. Si éste ya existe, envía un mensaje de error, indicando que el nombre está duplicado y regresa al *Menú de archivos*. En caso contrario, añade el nombre a la lista de programas del usuario, limpia la pantalla y despliega en ella dos ventanas: sobre la ventana del lado derecho aparecerá el *Menú de diseño*, que muestra la lista de las operaciones que pueden llevarse a cabo para crear el programa. La ventana del lado izquierdo servirá como área de trabajo sobre la cual se irá construyendo la representación gráfica del programa.

El *Menú de diseño*, como ya se indicó en la sección 4.2.1, consta de trece opciones. Las primeras diez corresponden a las estructuras de control que pertenecen a Pascal. El usuario elige la estructura que desea incorporar a su programa y *PROGRE* lo irá guiando en la sintaxis de la misma. El ciclo se repetirá hasta que el usuario elija la opción de terminación (opción 13, Termina), para indicarle al sistema que desea regresar al *Menú de archivos*.

Cada estructura, excepto la que corresponde a la proposición secuencial o de proceso, puede tener al menos una estructura interna. A esta estructura se le considera como una unidad indivisible. El anidamiento dentro de las otras estructuras es ilimitado.

Para dibujar las estructuras, el procedimiento *CREAPROG*, llama a la rutina *DIBUJAFIGURA*, pasándole como parámetro el número de la estructura elegida. Este

procedimiento se encargará de dibujar las figuras que representan cada estructura en una forma secuencial, cambiando las coordenadas del cursor para que la figura quede en el lugar correcto. Este procedimiento llama a la función *VERIFICA*, la cual, como su nombre lo indica, se encarga de verificar si la estructura solicitada por el usuario, en un momento dado, puede ser dibujada, ya que, por ejemplo, un procedimiento no puede ser insertado dentro de un bloque de proposiciones, ya que, como el sistema conoce la sintaxis de Pascal, permite al programador insertar construcciones del lenguaje sólo cuando el lenguaje así lo indica. Por ejemplo, cuando el usuario invoca la estructura del IF, no puede invocar otra estructura hasta que haya llenado la condición o la haya dejado vacía para llenarla posteriormente (oprimiendo la tecla de "*Escape*"). Posteriormente, debe desarrollar la parte del THEN, terminar con la opción termina (opción 13) y, a continuación, desarrollar la parte del ELSE, terminar de la misma forma y, es entonces cuando ya puede agregar una estructura diferente. Esta función también verifica si una cierta secuencia de estructuras es válida o no. En caso de que no lo sea, se desplegará en la pantalla un mensaje de error.

Por otro lado, si la estructura es aceptada, el procedimiento *DIBUJAFIGURA* llama al procedimiento *LEETEXTO* para que lea el texto que le corresponde a esa estructura y el cursor se coloca en el lugar adecuado para escribir el texto dentro de la figura. En ese momento, *PROGRE*, a través de *CREAPROG*, toma el siguiente registro del arreglo *ESTRUCTURAS* y llena los campos que corresponden a **codigo, nivprof, appadre, aptexto**. Al mismo tiempo, guarda el texto indicado en el arreglo *TEXTOEST*. Sin embargo, el usuario tiene la opción de oprimir la tecla de "*Escape*" y dejar un "hoyo" para llenarlo posteriormente en la misma sesión o en otra.

El procedimiento *DIBUJAFIGURAS* sabe si la estructura tiene ramificaciones o si puede tener estructuras dentro de ella. Si se tiene el primer caso (estructuras IF y CASE), *PROGRE* le indica al usuario la ramificación correspondiente por la que debe tomar. En el segundo caso, el usuario puede invocarlas después de que ha escrito el texto de la estructura "padre" o, en caso contrario, elegir la opción 13 (Termina), ya que también, durante la fase de diseño, *PROGRE* permite al usuario dejar "hoyos" en una proposición o en el bloque de un programa, los cuales, pueden llenarse posteriormente en una o varias sesiones. El usuario puede indicarle al sistema que ha terminado de desarrollar la totalidad de su programa, o una parte de él y que desea regresar al *Menú de archivos* eligiendo la opción 13. El único requisito es que todas las estructuras invocadas hayan sido cerradas anteriormente, con la misma opción.

Como ya se mencionó anteriormente, todas las proposiciones, excepto la de proceso, pueden tener proposiciones dentro de ellas, lo que puede dar lugar a un anidamiento ilimitado. Esto puede producir problemas de legibilidad para observar la forma gráfica del programa en la pantalla. Sin embargo, es posible obtener un despliegue claro de cualquiera de las estructuras utilizando la opción Amplifica (opción 11). Esta opción limpia la pantalla y amplifica la estructura en cuestión, sin importar el nivel de profundidad en que se encuentre. La operación contraria se lleva a cabo eligiendo la opción 12 (Contrae). Estas operaciones pueden realizarse tanto en la creación como en

la edición de un programa.

PROGRE manipula un programa por medio de su estructura de árbol; pero no hay esperanza de desplegar mucho de este árbol gráficamente sobre una pantalla pequeña. Aunque una buena propuesta de la programación estructurada es producir procedimientos o módulos pequeños (v.g. el número de líneas que pueden caber en una página de listado), no hay garantía de que cada programador concuerde o cumpla con ésto. Para prevenir la situación en la cual el programador produzca un módulo más largo que el tamaño de la pantalla, *PROGRE* tiene un mecanismo automático que controla el tamaño del despliegue de la pantalla y que actúa de la siguiente forma: el procedimiento *DIBUJAFIGURA* llama a la función *ESPACIOSUF* que verifica si la estructura a ser dibujada cabe completamente en la pantalla. Si no es así, llama al procedimiento *GUARDAFIGURA*, el cual protege en memoria la parte gráfica que se tenga en ese momento sobre la pantalla, limpia la pantalla y comienza a dibujar la siguiente estructura en la parte superior del área de trabajo. También incrementa el contador del número de pantallas, guardando este valor en el campo correspondiente del registro de la estructura.

Cada vez que se le indica al sistema que una estructura va a contener una o varias estructuras, se incrementa el nivel de profundidad. Algunos campos se escriben tan pronto como *DIBUJAFIGURAS* acepta la estructura y los campos restantes se escriben hasta el momento en que la estructura ha terminado de ser definida.

4.3.1.2 Edita programa

La edición de un programa implica realizar modificaciones en un programa. Un editor de sintaxis deberá permitir cambios en términos de la estructura del árbol, con operaciones sobre sus ramificaciones, o por cambios arbitrarios en el texto. La estructura del árbol puede cambiarse al suprimir, insertar, copiar y mover subárboles. Como las modificaciones a un programa van a realizarse sobre su forma gráfica, si el usuario solicita esta opción antes de haber creado el programa correspondiente, el sistema desplegará un mensaje de error y retornará al *Menú de archivos.*

Las modificaciones a un programa pueden llevarse a cabo invocando la opción Edita, del *Menú de archivos.* Esta opción limpia la pantalla y, al igual que para la creación de un programa, despliega dos ventanas. Sobre la ventana del lado derecho aparecerá el *Menú de edición* con la lista de operaciones permitidas (ver sección 4.2.1). La ventana izquierda servirá también, en este caso, como área de trabajo. Hay que recordar que como las modificaciones van a realizarse sobre la forma gráfica de un programa, éste debe haber sido creado con anterioridad. Al invocar esta opción, el sistema solicitará el nombre del programa a editar y lo buscará en la lista de programas del usuario. Si el identificador no existe (lo cual quiere decir que tampoco existe su forma gráfica), el sistema enviará un mensaje de error y regresará al *Menú de*

archivos. En caso contrario, aparecerá en la pantalla el *Menú de edición*, mostrando las operaciones mediante las cuales se modificará el programa y que a continuación se describen.

MUEVE

La opción 1 (Mueve) mueve de lugar una estructura, eliminándola de su lugar original, por lo que el subárbol que la representa desaparece.

COPIA

Copiar una estructura (opción 2 Copia), consiste en crear una copia de la estructura en el lugar indicado por el usuario. Esto es, se reproduce el subárbol especificado, en el lugar indicado.

ELIMINA

La opción 3 (Elimina estructura) anula la estructura o conjunto de estructuras indicadas por el usuario y cuyo efecto en el árbol de sintaxis es el mismo que se indica en la opción 1. La figura 4.15 muestra los efectos de esas tres operaciones.

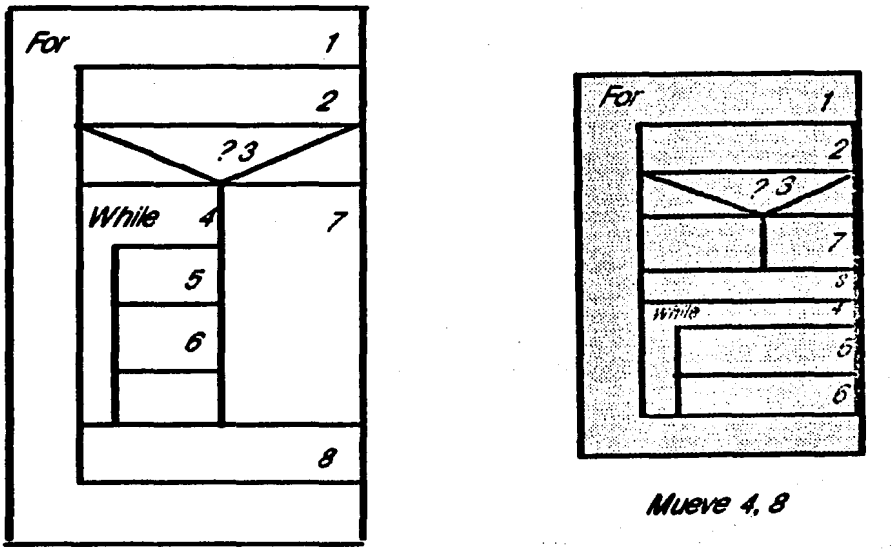
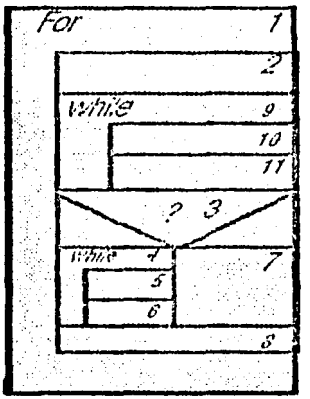
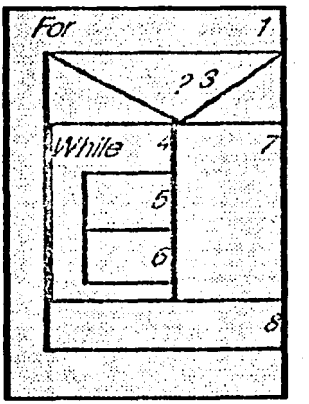


Figura 4.15 Efectos de las operaciones Mueve, Copia y Elimina estructura.



Copia 4, 2



Elimina 2

Figura 4.15 Continuación.

ABRE e INSERTA

Para *PROGRE*, un programa es una estructura jerárquica compuesta de estructuras. Cada figura en *PROGRE* representa una estructura. Por lo tanto, se pueden insertar nuevas estructuras dentro de una estructura ya existente (opción 7, Abre estructura) o, insertar estructuras existentes en una nueva estructura (opción 4, Inserta). Por ejemplo, se tienen algunas estructuras secuenciales o de proceso y se quiere poner-

las dentro de una estructura de WHILE (insertar estructuras ya existentes dentro de una nueva); el ejemplo se muestra en la figura 4.16.

