Universidad de Matanzas "Camilo Cienfuegos"
Facultad de Ingenierías Química – Mecánica

MONOGRAFÍA

PROGRAMANDO PARA AUTOCAD CON VBA

PRIMERA PARTE

Ramón Quiza Sardiñas

Departamento de Ingeniería Mecánica

Noviembre, 2006

Programando para AutoCAD con VBA – Primera Parte.

Ramón Quiza Sardiñas
e-mail: quiza@umcc.cu

Autopista a Varadero, km 3 ½, Matanzas CP 44740, Cuba.
http://www.umcc.cu

TABLA DE CONTENIDO

CAPÍTULO 1
Conceptos Generales

OBJETIVOS

Comparar VBA con otras herramientas de personalización de AutoCAD y con el Visual Basic estándar, considerando sus ventajas y desventajas.

Describir las características de los proyectos y macros de Visual Basic for Applications en AutoCAD.

1.1 – ¿Por qué VBA?

AutoCAD es uno de los programas de CADD más difundidos en todo el mundo, lo cual se debe, entre otras muchas excelentes cualidades, a su arquitectura abierta. Esta característica permite la personalización del entorno y, sobre todo, la creación y adición de subprogramas para realizar tareas específicas.

Mediante la creación de subprogramas personalizados, es posible lograr un entorno más amigable, desarrollando una alta velocidad y eficiencia en el trabajo, gracias a la eliminación y simplificación de las tareas tediosas o repetitivas. Por otro lado, a través de subprogramas adecuados, se puede lograr una alta integración entre los cálculos de ingeniería y su resultado visual: los dibujos y modelos.

Por supuesto, la creación de tales subprogramas representa un trabajo duro, pero los resultados que se obtienen recompensan con creces el tiempo y el esfuerzo empleados y, por tanto, el costo invertido.

Aunque desde de su propio origen AutoCAD proporcionó a sus usuarios diversas herramientas para crear subprogramas personalizados, o su capacidad era limitada, o requerían grandes habilidades de programación. No fue hasta la introducción del lenguaje de programación Visual Basic, que las oportunidades de personalización fueron realmente efectivas para la gran mayoría de los usuarios.

Los primeros intentos de introducir a Visual Basic, como lenguaje de programación alternativo para AutoCAD, fueron llevados a cabo en la versión R12, sin embrago, no fue hasta la versión R14 (en realidad, en la versión R14.01) que AutoCAD ofreció por primera vez soporte completamente funcional para *Visual Basic for Applications* (VBA).

VBA es un lenguaje de programación desarrollado por Microsoft Corporation para elaborar subrutinas que automaticen el trabajo con las aplicaciones que le ofrecen soporte. En los primeros tiempos, VBA fue concebido para Microsoft Office, pero se popularizó rápidamente gracias a sus bondades y a una excelente labor de marketing de

sus creadores. Desde su inclusión en AutoCAD, VBA comenzó a ganar popularidad entre los desarrolladores de aplicaciones para este sistema.

VBA surgió a partir del Visual Basic estándar, de quien heredó la mayoría de sus características. No obstante, VBA tiene algunas diferencias con su progenitor, siendo la más importante el ser un lenguaje interpretado, mientras que Visual Basic es capaz de compilar los proyectos en un archivo ejecutable. También poseen algunas diferencias de sintaxis y comportamiento, pero son insignificantes.

Gracias a su entorno de desarrollo integrado (*Integrated Development Environment*, IDE), VBA permite desarrollar aplicaciones con gran rapidez, controlando, desde la propia etapa de diseño, el ambiente de las diversas ventanas del programa. Además, Visual Basic es un lenguaje muy fácil de aprender y utilizar.

VBA permite también la integración con otras aplicaciones que admiten VBA. Lo que significa que AutoCAD puede, mediante las bibliotecas de objetos de otras aplicaciones, funcionar como controlador de automatización de otras aplicaciones como Microsoft Word o Excel.

Por último, es bueno señalar que VBA se ejecuta en el mismo espacio de proceso que AutoCAD lo que redunda en una ganancia en la velocidad de ejecución.

1.2 – Comparación con otras herramientas.

Existen, en AutoCAD, seis herramientas básicas de automatización de tareas: Archivos de lotes de comandos, AutoLISP, ADS, ARX, DIESEL y SQL. Una breve revisión de las características de cada uno de ellos permitirá una mejor comprensión de las bondades y las limitaciones de VBA.

Archivos de lotes de comandos (scripts).

Es posible, en AutoCAD, crear archivos de texto donde se especifiquen un conjunto de comandos que se ejecutarán sucesivamente. Los archivos de lotes de comandos pueden ejecutarse cuando se carga AutoCAD, o cada vez que se invoquen con el comando `SCRIPT`. Este tipo de personalización es muy útil cuando tenemos que ejecutar varias veces la misma secuencia de comando, o para realizar presentaciones de imágenes sucesivas.

AutoLISP.

AutoLISP es un lenguaje de programación desarrollado a partir de LISP, el cual fue creado en la década de los 50 y es usado, fundamentalmente, en los campos de la inteligencia artificial y los sistemas expertos. AutoCAD dispone de un intérprete de LISP integrado que le permite ejecutar código de AutoLISP directamente desde la línea de comandos, o cargarlo desde un archivo externo.

Las aplicaciones de AutoLISP son capaces de solicitarle datos al usuario, de acceder directamente a los comandos de AutoCAD y de modificar y crear entidades en el dibujo. Muchos de los comandos de AutoCAD son realmente aplicaciones de AutoLISP.

A pesar de ser un lenguaje compilado, AutoLISP es algo más lento que VBA en su ejecución, sin embargo, quizás su deficiencia principal sea lo enrevesado de sus sintaxis y la consecuente dificultad para depurar el código. No obstante, Autodesk continúa trabajando por solucionar estas limitaciones y ha apostado por el mantenimiento de AutoLISP dentro de AutoCAD. La principal ventaja de AutoLISP en estos momentos es la gran cantidad de código escrito en este lenguaje y su compatibilidad con versiones anteriores.

Desde la versión R14, se introdujo en AutoCAD el VisualLISP, que constituye una variante perfeccionada del AutoLISP. VisualLISP proporciona soporte para la manipulación de eventos y para el trabajo con Automatización ActiveX.

ADS.

El Sistema de Desarrollo de AutoCAD (*AutoCAD Development System*, ADS) es un ambiente de programación dirigido al desarrollo de aplicaciones para AutoCAD en lenguaje C. El soporte para ADS fue eliminado en la versión R14 de AutoCAD, pero algunas aplicaciones de versiones anteriores pueden continuar trabajando en la actualidad.

ARX.

La Extensión de Ejecución de AutoCAD (*AutoCAD Runtime eXtension*, ARX) es un lenguaje compilado para desarrollar aplicaciones de AutoCAD. Las aplicaciones ARX operan en el mismo proceso y espacio de memoria que AutoCAD, lo cual proporciona un comportamiento muy eficiente y gran rapidez en la ejecución.

Con AutoCAD 2000, Autodesk incluyó el ambiente de programación ObjectARX, el cual incluye bibliotecas de C++ para desarrolladores de aplicaciones de AutoCAD, extiende las clases y protocolos de AutoCAD, y crea nuevos elementos que operan exactamente como los comandos nativos de AutoCAD. La jerarquía de clases de objetos incluye todas las entidades de AutoCAD como objetos completos y derivables.

DIESEL.

El Lenguaje de Expresiones de Cadena Evaluadas Directa e Interpretativamente (*Direct Interpretively Evaluated String Expression Language*, DIESEL), el cual fue introducido en AutoCAD R12, permite modificar la barra de estado de AutoCAD, donde se muestra información como el estado de activación de los modos ORTHO y OSNAP, y la posición del cursor.

El DIESEL puede ser usado en los menús como un lenguaje de macros del mismo modo que AutoLISP. Las expresiones DIESEL toman cadenas de caracteres como argumento y las devuelven como resultado. Estas cadenas pueden ser evaluadas por AutoLISP mediante la función `menucdm`. A diferencia de AutoLISP, DIESEL consiste únicamente en funciones y no emplea variables.

SQL.

El Lenguaje de Consultas Estructurado (Structured Query Language, SQL) permite acceder a la información almacenada en sistemas de bases de datos. Mediante SQL, AutoCAD puede asociar atributos no gráficos almacenados en bases de datos externas (tales como dBASE III, Oracle y Microsoft Access) con objetos gráficos en el dibujo. El uso de bases de datos externas para almacenar los atributos del dibujo permite reducir el tamaño de los dibujos, simplificar la generación de reportes, y editar con facilidad los datos externos.

1.3 – Limitaciones de VBA.

Como contraparte de sus múltiples ventajas, VBA tiene algunas desventajas. En AutoCAD R14, era imposible acceder a la línea de comandos desde programación, ni tampoco ejecutar subrutinas de AutoLISP. En AutoCAD 2000 se solucionaron estas dificultades, sin embargo, no es una forma de trabajo completamente confiable y eficiente dentro de VBA.

Otro defecto de VBA (y en general de Visual Basic), que señalan muchos programadores es que no ofrecen un ambiente robusto de programación, comparable con C o C++. Afortunadamente esta diferencia se ha ido reduciendo con las versiones 5 y 6 de Visual Basic, que han incorporado muchas de las mejores características de la programación contemporánea. Aunque VBA incorpora sólo un grupo pequeño de controles ActiveX, permite al usuario adicionar otros si lo requiere.

Por último debe señalarse que VBA sólo opera en modo intérprete, por lo que no es capaz de crear archivos ejecutables.

1.4 – Proyectos.

Uno de los conceptos básicos de la programación con VBA es el proyecto, que no es más que una colección de formularios, módulos de código y módulos de clase, que trabajan en conjunto para realizar una función determinada. En AutoCAD, a diferencia de la mayoría de las aplicaciones, los proyectos pueden almacenarse en el propio archivo del dibujo (proyectos incrustados) o en un archivo separado (proyectos globales).

Los proyectos incrustados se cargan de forma automática al abrir el archivo que los contiene, por lo que son muy fáciles de distribuir y de utilizar. Sin embargo, estos proyectos tienen algunas desventajas, ya que no son capaces de abrir o cerrar documentos por estar limitada su funcionalidad al archivo que lo contiene.

Los proyectos globales, en cambio, no sólo permiten trabajar con un dibujo, sino también abrirlo o cerrarlo. No obstante, deben ser cargados manualmente cuando se van a ejecutar, de modo que el usuario debe conocer el nombre del archivo que lo contiene y su ubicación. A pesar de sus desventajas, los proyectos globales constituyen un medio excelente para almacenar bibliotecas de subprogramas.

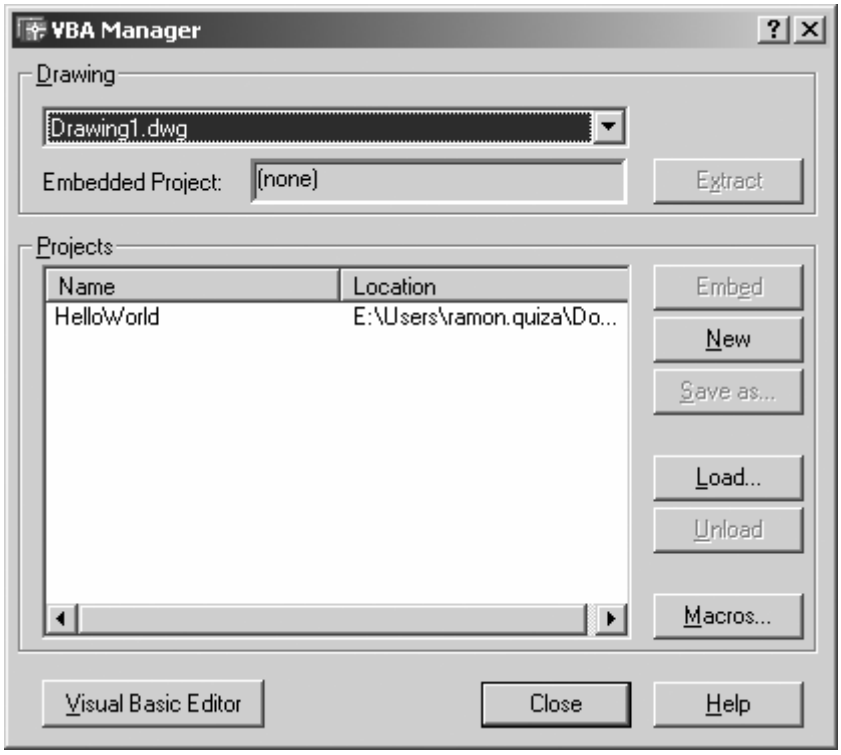


Fig. 1.1 – VBA Manager.

Durante el trabajo, se pueden tener varios proyectos globales cargados simultáneamente, pero sólo uno incrustado en cada dibujo.

Para administrar los proyectos se emplea el *VBA Manager* (ver Fig. 1.1), que puede ser cargado con el comando `VBAMAN`, o también, mediante la opción *Macro > VBA Manager...* del submenú *Tools* del menú estándar de AutoCAD.

Mediante el *VBA Manager* se pueden cargar, descargar y guardar los proyectos. También permite crear nuevos e incrustarlos.

Para crear un proyecto nuevo, una vez abierto el *VBA Manager*, se pulsa el botón *"New"*. Esta acción hace que se agregue un nuevo proyecto global a la lista de proyectos cargados. El botón *"Embed"* permite incrustar el proyecto seleccionado.

Mediante el botón *"Save as..."* es posible guardar el proyecto seleccionado, en un archivo determinado. Los botones "*Load*" y "*Unload*" permiten, respectivamente, cargar y descargar proyectos.

Aunque los proyectos de VBA para AutoCAD no son compatibles con el Visual Basic estándar, sus formularios, módulos de código y de clase pueden ser exportados e importados para utilizarlos en Visual Basic.

Los proyectos en VBA están compuestos por módulos que se clasifican en tres tipos:

- Formularios: Contienen cada uno de los cuadros de diálogo (ventanas) de la aplicación con todos sus controles y la programación correspondiente a ellos.
- Módulos de Código: Contienen definiciones de variables globales y código accesible desde cualquier parte de la aplicación.
- Módulos de Clase: Contienen definiciones de clases de objetos con las correspondientes métodos y propiedades.

1.5 – Macros.

Para el profano, las macros tienen un cierto significado lleno de misterio y peligro. De vez en cuando, al abrir un documento de Word o un libro de Excel, se muestra un cuadro de diálogo que dice: "El archivo que se desea abrir contiene macros. Las macros pueden contener virus. Siempre es más seguro deshabilitar las macros, pero si las macros son legítimas deshabilitándolas puede perder ciertas funcionalidades", dejando al pobre usuario en un grado de perplejidad similar al de Hamlet con su "*to be or not to be*". Afortunadamente, siempre hay un alma caritativa cerca que dice: "presiona el botón *Deshabilitar macros* y sigue trabajando, que eso no sirve para nada".

No obstante, si las macros no sirvieran para nada, no existirían. Si bien es verdad que las macros pueden ser un arma peligrosa en manos malintencionadas, en poder de un usuario conocedor y responsable son una herramienta maravillosa, que permite un ahorro sustancial de tiempo y esfuerzo.