La otra situación surge cuando se tiene una estructura de WHILE y se quiere insertar en en ella alguna otra estructura, ver figura 4.17.

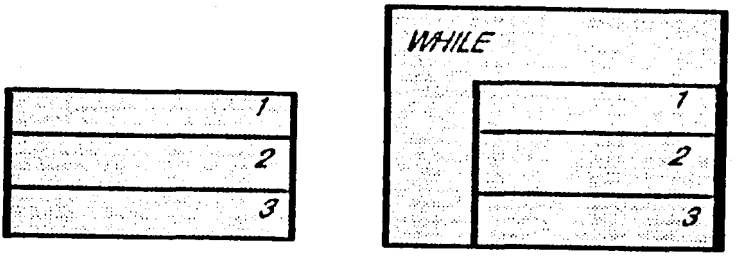


Figura 4.16 Efecto de la operación Inserta estructura.

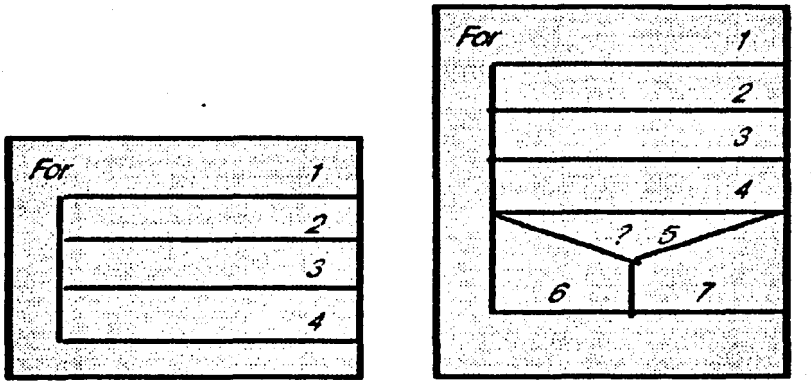
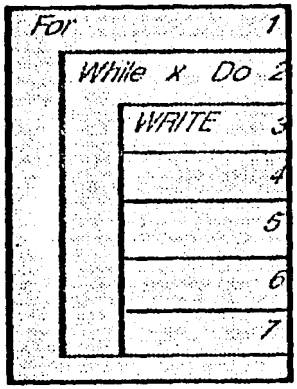
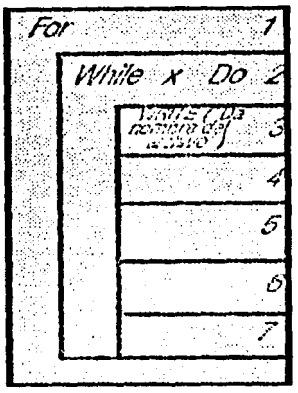


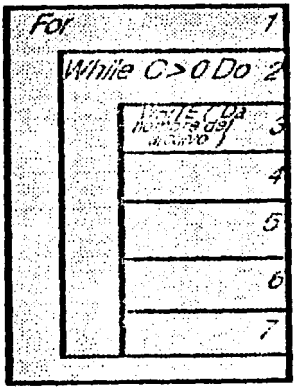
Figura 4.17 Efecto de la operación Abre estructura.

SUSTEXTO y ESCTEXTO

Las modificaciones de texto se realizan con la ayuda de dos operaciones. La operación más común para ejecutar un cambio en el texto consiste en substituir una cuerda de caracteres por otra (opción 5, Sustituye texto). *PROGRE* permite una segunda operación (opción 6, Escribe texto), por medio de la cual se llenan los "hoyos" dejados en las estructuras. Cuando el usuario se encuentra desarrollando un programa, o parte de él, puede pedir estructuras y dejarlas vacías (es decir, sin texto) y llenar los "hoyos" después en la misma sesión o en otra. En la siguiente figura (figura 4.18) se muestra un ejemplo de estas dos operaciones.







Escribe 3, ('Da nombre del archivo') *Sustituye 2, X, C>0*

Figura 4.18 Ejemplo de las operaciones Sustituye y Escribe texto.

AMPLIFICA y CONTRAE

Las opciones 8 y 9 (Amplifica estructura y Contrae estructura, respectivamente) producen los mismos resultados, descritos anteriormente, para la creación de archivos, un ejemplo de ello se muestra en la figura 4.19.

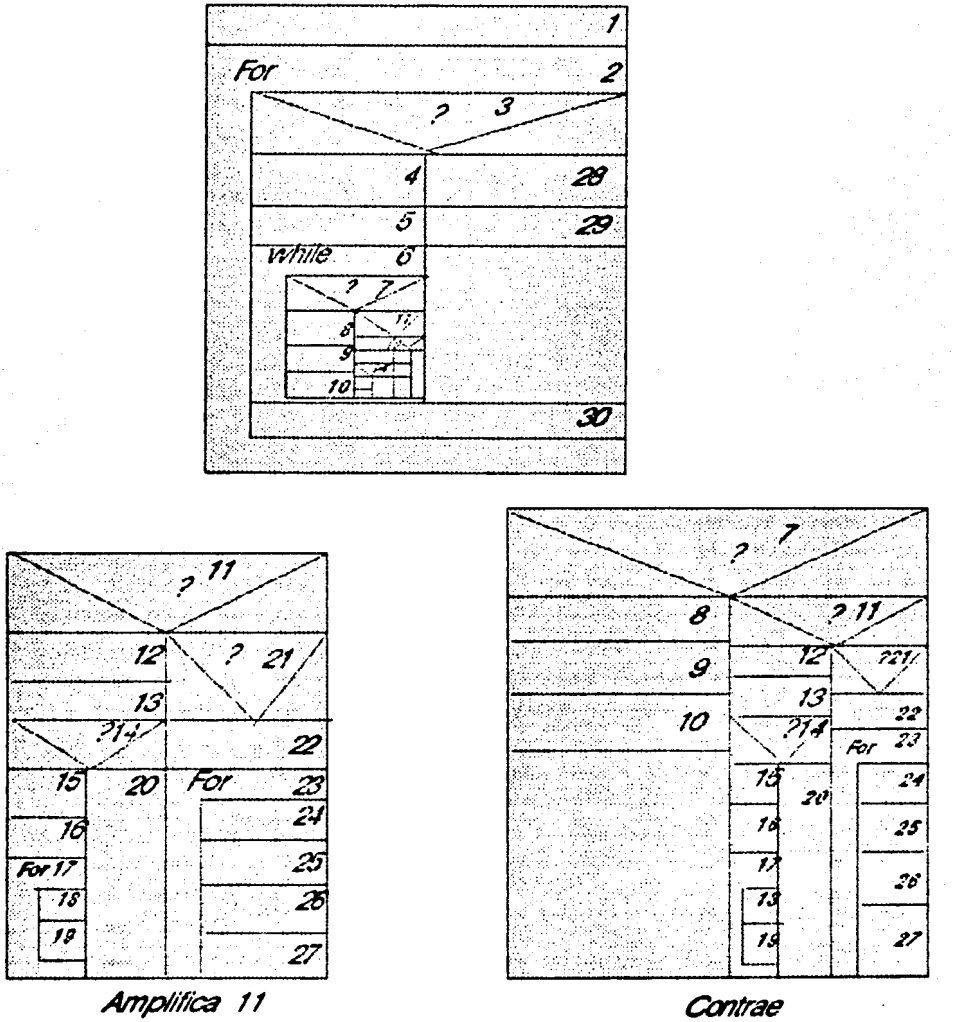


Figura 4.19 Ejemplo de las operaciones Amplifica y Contrae estructura.

LOCALIZA

Para localizar una estructura (opción 10) se da su número correspondiente. El apuntador del programa se posiciona en esta estructura y las siguientes operaciones se ejecutan sobre ella, hasta que una nueva estructura es solicitada.

FINMOD

Finalmente, la opción 11 (Termina), como su nombre lo indica, sirve para terminar la edición del programa. El sistema, al recibir esta opción, protege inmediatamente en disco los cambios realizados al programa y el control regresa al menú de archivos.

Todos los cambios al programa se llevan a cabo lógicamente y los números de las estructuras se conservan hasta que el usuario graba el programa.

4.3.1.3 Traduce programa

Cuando el programador ha terminado de desarrollar un programa, o una sección de él, y selecciona la opción 13 (Termina) del *Menú de diseño*, *PROGRE* le muestra el *Menú de archivos*, donde puede optar por traducir el programa (opción 2, Traduce) a su versión textual indentada en Pascal, siguiendo la sintaxis de este lenguaje de programación. Por otro lado, el sistema también permite solicitar otro tipo de traducción, ya que es capaz de aceptar programas escritos en Pascal, aunque no hayan sido desarrollados utilizando *PROGRE*, para producir su representación gráfica. Esto es posible ya que el sistema, al recibir el comando que le indica que el usuario desea traducir un programa, le pregunta que tipo de traducción desea, ya que cuenta con dos opciones:

1) GAL a Pascal y, 2) Pascal a GAL. Sin embargo, existen ciertas restricciones en ambos casos: en el primero ya debe existir la forma gráfica del programa y, en el segundo caso, el programa no puede tener proposiciones GOTO ni declaraciones de etiquetas.

La traducción de la representación gráfica (GAL) a la forma textual en Pascal del programa se lleva a cabo a través del procedimiento *TRADUCEATEXTUAL*, el cual toma, del arreglo de programas, el apuntador al primer registro del programa, en el archivo de estructuras. De ese primer registro, a su vez, toma el apuntador al primer registro del programa en el archivo donde está guardado el texto. A continuación, llama al procedimiento *GENERATEXTO*, que con la ayuda de los campos del archivo de estructuras, construye la forma textual del programa.

La forma textual del programa es un programa indentado, siguiendo la sintaxis del lenguaje de programación Pascal, con las claúsulas y los signos de puntuacion correspondientes. El usuario no tiene que preocuparse por las palabras reservadas de Pascal (begin, end, then, else, until, do, etcétera), ni por los signos de puntuación (; , . : etcétera), debido a que el sistema los inserta automáticamente por él, ni por la

indentación de las proposiciones, la cual, también, se obtiene en forma automática.

La forma textual obtenida se escribe en el archivo **TEXTOS** colocando los apuntadores correspondientes, tanto en este archivo como en el de programas y en el de estructuras.

Para el segundo tipo de traducción (forma textual en Pascal a representación gráfica en GAL), el sistema invoca al procedimiento *TRADUCEAGAL*. Esta rutina analiza el programa del usuario, tomando en cuenta la sintaxis de Pascal, pero sin considerar la estructura del texto, es decir, reconoce asignaciones, expresiones y condiciones, aunque no verifica si están bien formadas.

Si la proposición es una llamada a procedimiento, una llamada a función o una asignación (lo cual se reconoce porque la primera palabra de la proposición no es una palabra reservada), entonces *TRADUCEAGAL* la traduce a una estructura secuencial o de proceso. A continuación, se dedica a buscar un elemento que le indique la terminación de la proposición (v.g. un signo de puntuación, una palabra reservada de Pascal o alguna de las siguientes claúsulas: begin, end, else, until).