Pero bueno: ¿qué son las macros? Las macros no son más que subprogramas que pueden ejecutarse desde una aplicación (Word, Excel, AutoCAD...), para realizar determinada función. Por ejemplo, se desea dibujar perfiles de vigas doble T en AutoCAD (como se sabe estas vigas están normalizadas). Si sólo necesitamos dibujar una viga, una sola vez, buscamos el catálogo o la norma, extraemos las dimensiones de nuestro perfil y luego dibujamos, una a una, las entidades (líneas, arcos, etc.) que conforman el perfil.

En cambio, si se tiene que desarrollar un proyecto donde hay que dibujar varios cientos de vigas de diferentes tipos, es preferible escribir una macro que acceda a una base de datos donde están las características geométricas de las vigas y que una vez seleccionado el tipo deseado, lo dibuje de forma automática. Por supuesto, esto supone el trabajo adicional de programar la macro y, posiblemente, crear la base de datos, pero al final el tiempo que ahorra cada vez que se dibuja un perfil, recompensa el esfuerzo invertido.

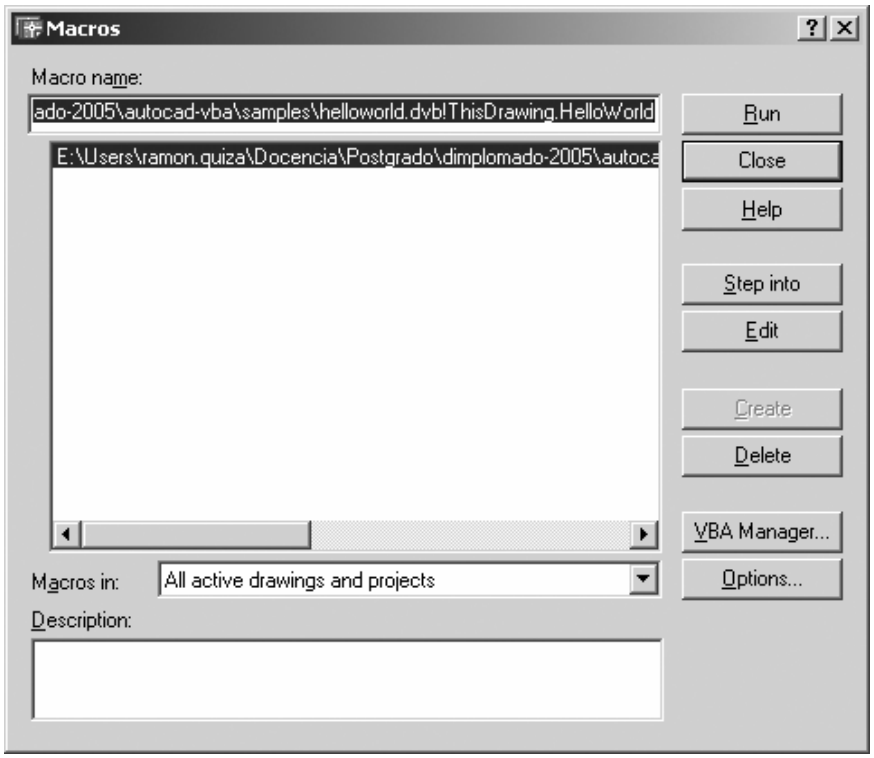


Fig. 1.2 – Cuadro de diálogo "Macros".

Para ejecutar una macro, se emplea el comando VBARUN, o la opción *Macros > Macros... (Atl+F8)*, del submenú *Tools* del menú estándar de AutoCAD. Este comando abre un cuadro de diálogo (Fig. 1.2) donde se muestra una lista de todas las macros de los proyectos abiertos en el dibujo.

Con el botón "*Run*" se ejecuta la macro seleccionada. El botón "*Close*" permite cerrar el cuadro de diálogo. "*Step into*" comienza la ejecución paso a paso de la macro, lo que permite depurarla (más adelante se explicará el proceso de depuración). "*Edit*" se utiliza para editar la macro y "*VBA Manager*" para acceder al Administrador de Proyectos como se vio en el epígrafe anterior. El botón "*Options*" muestra un pequeño cuadro de diálogo (Fig. 1.3) donde se establecen las opciones de ejecución de las macros de VBA: Habilitar Autoincrustación (*Enable Auto Enbedding*) crea, automáticamente, un proyecto de VBA incrustado para todo dibujo una vez que es abierto; Permitir Paradas en los Errores (*Allow Break on Errors*) permite detener, momentáneamente, la ejecución del programa cuando ocurre un error para depurarlo; y Habilitar Protección contra Virus Macros (*Enable Macro Virus Protection*), muestra un cuadro de diálogo al que hicimos

referencia al inicio del epígrafe, para advertir sobre la posible presencia de virus en las macros.

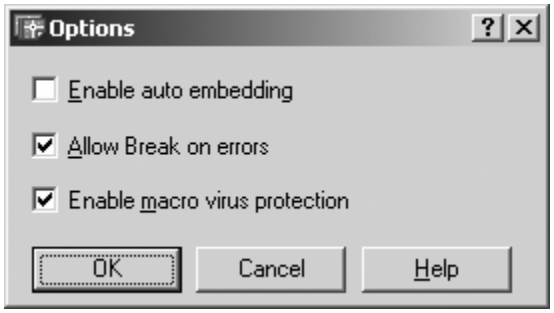


Fig. 1.3 – Opciones de ejecución de macros.

Preguntas.

1. ¿Qué importancia tiene *Visual Basic for Applications* dentro de AutoCAD?

2. ¿Qué ventajas y desventajas tiene VBA frente a AutoLISP? ¿Y frente a ARX?

3. ¿Qué es un proyecto de VBA?

4. ¿Qué diferencias hay entre un proyecto global y uno incrustado?

5. ¿Qué es una macro? ¿Ponga ejemplos de caso donde considere adecuado utilizar macros de VBA para AutoCAD?

CAPÍTULO 2
El Entorno de Desarrollo Integrado

OBJETIVO

Describir las características generales del Entorno de Desarrollo Integrado de VBA, detallando el uso de cada una de sus ventanas principales.

2.1 – Descripción general.

El Entorno de Desarrollo Integrado (IDE) de VBA es el ambiente de trabajo que permite crear, modificar y depurar aplicaciones de VBA para AutoCAD (Fig. 2.1).

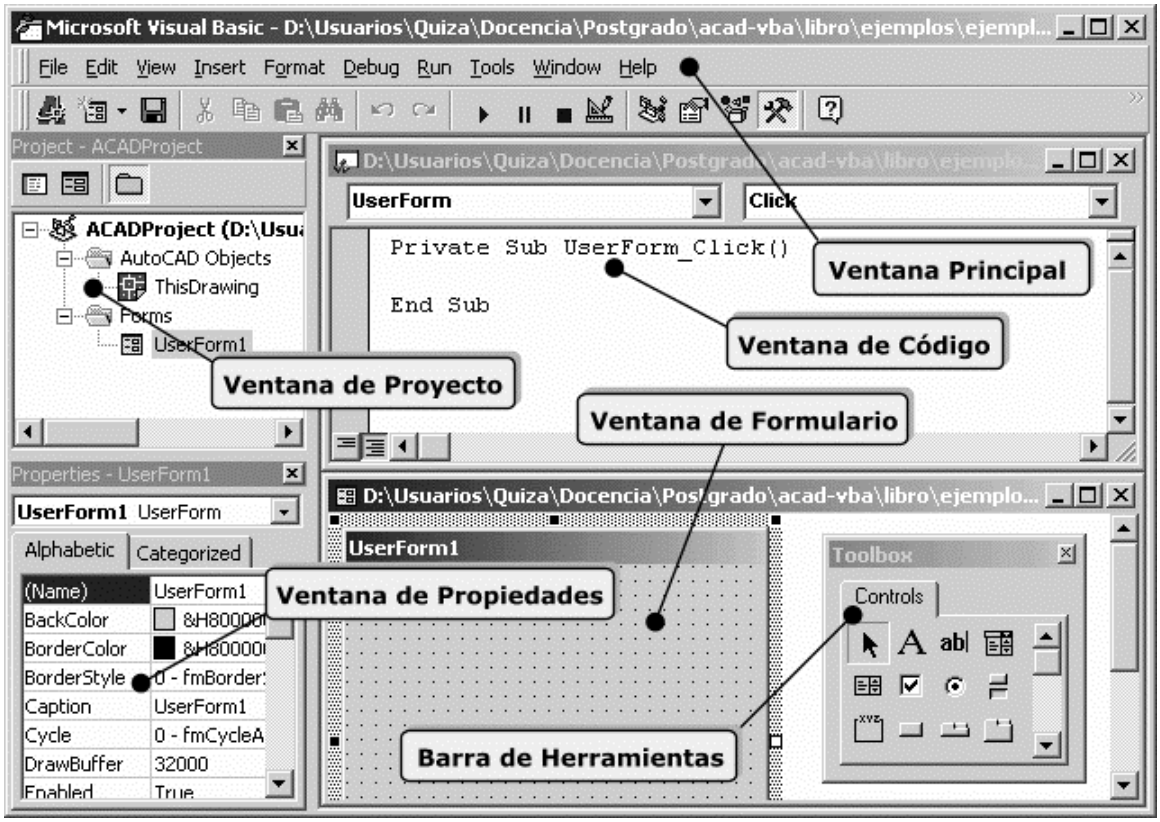


Fig. 2.1 – Entorno de Desarrollo Integrado de VBA.

Se puede acceder al IDE mediante el comando `VBAIDE`, o con la opción *Macros* > *Visual Basic Editor (Alt+F11)* del submenú *Tools* del menú estándar de AutoCAD. El IDE también puede abrirse mediante el botón "*Visual Basic Editor*" del *VBA Manager*. El IDE cuenta con varias ventanas:

- Ventana Principal: Es la ventana "madre" del IDE. Contiene el menú principal, las barras de herramientas, y el área de trabajo donde se muestran el resto de las ventanas.
- Ventana de Código: Muestra la programación asociada a cada formulario, a cada módulo de código o de clase, o al propio dibujo de AutoCAD.
- Ventana de Formulario: Muestra el diseño del formulario. Permite acceder a cada uno de los controles que lo componen.
- Ventana de Herramientas: Contiene los controles que pueden ser agregados a los formularios durante el diseño de la interfaz de usuario.
- Ventana de Proyecto: Muestra los módulos y demás elementos integrantes del proyecto.
- Ventana de Propiedades: Muestra y permite modificar las propiedades del formulario activo o de uno de sus controles que esté seleccionado.

En los siguientes tópicos se explicarán los detalles de cada una de estas ventanas y de los elementos que las componen.

2.2 – Ventana Principal.

La Ventana Principal (Fig. 2.2) se compone de los siguientes elementos:

- Barra de Título: Localizada en la parte superior de la ventana, muestra las palabras "Microsoft Visual Basic" seguidas del nombre del archivo que contiene el proyecto abierto.
- Barra de Menú: Está ubicada inmediatamente debajo de la barra de título. Proporciona los menús desplegables necesarios para desarrollar las aplicaciones. Muchos de los submenús presentes en la barra de menú tales como *File*, *Edit*, *View* y *Help* son similares a los del resto de los programas de Windows, en cambio, otros como *Debug* o *Run* son específicos de VBA. A lo largo de este material se irán viendo las diferentes opciones de la barra de menú.
- Barra de Herramientas: Está inmediatamente debajo de la barra de menú. Ofrece un conjunto de botones de comandos para acceder directamente a las opciones más utilizadas del programa. Cada acción se identifica con una imagen (ícono). La barra de herramientas mostrada en la Fig. 2.2 es la llamada "barra estándar" que aparece por defecto, pero también hay otras barras para usos específicos que pueden mostrarse mediante la opción *Toolbars* del submenú *View*.

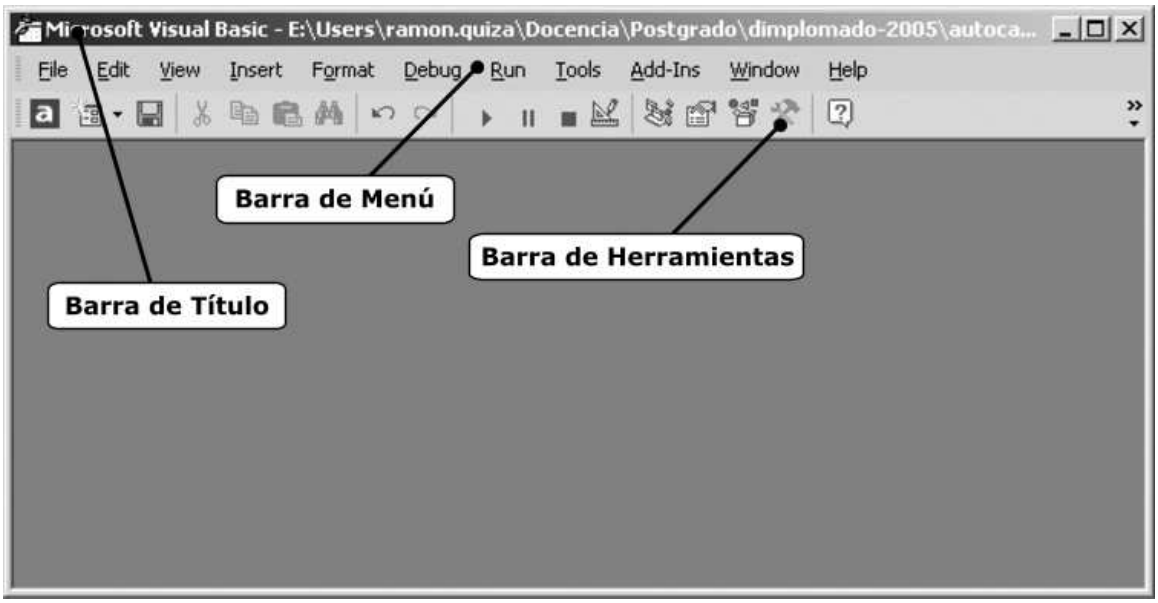


Fig. 2.2 – Ventana Principal.

2.3 – Ventana de Proyectos (Explorador de Proyectos).

La Ventana de Explorador de Proyecto (Fig. 2.3) muestra la información de todos los proyectos cargados en el dibujo actual, organizados jerárquicamente. Cuenta con un *Botón de Código* que muestra, en la Ventana de Código, el código correspondiente al objeto seleccionado, y con un *Botón de Objeto* que previsualiza el formulario en diseño.

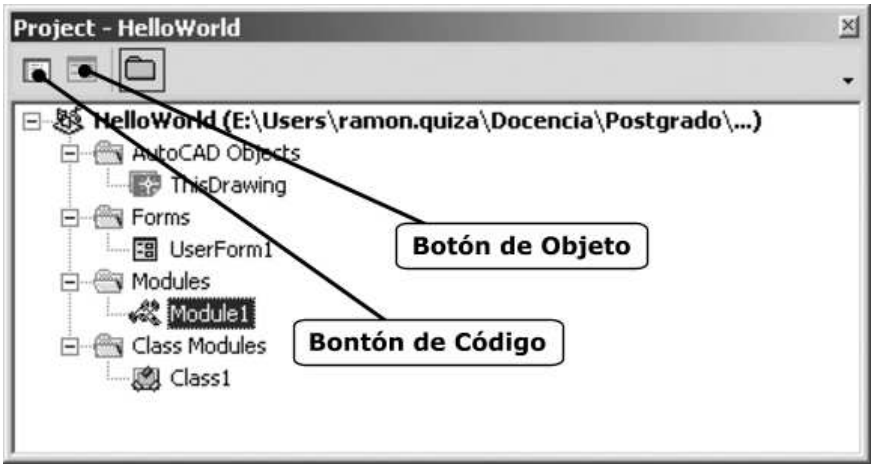


Fig. 2.3 – Explorador de Proyectos.

La Ventana de Proyecto aparece, por defecto, a la izquierda del IDE, pero puede arrastrarse y colocarse en cualquier otro lugar. También puede ocultarse y visualizarse con la opción *Project Explorer (Ctrl+R)* del submenú *View*.

2.4 – Ventana de Propiedades.

La Ventana de Propiedades (Fig. 2.4) permite visualizar y modificar las propiedades de los formularios y de cada uno de sus componentes. La lista de propiedades puede mostrarse ordenada alfabéticamente o por categorías. Mediante el cuadro desplegable de la parte superior, se puede seleccionar el componente deseado, lo cual también puede lograrse haciendo clic directamente sobre él, con el ratón, en la Ventana de Formulario.

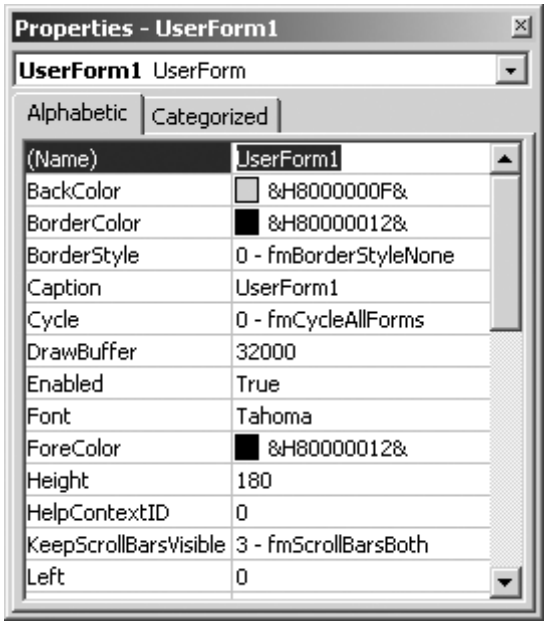


Fig. 2.4 – Ventana de Propiedades.

2.5 – Ventana de Código.

La Ventana de Código (Fig. 2.5) se utiliza para mostrar, escribir y modificar el código de programación de un formulario o módulo determinado.

En la parte superior tiene dos cuadros desplegables. El de la izquierda muestra todos los controles presentes (si es un formulario), además muestra una sección denominada *General* con la programación correspondiente a todo el módulo. En el cuadro de la derecha se muestran todos las subrutinas (procedimientos y funciones) del control seleccionado en el cuadro de la izquierda, incluyendo los procedimientos de evento. Ambos cuadros desplegables permiten cambiar el objeto seleccionado y la subrutina dentro del objeto.

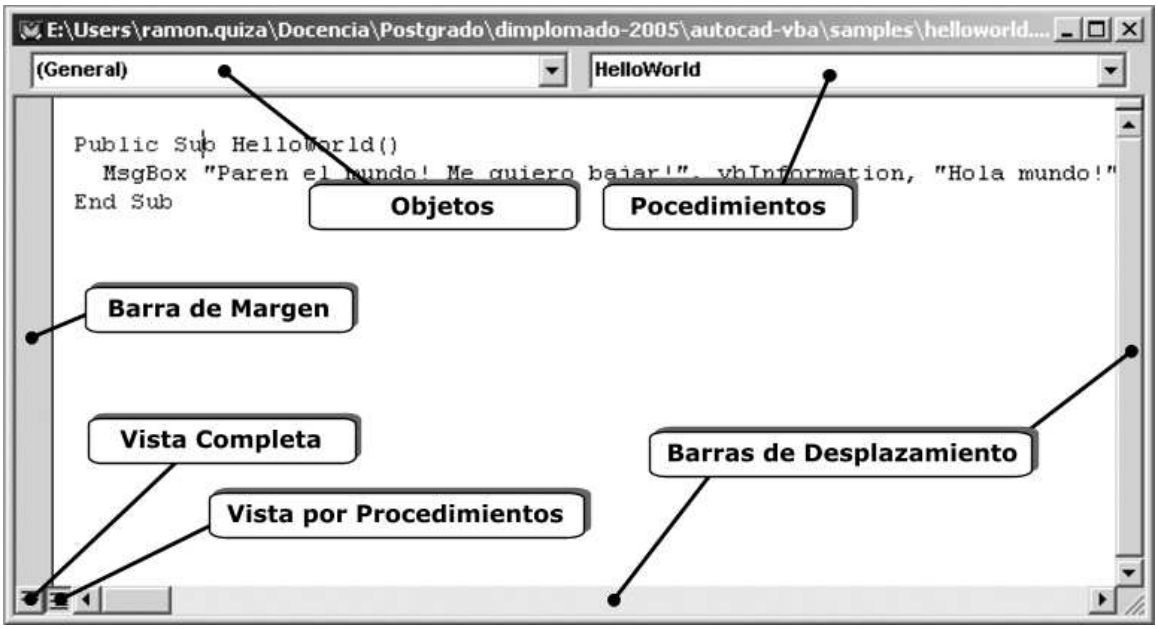


Fig. 2.5 – Ventana de Código.

La barra de desplazamiento de la derecha permite dividir la ventana en dos partes y mostrar dos subprogramas diferentes en ellas. Esto se logra con sólo desplazar dicha barra hacia abajo. Para volver a tener una sola área de código, basta con regresar la barra de desplazamiento a su posición original.

La que se encuentra en la izquierda se emplea para mostrar indicaciones de margen durante la edición. Los botones de vista completa y de vista por procedimientos están ubicados en la esquina inferior izquierda permiten mostrar el código completo, o sólo el del procedimiento seleccionado, respectivamente.

2.6 – Ventana de Formulario.

Para el usuario no avezado en temas de programación para Windows, el término formulario puede resultar poco claro. En realidad, se trata sólo de un problema de conceptualización: formularios, no es más que el nombre que se les da a las ventanas cuando se está diseñando o programando una aplicación; o sea, formulario significa ventana en la jerga del programador para Windows (quizás algún especialista en programación nos objete la exactitud de esta afirmación, pero al fin y al cabo esto no es un libro teórico sobre programación pura, sino un manual para aprender, lo más fácil y rápido posible, a programar con VBA para AutoCAD).

La Ventana de Formulario (Fig. 2.6) permite crear la interfaz gráfica de usuario de la aplicación: los formularios o cuadros de diálogo necesarios para intercambiar información con el usuario. El diseño y puesta a punto de los formularios es tan simple

como arrastrar a ellos los controles necesarios, desde la Caja de Herramientas, y luego, modificar sus propiedades, hasta lograr el aspecto y el comportamiento deseados.

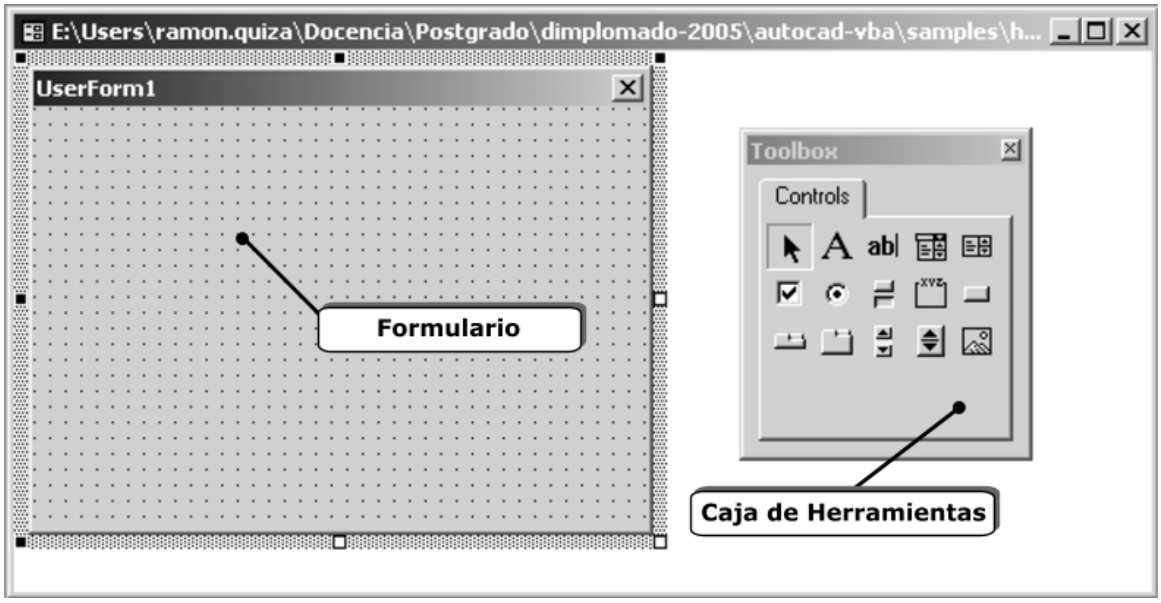


Fig. 2.6 – Ventana de Formulario.

2.7 – Personalización del IDE.

La apariencia del IDE de VBA puede modificarse para adaptarse al gusto del usuario. Esto se logra con la ventana *Options*, a la cual se accede mediante el submenú *Tools* de la barra de menú del IDE. La ventana de opciones contiene cuatro fichas:

Editor.

En ella se especifica la configuración de las Ventanas de Proyecto y de Código. Las opciones de código incluyen:

- Comprobación de sintaxis automática (*Auto Syntax Check*): Realiza el chequeo de la sintaxis del código mientras se escribe y avisa de los errores cometidos.
- Requerir declaración de variables (*Require Variable Declaration*): Exige que las variables que se utilicen en el código sean previamente declaradas. Si esta opción está chequeada, se adiciona automáticamente la línea `Option Explicit` en las declaraciones generales de cada módulo.
- Lista de miembros automática (*Auto List Member*): Despliega una lista con información que pueda completar, de forma lógica, la instrucción que se está escribiendo.
- Información rápida automática (*Auto Quick Info*): Muestra información sobre las funciones y sus parámetros, a medida que se escriben.

- Sugerencias de datos automáticas (*Auto Data Tips*) Muestra el valor de la variable sobre la cual se encuentra el cursor. Sólo funciona cuando se está trabajando en modo de interrupción.
- Sangría automática (*Auto Indent*): Permite ajustar automáticamente la posición inicial de las líneas a la de sus antecesoras.
- Ancho de tabulación (Tab Width): Establece el ancho de las tabulaciones, que puede variar entre 1 y 32 espacios. El valor por defecto es 4 espacios.

Por su parte, las opciones de configuración de ventana son:

- Modificar texto mediante "arrastrar y colocar" (*Drag-and-Drop Text Editing*): Permite arrastrar y colocar texto del código actual, desde la ventana de Código en la Ventana Inmediata (*Immediate Window*) o en la Rastreo (*Watch Window*) (ambas ventanas se explican en el capítulo dedicado a la depuración).
- Vista completa predeterminada del módulo (*Default to Full Module View*): Establece el estado por defecto para los nuevos módulos para permitir que se muestren todos los procedimientos a la vez.
- Separador de procedimientos (*Procedure Separator*): Permite mostrar u ocultar barras de separación que aparecen al final de cada procedimiento en la Ventana de Código.

Formato del Editor (*Editor Format*).

En esta hoja se establece la apariencia del código de Visual Basic. Cuenta con las siguientes opciones:

- Colores del código (*Code Color*): Determina el color de primer plano, de fondo y del indicador de margen de cada uno de los tipos de texto mostrado en la lista.
- Fuente (*Font*): Especifica el nombre de la fuente utilizada para todo el código.
- Tamaño (*Size*): Especifica la altura de la fuente usada para el código.
- Barra indicadora al margen (*Margin Indicator Bar*): Muestra u oculta la Barra de Margen de la Ventana de Código.
- Ejemplo (*Sample*): Muestra ejemplo de cada texto seleccionado.

General.

En ella se especifican la configuración, la manipulación de errores y las opciones de compilación para el proyecto de Visual Basic actual. Incluye:

Configuración de la Rejilla de Formulario (*Form Grid Settings*): Determina la apariencia de la rejilla del formulario durante el proceso de edición mediante las opciones:

- Mostrar cuadrícula (*Show Grid*): Muestra u oculta la cuadrícula.
- Unidades de la cuadrícula (*Grid Units*): Muestra las unidades de medida utilizada para medir la separación entre los nodos de la cuadrícula.

- Ancho (*Width*): Determina la distancia horizontal entre dos nodos de la cuadrícula consecutivos. Oscila en el rango de 2 a 60 puntos.
- Altura (*Height*): Determina la distancia vertical entre dos nodos de la cuadrícula consecutivos. Oscila en el rango de 2 a 60 puntos.
- Alinear controles a cuadrícula (*Align Controls to Grid*): Coloca los bordes exteriores de los controles alineados automáticamente con la cuadrícula.

Información sobre herramientas (*Show ToolTips*): Muestra información rápida de los botones de las barras de herramienta.

Ocultar ventanas al contraer el proyecto (*Collapse Proj. Hides Windows*): Determina si las ventanas de Formulario, Objeto o Módulo son cerradas automáticamente cuando un proyecto es contraído en el Explorador de Proyectos.

Notificar antes de perder estado (*Notify Before State Loss*): Determina si el usuario recibirá un mensaje notificando que la acción indicada causará que todas las variables a nivel de módulo serán reiniciadas para el proyecto en ejecución.

Interceptación de Errores (*Error Trapping*): Determina como los errores son manipulados en el entorno de desarrollo de Visual Basic. Los cambios en estas opciones afectan a todas las instancias de Visual Basic lanzadas luego de la modificación. Incluye:

- Interrumpir en todos los errores (*Break on All Errors*): Ante cualquier error, el proyecto detiene su ejecución y pasa a modo de interrupción.
- Interrumpir en módulos de clases (*Break in Class Modules*): Cualquier error no manipulado que tenga lugar en un módulo de clase, provoca que el proyecto detenga su ejecución y pase a modo de interrupción, colocando el indicador del cursor en la línea de código dentro del módulo de clase que produjo el error.
- Interrumpir en errores no controlados (*Break on Unhandled Errors*): Si hay un manipulador de errores activos, el error es manipulado sin entrar en modo de interrupción. Si no hay manipulador de error activo, el error provoca una parada con paso a modo de interrupción.

Compilar (*Compile*): Cuenta con las siguientes opciones:

- Compilación a petición (*Compile On Demand*): Determina si un proyecto es completamente compilado antes de ejecutarse, a si sus partes se van compilando a medida que se necesitan. La segunda variante permite ejecutar el proyecto con mayor rapidez.
- Compilar en segundo plano (*Background Compile*): Determina si se emplea tiempo ocioso durante el tiempo de ejecución para completar la compilación del proyecto en segundo plano. Este elemento no esta disponible a menos que se haya seleccionado la opción de Compilar a petición.

Acople (*Docking*).

Permite seleccionar que ventanas van a ser acoplables. Una ventana es acoplable cuando se adjunta o "ancla" a un borde de otra ventana acoplable o de la aplicación. Cuando se mueve una ventana acoplable, esta "captura" su ubicación. Si la ventana no es acoplable, es posible moverla por toda la aplicación y dejarla en cualquier sitio.

Ejemplo Resuelto.