Si la proposición comienza con alguna de las siguientes palabras reservadas de Pascal: IF, CASE, WHILE, FOR, REPEAT o WITH, *TRADUCEAGAL* la traduce a la estructura correspondiente. Por otro lado, si la proposición comienza con alguna de las siguientes palabras: LABEL, CONST, TYPE o VAR, el procedimiento de traducción construirá una estructura de declaración, en la que colocará el texto que encuentre, como si fuese una cuerda de caracteres, hasta que localice un elemento que le indique la terminación de las declaraciones (BEGIN, PROCEDURE o FUNCTION).

Para cada proposición del programa, *TRADUCEAGAL* construye un nuevo registro en el arreglo *FORMAGRAFICA*, llenando los campos correspondientes y colocando el texto en el arreglo *TEXTOEST*, sin las palabras reservadas de Pascal ni los signos de puntuación.

4.3.1.4 Graba

Cuando el usuario elige la opción 4 (Graba) del *Menú de archivos*, el sistema manda llamar al procedimiento *PROTEGE* el cual, al ejecutarse protege todos los archivos del sistema en disco.

4.3.2 Documentación

La documentación del sistema consiste en un manual en que se hace una breve descripción de cada uno de los procedimientos y funciones que conforman el sistema, el número y tipo de los parámetros que requiere cada uno de ellos, los procedimientos y funciones que estos procedimientos y funciones invocan a su vez y, por último, los datos que debe proporcionar el usuario.

La programación de *PROGRE* se llevó a cabo en una microcomputadora PS/2 modelo 50 de IBM, que cuenta con tarjeta de graficación a color VGA y tiene adaptado un mouse. Se usó el lenguaje de programación Turbo Pascal versión 4.0 [**Borland 87**].

El sistema realiza, esencialmente, las cuatro funciones del *Menú de archivos*: Crea programa, Edita, Traduce y Graba. Cada una de estas funciones, para cumplir su objetivo, cuenta con un conjunto de procedimientos y funciones que llevan a cabo, en conjunto, la función requerida.

El sistema está programado a partir de "*Units*" y trabaja en forma interactiva utilizando menús. Antes de continuar con la explicación del sistema, se explicará, brevemente, en qué consisten las "units".

Una "unit" o unidad es una colección de constantes, tipos de datos, variables, procedimientos y funciones. Una unidad es semejante a un programa escrito en Pascal consistente de una biblioteca de declaraciones, la cual se puede incluir dentro de otro programa y así permitir que el programa sea dividido y compilado en varias partes. También proporciona un conjunto de capacidades, por medio de procedimientos y de funciones, con constantes, tipos de datos y variables de soporte, cuya implementación se mantiene oculta, separando la unidad en dos secciones: la de *interface* y la de *implementación*. Cuando un programa utiliza una unidad, todas las declaraciones de ésta están disponibles como si hubiesen sido definidas dentro del programa mismo.

La estructura de una unidad difiere poco de la de un programa; sin embargo, existen algunas diferencias significativas, por ejemplo:

```
Unit <identificador>;
Interface
Uses <lista de unidades>; (* opcional *)
    (declaraciones públicas)
Impementation
    (declaraciones privadas)
    (procedimientos y funciones)
BEGIN
    (código de inicialización)
END.
```

La porción de interface, o parte pública, de una unidad comienza con la palabra reservada *Interface* y termina cuando encuentra la palabra *Implementation.*

La interface determina qué es "visible" a cualquier programa (o a otra unidad) que utiliza esa unidad. Cualquier programa que usa la unidad tiene acceso a los elementos "visibles" (en la sección de interface es posible declarar constantes, tipos de datos, variables, procedimientos y funciones).

La sección de implementación - la parte "privada" - comienza con la palabra *Implementation.* Todo lo que sea declarado en la porción de interface es visible en la sección de implementación: constantes, tipos de datos, variables, procedimientos y funciones. Además, en la implementación es posible hacer declaraciones adicionales, aunque éstas no sean visibles a los programas (o unidades) que utilicen esa unidad.

PROGRE utiliza las siguientes unidades:

- *Globales.* Contiene las declaraciones globales para el sistema.

- *Inicializa.* Es la encargada de inicializar las variables y arreglos necesarios para el comienzo de la ejecución.

- *Marcos.* Comprende los procedimientos encargados de dibujar los distintos tipos de marcos utilizados en la creación de las ventanas.

- *Figuras.* Está constituída por las rutinas que dibujan las estructuras de Pascal

-*Utilerías.* En esta unidad se agrupan las rutinas que realizan las funciones de utilería (Raton, HandleKey, HandleFuncKey, Beep, IniMenus, EscMenu, Portada, Expresion, ExpresionBoo, etcétera).

Estas unidades se invocan en los procedimientos que realizan las funciones principales del *Menú de archivos* y que se describen a continuación:

Crea programa

Para crear un programa, utilizando *PROGRE*, se elige la primera opción del *Menú de archivos* (opción 1, Crea programa) la cual a su vez, despliega el *Menú de diseño.* El usuario, a través de éste, construirá el programa, ya que las primeras diez opciones del *Menú de diseño* son las estructuras del lenguaje de programación Pascal. El sistema dibujará en la pantalla el diagrama de flujo estructurado del programa.

Si un programa escrito en Pascal es invocado, y éste ya existe, entonces puede ser incorporado a *PROGRE*, seleccionando la opción 3 del *Menú de archivos* (Traduce), indicándole, cuando el sistema así lo solicite, que se trata de traducir de un programa escrito en Pascal a su forma gráfica estructurada en GAL. El sistema pedirá también

el nombre del programa correspondiente (el nombre consistirá de hasta 8 caracteres alfanuméricos, cuyo primer carácter debe ser un alfabético). Al solicitar esta opción, el sistema crea los registros de las estructuras en el arreglo correspondiente y a partir del cual se podrá construir la forma gráfica.

Al elegir cualquiera de las dos opciones (1 ó 3) se obtendrán dos archivos: uno que contiene la estructura de apuntadores, mediante la cual se puede generar el diagrama de flujo estructurado y uno de texto conteniendo la forma textual indentada en Pascal.

Para construir un programa, utilizando *PROGRE*, se invoca al procedimiento *DIBUJAFIGURA*, el cual dibujará las distintas figuras que representan las estructuras de Pascal. De esta forma, el programa se va construyendo por medio de la unión de las diferentes figuras indicadas por el usuario, el cual no necesita estar familiarizado con las palabras reservadas de Pascal, tales como: [BEGIN, END, ELSE, UNTIL, THEN, etcétera]; o conocer cuándo utilizar algunos signos de puntuación, tales como: [; , . , : etcétera], ya que el sistema se encarga automáticamente de su inserción.

Cuando el usuario elige una estructura, él únicamente observa en la pantalla el inicio de la figura que representa esa estructura, posteriormente, el sistema le solicita el texto respectivo, es decir, la condición, en caso de que se trate de un IF o de un WHILE, la expresión, para el caso de un WITH o de un CASE, etcétera. Cuando el usuario elige otras estructuras, éstas formarán el cuerpo de la primera, mientras no sea cerrada con la opción 13 del *Menú de diseñõ*. En ese momento, el sistema dibujará, en la pantalla, la parte final de la figura que representa la estrutura.

Todas las estructuras requieren ser cerradas, excepto la que representa una proposición secuencial o de proceso, ya que ésta no puede contener otras estructuras en su interior.

No existe un límite para la longitud del texto que se incluye en las estructuras, ya que se implementó un mecanismo de "scroll". Para indicar fin de texto sólo se requiere oprimir la tecla de "escape". El cursor se posiciona automáticamente, en el lugar correcto de la figura y espera el texto respectivo. Si, en ese momento, el usuario no desea escribir nada, basta con que oprima "escape" y el sistema dejará un "hoyo", que podrá ser llenado posteriormente.

Para indicar que ya no se desea añadir otra estructura, se elige la opción 13 (Termina) del *Menú de diseño.*

Dado que cualquier estructura de Pascal (excepto la de proceso) admite estructuras internas, el anidamiento puede resultar ilimitado, lo cual abre la posibilidad de que la imagen de una estructura, en un momento dado, sea tan pequeña que resulte confusa o poco visible. *PROGRE*, para solucionar dicho problema, proporciona dos operaciones, opeciones 11 y 12 del *Menú de diseño*, con las cuales se puede amplificar o contraer la figura. Al solicitarse la opción 11 (Amplifica), el sistema limpia la pantalla y dibuja la estructura elegida por el usuario como si estuviese a nivel uno de profundidad. Para regresar a la imagen original, el usuario sólo tiene que elegir la opción 12

(Contrae).

Los procedimientos que realizan cada una de las funciones del *Menú de diseño* se explican a continuación:

ProcFunc (opción 1 Proceso)

Este procedimiento se encarga de dibujar un rectángulo (ver figura 4.1.1) dentro del cual podrán definirse asignaciones, llamadas a procedimientos o a funciones, etcétera, es decir, aquéllas proposiciones de Pascal que no contengan otras proposiciones en su interior. El cursor se posiciona en el extremo superior izquierdo donde el usuario puede escribir el texto correspondiente y terminar oprimiendo la tecla de "escape".

IFfunc (opción 2 IF)

La figura que dibuja este procedimiento es un rectángulo, en cuyo interior se dibujan otros tres (ver figura 4.1.4). El primero se coloca en la parte superior y dentro de él está colocado un triángulo donde irá escrita la expresión booleana o condición. La parte restante se divide en dos rectángulos iguales. El rectángulo dibujado a la derecha corresponde al bloque del THEN y el izquierdo al del ELSE.

Cuando se invoca esta opción, el sistema dibuja únicamente la parte superior (el rectángulo pequeño con el triángulo en su interior) y posiciona el cursor dentro del triángulo. En ese momento, el usuario puede escribir la condición del IF y terminar oprimiendo la tecla de "escape" u oprimir solamente esta tecla y dejar un "hoyo" para llenarlo, posteriormente, en otra sesión de edición. Al recibir el "escape", el sistema dibuja la parte faltante, y se posiciona, inmediatamente, en el lugar correspondiente para esperar las estructuras del THEN. Las siguientes estructuras que se invoquen corresponderán a esta parte hasta que el usuario elija la opción 13 (Termina), para cerrar la estructura correspondiente al THEN. En este momento, el sistema se encuentra listo para recibir las estructuras correspondientes al ELSE. Al elegir la opción 13, que cierra esta parte, también se cierra la estructura del IF y el sistema regresa al *Menú de diseño.*

WHILEfunc y FORfunc (opciones 3 y 4)

Como estas proposiciones son similares en su sintaxis, se describirán en forma conjunta. Al invocar cualquiera de estas opciones, el sistema dibuja la figura respectiva según sea el caso. Estas estructuras corresponden al WHILE-DO y al FOR-TO de Pascal (ver figuras 4.1.2 y 4.2.2 respectivamente). Posiciona el cursor en el lugar correcto y, en el primer caso, espera la condición correspondiente, y en el segundo, la iteración del FOR. El usuario no requiere escribir la cláusula DO del WHILE ni el TO del FOR, ya que el sistema las inserta en forma automática.

El texto correspondiente a cada una de las proposiciones se termina oprimiendo la tecla de "escape". Las estructuras que se invoquen a continuación corresponderán al siguiente nivel de la estructura seleccionada en ese momento (WHILE o FOR). Para cerrar esta estructura basta seleccionar la opción 13 (Termina).