Para ilustrar el uso del IDE de AutoCAD, y para ir viendo algún resultado positivo del curso, se pasará a elaborar una primera macro en VBA para AutoCAD. Naturalmente, esta será muy simple, con fin puramente didáctico.

El objetivo de esta macro será enviar un mensaje a la pantalla (por supuesto, para que lo lea el usuario). Para ello se empleará la función de VBA `MsgBox`. Se verá, paso a paso, el proceso de creación de la macro.

En primer lugar, en la ventana de comandos de AutoCAD se teclea el comando `VBAMAN` que, como ya se vio, muestra el administrador de proyectos (ver Fig. 1.1). Éste debe mostrar una listado de proyectos vacío. Para agregar el nuevo proyecto, se presiona el botón "*New*"; esto trae como consecuencia que un proyecto llamado "ACADProject", de tipo global, sea creado y agregado a la lista.

El siguiente paso consiste en guardar el proyecto que recién se ha creado. Para ello se selecciona el nombre del proyecto de la lista y luego se presiona el botón "*Save as...*". En el cuadro de diálogo de guardar que aparecerá se establece la ubicación y el nombre que se quiere dar al archivo que contenga el proyecto.

Una vez guardado el archivo del proyecto, se pasa a abrir el IDE de VBA, para lo cual se puede presionar el botón "Visual Basic Editor" del propio administrador de proyectos. Esto abre el editor, el cual aparecerá en pantalla y como una aplicación más en la barra de tareas. Dentro de la ventana de proyecto, aparecerá solamente el que hemos creado (ver Fig. 2.7).

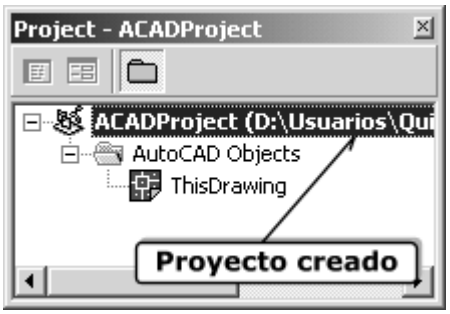


Fig.2.7 – Proyecto creado.

Es buena política de programación, establecer nombres a los proyectos (y en general, a las diferentes entidades que sean creadas) para poder identificarlos con facilidad. Una costumbre aconsejable es nombrarlos con un sufijo de tres letras que indique claramente que tipo de objeto es. En el caso de los proyectos, se utilizará el sufijo ”pro”, por lo cual se establece como nombre para el proyecto recién creado “proHolaMundo”. Para asignarle el nombre al proyecto, en la ventana de propiedades – habiendo, previamente, seleccionado el proyecto en la ventana correspondiente – se escribe el nuevo nombre en la casilla “*(Name)*”.

Fig. 2.8 – Cambio del nombre del proyecto.

Después de los pasos anteriores, se guardan los cambios realizados con la opción *Save* del menú *File* del IDE.

Para crear la macro, dentro del proyecto, se muestra la hoja de código asociada al objeto “*ThisDrawing*” (este objeto controla el dibujo activo en AutoCAD). Para ello, se selecciona el elemento “*ThisDrawing*” en la ventana de proyecto y se presiona el botón de código. Aparecerá la ventana de proyecto que, inicialmente estará vacía.

A continuación, se selecciona la opción “*Procedure*” del menú “*Insert*”, del IDE, para insertar un procedimiento en la hoja de código activa. Se mostrará el cuadro de diálogo “*Add Procedure*”, donde se establece el nombre del procedimiento (en este caso se le llamará “HolaMundo”), el tipo de procedimiento: subprograma, función o propiedad (en este caso, subprograma) y el alcance del procedimiento: público o privado (en este caso, público).

Fig. 2.9 – Adicionar procedimiento.

Como resultado, se insertará, en la ventana, el siguiente código (ver Fig. 2.10)

```
Public Sub HolaMundo()

End Sub
```

El código anterior, que como se puede ver es muy simple, es la definición de un subprograma público. La primera línea indica el inicio del subprograma (instrucción **Sub**), su alcance (**Public**) y su nombre (HolaMundo).

La segunda línea indica el final del subprograma (**End Sub**). Todo el código que se agregue al subprograma, se escribirá entre estas dos líneas.

Para que mostrar el cuadro de mensaje, se utiliza el procedimiento MsgBox, de Visual Basic, escribiendo el siguiente código:

```
Public Sub HolaMundo()
  MsgBox "Paren el mundo! Me quiero bajar!"
End Sub
```

A este procedimiento se le pasa como argumento o parámetro el texto que aparecerá en el cuadro; en el ejemplo, "¡Paren el mundo! ¡Me quiero bajar!". En realidad el procedimiento MsgBox es algo más complejo y cuenta con otros parámetros, pero para este ejemplo así será suficiente. En la Fig. 2.10 se muestra la ventana de código con la macro escrita.

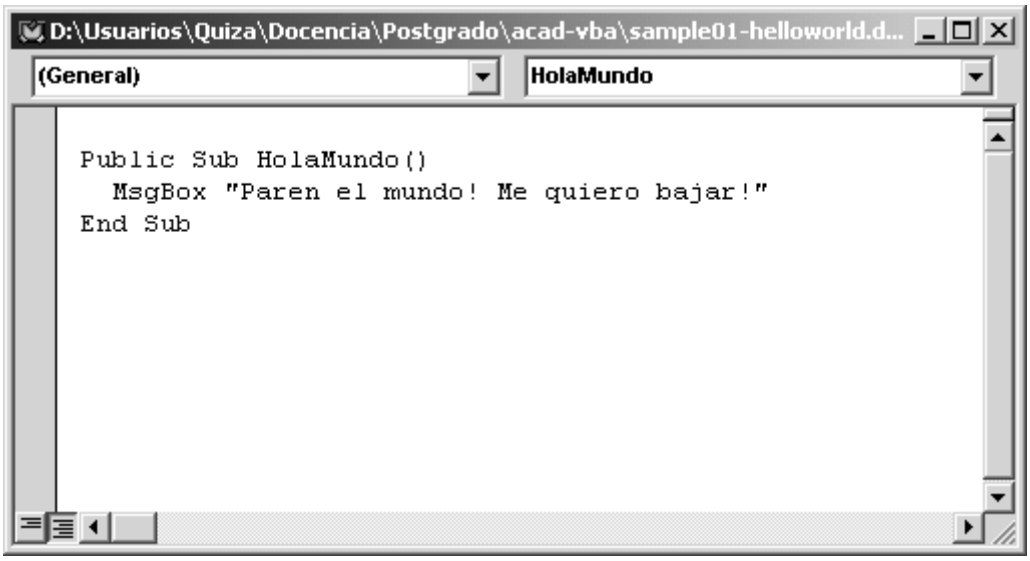


Fig. 2.10 – Ventana de Código con la programación de la macro.

Se guarda el trabajo hecho, y se regresa a AutoCAD, para probar la macro. Para ejecutar una macro cualquiera, se teclea el comando VBARUN y, en el cuadro de diálogo que aparece, se selecciona la macro que se desea ejecutar y se presiona el botón "*Run*". Si se ejecuta la macro que se acaba de crear (y que debe ser la única que aparece en el listado) se obtiene como resultado, el cuadro de mensaje que se muestra en la Fig. 2.11.

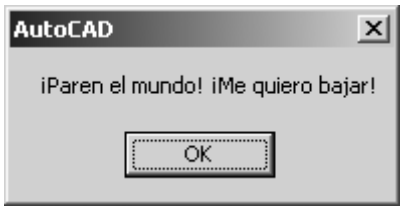


Fig. 2.11 – Mensaje generado por la macro.

Con esta comprobación, se puede dar por terminada la creación de esta primera macro, la cual, como se dijo al inicio, es muy simple y de escasa utilidad salvo como ejemplo. No obstante, es un primer paso en programación para AutoCAD. ¡Ya estamos adentro!

Preguntas y Ejercicios Propuestos.

1. ¿Qué es el Entorno de Desarrollo Integrado (IDE)?

2. ¿Qué ventanas componen el IDE? Descríbalas.

3. Cree un proyecto global y guárdelo. Escriba, dentro de él, una macro que muestre un mensaje en pantalla con su nombre y sus apellidos.

CAPÍTULO 3
Estructuras Básicas del Lenguaje VBA

OBJETIVO

Utilizar las estructuras básicas del lenguaje Visual Basic for Applications para desarrollar subprogramas.

3.1 – Introducción.

Aunque este libro no pretende ser un manual de Visual Basic, está claro que una comprensión básica de las principales estructuras de este lenguaje es una premisa para desarrollar buenos programas. Como se ha presumido que el lector no está previamente familiarizado con Visual Basic (o con cualquier otro lenguaje de programación), se dedicará este capítulo para explicar los conceptos elementales de este lenguaje, ilustrando la explicación con ejemplos. Naturalmente, no se pretende que tras leer un par de páginas el lector sea un consumado programador; si al final del mismo, es capaz de utilizar Visual Basic para crear subprogramas en AutoCAD que le permitan automatizar su trabajo, el objetivo habrá sido cumplido.

3.2 – Comentarios y otras utilidades básicas.

Visual Basic interpreta (o al menos intenta hacerlo) todo el código que se escriba, excepto aquel que esté a la derecha del signo apóstrofe (`'`), el cual es completamente ignorado. Esta característica es muy útil para escribir comentarios dentro del código. Los comentarios pueden ocupar toda una línea, o estar a continuación de una línea de código, tal como se muestra a continuación:

```
' Esto es un comentario que ocupa toda la línea
A = B + C  ' Este comentario está en la misma línea
           ' que un fragmento de código (A = B + C)
```

Otro rasgo muy usado de Visual Basic es que puede escribirse una sentencia en más de una línea, empleando el carácter subrayado (_). Por ejemplo, las dos sentencias siguientes son exactamente equivalentes:

```
TiempoMaquinado = Longitud/(Avance*Revoluciones)
TiempoMaquinado = Longitud/ _
                  (Avance*Revoluciones)
```

También es posible escribir varias sentencias en una misma línea, separándolas por dos puntos (:), de la forma:

```
A = 5: B = 8: C = A + B
```

Lo anterior sería equivalente a:

```
A = 5
B = 8
C = A + B
```

3.3 – Proyectos y módulos.

El proyecto (como ya se explicó anteriormente), es el conjunto de todos los archivos necesarios para realizar la acción deseada. A estos archivos se les denomina, en forma genérica, módulos.

Los módulos pueden ser varios tipos:

- Formularios: Contienen la información de un formulario y el código asociado a él.
- Módulos de código: Contienen código que no está asociado a ningún formulario, además de definiciones de variables.
- Módulos de clase: Contienen la definición de una clase de objeto.

3.4 – Variables y constantes.

Las variables son una parte muy importante de cualquier lenguaje de programación. Estas pueden entenderse bastante bien como espacios de memoria que se reservan en la computadora para almacenar en ellos determinados valores. Por ejemplo, en un programa para calcular el área de un circulo, se puede definir una variable para almacenar el valor del radio. Esta acción reservará en la memoria un cierto espacio, y durante la ejecución del programa el valor que tome este radio será almacenado en el mismo.

Las variables siempre se definen para un tipo de datos determinado. En Visual Basic existen diferentes tipos de datos, entre los cuales cabe destacar los siguientes:

- `Boolean` (booleano): Representa variables que sólo pueden tomar dos valores (verdadero ó falso, sí ó no, 0 ó 1). Se almacena en dos bytes de memoria. Se puede usar para representar, por ejemplo, el estado de una válvula (abierto ó cerrado) o el estado de un indicador de seguridad (encendido ó apagado).

- `Integer` (entero): Representan valores numéricos enteros, tales como la cantidad de remaches que hay en una estructura metálica, o la cantidad de válvulas

que existen en una red de tuberías. Puede tomar valores entre –32 768 y 32 767 y se almacena en dos bytes (16 bits) de memoria.

- `Long` (entero largo): También representa valores numéricos enteros pero en un rango mucho más amplio: desde –2 147 483 648 hasta 2 147 483 647. Se almacena en cuatro bytes (32 bits).

- `Byte`: Es otro tipo de datos para representar valores numéricos enteros, pero sólo en el rango de 0 a 255. Se almacenan en un solo byte.

- `Single` (real de precisión simple): Representa valores numéricos reales en el rango desde $1{,}4 \cdot 10^{-45}$ hasta $3{,}4 \cdot 10^{38}$ (para números negativos, desde $-3{,}4 \cdot 10^{38}$ hasta $-1{,}4 \cdot 10^{-45}$). Es el tipo de datos apropiado para representar, por ejemplo, el peso de un perfil laminado de determinada longitud, la velocidad de un elemento de máquina en un instante dado o la potencia de un motor eléctrico. Se utilizan 4 bytes para cada variable de este tipo.

- `Double` (real de precisión simple): Al igual que el anterior, también representa valores numéricos reales, pero con una precisión dos veces superior. Admite valores desde $1{,}8 \cdot 10^{-308}$ hasta $4{,}9 \cdot 10^{324}$ (o desde $-4{,}9 \cdot 10^{324}$ hasta $-1{,}8 \cdot 10^{-308}$, para números negativos). En este tipo de datos se almacenan gran número de variables típicas del trabajo con AutoCAD, como coordenadas, longitudes y ángulos. Requiere 8 bytes para su almacenamiento.

- `String` (cadena): Representa variables de cadena de caracteres del código ASCII, tales como la mayor parte de los textos ordinarios. Ejemplo de su utilización puede ser para almacenar el código del plano de una pieza, el modelo de una máquina, el nombre de un operario, etc.

- `Object` (objeto): Se emplea para representar referencias a objetos (desde el punto de vista de programación) de cualquier tipo. Requiere cuatro bytes.

- `Variant` (variante): Puede almacenar tanto valores numéricos como de texto. Es un tipo de variable muy versátil ya que puede contener casi cualquier información, pero su uso es poco eficiente, por lo que debe evitarse su empleo tanto como se pueda.

Las variables se definen con la instrucción `Dim`, cuya sintaxis es:

```
Dim <NombreDeLaVariable> As <TipoDeDatos>
```

Por ejemplo, para definir una variable llamada `LongitudTornillo`, de tipo `Double`, la instrucción será:

```
Dim LongitudTornillo As Double
```

En la declaración se puede omitir el tipo de variable, pero entonces la variable declarada será de tipo `Variant`.

Los nombres de las variables pueden tener hasta 255 caracteres, que pueden ser letras, números o el carácter subrayado (_), aunque el primer carácter debe ser, obligatoriamente una letra. No se admiten caracteres especiales como espacios, signos #, $, % y @ o letras con tildes o diéresis. Así por ejemplo: `DiametroDelTanque`, `Longitud_Cordon`, `PresionPunto1` son nombres válidos para variables, mientras que `Cateto Costura`, `3Altura` o `EficienciaEn%` no lo son, ya que la primera contiene un espacio, la segunda no empieza con una letra y la tercera contiene el símbolo %. Se debe señalar que en VBA no se diferencian las mayúsculas de las minúsculas, así `PasoRosca`, `pasorosca` y `PaSoRoSCa` serían la misma variable.

La declaración de varias variables puede hacerse en diferentes instrucciones Dim, o en una sola, de la forma:

```
Dim <Variable1> As <Tipo1>, <Variable2> As <Tipo2>, ...
```

Se debe tener cuidado en no hacerlo de la forma:

```
Dim <Variable1>, <Variable2> As <Tipo>
```

ya que en este caso, sólo `Variable2` será del tipo especificado, y `Variable1` será de tipo `Variant`.

La declaración de las variables no es imprescindible en VBA, pero cada variable no declarada será de tipo `Variant`, con lo que se perderá en eficiencia. Escribiendo la instrucción `Option Explicit`, al inicio de un módulo la declaración de las variables se vuelve obligatoria y de no hacerlo se genera un error durante la ejecución del código. La utilización de la instrucción `Option Explicit` es una buena política de programación que permite generar códigos más eficientes y limpios.

Las constantes son elementos que almacenan un valor que, como lo indica su nombre, no varía durante toda la ejecución del programa. Se definen con la instrucción `Const`, según la sintaxis:

```
Const <NombreDeLaConstante> [As <TipoDeDato>] = <Valor>
```

Como se puede ver, el tipo de dato de la constante puede omitirse, pero en este caso no hay garantía de que Visual Basic utilice el tipo de datos más eficiente. A continuación se muestran algunos ejemplos de declaración de constantes:

```
Const PI As Double = 3.1416
Const CantidadEmpleados As Integer = 1251
Const DensidadAcero As Single = 7.83
Const NombreAutor As String = "Ramón Quiza Sardiñas"
```

Visual Basic dispone de un gran número de variables ya definidas. El listado de las mismas se puede ver con el explorador de objetos, el cual se muestra con la opción *Object Browser* del menú *View* del IDE de VBA.

3.5 – Funciones y procedimientos.

En los módulos, el código está organizado en subrutinas, las cuales pueden ser procedimientos o funciones. Los procedimientos (*procedures*) son un conjunto de instrucciones encerradas entre las instrucciones `Sub` y `End Sub`, que realizan un conjunto de acciones determinadas. A los procedimientos se les pueden suministrar argumentos, que son variables que las instrucciones del mismo procesarán.

En el siguiente ejemplo, se muestra un procedimiento llamado `MiMensaje`, al cual se le suministran, como argumentos las variables `Nombre` y `Mensaje`, ambas del tipo `String`, y muestra, como acción, un mensaje que combina ambas variables, utilizando en procedimiento `MsgBox` que ya se vio anteriormente.

```
Sub MiMensaje(Nombre As String, Mensaje As String)
  ' Muestra un mensaje en pantalla
  MsgBox Mensaje, vbInformation, "Mensaje de " & Nombre
End Sub
```

Las funciones (*functions*) son similares a los procedimientos pero, además, como resultado devuelven un valor que será de un tipo determinado. Se implementan entre las etiquetas `Function` y `End Function`. El ejemplo siguiente, se muestra una función que convierte longitudes de pulgadas a metros.

```
Function PulgAMetros(Longitud As Double) As Double
  ' Convierte una longitud de pulgadas a metros
  PulgAMetros = 0.0254 * Longitud
End Function
```

Las llamadas a procedimientos y funciones se lleva a cabo mediante el nombre de los mismos, seguidos de los nombres de las variables que se le pasan. En las funciones, se encierran las variables entre paréntesis, mientras que en los procedimientos no. A continuación se muestra una macro que utiliza el procedimiento y la función anteriormente explicados:

```
Public Sub Msg()
  ' Muestra las llamadas a procedimientos y funciones
```

```
  Dim LPulg As Double, LMetros As Double, _
      MiMsg As String
  LPulg = 24
  LMetros = PulgAMetros(LPulg) ' Llamada a la función
  MiMsg = LPulg & " pulgadas equivalen a " & _
          LMetros & " metros."
  MiMensaje "Ramón Quiza", Msg ' Llamada al procedim.
End Sub
```

3.6 – Ámbito de las subrutinas y variables.

Se denomina ámbito de una subrutina o variable, a la parte de un programa desde la cual ésta es accesible y desde donde, por lo tanto, puede ser llamada o utilizada. Las subrutinas y variables pueden ser de ámbito global o modular. Las variables, además, pueden ser de ámbito local.

Son de ámbito global, aquellas subrutinas y variables accesibles desde cualquier lugar del programa. Se declaran anteponiendo la palabra clave `Public` antes de la declaración, tal como se muestra a continuación:

```
Public DiametroTubo As Double
Public Function Area(Diametro As Double) As Double
```

Las subrutinas y variables de ámbito modular, sólo son accesibles desde el módulo en el cual fueron declaradas. Se especifican con la palabra clave `Private`.

```
Private AlturaTanque As Double
Private Sub AbrirValvula()
```

Nótese que las instrucciones `Public` o `Private`, en las definiciones de variables, sustituyen a la instrucción `Dim`, la cual se mantiene sólo por compatibilidad con versiones antiguas de Basic.

Por último, las variables locales, son accesibles sólo dentro de la subrutina donde fueron definidas.

```
Sub AlertarSobrepresion()
  ' Muestra la definción de una variable local
  Dim MensajeAlerta As String
  MensajeAlerta = "Existe sobrepresión en el punto 14."
  MsgBox MensajeAlerta, vbExclamation, "Alerta"
End Sub
```

3.7 – Arreglos.

Los arreglos son estructuras que permiten hacer referencia a varios valores mediante un mismo nombre. Para referirse a un elemento específico dentro del conjunto se utilizan uno o más índices. El concepto de arreglo dentro de los lenguajes de programación es similar a los de vector o matriz, en Álgebra.

Todos los elementos de un arreglo deben ser de un mismo tipo de datos. Por supuesto, si el tipo dato del arreglo es `Variant`, entonces puede contener diversos tipos de datos.

Los arreglos pueden ser estáticos o dinámicos. Los primeros contienen un número fijo de elementos, establecido en su declaración. Por ejemplo, la instrucción

```
Dim Nombres(0 To 9) As String
```

declara un arreglo llamado `Nombres`, que puede contener 10 valores de tipo `String`, a los cuales se les puede referir como `Nombres(0)`, `Nombres(1)`... `Nombres(9)`.

A los arreglos dinámicos no se les predetermina el tamaño durante la declaración, sino que se le establece posteriormente con la instrucción `Redim`. Por ejemplo:

```
Dim Nombres() As String
ReDim Nombres(0 To 2)
Nombres(0) = "Paco"
Nombres(1) = "Cuca"
Nombres(2) = "Cheo"
```

Los arreglos dinámicos son especialmente útiles cuando no conocemos de antemano el tamaño del arreglo.

3.8 – Operadores.

Los operadores son elementos que realizan una acción determinada (operación) sobre una o más variables. En Visual Basic existen diferentes tipos de operadores. En la Tabla 3.1, se muestra un resumen de los mismos, con ejemplos de su aplicación:

Tabla 3.1 – Operadores más usados en Visual Basic.

Tipo	Operador	Operación	Ejemplo
Aritméticos	+	Adición	A + B
	-	Substracción,	A - B
	*	Multiplicación	A * B
	/	División	A / B
	\	División entera	A \ B
	Mod	Resto de la división entera	A Mod B
	-	Cambio de signo	- A
	^	Elevación a potencia	A ^ B
De texto	&	Concatenación de cadenas	A & B
	Like	Comparación de cadenas	A Like B
Relacionales	=	Igualdad	A = B
	<>	Diferencia	A <> B
	<	Minoría	A < B
	>	Mayoría	A > B
	<=	Minoría o igualdad	A <= B
	>=	Mayoría o igualdad	A >= B
Lógicos	Not	Negación	Not A
	And	And	A And B
	Or	Or inclusivo	A Or B
	Xor	Or exclusivo	A Xor B

Estos no son los únicos operadores de que dispone Visual Basic, pero sí los más usados. Para ver todos los operadores puede consultar la ayuda de VBA donde se da una excelente explicación de los mismos.

3.9 – Estructuras condicionales.

Es común que, en cualquier problema, nos encontremos frente a tomas de decisiones. Por ejemplo, si la presión es demasiado alta, entonces abrimos la válvula de seguridad; si el número de Reynolds dentro de la tubería es mayor que $5 \cdot 10^5$, entonces el régimen del flujo es turbulento, sino el régimen es laminar. Para el tratamiento de estas tomas de decisiones, los lenguajes de programación incorporan las estructuras condicionales.

La estructura condicional más usada en Visual Basic es `If...Then...Else`, la cual permite evaluar una condición, si se cumple se ejecutará un conjunto de instrucciones (llamadas instrucciones positivas), de lo contrario, se ejecutará otro conjunto de instrucciones (instrucciones negativas). La sintaxis de la misma es:

```
If (<Condicion>) Then
  <InstruccionesPositivas>
Else
  <InstruccionesNegativas>
End If
```

Por ejemplo, la condición sobre el número de Reynolds podrá quedar de la forma:

```
If (Re > 500000) Then
  Regimen = "Laminar"
  Nu = 0.032 * Re ^ 0.8
Else
  Regimen = "Turbulento"
  Nu = 0.67 * Re ^ 0.5 * Pr ^ 0.333
End If
```

Las condiciones pueden hacerse más complejas, combinando varias condiciones mediante operadores lógicos. En el siguiente ejemplo, la condición se satisface sólo si la sección transversal del conducto es circular y el número de Reynolds es mayor que $5 \cdot 10^5$:

```
If ((Seccion Like "Circular") And (Re > 500000)) Then
```

La parte negativa de la estructura puede ser omitida, tal como muestra el siguiente ejemplo:

```
If (Presion > PresionLimite) Then
  MsgBox "La presión ha alcanzado el valor límite", _
         vbCritical, "Alerta Crítica"
  Abrir_Valvula_Seguridad
End If
```

Por último, existe la palabra clave `ElseIf` que permite evaluar otra condición en caso de que la primera no sea satisfecha. En el siguiente ejemplo se ilustra su uso:

```
If (Figura Like "Cuadrado") Then
  Area = Arista ^ 2
ElseIf (Figura Like "Circulo") Then
  Area = 3.1416 * Radio ^ 2
ElseIf (Figura Like "Elispe") Then
  Area = 3.1416 * SemiejeA * SemiejeB
Else
  MsgBox "Figura desconocida"
End If
```

Existen otras estructuras condicionales como `Select Case`, pero por su uso menor no se cubren en este material. El usuario interesado puede consultar la ayuda de VBA o un manual especializado.

3.10 – Estructuras repetitivas.

Las estructuras repetitivas permiten realizar acciones similares un número determinado de veces con las mismas instrucciones. Dentro de estas se destaca la sentencia `For...Next`, que tiene la sintaxis:

```
For <var> = <expres1> To <expres2> [Step <expres3>]
  [<sentencias>]
  [Exit For]
  [<sentencias>]
Next [<var>]
```

La variable del ciclo (llamada "`var`" en la sintaxis) controla el proceso. El ciclo se ejecuta desde el valor dado por la primera expresión (`expres1`) hasta el valor dado por la segunda expresión (`expres2`). La variable del ciclo aumenta su valor en cada iteración del lazo, aumentando cada vez en el valor dado por la tercera expresión (`expres3`). La instrucción `Exit For` es opcional y permite interrumpir la ejecución del lazo en un punto dado.

Por ejemplo, la función siguiente permite calcular el factorial del número que se le pase como argumento.

```
Function Factorial(Numero As Integer) As Integer
  ' Calcula el factorial de un número
  Dim I As Integer, C As Integer
  C = 1
  For I = 1 To Numero
    C = C * I
  Next I
  Factorial = C
End Function
```

En el mismo, `I` es la variable del ciclo; `C` es una variable para almacenar las sucesivas multiplicaciones que, como resultado, darán el factorial. El ciclo se ejecuta desde 1 hasta el valor del número pasado como parámetro. Como no se especifica el paso, éste será igual a 1.

Una variante del ciclo anterior, es la estructura `ForEach...Next`, la cual permite recorrer un arreglo o colección elemento a elemento. En el próximo capítulo será visto un ejemplo del uso de este tipo de lazo para recorrer las diversas entidades de un dibujo de AutoCAD.

Otras estructuras repetitivas son la `While...Wend` y la `Do...Loop`, cuya sintaxis y uso se pueden ver la ayuda.

Ejemplo Resuelto.

A modo de ejemplo, se hará una macro que permita calcular el volumen de tornillos, de tres tipos diferentes: de hexágono embutido (Allen), de cabeza hexagonal y de cabeza cilíndrica (ver Fig. 3.1).

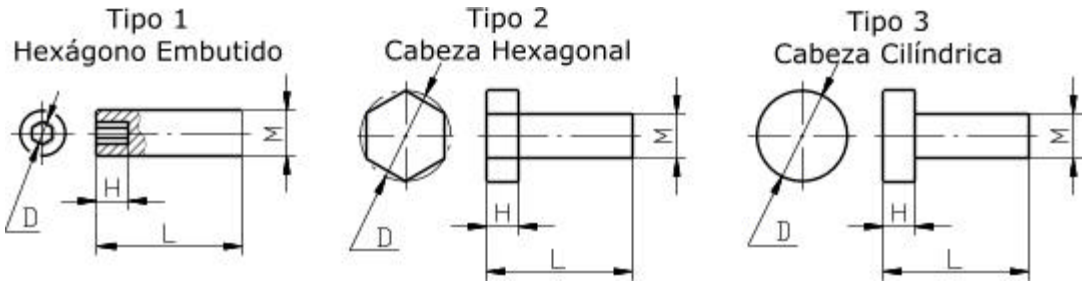


Fig. 3.1 – Dimensiones de los tornillos del ejemplo resuelto.

Para cada una de los tipos, se establecen las dimensiones D, H, L y M, tal como puede verse en la propia figura.

Un paso previo, muy importante, para desarrollar cualquier programa de computación es establecer su algoritmo. Para el problema considerado, el algoritmo en forma gráfica se muestra en la Fig. 3.2. Como se puede observar, hay tres tomas de decisiones, para realizar los cálculos correspondientes a cada tipo de tornillo.

La primera sentencia escrita en el código (exceptuando los comentarios) es `Option Explicit`, la cual indica que todas las variables utilizadas deberán ser declaradas.

A continuación, y para facilitar la escritura del programa, se declaran las constantes de tipo de tornillo que tomarán valores 1, 2 y 3, respectivamente.

Las ecuaciones para calcular el del cilindro y del prisma de base hexagonal son:

$$V_{CIL} = \frac{p}{4} d^2 \cdot h = 0{,}7854 \cdot d^2 \cdot h \text{; y}$$

$$V_{PRIS} = \frac{3}{4} d^2 \cdot h \cdot \text{sen}(60`) = 0{,}6495 \cdot d^2 \cdot h \text{;}$$

respectivamente. Como ambas expresiones se repiten en varios lugares del algoritmo, se han implementados las respectivas funciones, que han sido declaradas como privadas, o sea, accesibles sólo para este módulo.