WITHfunc (opción 5)

Este procedimiento es parecido a los anteriores, tanto en la sintaxis como en la forma de la figura de la estructura (ver figura 4.2.1). Corresponde al WITH-DO de Pascal, al igual que en el caso anterior, el usuario no requiere escribir la cláusula DO. El sistema posiciona el cursor en el lugar adecuado y espera la expresión correspondiente. Para terminar basta oprimir la tecla de "escape". El sistema actúa de la misma forma que en el caso anterior, con respecto a las estructuras internas de la proposición WITH.

CASEfunc (opción 6)

Al invocarse esta opción, el procedimiento CASEfunc dibuja la estructura correspondiente al CASE-OF de Pascal (ver figura 4.3). Esta estructura se representa como un rectángulo vertical e inicialmente sólo se dibuja la parte superior de él. El cursor se posiciona en el lugar correcto y espera la expresión correspondiente. Para terminar se oprime "escape". El sistema se encarga de insertar automáticamente la palabra OF.

A continuación, el cursor se posiciona en forma automática, para esperar la primera etiqueta del CASE. Al oprimir "escape", el usuario indica que ha terminado y el sistema inserta inmediatamente ":". A partir de ese momento, las siguientes estructuras que se invoquen, corresponderán a dicha etiqueta, hasta que se oprima la tecla de "escape", indicando su terminación.

El sistema esperará la siguiente etiqueta y el ciclo se repetirá de la misma forma descrita en el párrafo anterior, hasta que el usuario indique que la estructura del CASE debe cerrarse eligiendo la opción 13 (Termina).

REPfunc (opción 7 REPEAT)

La forma de la estructura dibujada por este procedimiento es muy similar a la de las proposiciones WHILE, FOR y WITH, y consiste de un rectángulo dentro de otro de mayor tamaño (ver figura 4.1.3). Corresponde a la proposición REPEAT-UNTIL de Pascal.

A diferencia de las demás estructuras, el texto correspondiente a esta proposición se solicita hasta el final, cuando las estructuras internas ya han sido invocadas y la estructura va a ser cerrada. El usuario, para indicar su terminación, después de elegir la opción 13 que cierra la última estructura interna al REPEAT, oprime la tecla de "escape". En ese momento, el sistema inserta la palabra UNTIL y posiciona el

cursor para esperar la condición correspondiente. De nuevo, para terminar, el usuario oprime "escape" y, entonces, el sistema cierra la estructura del REPEAT.

DECLARA (opción 8 Declaraciones)

Este procedimiento se encarga de dibujar dos rectángulos de distinto tamaño, uno menor dentro de otro mayor (ver figura 4.2.4). Se utiliza para declarar los identificadores del programa. El usuario puede escribir tantas líneas como lo desee e indicar su terminación oprimiendo "escape". En ese momento, el sistema cierra la figura.

Esta opción sólo puede ser invocada al inicio de la creación de un programa o inmediatamente después de invocar las opciones 8 ó 9 (PROCEDURE o FUNCTION). En el primer caso, los identificadores declarados serán globales y, en el segundo, locales.

PROC y FUNC (opciones 9 PROCEDURE y 10 FUNCTION)

Estas dos estructuras son iguales en su forma (ver figura 4.2.3) por lo que se les tratará de igual manera. Permiten al usuario definir los módulos de su programa. El sistema dibuja la parte superior de la estructura y posiciona el cursor para recibir la definición del módulo, ésto es, su identificador, nombre y tipo de sus parámetros y, en el caso de que se trate de una función, el tipo de ella. Para terminar se oprime la tecla de "escape".

A continuación, y de la misma forma que si se tratase de un programa, el usuario puede invocar la opción 8 para declarar los identificadores locales; la opciones 9 ó 10, para definir otros módulos o, definir las estructuras internas del módulo. Para cerrar la parte inferior de la estructura del módulo, el usuario elige la opción 13 (Termina).

AMPLIFICA (opción 11)

Este procedimiento permite al sistema tomar una estructura, seleccionada por el usuario, que se encuentre en cualquier nivel de profundidad, y desplegarla en la pantalla, después de limpiar esta última, como si estuviese a nivel uno. El usuario sólo requiere proporcionar el número de la estructura elegida.

CONTRAE (opción 12)

El efecto contrario a la operación anterior se obtiene invocando esta opción. Al recibir esta orden, el sistema contrae nuevamente la estructura que había sido amplificada y la regresa a su tamaño original, a su nivel de anidamiento correspondiente. La estructura completa aparece de nuevo en la pantalla.

Edita programa

El usuario puede requerir hacer modificaciones a un programa por distintas razones: el programa contiene errores, cambian los requerimientos, es necesario llenar los "hoyos" dejados en las proposiciones, el usuario está utilizando el método de *refinamiento por pasos sucesivos*, etcétera.

Un programa, para ser modificado utilizando *PROGRE* requiere estar presente en memoria, en su forma gráfica. Para modificar un programa, el usuario invoca la opción 2 (Edita) del *Menú de archivos*, en ese momento, el sistema borra la pantalla y despliega el *Menú de edición* con la lista de las opciones con que cuenta. Para terminar cualquier operación de edición y para terminar la edición misma, el usuario debe invocar la opción 11 (Termina).

Para llevar a cabo la modificación de cualquier estructura, el usuario debe llamarla por su número. Cada estructura, dentro del programa, tiene un número que la identifica y que le fue asignado en el momento de su creación. La numeración se realiza en forma secuencial. Algunas de las operaciones de edición necesitan sólo un número, ya que la operación se efectuará únicamente sobre una estructura; sin embargo, algunas operaciones pueden aplicarse sobre una o varias estructuras. En el segundo caso, se indicarán dos números, que representarán el rango de las estructuras sobre las cuales desea realizarse la operación. Después de llevar a cabo varias modificaciones, es posible que la numeración ya no conserve el orden inicial que tenía cuando fue creado el programa. Esto puede ser resultado de la inserción y/o eliminación de algunas estructuras. Para obtener nuevamente un orden secuencial, basta invocar la opción 4 (Graba) del *Menú de archivos* y el orden será restaurado. Los procedimientos, que realizan las funciones de edición, se describen a continuación.

MUEVE (opción 1)

Este procedimiento mueve una estructura, o un conjunto de estructuras, a la posición indicada por el usuario, borrándolas de la posición original. Su sintaxis es:

<num. estructura, nueva posición>

donde num. estructura corresponde a un sólo número (p.e. 3), o a un rango (v.g. 2-6, de la estructura 2 a la 6).

COPIA (opción 2)

Si el usuario desea copiar una sola estructura, o un conjunto de estructuras, a una nueva posición, invoca esta opción, la cual llama al procedimiento que se encarga de realizar la operación correspondiente, dejando la estructura (o estructuras) también en su lugar original. La sintaxis es la misma que para el caso anterior.

ELIMINA (opción 3)

Si el usuario desea eliminar una estructura, o conjunto de estructuras, del programa puede invocar esta opción. La sintaxis es similar a los casos anteriores.

INSERTA y ABRE (opciones 4 y 7)

Para insertar una o varias estructuras dentro del programa, el sistema cuenta con dos procedimientos, a través de los cuales es posible llevar a cabo esta operación.

Es posible insertar nuevas estructuras dentro de una estructura ya existente invocando la operación ABRE. Por otro lado, para insertar estructuras ya existentes en el programa dentro una nueva se invoca la operación INSERTA. En ambos casos la sintaxis es:

<num. estructura, num1. estructura>

donde num. estructura corresponde al número de una sola estructura o a un rango de estructuras y num1. estructura corresponde, en la operación ABRE, al número de la estructura después de la cual van a ser insertadas las nuevas estructuras y, en el segundo caso, al número de la nueva estructura que va a contener las estructuras ya existentes.

SUSTITUYE y ESCRIBE (opciones 5 y 6)

En *PROGRE* hay dos operaciones para manipulación de texto. La primera permite sustituir una cuerda de caracteres por otra. Su sintaxis es:

<cuerda de texto anterior, cuerda de texto nueva>

donde cuerda de texto anterior corresponde al texto que va a ser substituído por el texto nuevo.

La segunda operación permite llenar los "hoyos" que aparecen en las estructuras. Con esta operación es posible escribir el texto, que no fue proporcionado por el usuario, en el momento de la creación del programa. Su sintaxis es:

<num2. estructura, texto>

donde num2. estructura corresponde al número de la estructura donde va a ser escrito el texto (en este caso no se permiten rangos) y texto es la cuerda de caracteres que va a ser insertada en el lugar correspondiente.

ELIMINA y CONTRAE (opciones 8 y 9)

Estos procedimientos actúan de la misma forma que en la creación de un pro-

grama.

LOCALIZA (opción 10)

Esta opción hace posible localizar una estructura. El usuario únicamente requiere conocer y proporcionar al sistema el número de dicha estructura. Al localizarla, el sistema posiciona el cursor en el lugar correspondiente.

Traduce programa

PROGRE cuenta con dos procedimientos para llevar a cabo la traducción de un programa: 1) el usuario puede solicitar traducir un programa de su forma gráfica a su forma textual indentada y, 2) puede pedir la traducción de un programa que esté escrito en Pascal a su representación gráfica (aún cuando éste no haya sido creado con *PROGRE*). En ambos casos, los programas ya deben existir en memoria y deben cumplir las siguientes restricciones: en el primer caso, debe existir la forma gráfica y, en el segundo, el programa no debe contener proposiciones GOTO ni declaraciones de etiquetas. El sistema, únicamente, solicita al usuario el nombre del programa.

Graba

Al terminar de crear o de editar un programa, es necesario proteger los cambios efectuados. El usuario, al invocar la opción 4 (Graba) del *Menú de archivos*, puede solicitar al sistema que proteja en disco el programa que se encuentra en memoria.

El procedimiento *PROTEGE* se encarga de grabar en disco el programa indicado. Al invocar esta opción, después de crear un programa, el sistema solicita al usuario el nombre que desea darle al programa. Si el programa ya existía, y fue editado, el usuario puede pedir que se proteja con el mismo nombre o con un nombre distinto. En este último caso la versión anterior, con el nombre original, permanece sin cambio. Para llevar a cabo ésto, el procedimiento cuenta con un parámetro, el cual le indica queácciones debe realizar, según sea el caso.

PROGRE cuenta con otro mecanismo de protección, pero éste se encarga de proteger los archivos del sistema. El usuario puede solicitar la protección después de realizar modificaciones a su programa, o al terminar la sesión; sin embargo, si no lo hace, el sistema lleva a cabo esta protección, en forma automática, si alguno de los archivos fue modificado durante la sesión.

CAPITULO 5

CONCLUSIONES

Generalmente, cuando se habla de computación aplicada a la enseñanza se piensa en programas que enseñan matemáticas, geografía, gramática, etcétera, es decir, se utiliza la computación como herramienta pero no como el objetivo mismo del aprendizaje. En *PROGRE* (**PRogramación GRáfica Estructurada**) la computación es el objetivo y la herramienta.