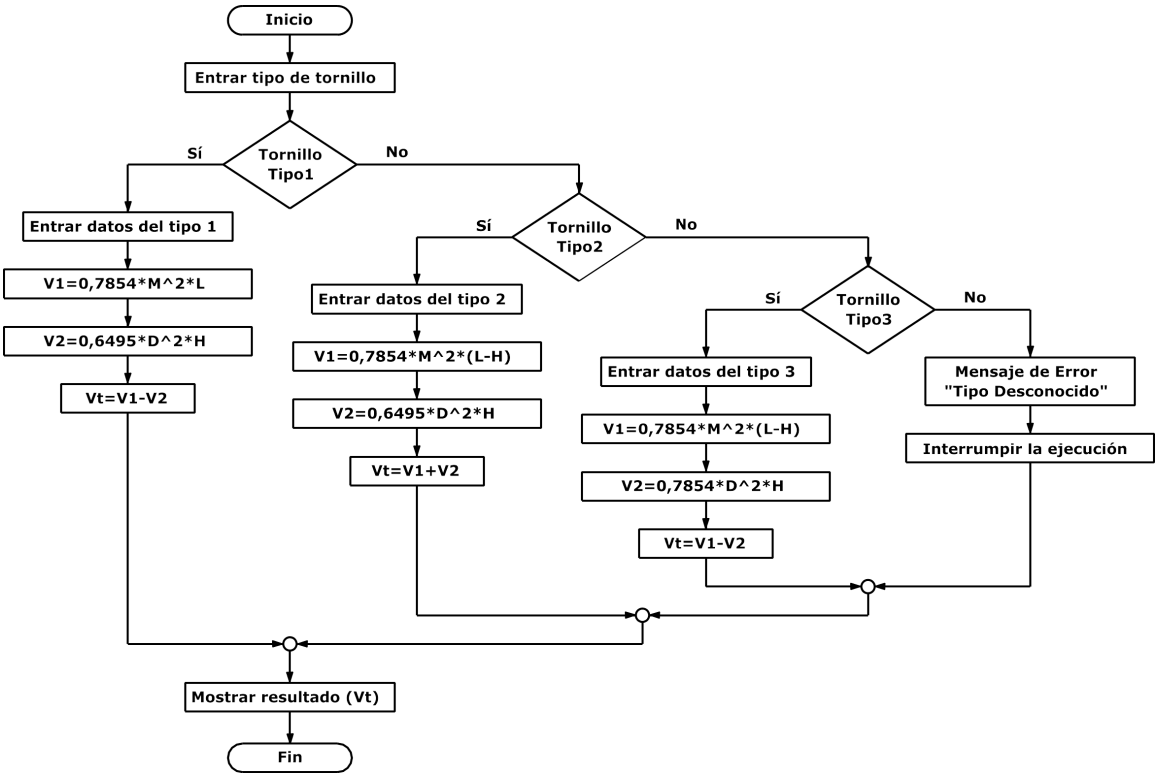


Fig. 3.2 - Representación gráfica del algoritmo de solución del ejemplo resuelto.

Código del ejemplo resuelto

```
'*******************************************************
'*  Macro para el cálculo del volumen de tornillos    *
'*  de diferentes tipos (ejemplo resuelto Cap. 3)     *
'*******************************************************

' Instrucción para forzar la declaración de variables
Option Explicit

' Constantes de tipo de tornillo
Const ttAllen As Integer = 1
Const ttCabHexagonal As Integer = 2
Const ttCabCilindrica As Integer = 3

' Función para calcular el volumen de un cilindro
Private Function VolCilindro(Dc As Double, _
                             Hc As Double) As Double
  ' Dc y Hc son el diámetro y la altura del cilindro
  VolCilindro = 0.7854 * Dc ^ 2 * Hc
End Function
```

```
' Función para calc. el volumen de un prisma hexagonal
Private Function VolPrisma(Dp As Double, _
                           Hp As Double) As Double
  ' Dp y Hp son el diámetro y la altura del cilindro
  VolPrisma = 0.6495 * Dp ^ 2 * Hp
End Function

' Macro para calcular el volumen de los tornillos
Public Sub VolTornillo()
  ' Declaración de las variables usadas
  Dim Tipo As Integer, M As Double, D As Double, _
      L As Double, H As Double, V1 As Double, _
      V2 As Double, Vt As Double
  ' Petición del tipo de tornillos
  Tipo = ThisDrawing.Utility.GetInteger("Indique " & _
         "el tipo de tornillo (1=Allen; 2=Cabeza " & _
         "Hexagonal; 3=Cabeza Cilíndrica): ")
  If (Tipo = ttAllen) Then
    ' Tornillo Allen
    M = ThisDrawing.Utility.GetReal("Diámetro " & _
          "del tornillo: ")
    L = ThisDrawing.Utility.GetReal("Longitud " & _
          "del tornillo: ")
    D = ThisDrawing.Utility.GetReal("Longitud " & _
          "entre aristas del : ")
    M = ThisDrawing.Utility.GetReal("Altura " & _
          "del hexágono embutido: ")
    V1 = VolCilindro(M, L)
    V2 = VolPrisma(D, H)
    Vt = V1 - V2
  ElseIf (Tipo = ttCabHexagonal) Then
    ' Tornillo de Cabeza Hexagonal
    M = ThisDrawing.Utility.GetReal("Diámetro " & _
          "del tornillo: ")
    L = ThisDrawing.Utility.GetReal("Longitud " & _
          "del tornillo: ")
    D = ThisDrawing.Utility.GetReal("Longitud " & _
          "entre aristas de la cabeza: ")

    M = ThisDrawing.Utility.GetReal("Altura de " & _
          "la cabeza: ")
    V1 = VolCilindro(M, L - H)
    V2 = VolPrisma(D, H)
    Vt = V1 + V2
  ElseIf (Tipo = ttCabCilindrica) Then
    ' Tornillo de Cabeza Cilíndrica
    M = ThisDrawing.Utility.GetReal("Diámetro " & _
          "del tornillo: ")
```

```
    L = ThisDrawing.Utility.GetReal("Longitud " & _
          "del tornillo: ")
    D = ThisDrawing.Utility.GetReal("Diámetro " & _
          "de la cabeza: ")
    M = ThisDrawing.Utility.GetReal("Altura de " & _
          "la cabeza: ")
    V1 = VolCilindro(M, L - H)
    V2 = VolCilindro(D, H)
    Vt = V1 + V2
  Else
    ' Tipo desconocido
    ThisDrawing.Utility.Prompt "Tipo de tornillo " & _
      "desconocido" & Chr(13) & Chr(10) & _
      "Ejecución abortada"
    Exit Sub
  End If
  ' Impresión de la respuesta
  ThisDrawing.Utility.Prompt "Volumen del tornillo:" &_
      Vt & " unidades cúbicas"
End Sub
```

En estas funciones, se han introducidos simplificaciones, ya que ni el cuerpo ni la cabeza del tornillo tienen la forma cilíndrica o prismática hexagonal realmente. No obstante, como ejemplo, es aceptable esta simplificación.

La macro implementada recibió por nombre `VolTornillo`. En ella se declararon las variables `Tipo` (guardará el tipo de tornillo, auxiliado por las constantes ya declaradas), `M` (el diámetro nominal de la rosca del tornillo), `L` (la longitud del tornillo), `D` (el diámetro o la distancia entre aristas de la cabeza o el hexágono embutido, según corresponda), `H` (la altura de la cabeza o el hexágono embutido), `V1` (el volumen del cuerpo del tornillo), `V2` (el volumen de la cabeza o el hexágono embutido) y `Vt` (el volumen total).

Para la toma de datos, se emplean los métodos `ThisDrawing.Utility.GetInteger` y `ThisDrawing.Utility.GetInteger`, que permiten obtener un número entero o real, respectivamente, desde la consola de comandos de AutoCAD. El uso detallado de estas utilidades y otras similares será explicado posteriormente.

Luego de obtenido el tipo de tornillo, se evalúa la primera condición, o sea, si el tipo de tornillo es Allen. Si esta se cumple, se solicitan los datos necesarios y se calculan los volúmenes `V1` y `V2`, según las expresiones correspondientes, y el volumen total por la diferencia de estos.

Si la primera condición no se cumple, se evalúa la segunda (que el tipo de tornillo sea de cabeza hexagonal). Si esta se satisface, se toman los datos correspondientes y se calculan los volúmenes `V1` y `V2`, así como el volumen total que es la suma de ambos.

Si tampoco esta segunda condición se satisface, se evalúa la tercera (que el tipo de tornillo sea de cabeza cilíndrica). Si se cumple, entonces se procede de forma similar a la anterior, aunque con los datos y las expresiones correspondientes.

Finalmente, si tampoco esta tercera condición se satisface, se envía un mensaje de error (con la herramienta `ThisDrawing.Utility.Prompt`) y se termina le ejecución de la macro (con la instrucción `Exit Sub`).

Como resultado final, se envía un mensaje a la consola de comandos indicando el valor del volumen total calculado.

Preguntas y Ejercicios Propuestos.

1. Identifique ejemplos, dentro de su campo profesional, de información que pudiera representarse como variables de diferentes tipos de datos.

2. Ponga ejemplos de posibles usos de estructuras condicionales y ciclos repetitivos dentro de los problemas cotidianos en su práctica profesional.

3. Modifique el código del ejemplo anterior para garantizar que no se introduzcan datos incorrectos. Por ejemplo que la distancia entre aristas del hexágono embutido sea menor que el diámetro del tornillo.

4. Escriba una macro que, dadas coordenadas (x, y) de tres puntos diga si están sobre una línea recta o no.

CAPÍTULO 4
Fundamentos de Automatización ActiveX

OBJETIVO

Describir las principales características de la Automatización ActiveX y del Modelo de Objetos de AutoCAD.

4.1 – Introducción.

La tecnología de ActiveX[1] permite manipular completamente AutoCAD mediante programación, bien sea desde el propio programa o desde otro entorno de programación que soporte esta tecnología. El fundamento de la misma es tener un conjunto de objetos que permitan controlar el programa y que estos objetos sean accesibles desde otras aplicaciones.

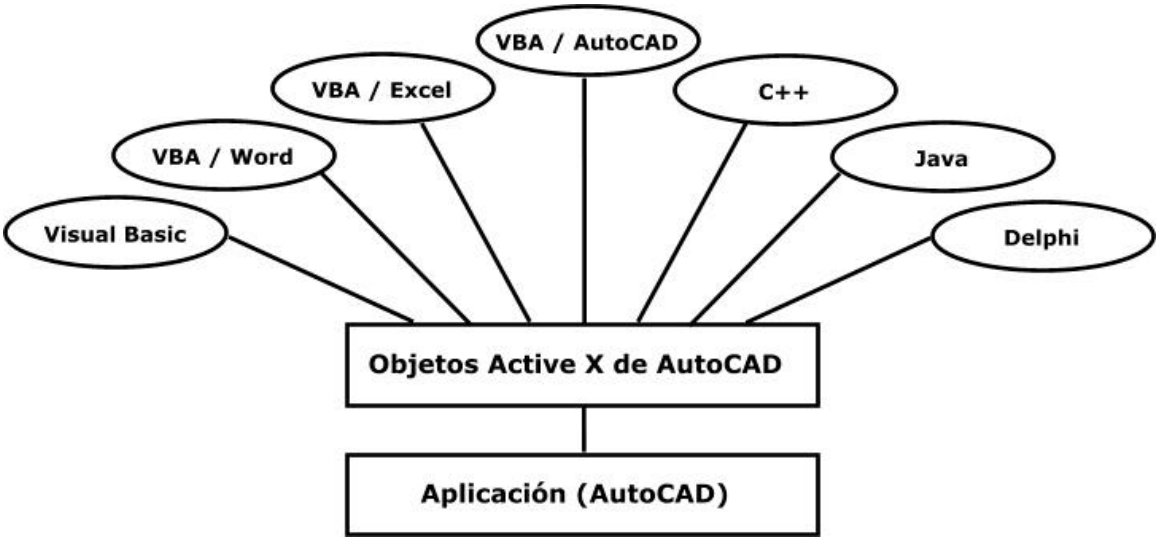


Fig. 4.1 – Esquema de la integración de ActiveX.

La inclusión de una interfaz de ActiveX para AutoCAD presenta dos ventajas básicas:

- Es posible acceder mediante programación a los dibujos de AutoCAD, lo cual puede realizarse desde diferentes entornos (incluyendo Visual Basic for Applications – VBA –, Visual Basic Estándar, Visual Studio – la plataforma .NET inclusive –, y Borland Delphi).
- Se facilita el intercambio de información con otras aplicaciones de Windows, tales como Microsoft Excel, Microsoft Word, etc.

[1] ActiveX es una marca registrada de Microsoft Corporation.

Los objetos constituyen el bloque de integración principal de todas las aplicaciones ActiveX. Cada objeto expuesto representa un componente concreto de AutoCAD. Existen multitud de tipos de objetos diferentes en la interfaz de ActiveX de AutoCAD, incluyendo los elementos gráficos como las líneas, los arcos, el texto y las cotas; parámetros de estilo como el tipo de línea y el estilo de cota; estructuras de organización como las capas, los grupos y los bloques; las pantallas de dibujo, como vistas y ventanas gráficas; e, incluso, el propio dibujo y la aplicación AutoCAD.

4.2 – El modelo de objetos de AutoCAD.

Para hacer un uso eficaz de ActiveX de AutoCAD, es muy conveniente estar familiarizado con los objetos, entidades y funciones de AutoCAD relativos al tipo de aplicación que esté desarrollando.

Como ya se dijo anteriormente, los objetos son la espina dorsal de la tecnología ActiveX. Todos los elementos de AutoCAD, tanto en los dibujos, como en la propia aplicación, son representados mediante los correspondientes objetos.

Los objetos se estructuran de forma jerárquica, relacionándose objetos de nivel superior con otros de nivel inferior. Por ejemplo el objeto que representa un dibujo puede contener varios objetos que representen líneas, arcos, textos, etc. A esta estructura jerárquica se le conoce como Modelo de objetos. En la Fig. 4.2 se puede observar un diagrama del Modelo de objetos de AutoCAD.

Dentro del modelo de objetos, hay algunos de gran importancia, por lo que se describirán brevemente. Estos son:

- `Application`: Es la raíz de todo el Modelo de Objetos de AutoCAD y permite acceder a cualquiera de los demás objetos que componen el Modelo. Los métodos del objeto `Application` permiten realizar acciones específicas de la aplicación, como enumeración, carga y descarga de aplicaciones ADS y ARX, así como salir de AutoCAD. Contiene otros objetos que permiten acceder a las preferencias (objeto `Preferences`), a los documentos de trabajo (colección `Documents`), a las barras de menús y de herramientas (colecciones `MenuBar` y `MenuGroups`), y al IDE de VBA (objeto `VBE`), entre otras. El objeto `Application` también es el objeto global de la interfaz de ActiveX. Es decir, todos los métodos y propiedades del objeto Application están disponibles en el espacio de nombres global.

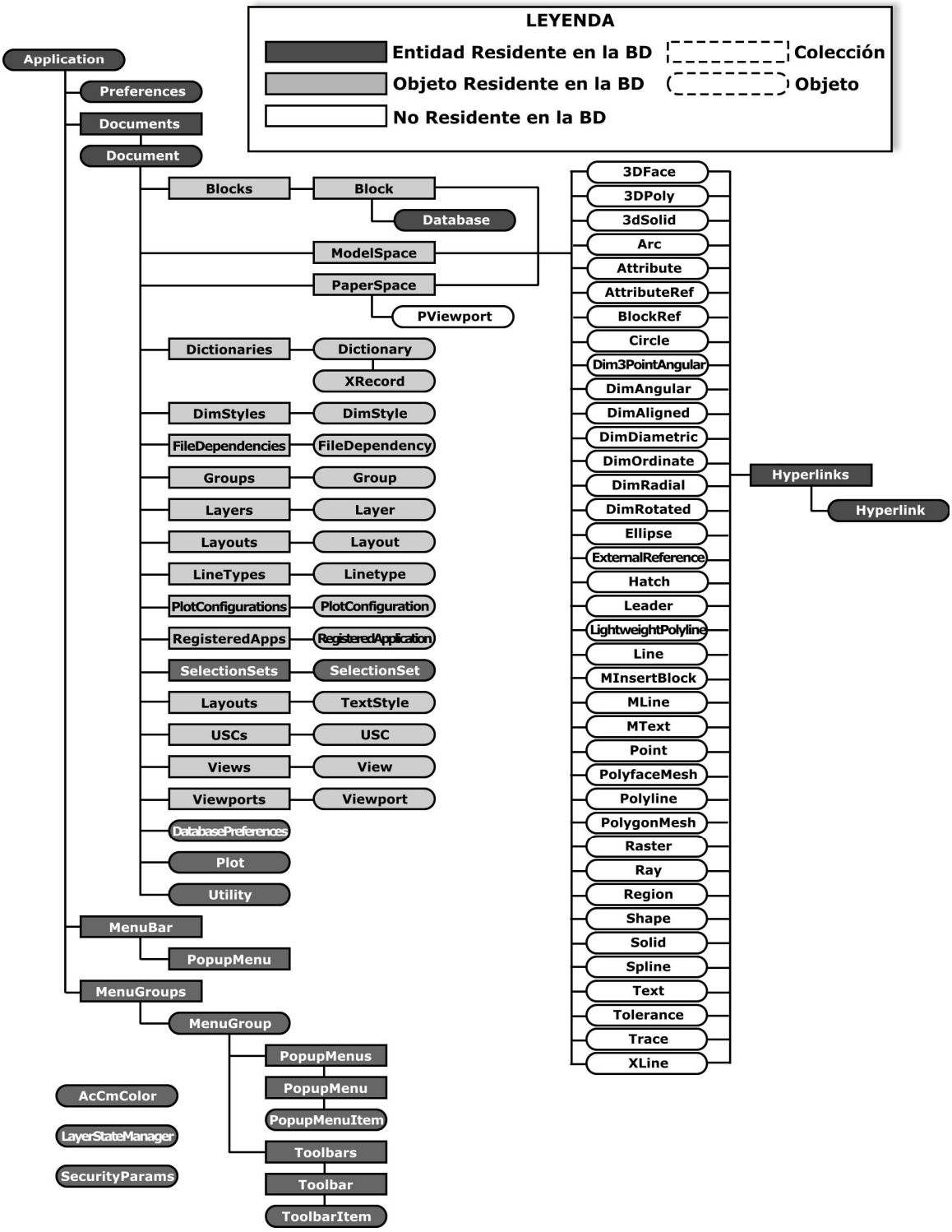


Fig. 4.2 – Modelo de Objetos de AutoCAD

- `Documents`: Es una colección de objetos de tipo `Document`, cada uno de los cuales contiene la información y los métodos para manipular los dibujos abiertos en

AutoCAD. Proporciona acceso a los objetos gráficos y muchos de los que no són gráficos también. El acceso a los objetos gráficos (líneas, círculos, arcos, etc.) se realiza a través de las colecciones `ModelSpace` y `PaperSpace`, mientras que el acceso a los objetos no gráficos (capas, tipos de línea, estilos de texto, etc.) se realiza a través de colecciones del mismo nombre, como `Layers`, `Linetypes` y `TextStyles`. El objeto `Document` también proporciona acceso a los objetos `Plot` y `Utility`.

- *Objetos gráficos*: También conocidos como entidades, incluyen los objetos visibles (líneas, círculos, cotas, imágenes, etc.) que conforman un dibujo. Cada objeto gráfico contiene métodos que permiten realizar la mayoría de las acciones de edición sobre ellos. También cuentan con métodos para establecimiento y acceso a datos extendidos (*xdata*), selección y actualización. Poseen propiedades típicas, como capa, tipo de línea y color y también propiedades específicas como pueden ser centro, radio y área.

- *Objetos no gráficos*: Son aquellos objetos invisibles que se incluyen para propósitos de información. Dentro de ellos tenemos las capas (`Layers`), los tipos de líneas (`Linetypes`), los estilos de acotado (`DimStyles`) entre otros. Para modificar o consultar estos objetos, se usan los métodos o propiedades del mismo. También cuentan con métodos para establecer y acceder a datos extendidos (*xdata*).

- `Preferences`: Incluye un conjunto de objetos que corresponden a las fichas del diálogo `Options`, de AutoCAD. Estos objetos permiten acceder y modificar todas las opciones guardadas por AutoCAD en el registro. Los elementos de configuración almacenados en el dibujo están contenidos en el objeto `DatabasePreferences`. También existen los métodos `SetVariable` y `GetVariable` que permiten modificar y consultar variables del sistema que no forman pare directamente del cuadro de diálogo `Options`.

- `Plot`: Proporciona acceso a la configuración de la impresión y ofrece herramientas para imprimir los dibujos según diversos métodos.

- `Utility`: Ofrece herramientas para la entrada y la conversión de datos. Las funciones de entrada son aquellas que interaccionan con el usuario a través de la consola de comandos de AutoCAD, solicitando datos como textos, valores numéricos enteros reales, puntos etc. Los métodos de conversión operan sobre tipos de datos específicos de AutoCAD, tales como puntos y ángulos.

El acceso a la jerarquía de objetos es muy simple en VBA, ya que éste corre junto con AutoCAD y, por lo tanto, no es necesario realizar ningún paso previo para interconectarlos.

VBA proporciona un vínculo al objeto activo en la sección actual de AutoCAD mediante el objeto `ThisDrawing`. A través del mismo se puede acceder al objeto `Document`

que corresponde a dicho dibujo activo, a todos sus métodos y propiedades y a todos los otros objetos del modelo que descienden de él.

En los proyectos globales, `ThisDrawing` siempre se refiere al dibujo activo en AuoCAD; por su parte, en los proyectos incrustados, se refiere al dibujo que contiene al proyecto.

4.3 – Colecciones.

Las colecciones (*collections*) son objetos que contienen varias instancias de un mismo tipo de objeto. Por ejemplo, en el modelo de objetos de AutoCAD existen, entre otras, las siguientes colecciones:

- `Documents`: Contiene a todos los objetos de tipo `Document` que hacen referencia a cada uno de los dibujos abiertos por AutoCAD.
- `ModelSpace`: Contiene a todos los objetos gráficos (entidades) en el espacio de modelo.
- `PaperSpace`: Contiene a todos los objetos gráficos (entidades) en el espacio de papel activo.
- `Block`: Contiene a todas las entidades dentro de un bloque.
- `Blocks`: Contiene a todos los bloques en un dibujo.
- `DimStyles`: Contiene a todos los estilos de acotado.
- `Layers`: Contiene a todas las capas del dibujo.
- `MenuBar`: Contiene a todos los menús mostrados en ese instante.
- `MenuGroups`: Contiene a todos los menús y barras de herramientas que se encuentren cargados en ese instante.

La mayor parte de las colecciones son accesibles a través del objeto `Document`, el cual contiene una propiedad que corresponde a cada objeto colección. Por ejemplo, el siguiente código define una variable y le asigna la colección de las capas del dibujo activo.

```
Dim MisCapas As AcadLayers
Set MisCapas = ThisDrawing.Layers
```

Las colecciones `Documents`, `MenuBar` y `MenuGroups` son accesibles a través del objeto `Application`, el cual contiene una propiedad para cada una de estas colecciones. El siguiente código muestra como definir una variable y asignarle la colección `MenuGroups`.

```
Dim MisMenus As AcadMenuGoups
Set MisMenus = ThisDrawing.Application.MenuGroups
```

Para adicionar un Nuevo miembro a una colección se emplea el método `Add`. El siguiente código muestra como crear una nueva capa y asignársela a una variable (`MiCapa`).

```
Dim MiCapa As AcadLayer
Set MiCapa = ThisDrawing.Layers.Add("NuevaCapa")
```

Para acceder a un miembro específico de un objeto colección se utiliza el método `Item`. Éste requiere de un índice numérico que identifica la posición del elemento dentro de la colección. El ejemplo siguiente permite recorrer todos los elementos de la colección `Layers` y muestra un mensaje con todos los nombres de las capas.

```
Sub NombreCapas()
  ' Recorre la colección Layers y muestra un mensaje
  ' con el nombre de las capas
  Dim I As Integer, Msg As String
  Msg = ""
  For I = 0 To ThisDrawing.Layers.Count - 1
    Msg = Msg & ThisDrawing.Layers.Item(I).Name & vbCrLf
  Next I
  MsgBox Msg
End Sub
```

Para eliminar un elemento de una colección, se emplea el método `Delete`, del objeto que contiene al elemento que se desea eliminar, tal como muestra el siguiente ejemplo.

```
Dim LayerCotas As AcadLayer
Set LayerCotas = ThisDrawing.Layers.Item("Cotas")
LayerCotas.Delete
```

4.4 – Propiedades y métodos.

Un elemento muy importante dentro de la programación orientada a objetos son las propiedades y los métodos, a los cuales se ha hecho referencia aunque sin entrar en detalles.

Las propiedades y los métodos están asociados a cada objeto. Las propiedades describen aspectos del objeto individualmente considerado, mientras que los métodos son acciones que pueden realizarse sobre él. La obtención de información a partir del objeto, y su modificación, son posibles a través del acceso a sus propiedades y métodos.

Para ilustrar lo anterior se puede considerar el objeto `Circle` (círculo). Éste tiene una propiedad llamada `Center` (centro), la cual representa las coordenadas tridimensionales del punto central de la circunferencia, dadas para el Sistema de Coordenada Universal. Para modificar el centro de un círculo, solamente hay que establecer esta propiedad en el valor de las nuevas coordenadas.

```
Circulo.Center = NuevoCentro
```

El objeto `Circle` posee, además, un método llamado `Offset` (sobreimpresión), el cual crea un nuevo objeto a una determinada distancia del círculo existente.

```
OtroCirculo = Circulo.Offset(20)
```

Para ver el listado completo de las propiedades de un objeto cualquiera, dentro del modelo de objetos de AutoCAD, se puede referir a la ayuda o utilizar el Explorador de Objetos (*Object Browser*), disponible a través del menú *View* del IDE de VBA.

Ejemplo Resuelto.

Se necesita crear, desde programación una nueva capa llamada "Azul", a la cual se le asignará color azul. A continuación todas las entidades existentes en el dibujo serán pasadas a la capa recién creada.

```
Public Sub CrearCapaAzul()
  ' Definición de variables
  Dim CapaAzul As AcadLayer
  Dim MiEntidad As AcadEntity
  Dim ColorAzul As AcadAcCmColor
  ' Creando e inicialiando la variable de color
  Set ColorAzul = AcadApplication.GetInterfaceObject( _
                    "AutoCAD.AcCmColor.16")
  ColorAzul.SetRGB 0, 0, 255
  ' Creando la capa y asignandole el color azul
  Set CapaAzul = ThisDrawing.Layers.Add("Azul")
  CapaAzul.TrueColor = ColorAzul
  ' Recorriendo los elementos del espacio de
  '   modelo y cambiando la capa
  For I = 0 To ThisDrawing.ModelSpace.Count - 1
    Set MiEntidad = ThisDrawing.ModelSpace.Item(I)
    MiEntidad.Layer = "Azul"
  Next I
  ' Regenerando el dibujo
  ThisDrawing.Regen acAllViewports
End Sub
```

El primer paso consiste en declarar tres variables llamadas `CapaAzul`, de tipo `AcadLayer` y que contendrá a la capa que será creada; `MiEntidad`, de tipo `AcadEntity` y que contendrá a cada una de las entidades presentes en el espacio de modelo según se recorran; y `ColorAzul`, de tipo AcadAcCmColor y que contiene el color que le asignaremos a la capa.

A continuación, se crea el color como una instancia de la interfaz `AutoCAD.AcCmColor`. Luego se le asigna el valor del color con el método `SetRGB`.

La nueva capa se crea con el método `Add` de la colección `Layers` y se le asigna al objeto `CapaAzul`. Seguidamente a la propiedad `TrueColor` de `CapaAzul`, se le asigna el objeto `ColorAzul`. Con esta acción queda creada la capa y asignado el color azul.

La segunda parte consiste en un ciclo repetitivo que recorre la colección `ModelSpace`. Dentro del ciclo, cada una de las entidades que forma el espacio de modelo es asignada a la variable `MiEntidad`. Luego, a la propiedad `Layer` se le asigna el valor "Azul", que es el nombre de la capa creada.

Finalmente, se regenera el dibujo mediante el método `Regen` del objeto `ThisDrawing`.

Preguntas y Ejercicios Propuestos.

1. ¿Qué ventajas presenta el uso de la tecnología ActiveX en AutoCAD?

2. ¿Qué representa el objeto `ThisDrawing`? ¿Qué diferencia hay en este objeto en los proyectos globales e incrustados?

3. Elabore una macro que permita mostrar, mediante mensajes en pantalla, los nombres de las capas que existen en el dibujo activo.

CAPÍTULO 5
Creación de Entidades

OBJETIVO

Crear entidades elementales en el espacio de modelo de AutoCAD mediante programación de VBA.

5.1 – Consideraciones Generales.

Las entidades gráficas de AutoCAD, normalmente, se crean en el espacio de modelo, aunque pueden crearse en bloques, o en los espacios de papel. Una diferencia sustancial entre la forma de crear entidades en el entorno de AutoCAD y hacerlo desde programación es que, mientras en el primero usualmente hay varias formas de crear un objeto, desde programación sólo hay un método para cada objeto.

Por ejemplo, para crear un arco, desde AutoCAD, podemos hacerlo de varias maneras: especificando su centro, su punto de origen y su punto final; especificando su centro, su origen y el ángulo barrido; especificando su centro, su origen y la longitud de la cuerda; especificando el origen, el final y un punto intermedio; etc. Sin embargo, desde programación, la única forma de hacerlo es mediante el método `AddArc`, cuyos parámetros son: el punto central, el radio, el ángulo inicial y el ángulo final.

5.2 – Creación de entidades.

Entidades lineales.

Prácticamente la totalidad de los dibujos de AutoCAD contienen algún elemento lineal. Las técnicas de creación de líneas desde VBA se basan en la especificación de las coordenadas de sus vértices. Existen cuatro métodos para crear entidades lineales:

- `AddLine`: Crea un segmento de recta mediante la especificación de las coordenadas se sus extremos.
- `AddLightweightPolyline`: Crea una polilínea ligera bidimensional a través de las coordenadas de sus vértices.
- `AddMLine`: Crea una multilínea.
- `AddPolyline`.: Crea una polilínea que puede ser bidimensional o tridimensional.

Un aspecto importante en la creación de entidades es la forma en que se especifican las coordenadas de los puntos que se usan para definirla (por ejemplo, los puntos extremos de un segmento de recta). Generalmente, estas coordenadas se dan como un arreglo

(array) de tipo `Double`, que tiene tantos elementos como coordenadas se necesitan. Para ilustrar lo anterior, se muestra un ejemplo de código que crea una línea desde el punto (0, 0, 0) hasta el punto (100, 100, 0).

```
Public Sub Linea()
  ' Declaración de las variables
  Dim Punto1(0 To 2) As Double
  Dim Punto2(0 To 2) As Double
  Dim Linea As AcadLine
  ' Inicialización de los puntos extremos
  Punto1(0) = 0: Punto1(1) = 0: Punto1(2) = 0
  Punto2(0) = 100: Punto2(1) = 0: Punto2(2) = 0
  ' Creación de la línea
  Set Linea = ThisDrawing.ModelSpace.AddLine(Punto1, _
                Punto2)
End Sub
```

Nótese que para las variables `Punto1` y `Punto2`, los elementos de subíndices 0, 1 y 2 representan las coordenadas *x*, *y* y *z*, respectivamente.