Aprender a programar no es fácil. La experiencia nos ha demostrado que, cuando una persona comienza a aprender a programar donde encuentra mayor dificultad es en el diseño de los algoritmos (elección de estructuras, flujo de control, modularización, interconexión entre módulos, etcétera) y no tanto en la sintaxis específica del lenguaje de programación. Sin embargo, el tener que ocuparse de ambas cosas a la vez hace más tardía y difícil la tarea del aprendizaje. Si el programador tuviese que preocuparse sólo de una de estas dos partes en un principio y posponer la otra para cuando dominase la primera, seguramente aprendería más rápido y con menos dificultad. En *PROGRE* el programador aprende primero a diseñar algoritmos estructurados y posteriormente se ocupa de aprender los detalles sintácticos del lenguaje. Para ayudar, a quien aprende a programar, en esta labor, se diseñó e implementó una herramienta, con la cual el nuevo programador desarrolla su programa en forma gráfica (diagrama de flujo estructurado) con la peculiaridad de que, la herramienta lo dirige en la sintaxis del lenguaje, es decir, sólo le permite introducir estructuras donde el lenguaje así lo indica, le pide la condición en una estructura condicional en el lugar preciso, le indica cuando hay bifurcación de control, etcétera. Los detalles sintácticos tales como signos de puntuación [; , . :] así como ciertas cláusulas [begin, end, until, do, etcétera] son responsabilidad de *PROGRE*. Una vez que el programador ha terminado de crear el programa obtiene la versión correspondiente en forma textual indentada para mostrar claramente la modularidad y la estructura del programa.

Por otro lado, la utilidad que presenta la operación inversa, es decir, traducir el programa textual a su representación gráfica, es, principalmente, proporcionar el diagrama de flujo estructurado actualizado del programa, sin mayor esfuerzo por parte del programador. Este diagrama, siendo parte de la documentación, puede ser muy útil para el mantenimiento del programa, además de proporcionar una historia gráfica del desarrollo del mismo, es decir, se puede observar el diseño estático y el dinámico

(el desarrollo) del programa.

Con *PROGRE* el programador comienza a aprender a programar fijando su atención en la algorítmica y pospone el aprendizaje de la sintaxis específica del lenguaje de programación, ya que al utilizar *PROGRE*, el programador, novato o experto, puede tener la confianza en que no existe la posibilidad de cometer un error de sintaxis al olvidar poner un signo de puntuación o una cierta claúsula (v.g. ELSE, UNTIL, DO, etcétera), porque el sistema se encarga de introducir, autoáticamente, esos elementos sintácticos por él.

Con *PROGRE*, el programador tan sólo puede crear programas estructurados ya que la representación gráfica que se utiliza es un lenguaje gráfico estructurado. GAL es un lenguaje de programación abstracto gráfico que representa las estructuras del lenguaje en vez de cadenas de caracteres y donde el programa es una jerarquía de jerarquías.

PROGRE no hace un análisis sintáctico del texto proporcionado por el usuario, como parte de las estructuras, por lo que puede utilizarse tanto lenguaje natural como seudocódigo, o adaptarse a otros lenguajes estructurados de programación (tipo Pascal). Gracias a esta facilidad, es posible desarrollar el programa desde el diseño y durante todas las fases del ciclo de vida de su desarrollo, en una forma interactiva, facilitándose y minimizándose el trabajo.

Aunque esta herramienta está pensada para ayudar a aprender a programar, también puede ser de gran utilidad para los programadores expertos, de varias maneras: ayudándoles en la inserción de ciertas cláusulas y signos de puntuación; proporcionándoles el diagrama de flujo actualizado; creándo la versión textual (programa fuente) del programa en forma indentada y, finalmente, mostrándoles gráficamente el desarrollo estructural del mismo.

PROGRE puede integrarse en un gran entorno, así como transportarse a otras computadoras, porque es autosuficiente, no necesita información externa, tan sólo requiere de una microcomputadora personal (PC) con disco duro, un ratón (*mouse*), un compilador de Pascal y dispositivo de despliegue (pantalla) con opción de graficación.

Aunque *PROGRE* ya cumple con su objetivo principal, tiene posibilidades de extenderse y mejorarse.

Uno de los principales problemas para llevar a cabo la implementación de *PROGRE* son los límites del tamaño de la pantalla que, en la mayoría de los casos, no permite desplegar el programa en su totalidad. Este problema podría solucionarse con la implementación de pantallas y ventanas virtuales que permitiesen observar una pantalla virtual que contuviese al programa completo. Una vez que la ventana virtual fuese abierta y posicionada en la pantalla física, ésta no se movería. Para ver diferentes porciones de la pantalla virtual, la ventana virtual sería reposicionada dentro de la pantalla. La ventana permanecería siempre en el mismo lugar en la pantalla física y la pantalla virtual se deslizaría dentro de la ventana para permitir ver al usuario diferentes secciones del programa.

Hasta el momento, *PROGRE* sólo puede utilizarse para desarrollar programas en lenguaje PASCAL; sin embargo, a futuro, podría extenderse a otros lenguajes del tipo de Pascal, es decir, lenguajes estructurados. Estas modificaciones al programa no son muy complejas ya que, únicamente, se requiere especificar las diferencias de la sintaxis del nuevo lenguaje con Pascal, de tal forma que se pueda generar el texto con la sintaxis apropiada, al traducir el programa de su forma gráfica (GAL) a su forma textual. Otra modificación adicional sería añadir un comando para indicar el lenguaje de programación que va a utilizarse.

CAPITULO 6

Referencias

[**Albizuri 87**] Albizuri, B., "La Programación Estructurada y el Diagrama de Flujo Nassi-Shneiderman Charts", *Comunicaciones Técnicas del IIMAS-UNAM*, Serie Naranja (Investigaciones), núm. 493, 1987.

[**Bauer 75**] Bauer, F. L., *Software Engineering*, Software Engineering As An Advanced Course, Springer 1975.

[**Boehm 74**] Boehm, B. W., J. R. Brown, H. Kasper, M. Lipow, G. J. MacLeod y M. J. Merrit, "Characteristics of Software Quality", *TRW Software Series Report TRW -SS-73-09*, Redondo Beach, California, 1973.

[**Boehm 79**] Boehm, B. W., *Software Engineering*, Classics in Software Engineering, Yourdon Press, 1979.

[**Bohm 66**] Bohm, C. y G. Jacopini, "Flow Diagrams, Turing Machines and Languages with only two Formation Rules", *Communications of the ACM*, vol. 9, núm. 5, 1966, págs. 366-371.

[**Borland 87**] Borland International, *Turbo Pascal: Owner Handbook*, versión 4.0, Scootts Valley, California, 1987.

[**Brown 77**] Brown P. J., *Software Portability*, Cambridge University Press, 1975.

[**Budde 84**] Budde, R., et. al., *Approaches to Prototyping*, Springer Verlag, 1984.

[**Butler 89**] Butler C. J., "Turbo Pascal Windowing System", *BYTE*, febrero, 1989.

[**Coleman 78**] Coleman, D., *A Structured Approach to Data*, The Macmillan Press LTD., 1978.

[**Coleman 79**] Coleman, D., "A Structured Programming Approach to Data", *An Overview of Program Desig*, 1979.

[**Constantine 74**] Constantine, L. L., W. P. Stevens y G. J. Myers, "Structured Design", *IBM Systems Journal*, mayo, 1974.

[**Cheatman 81**] Cheatman, T. E., Glenn H. Holloway y Judy A. Townley, *Program Refinement by Transformation*, IEEE, CH1627-9, 1981.

[**Dahl 72**] Dahl, O. J., E. W. Dijkstra y C. A. Hoare, *Structured Programming*, Academic Press, Londres y Nueva York, 1972.

[**Dennis 73**] Dennis J. B., "Modularity", *Lecture Notes in Economics and Mathematical Systems*, núm. 18, Springer, 1973.

[**Dennis 75**] Dennis J. B., "The Design and Construction of Software Systems", *Software Engineering As An Advanced Course*, Springer, 1975.

[**Dijkstra 68**] Dijkstra, E. W., "The Structure of the THE-Multiprogramming System", *Communications of the ACM*, mayo, 1968, págs. 341-346.

[**Dijkstra 68b**] Dijkstra, E. W., "GOTO Statements Considered Harmful", *Communications of the ACM*, vol. 11, núm. 3, 1968, págs. 147-148.

[**Dijkstra 69**] Dijkstra, E. W., "Structured Programming", *Software Engineering Techniques*, NATO Scientific Affairs Division, Brussels 39, Bélgica, 1968, págs. 84-88. (Report on a Conference, Roma, 1969.)

[**Dijkstra 69b**] Dijkstra, E. W., *Notes on Structured Programming*, EWD 249, Technical University, Eindhoven, Netherlands, 1969.

[**Dijkstra 72b**] Dijkstra, E. W., *Notes on Structured Programming*, Academic Press, 1972.

[**Donzeau-Gouge 80**] Donzeau-Gouge, V., et al., "Programming Environments based on Structured Editors: The Mentor Experience", *INRIA Research Report*, núm. 26, Rocquencourt Francia, 1980.

[**Fairley 85**] Fairley, R. E., *Software Engineering Concepts*, McGraw Hill Inc., 1985.

[**Fisher 80**] Fisher, D. A., *Requirements for Ada Programming Support Environments "Stoneman"*, US Departament of Defense, 1980.

[**Floyd 67**] Floyd, R. W., "Assigning Meanings to Programs", in *Mathematical Aspects of Computer Science*, Ed. J. T. Schwartz, núm. 19, Nueva York, 1967, págs. 19-32.

[**Frei 78**] Frei, H. P., et al., "Graphics-Based Programming Support Systems", *Computer Graphics*, ACM Siggraph, vol. 12, núm. 3, 1978.

[**Gerthart 76**] Gerthart, S., y L. Yelowitz, "Observations of Fallibility in Applications of Modern Programming Methodologies", *IEEE Trans. on Software Engineering*, SE-2, núm. 3, 1976, págs. 175-207.

[**Goos 73**] Goos, G., "Hierarchies" in M. Beckman et al., Ed. *Advanced Course in Software Engineering*, Berlin, Springer-Verlag, 1973, págs. 29-46.

[**Halsteadt 72**] Halsteadt, M. H., "Natural Laws Controlling Algorithm Structure", *Sigplan Notices*, vol. 7, núm. 2, 1972.

[**Hoare 69**] Hoare, C. A. R., "An Axiomatic Basis for Computer Programming", *Communications of the ACM*, vol. 12, núm. 10, 1969, págs. 576-580, 583.

[**IBM 75**] IBM: *HIPO-A Design Aid and Documentation Technique*, GC 20-1851-11, White Plains, Nueva York, 1975.

[**IEEE 83**] *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Standard 729, 1983.

[**Jacobi 82**] Jacobi, C., *The Debugger. In: Lilith Handbook-A Guide for Lilith users and Programmers*, Institut fur Informatik, ETH Zurich, 1982.

[**Jones 79**] Jones, C., "A Survey of Programming Design and Specification Techniques", *Proceedings Specification of Reliable Software*, IEEE, Nueva York, 1979.

[**Knuth 68**] Knuth, D. E., *The Art of Computer Programming: Fundamental Algorithms*, vol. 1, Addison Wesley Publishing Company Inc. 1968.

[**Knuth 74**] Knuth, D. E., "Structured Programming with GOTO Statements", *Computing Survey*, vol. 6, núm. 4, 1974, págs. 261-301.

[**Kopetz 79**] Kopetz, H., *Software Reliability*, The MacMillan Press LTD, Reino Unido, 1979.

[**Lientz 80**] Lientz B.P., and Swanson, E.B., "*Software Maintenance Management*, Addison Wesley, 1980.

[**Liskov 72**] Liskov, B. H., "A Design Methodology for Reliable Software Systems", *Proceedings of the 1972 Fall Joint Computer Conference*, Montvale, N.J., AFIPS Press, 1972, págs. 191-199.

[**Liskov 72b**] Liskov, B. H., "The Design of the Venus Operating System", *Communications of the ACM*, vol. 15, núm. 3, 1972, págs. 144-149.

[**Liskov 75**] Liskov, B. y S. Zilles, "Specification Techniques for Data Abstractions", *IEEE Transactions on Software Engineering*, SE-1, marzo 1975, págs. 7-19.

[**Mills 72**] Mills, H. D., *Mathematical Foundations for Structured Programming*, IBM Technical Report FSC-72-6012, 1972.

[**Myers 75**] Myers, G. J., *Reliable Software Through Composite Design*, Nueva York, Petrocelli/Charter, 1975.

[**Myers 76**] Myers, G. J., *Software Reliability: Principles and Practices*, John Wiley and Sons, 1976.

[**Nassi 73**] Nassi, I. y B. Shneiderman, "Flowchart Technique for Structured Programming", *SIGPLAN Notices of the ACM*, vol. 8, núm. 8, agosto 1973.

[**NAU 76**] Naur, P., et al., *Software Engineering: Concepts and Techniques*, Petrocelli/Charter, Nueva York, 1976.

[**Oliver 75**] Oliver P., "Cobol´ 74 - Contributions to Structured Programming", *AFIPS Conference Proceedings*, vol. 44, 1975.

[**Parnas 71**] Parnas, D. L., "Information Distribution Aspects of Design Methodology", *Proceedings of the IFIP Congress*, 1971, págs. 339-344.

[**Parnas 72**] Parnas, D. L., "On the Criteria to be Used in Descompositing Systems in Moduls", *Communications of the ACM*, vol. 15, núm. 12, 1972.

[**Parnas 74**] Parnas, D. L., "Software Engineering or Methods for the Multi-Person Construction of Multi-Version Programs", *Lecture Notes in Computer Science: Programming Methodology*, Springer, 1974.

[**Parnas 75**] Parnas, D. L., "The Influence of Software Structure on Reliability", *International Conference on Reliable Software*, Los Angeles, Calif., 1975, págs. 358-362 (ACM SIGPLAN Notices 10, núm. 6, junio, 1975).

[**Pomberger 82**] Pomberger, G., "Ein Werkzeug zur interaktiven Programmentwicklung und-dokumentation", *Elektronische Rechenanlagen*, Oldenbourg, 1982.

[**Pomberger 84**] Pomberger, G., *Software Engineering and Modula-2*, Prentice-Hall International, 1984.

[**Pressman 82**] Pressman, R. S., *Software Engineering: A Practicioner Approach*, McGraw Hill International, 1982,

[**Rault 73**] Rault, J. C., *Extension of Hardware Fault Detection Models to the Verification of Software in Program Test Methods*, ed. W.C. Hetzel, Prentice Hall, Englewood Cliffs Nueva Jersey, 1973, págs. 255-262.

[**Riddle 78**] Riddle, W.E. y J. C. Wileden, "Languages for Representing Software Specifications and Designs", *ACM SIGSOFT*, Software Engineering Notes, octubre 1978.

[**Schoman 77**] Schoman, K. y D. T. Ross, "Structured Analysis for Requirements Definition", *IEEE Transactions on Software Engineering*, SE-3, 1977.

[**Sloughter 74**] Sloughter, J. B., "Understanding the Software Problem", *Proceedings of the IFIP Congress*, 1971, págs. 249-266.

[**Tanenbaum 78**] Tanenbaum. A. S. et al., "Guidelines for Software Portability", *Software Practice and Experience*, vol. 8, 1978.

[**Teichrow 77**] Teichrow, D. y E. A. Hersley, "PSL/PSA: A Computer Aided Technique for Structured Documentation and Analysis of Information Processing Systems", *IEEE Transactions on Software Engineering*, SE-3, 1977.

[**Teitelbaum 81**] Teitelbaum, T. y T. Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment", *Communication Ass. Computer*, vol. 24., marzo 1981.

[**Truol 81**] Truol, K., "DIPROTOR: ein Softwarewerkzeug zur Erstellung von Diagrammen und Programmrahmen fur die Datenstruktur orientierte Methode des Programmentwurfs", *Informatik Fachberichte*, vol. 43, Springer, 1981.

[**Wasserman 81**] Wasserman A. I., "Automated Development A Environments", *Computer*, abril 1981.

[**Willis 81**] Willis, R. R., "AIDES: Computer Aided Design of Software Systems", *Software Engineering Environments*, North-Holland, 1981.

[**Wirth 71**] Wirth, N., "Program Development by Stepwise Refinement", *Communications of the ACM*, vol. 14, núm. 4, abril, 1971.

[**Wirth 73**] Wirth, N., *Systematic Programming*, Prentice Hall, 1973.

[**Wirth 74**] Wirth, N., "On the Composition of Well Structured Programs", *Computing Surveys*, vol. 6, núm. 4, 1974, págs. 247-259.

[**Yourdon 74**] Yourdon, E., "A Brief Look at Structured Programming and Top-Down Design", *Modern Data*, Junio, 1974, págs. 30-35.

[**Yourdon 75**] Yourdon, E., *Techniques of Program Structure and Design*, Prentice Hall, Inc., Englewood Cliffs, Nueva Jersey, 1975.

[**Yourdon 79**] Yourdon, E. y L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and System Design*, Prentice Hall Inc. Englewood Cliffs, Nueva Jersey, 1979.

[**Zelkowitz 79**] Zelkowitz, M. et al, *Principles of Software Engineering and Design*, Prentice-Hall International, 1979.

[**Zilles 75**] Zilles S. N., "Modularization around a Suitable Abstraction", *AFIPS, Conference Proceedings*, vol. 44, 1975.

PROGRE: UNA HERRAMIENTA GRAFICA PARA APRENDER A PROGRAMAR ESTRUCTURADAMENTE

H. R. Albizuri Romero
M. Ardisson Pérez
IIMAS - UNAM
México, D. F.

Generalmente, cuando se habla de computación aplicada a la enseñanza se piensa en programas que enseñen matemáticas, geografía, gramática, etc., o sea, se utiliza la computación como herramienta pero no como el objetivo mismo del aprendizaje. En PROGRE la computación es el objetivo y la herramienta.

PROGRE (PROgramación GRáfica Estructurada) es el desarrollo práctico de una herramienta de software consistente de un traductor de programas escritos en un lenguaje de programación estructurado (Pascal) a su representación gráfica en un lenguaje de programación gráfico abstracto (GAL) y viceversa, es decir, la traducción de un programa desarrollado utilizando un lenguaje de programación gráfico abstracto a su correspondiente versión en un lenguaje de programación estructurado.

Introducción.

Aprender a programar no es fácil. La experiencia nos ha demostrado que cuando una persona empieza a aprender a programar donde encuentra mayor dificultad es en el diseño de los algoritmos (selección de estructuras, flujo de control, modularización, interconexión entre módulos, etc.) y no tanto en la sintáxis específica del lenguaje de programación. Sin embargo, el tener que ocuparse de ambas cosas a la vez hace más tardía y difícil la tarea de aprendizaje. Si el programador tuviese que preocuparse sólo de una de estas dos partes en un principio y posponer la otra para cuando dominase la primera, seguramente aprendería más rápido y con menos dificultad. En PROGRE el programador aprende primero a diseñar algoritmos estructurados y posteriormente se ocupa de aprender los detalles sintácticos del lenguaje. Para ayudar a quien aprende a programar en esta labor diseñamos e implementamos una herramienta con la cual el nuevo programador desarrolla su programa en forma gráfica (diagrama de flujo estructurado) con la peculiaridad de que la herramienta lo dirige en la sintáxis del lenguaje, es decir, sólo le permite introducir estructuras donde el lenguaje así lo indica, le pide la condición en una estructura condicional en el lugar preciso, le indica cuándo hay bifurcación de control, etc. Los detalles sintácticos tales como signos de puntuación (; : , .) así como ciertas palabras (begin, end, until, do, etc.) son responsabilidad de PROGRE. Una vez que el programador ha terminado de crear el programa obtiene la versión correspondiente en forma textual indentada para mostrar claramente la modularidad y estructura del programa.

Por otro lado, la utilidad que presenta la operación inversa, es decir, traducir el programa textual a su representación gráfica, es, principalmente, proporcionar el diagrama de flujo estructurado del programa sin mayor esfuerzo por parte del programador. Este diagrama, siendo parte de la documentación, puede ser muy útil para el mantenimiento del programa, además de proporcionar una historia gráfica del desarrollo del programa.

El programador por medio de un menú elige la instrucción que desea incorporar en su programa, en ese momento PROGRE dibuja en la pantalla la estructura que representa la instrucción solicitada y dirige la sintáxis de la misma. Con PROGRE el programador tan sólo puede crear programas estructurados ya que la representación gráfica que se utiliza es un lenguaje gráfico estructurado.

GAL- Graphical Abstract Language

GAL es un lenguaje abstracto gráfico para desarrollar programas estructurados. GAL representa las estructuras del lenguaje en vez de cadenas de caracteres. El lenguaje abstracto gráfico intenta acercarse a los lenguajes sometidos a la programación y diseño estructurado.

GAL está compuesto por nueve símbolos cuyos cuerpos, excepto el de proceso, son estructuras de complejidad arbitraria. No hay flechas para conectar los símbolos sino que se colocan adyacentemente con lo que se evitan las transferencias de control arbitrarias.

En GAL se adoptaron cuatro símbolos de Nassi-Shneiderman charts (Nassi 73) (fig. 1), se diseñaron cuatro más (fig. 2) y se modificó la forma del símbolo del case (fig. 3).

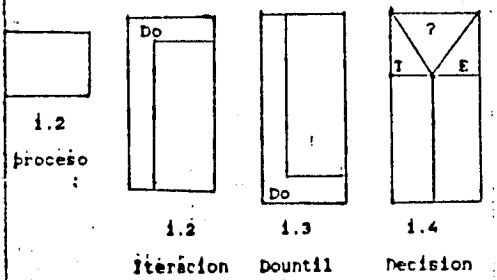


Figura 1. Símbolos de N-S charts adoptados por GAL.

El símbolo de proceso se usa para representar asignaciones, instrucciones de entrada/salida y llamadas a subrutinas (fig. 1.1).

Hay dos tipos de instrucciones de loop, aquéllas que tienen la condición para terminar el loop arriba y aquéllas que la tienen abajo. Las instrucciones de loop que tienen la condición arriba se dividen en dos clases: las instrucciones para las cuales el fin del loop depende de una condición booleana; como el WHILE-DO de Pascal (fig. 1.2), y aquéllas en las cuales la condición es una variable indexada, como el FOR-TO de Pascal (fig. 2.2). Las instrucciones con la condición del loop abajo, como el REPEAT-UNTIL (fig. 1.3).

El símbolo de decisión (fig. 1.4) se utiliza para representar instrucciones donde la acción a llevarse a cabo depende de una expresión booleana, como el IF-THEN-ELSE de Pascal.

Un grupo de instrucciones relacionadas en un módulo se inclujen en un marco (fig. 2.3). Este símbolo le permite al programador reconocer fácilmente los módulos y la estructura modular del programa; una aplicación de esta estructura es el PROCEDURE o FUNCTION de Pascal.

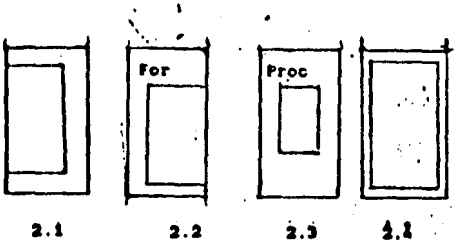


Figura 2. Símbolos de GAL diseñados.

El símbolo de declaración (fig. 2.4) le permite al programador establecer la definición de los identificadores locales de cada módulo y determinar fácilmente su alcance.

Con el símbolo de la figura 3 se obtiene la generalización del símbolo de decisión permitiendo tener n valores para la condición, tal como sucede con el CASE de Pascal.

Finalmente la instrucción compuesta, la cual es un grupo de instrucciones, se encuentra claramente representada por el símbolo compuesto (fig. 2.1), por ejemplo el WITH-DO de Pascal.

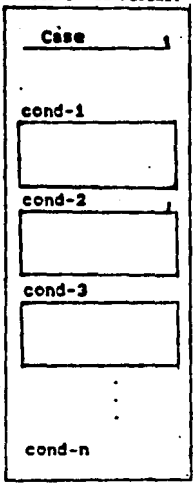


Figura 3. Símbolo CASE.

Un ejemplo usando PROGRE.

PROGRE es útil a través de los estados de especificación, diseño, implementación y mantenimiento del ciclo de vida del desarrollo de un programa proporcionando soluciones legibles, entendibles y modificables.

En PROGRE el usuario puede desarrollar su programa interactivamente en forma gráfica ó a partir del programa textual en Pascal obtener la representación gráfica o diagrama de flujo del mismo. Cuando existe ya la representación gráfica del programa, éste puede ser modificado (editado). La edición de un programa se puede realizar en la misma sesión en que se creó o en otra, siempre y cuando se encuentre presente en memoria. Por último, el programa se puede grabar en disco para futuros usos. Estas cuatro operaciones con los programas constituyen el primer nivel de menús. De tal forma que cuando PROGRE es invocado, el usuario selecciona del primer menú o menú de archivos (fig. 4) la operación a realizarse.

MENU DE ARCHIVOS

1. Crea programa
2. Traduce programa
3. Edita
4. Graba

Seleccione una opcion

Figura 4. Menu de archivos.

La selección de opciones se realiza con un "mouse".

Para mostrar el uso de PROGRE desarrollaremos un ejemplo en el que el programador va a desarrollar su programa en forma gráfica (opción 1. Crea programa). Ahora el usuario podrá seleccionar una de las trece opciones del menú de diseño (fig. 5).

MENU DE DISEÑO

1. Proceso
2. IF
3. WHILE
4. FOR
5. WITH
6. CASE
7. REPEAT
8. Declaraciones
9. PROCEDURE
10. FUNCTION
11. Amplifica
12. Contrae
13. Termina

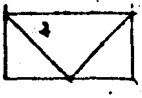
Seleccione una opcion

Figura 5. Menu de diseño.

Las diez primeras opciones corresponden a las estructuras de control proporcionadas por Pascal. El usuario selecciona la estructura que desea y PROGRE lo irá guiando en su sintáxis. Supóngase que se selecciona la opción 2, IF, entonces aparecerá el rectángulo superior de la gráfica y el cursor se posicionará en el triángulo correspondiente a la condición para que el programador la escriba (fig. 6).

MENU DE DISEÑO

1. Proceso
2. IF
3. WHILE
4. FOR
5. WITH
6. CASE
7. REPEAT
8. Declaraciones
9. PROCEDURE
10. FUNCTION
11. Amplifica
12. Contrae
13. Termina



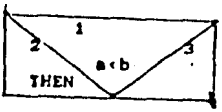
Seleccione una opcion

Figura 6. Opcion 2 del menu de diseño.

En este momento el programador tiene dos opciones: escribir la condición terminándola con "return" o no escribir nada y tan sólo presionar "return", o sea, dejar un "hoyo" para rellenarlo en una futura edición. Posteriormente PROGRE le informará al usuario que continúe el bloque del THEN (fig. 7)

MENU DE DISEÑO

1. Proceso
2. IF
3. WHILE
4. FOR
5. WITH
6. CASE
7. REPEAT
8. Declaraciones
9. PROCEDURE
10. FUNCTION
11. Amplifica
12. Contrae
13. Termina



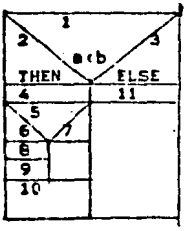
Seleccione una opcion

Figura 7. Desarrollo del bloque del THEN.

También en este caso se pueden dejar hoyos, es decir, se puede dejar sin desarrollar el bloque, bien porque es nulo o porque se desarrollará posteriormente. El usuario selecciona las estructuras que contiene el THEN, cada vez que se selecciona termina (opción 13) se cierra la última estructura abierta. El bloque del THEN también se cierra con esta opción. Una vez concluido el THEN, PROGRE dirige al usuario para que le proporcione el bloque del ELSE (fig. 8).

MENU DE DISEÑO

1. Proceso
2. IF
3. WHILE
4. FOR
5. WITH
6. CASE
7. REPEAT
8. Declaraciones
9. PROCEDURE
10. FUNCTION
11. Amplifica
12. Contrae
13. Termina



Seleccione una opcion

Figura 8. Desarrollo del bloque del ELSE.

El desarrollo del bloque del ELSE sigue la misma filosofía que en el del THEN.

En cualquier estructura se pueden dejar "hoyos" de texto o de cuerpo.

Cuando el usuario termina el bloque del ELSE, automáticamente queda terminado el IF y la siguiente estructura que seleccione el usuario quedará al mismo nivel de definición que el IF.

Para PROGRE un programa es una estructura jerárquica compuesta de estructuras, de tal forma que el desarrollo del programa termina también con la opción 13, pero siempre y cuando se encuentre a nivel 1 de definición de estructuras de control.

Obsérvese que en el menú de diseño hay dos opciones de las cuales no se ha hecho mención, 11. amplifica y 12. contrae. Estas opciones se utilizan para amplificar o contraer las estructuras, respectivamente. Supóngase que el IF que diseñó el usuario es el que se presenta en la figura 8.

Nótese que cada estructura está numerada, esto es para facilitar la referencia a ellas, la cual se puede hacer por su número o con el uso del mouse. En el THEN del IF de la figura 8 hay definido otro IF (estructura 5) que por las limitaciones del tamaño de la pantalla puede resultar difícil ver con claridad su contenido, entonces se puede solicitar una ampliación de la estructura. El usuario solicita la opcion 11 y posteriormente selecciona la estructura, por su número o con el mouse, entonces se borra la pantalla y tan sólo aparece la estructura seleccionada amplificado el dibujo, como si estuviese a nivel 1 de definición (fig. 9).

```
MENU DE DISENO

 1. Proceso
 2. IF
 3. WHILE
 4. FOR
 5. WITH
 6. CASE
 7. REPEAT
 8. Declaraciones
 9. PROCEDURE
10. FUNCTION
11. Amplifica
12. Contrae
13. Termina

Seleccione una opcion
```

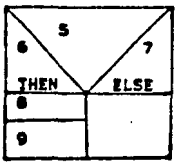


Figura 9. Amplificacion de la estructura 5.

Para regresar a la figura original se selecciona la opción 12 (contrae).

El programador al desarrollar el IF de la figura 8 no tuvo que preocuparse en conocer la sintáxis de esta estructura, por ejemplo, que tenía que poner BEGIN después de la cláusula THEN ya que este bloque está compuesto de más de una estructura y que el BEGIN se tiene que cerrar con un END sin el signo de puntuación ; puesto que continúa la cláusula ELSE después de la cual no se requiere de las cláusulas BEGIN-END ya que este bloque sólo contiene una estructura.

Cuando el programador ha terminado de desarrollar su programa, o una sección de él, y selecciona la opción 13, PROGRE le muestra el menú de archivos, donde se puede optar por traducir el programa (opción 2) a su versión textual en Pascal indentada siguiendo la sintáxis de este lenguaje de programación. El diagrama de la figura 8 después de esta traducción quedaría como se muestra en la siguiente figura.

```
IF a<b
THEN
  BEGIN
    estructura-4;
    IF cond
    THEN
      BEGIN
        estructura-8;
        estructura-9
      END;
    estructura-10
  END
ELSE
  estructura-11;
```

Figura 10. Version en Pascal del programa de la figura 8.

Después o antes de haber seleccionado esta opción el programador puede regresar a su programa para editarlo (opción 3). Las operaciones de creación y edición de programas siempre se realizan a través de su versión gráfica. Para editar el programa, el programador cuenta con un grupo de once operaciones proporcionadas en el menú de edición (fig. 11).

Menu de edicion .

1. Mueve
2. Copia
3. Elimina
4. Inserta
5. Sustituye texto
6. Escribe texto
7. Habre estructura
8. Amplifica
9. Contrae
10. Localiza estructura
11. Termina

Figura 11. Menú de edicion.

La opción 1 mueve de lugar una estructura, eliminándola de su lugar original. La opción 2 crea una copia de la estructura en el lugar indicado por el usuario. La tercera opción anula la estructura o conjunto de estructuras. Con la opción 4 se inserta la estructura existente en una nueva que se define cuando se selecciona esta opción. Las opciones 5 y 6 permiten modificar el texto de las estructuras.

Las operaciones de amplificación y contracción producen los mismos resultados que en la creación de programas. La opción 10 permite localizar estructuras por su número y así seleccionarlas para realizar cualquiera de las operaciones del presente menú. Por último, con la opción 11 se termina la operación seleccionada del menú así como también la edición del programa y entonces regresar al menú de archivos.

Conclusiones.

Con PROGRE el programador comienza a aprender a programar fijando su atención en la algorítmica y pospone el aprendizaje de la sintáxis específica del lenguaje de programación.

PROGRE utiliza un lenguaje gráfico de programación estructurada por lo que el programador aprende a programar estructuradamente.

La herramienta de programación que se presenta en este artículo no hace un análisis sintáctico del texto escrito por el usuario como parte de las estructuras por lo que se puede utilizar desde el diseño del programa en lenguaje natural o adaptarse a otros lenguajes de programación del tipo de Pascal.

La documentación de un programa es un elemento importante para el futuro mantenimiento del mismo. Con PROGRE se obtiene el diagrama de flujo actualizado que corresponde exactamente al programa fuente sin esfuerzo por parte del programador.

Aunque esta herramienta está pensada para ayudar a aprender a programar, también puede ser de gran utilidad para programadores expertos ayudándoles en la inserción de ciertas cláusulas, signos de puntuación, proporcionándoles el diagrama de flujo actualizado, indentando su programa y mostrándoles gráficamente el desarrollo estructural de su programa.

Referencias.

(Nassi 73)
Nassi, I. y B. Shneiderman, "Flowchart technique for structured programming", SIGPLAN Notices of the ACM, vol. 8, núm. 8, agosto, 1973.

PROGRAMACION GRAFICA ESTRUCTURADA CON AYUDA DE UNA HERRAMIENTA DE SOFTWARE

Mónica Ardisson Pérez
y
Miren Begoña Albizuri Romero

Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas

Universidad Nacional Autónoma de México

RESUMEN

PROGRE (PROgramación GRáfica Estructurada) tiene como objetivo desarrollar programas estructuradamente en forma interactiva. Consiste de un traductor de programas escritos en un lenguaje de programación estructurado (**Pascal**) a su representación gráfica en un lenguaje de programación gráfico abstracto (**GAL**) y viceversa, es decir, la traducción de un programa desarrollado utilizando dicho lenguaje de programación gráfico abstracto a su correspondiente versión en un lenguaje de programación estructurado.

Esta herramienta de Software es útil a través de los estados de especificación, diseño, implementación y mantenimiento del ciclo de vida del desarrollo de un programa, proporcionando soluciones legibles, entendibles y modificables.

INTRODUCCION

Generalmente, cuando se habla de computación aplicada a la enseñanza se piensa en programas que enseñen matemáticas, geografía, gramática, etc., o sea, se utiliza la computación como herramienta pero no como el objetivo mismo del aprendizaje. En **PROGRE** la computación es el objetivo y la herramienta.

En **PROGRE** el programador aprende primero a diseñar algoritmos estructurados y posteriormente se ocupa de aprender los detalles sintácticos del lenguaje. Para ayudar a quien aprende a programar se diseñó e implantó una herramienta con la cual el nuevo programador desarrolla su programa en forma gráfica (diagrama de flujo estructurado) con la peculiaridad de que la herramienta lo dirige en la sintaxis del lenguaje, es decir, sólo le permite introducir estructuras donde el lenguaje así lo

indica, le pide la condición en una estructura condicional en el lugar preciso, le indica cuando hay bifurcación de control, etc. Los detalles sintácticos tales como signos de puntuación [; , . :] así como ciertas claúsulas [begin, end, until, do, etc.] son responsabilidad de **PROGRE**. Una vez que el programador ha terminado de crear el programa obtiene la versión correspondiente en forma textual indentada para mostrar claramente la modularidad y la estructura del programa.

La documentación de un programa es necesaria y fundamental para el futuro mantenimiento del mismo. El diagrama de flujo estructurado de tanto del diseño como de la codificación del programa puede ser de gran utilidad cuando se requiere cualquier tipo de modificaciones. **PROGRE** le proporciona al usuario, en forma automática, dichos diagramas sin mayor esfuerzo por su parte, basta solicitárselo.

GAL- Graphical Abstract Language

GAL[Albizuri 84] es un lenguaje abstracto gráfico para desarrollar programas estructurados. **GAL** representa las estructuras del lenguaje en vez de cadenas de caracteres. El lenguaje abstracto gráfico intenta acercarse a los lenguajes sometidos a la programación y diseño estructurado.

GAL está compuesto por nueve símbolos cuyos cuerpos, excepto el de proceso, son estructuras de complejidad arbitraria. No hay flechas para conectar los símbolos sino que se colocan adyacentemente con lo que se evitan las transferencias de control arbitrarias.

En **GAL** se adoptaron cuatro símbolos de Nassi- Shneiderman charts [**Nassi 73**] (fig. 1), se diseñaron cuatro más (fig. 2) y se modificó la forma del símbolo del case (fig. 3).

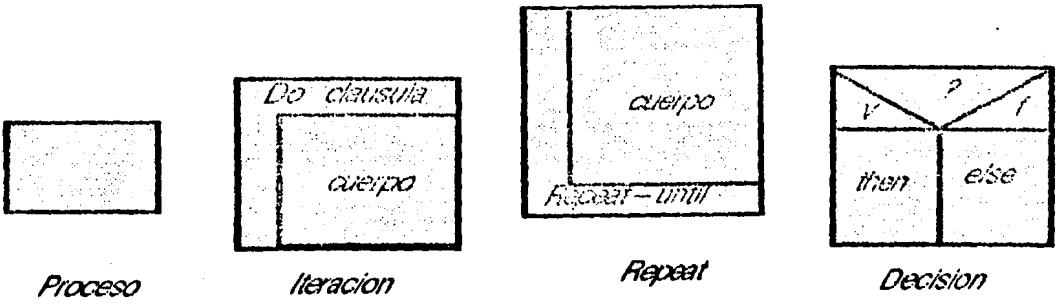


Figura 1. Símbolos de N-S charts adoptados por GAL.

El símbolo de proceso se usa para representar asignaciones, instrucciones de entrada/salida y llamadas a subrutinas (fig. 1.1).

Hay dos tipos de instrucciones de loop, aquéllas que tienen la condición para terminar el loop arriba y aquéllas que la tienen abajo. Las instrucciones de loop que tienen la condición arriba se dividen en dos clases: las instrucciones para las cuales el fin del loop depende de una condición booleana, como el **WHILE-DO** de Pascal (fig. 1.2) y aquéllas en las cuales la condición es una variable indexada, como el **FOR-TO** de Pascal (fig. 2.2). Las instrucciones con la condición del loop abajo, como el **REPEAT-UNTIL** (fig. 1.3)

El símbolo de decisión (fig. 1.4) se utiliza para representar instrucciones donde la acción a llevarse a cabo depende de una expresión booleana, como el **IF-THEN-ELSE** de Pascal.

Un grupo de instrucciones relacionadas en un módulo se incluyen en un marco (fig. 2.3). Este símbolo le permite al programador reconocer fácilmente los módulos y la estructura modular del programa; una aplicación de esta estructura es el **PROCEDURE o FUNCTION** de Pascal.

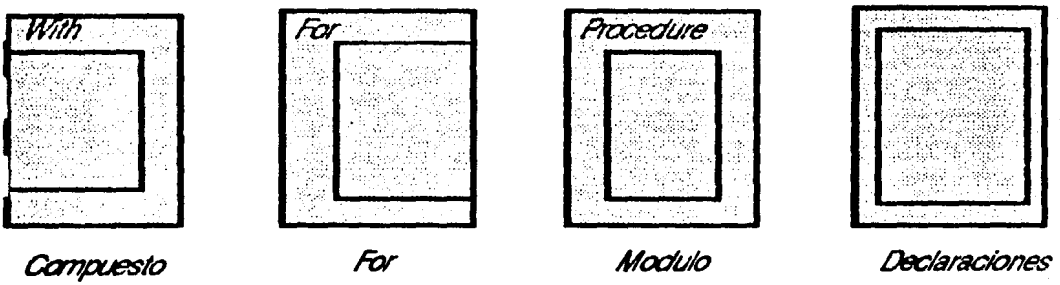


Figura 2. Símbolos de GAL diseñados.

El símbolo de declaración (fig. 2.4) le permite al programador establecer la definición de los identificadores locales de cada módulo y determinar fácilmente su alcance.

Con el símbolo de la figura 3 se obtiene la generalización del símbolo de decisión permitiendo tener n valores para la condición, tal como sucede con el **CASE** de Pascal.

Finalmente, la instrucción compuesta, la cual es un grupo de instrucciones, se encuentra claramente representada por el símbolo compuesto (fig. 2.1), por ejemplo el **WITH-DO** de Pascal.

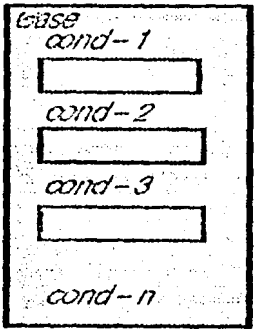


Figura 3. Símbolo CASE.

Funciones de PROGRE.

La interface entre PROGRE y el usuario se realiza a través de menús, seleccionando la opción deseada por tres medios distintos: con un "mouse", utilizando las flechas para posicionarse y oprimiendo la tecla de "return", u oprimiendo el número que aparece a la izquierda de la opción.

PROGRE trabaja principalmente con archivos, de tal forma que el primer menú muestra las cuatro posibles operaciones a realizarse con éstos (figura 4).

MENU DE ARCHIVOS

1. Crea programa
2. Traduce programa
3. Edita
4. Graba

Seleccione una opción

Figura 4. Menú de archivos.

Si el usuario selecciona la opción 1 (Crea Programa o la opción 3 (Edita, entonces PROGRE le proporciona el menú correspondiente (figura 5 o figura 6 respectivamente).

MENU DE DISENO

1. Proceso
2. IF
3. WHILE
4. FOR
5. WITH
6. CASE
7. REPEAT
8. Declaraciones
9. PROCEDURE
10. FUNCTION
11. Amplifica
12. Contrae
13. Termina

Figura 5. Menú de diseño.

MENU DE EDICION

1. Mueve
2. Copia
3. Elimina
4. Inserta
5. Sustituye texto
6. Escribe texto
7. Abre estructura
8. Amplifica
9. Contrae
10. Localiza
11. Termina

Figura 6. Menú de edición.

Si la elección del usuario fue "Crea programa", con el menú de diseño (figura 5) elige la instrucción que desea incorporar en su programa y en ese momento **PROGRE** dibuja en la pantalla la estructura que representa la instrucción solicitada y dirige la sintaxis de la misma.

Las opciones 11 y 12 se utilizan para amplificar o contraer las estructuras, respectivamente.

Si la operación elegida fue la de edición, el programador cuenta con un grupo de once operaciones(figura 6): La opción 1 mueve de lugar una estructura, eliminándola de su lugar original. La opción 2 crea una copia de la estructura en el lugar indicado por el usuario. La tercera opción anula la estructura o conjunto de estructuras. Con la opción 4 se inserta la estructura existente en una nueva que se define cuando se selecciona esta opción. Las opciones 5 y 6 permiten modificar el texto de las estructuras. Las operaciones de amplificación y contracción producen los mismos resultados que en la creación de programas. La opción 10 permite localizar estructuras por su número y así seleccionarlas para realizar cualquiera de las operaciones del presente menú. Por último, con la opción 11 se termina la operación seleccionada del menú así como también la edición del programa y entonces regresar al menú de archivos. Cuando el programador ha terminado de desarrollar su programa, o una sección de él, y selecciona la opción 13, **PROGRE** le muestra el menú de archivos, donde se puede optar por traducir el programa (opción 2) a su versión textual en Pascal indentada siguiendo la sintaxis de este lenguaje de programación.

Por último, la cuarta operación que se puede realizar con archivos, utilizando **PROGRE** es grabar (opción 4. Graba, del menú de archivos) el programa gráfico y/o textual en disco para un posible futuro uso.

CONCLUSIONES.

Con **PROGRE** el programador comienza a aprender a programar fijando su atención en la algorítmica y pospone el aprendizaje de la sintaxis específica del lenguaje de programación.

PROGRE utiliza un lenguaje gráfico de programación estructurada por lo que el programador aprende a programar estructuradamente.

La herramienta de programación que se presenta en este artículo no hace un análisis sintáctico del texto escrito por el usuario como parte de las estructuras por lo que se puede utilizar desde el diseño del programa en lenguaje natural o adaptarse a otros lenguajes de programación del tipo de Pascal.

La documentación de un programa es un elemento importante para el futuro mantenimiento del mismo. Con **PROGRE** se obtiene el diagrama de flujo actualizado que corresponde exactamente al programa fuente sin esfuerzo por parte del programador.

Aunque esta herramienta está pensada para ayudar a aprender a programar, también puede ser de gran utilidad para programadores expertos ayudándoles en la inserción de ciertas claúsulas, signos de puntuación, proporcionándoles el diagrama de flujo actualizado, indentando su programa y mostrándoles gráficamente el desarrollo estructural de su programa.

REFERENCIAS

[Nassi 73]

Nassi, I. y B. Shneiderman, "*Flowchart technique for structured programming*", SIGPLAN Notices of the ACM, vol. 8, núm. 8, agosto, 1973.

[Albizuri 84]

Albizuri Romero M.B., "*A Graphical Abstract Programming Language*", *SIGPLAN Notices of the ACM, vol. 19, núm. 1, enero 1984.*