En otros casos, como al adicionar polilíneas, todas las coordenadas se almacenas en la misma variable de arreglo. En el siguiente ejemplo se muestra el código para crear una polilínea que forme un triángulo rectángulo.

```
Public Sub Polinea()
  ' Declaración de las variables
  Dim Puntos(0 To 11) As Double
  Dim Polilinea As AcadPolyline
  ' Inicialización de los vértices
  Puntos(0) = 0: Puntos(1) = 0: Puntos(2) = 0
  Puntos(3) = 40: Puntos(4) = 0: Puntos(5) = 0
  Puntos(6) = 40: Puntos(7) = 30: Puntos(8) = 0
  Puntos(9) = 0: Puntos(10) = 0: Puntos(11) = 0
  ' Creación de la polilínea
  Set Polilinea = ThisDrawing.ModelSpace. _
        AddPolyline(Puntos)
End Sub
```

Aquí todas las coordenadas se almacenan en la variable `Puntos`, agrupando los índices de tres en tres, de forma que los índices 0, 1 y 2 representan las coordenadas del primer vértice; los índices 3, 4 y 5, las coordenadas del segundo vértice; y así sucesivamente.

Entidades curvas.

Otro grupo importante de elementos, dentro de los dibujos más comunes en AutoCAD, lo constituyen las entidades curvas (arcos, circunferencias, elipses...). Para crear estas entidades, el modelo de objetos de AutoCAD dispone de los siguientes métodos:

- `AddCircle`: Crea una circunferencia dados el punto central y el radio.
- `AddArc`: Crea un arco de circunferencia, especificando el punto central, el radio, el ángulo inicial y el ángulo final.
- `AddEllipse`: Crea una elipse dados su centro, un punto extremo del eje principal y la relación entre las longitudes de los ejes.
- `AddSpline`: Crea una curva de interpolación (NURBS) cuadrática o cúbica, mediante la indicación de los puntos por los cuales pasa la curva y dos vectores que especifican la tangencia de los puntos inicial y final.

El siguiente código ejemplifica como trazar dos arcos de circunferencia para formar una "s", tal como lo muestra la Fig. 5.1. Como se puede ver, los centros tienen la misma coordenada *x*, y una diferencia de dos radios en la coordenada *y*. El ángulo inicial del primer arco será igual a –90º (todos los ángulos se dan con respecto al eje o-x), coincidiendo con ángulo final del segundo arco. Por su parte, el ángulo final del primer arco será, lo mismo que el ángulo inicial del segundo, igual a 90º.

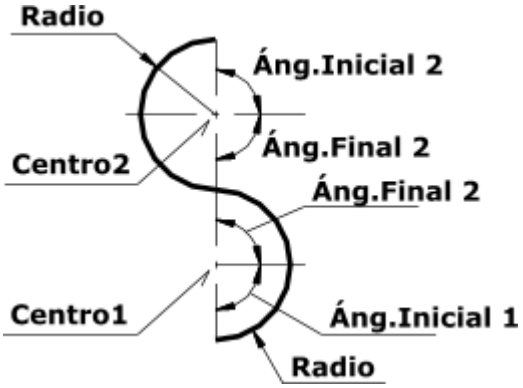


Fig. 5.1 – Representación gráfica del ejemplo de trazado de curvas.

Tomando como radio un valor de 10 unidades, y situando las coordenadas del centro del primer arco en el punto (50, 20, 0), tenemos que las coordenadas del segundo centro serán iguales a (50, 40, 0). Los valores de los ángulos se determinarán en radianes. Con estos datos, se construye el código:

```
Public Sub Arcos()
  ' Declaración de variables
  Dim Punto1(0 To 2) As Double
  Dim Punto2(0 To 2) As Double
  Dim Radio As Double
  Dim Angulo1 As Double, Angulo2 As Double
  Dim Arco1 As AcadArc, Arco2 As AcadArc
  ' Inicialización de los puntos centrales
  Punto1(0) = 50: Punto1(1) = 20: Punto1(2) = 0
  Punto2(0) = 50: Punto2(1) = 40: Punto2(2) = 0
  ' Inicialización del radio y los ángulos
  Radio = 10
  Angulo1 = 3.1416 * (-90) / 180
  Angulo2 = 3.1416 * (90) / 180
  ' Creación de el primer arco
  Set Arco1 = ThisDrawing.ModelSpace.AddArc(Punto1, _
            Radio, Angulo1, Angulo2)
```

```
  ' Creación de el segundo arco
  Set Arco2 = ThisDrawing.ModelSpace.AddArc(Punto2, _
            Radio, Angulo2, Angulo1)
End Sub
```

Regiones.

Las regiones son áreas bidimensionales y cerradas, que pueden crearse a partir de curvas cerradas llamadas lazos. Un lazo es una curva o una secuencia de curvas conectadas que definen un área en un plano, de forma tal que su frontera no se intercepta a sí misma. Los lazos pueden ser combinaciones de líneas, polilíneas, círculos, arcos, elipses, arcos elipses u otros elementos, que formen un área cerrada y que estén situados en el mismo plano.

Para crear una región, se usa el método `AddRegion`, el cual crea una región que bordea los contornos de los lazos pasados como entradas. Estos lazos se pasan como un arreglo de curvas. Como resultado del método `AddRegion`, se obtiene una variable `Variant`, que contiene, luego de la aplicación del método, un arreglo formado por tantos elementos como regiones fueron creadas. Posteriormente, los elementos de este arreglo deberán ser asignados a variable de tipo `AcadRegion` para su ulterior tratamiento.

A las regiones se les puede aplicar cualquiera de las operaciones booleanas unión, intersección o sustracción, utilizando el método `Boolean` proporciona el objeto `AcadRegion`.

En el siguiente ejemplo, se crean dos círculos que se intersecan, con sus centros en los puntos (100, 100, 0) y (120, 0, 0). El radio de ambos círculos es de 20 unidades. Los dos son convertidos a regiones las cuales, posteriormente son unidas en una única región, mediante la operación booleana unión. Finalmente, los círculos que sirvieron para la creación de las regiones son eliminados, a la región se le asigna color turquesa (cyan) y se muestra un mensaje indicando el área de la región creada.

```
Public Sub Regiones()
  ' Declaración de variables
  Dim Circulos(0 To 1) As AcadCircle
  Dim Centro1(0 To 2) As Double
  Dim Centro2(0 To 2) As Double
  Dim Radio As Double
  Dim Regiones As Variant
  Dim Region1 As AcadRegion
  Dim Region2 As AcadRegion
  Dim RegionTotal As AcadRegion
  ' Inicialización del radio y los centros
  Radio = 20
  Centro1(0) = 100: Centro1(1) = 100: Centro1(2) = 0
  Centro2(0) = 120: Centro2(1) = 100: Centro2(2) = 0
```

```
  ' Creación de los círculos
  Set Circulos(0) = ThisDrawing.ModelSpace. _
                      AddCircle(Centro1, Radio)
  Set Circulos(1) = ThisDrawing.ModelSpace. _
                      AddCircle(Centro2, Radio)
  ' Creación de las regiones
  Regiones = ThisDrawing.ModelSpace.AddRegion(Circulos)
  Set Region1 = Regiones(0)
  Set Region2 = Regiones(1)
  ' Unión de ambas regiones
  Region1.Boolean acUnion, Region2
  ' Eliminación de los círculos
  Circulos(0).Delete
  Circulos(1).Delete
  ' Cambio de color de la región
  Region1.Color = acCyan
  ' Envío del mensaje mostrando el área de la region
  MsgBox "El área de la región es " & Region1.Area
End Sub
```

Rayados.

El proceso de creación de rayados desde programación puede parecer un poco raro al usuario no familiarizado. En primer lugar se crea un objeto de rayado (`Hatch`) con el método `AddHatch`. Luego, a este objeto, se le asigna los contornos donde se aplicará.

El método AddHatch tiene tres parámetros. El primero es el tipo de patrón, que puede tomar los valores definidos por las siguientes constantes:

- `acHatchPatternTypePredefined`: Seleccionará el nombre del patrón de los definidos en el archivo *acad.pat*.
- `acHatchPatternTypeUserDefined`: Define un patrón de líneas utilizando el tipo de línea actual.
- `acHatchPatternTypeCustomDefined`: Selecciona el patrón desde otro archivo que no sea *acad.pat*.

El segundo parámetro es el nombre del patrón de rayado a utilizar. Finalmente, el tercer parámetro es la asociatividad, la cual define si un rayado se reajusta o no, cuando se modifican los contornos a los cuales ha sido aplicado. Un rayado asociativo puede desasociarse, pero no viceversa.

Los contornos del rayado se asignan con los métodos `AppendOuterLoop` (para el contorno exterior) y `AppendInnerLoop` (para el contorno interior). Los contornos son siempre arreglos de entidades (`AcadEntity`), que pueden incluir líneas, polilíneas, circunferencia, arcos, regiones, etc.

En el código siguiente se crea un objeto de rayado, al cual se le establecen las propiedades ángulo de rayado (`PatternAngle`) y escala del rayado (`PatternScale`). Luego, se le asignan como contornos interior y exterior dos circunferencias previamente creadas, y se ejecuta el rayado.

```
Public Sub Rayado()
  ' Definición de variables
  Dim Rayado As AcadHatch
  Dim Punto(0 To 2) As Double
  Dim Contorno1(0 To 0) As AcadEntity
  Dim Contorno2(0 To 0) As AcadEntity
  ' Creación del objeto de rayado
  Set Rayado = ThisDrawing.ModelSpace.AddHatch( _
                 acHatchPatternTypePreDefined, _
                 "Line", False)
  ' Establecimiento de las propiedades del rayado
  Rayado.PatternAngle = 3.1416 / 4  ' (45 grados)
  Rayado.PatternScale = 2
  ' Definición del punto central de los círculos
  Punto(0) = 100: Punto(1) = 100: Punto(2) = 0
  ' Creación del contorno interior
  Set Contorno1(0) = ThisDrawing.ModelSpace. _
        AddCircle(Punto, 25)
  ' Creación del contorno exterior
  Set Contorno2(0) = ThisDrawing.ModelSpace. _
        AddCircle(Punto, 50)

  ' Asignación de los contornos al rayado
  Rayado.AppendOuterLoop Contorno2
  Rayado.AppendInnerLoop Contorno1
  ' Ejecución del rayado
  Rayado.Evaluate
End Sub
```

5.3 – Trabajo con objetos con nombre.

Además de los objetos gráficos, en AutoCAD existen diversos tipos de elementos no gráficos, los cuales, muchas veces, tienen designaciones asociados con ellos. En este caso están los bloques, las capas, los tipos de líneas, etc., poseen un nombre que los identifica. Todos los identificadores de nombres utilizados en un dibujo se almacenan en su tabla de símbolos. En realidad, cuando se especifica un objeto por su nombre, se está haciendo referencia al nombre y los datos asociados que existen en esta tabla de símbolos.

Una acción importante, que se debe realizar con frecuencia, es la purga de los objetos nombrados a los cuales no se hace referencia en el dibujo. Esto disminuye el tamaño de

los dibujos al liberar el espacio que estos objetos, no utilizados, empleaban. Por supuesto, es imposible purgar objetos que estén siento referenciados por otros.

Para purgar un dibujo, desde programación, se emplea el método `PurgeAll`, del objeto `ThisDrawing`:

```
ThisDrawing.PurgeAll
```

Según aumenta la complejidad de los dibujos, será necesario renombrar objetos para mantener cierto sentido en los nombres asignados y evitar conflictos con los nombres de otros dibujos o bloques que se inserten en el dibujo principal.

Cualquier objeto puede ser renombrado, excepto aquellos a los cuales AutoCAD les asigna un nombre con significado especial, tales como la capa *0* o el tipo de línea *Continuous*.

Los nombres pueden contener hasta 255 caracteres incluyendo letras, números y espacios en blanco (aunque AutoCAD elimina cualquier espacio en blanco antes y después del nombre). En cambio, los caracteres simbólicos (> < / \ “ ‘ , ; : ? ! * | =) o los especiales extendidos por las fuentes Unicode no se admiten.

Para renombrar un objeto se emplea la propiedad `Name` del mismo. En el siguiente ejemplo, se busca dentro de la colección de capas una nombrada “Ocultas” y si existe, su nombre se cambia a “Lineas Ocultas”.

```
Sub RenombrarCapa()
  Dim Capa As AcadLayer
  For Each Capa In ThisDrawing.Layers
    If Capa.Name Like "Ocultas" Then
      Capa.Name = "Lineas Ocultas"
    End If
  Next
End Sub
```

5.4 – Conjuntos de selección.

Un conjunto de selección es un grupo de objetos de AutoCAD que se especifican para ser procesados como una única unidad. La definición de un conjunto de selección se realiza en dos etapas: primero, se crea un nuevo conjunto y se agrega a la colección de conjuntos de selección. Luego, se adicionan al conjunto los objetos que formarán parte de él.

Para crear un nuevo conjunto de selección se utiliza el método `Add` del objeto colección `SelectionSets`, al cual se le pasa, como único parámetro, el nombre que se asignará al conjunto de selección. La mayor dificultad consiste en que, si existe ya un conjunto con el nombre especificado, VBA producirá un error en tiempo de ejecución y detendrá

la macro. Para evitar esto, es muy buena práctica de programación, verificar si ya existe un conjunto con el nombre a utilizar y si así es, eliminarlo utilizando el método `Delete` del propio conjunto, tal como se muestra más adelante en el ejemplo.

Para adicionar elementos a un conjunto de selección se puede utilizar cualquiera de los siguientes métodos:

- `AddItems`: Adiciona un elemento haciendo referencia a él, directamente por su nombre.

- `Select`: Selecciona uno o varios objetos y los adiciona al conjunto. La selección puede hacerse a través de un área rectangular (bien sea seleccionando todos los objetos que queden completamente, o parcialmente dentro del área), un área poligonal, una línea de selección, u otras opciones).

- `SelectAtPoint`: Adiciona al conjunto los objetos que pasen por un punto especificado.

- `SelectAtPolygon`: Adiciona los objetos que toquen una línea de selección.

- `SelectOnScreen`: Permite al usuario seleccionar directamente, sobre el área de trabajo, los objetos que desea adicionar al conjunto de selección.

Para hacer referencia a los objetos de una selección, se utiliza la propiedad `Item`, junto al índice específico. También, los objetos de conjunto de selección tienen una propiedad `Count` que almacena la cantidad de elementos que tiene la selección.

Para eliminar elementos de una selección se pueden emplear los métodos:

- `RemoveItems`: Elimina uno o más elementos del conjunto de selección. Los elementos eliminados de la selección, no obstante, continuan existiendo en el dibujo.

- `Clear`: Elimina todos los objetos de la selección, aunque todos continuan existiendo en el dibujo.

- `Erase`: Borra todo los elementos de una selección, tando de la propia selección como del dibujo.

- `Delete`: Borra todos los elementos de una selección, tanto de la propia selección como del dibujo, y elimina el conjunto de selección.

En el siguiente ejemplo se muestra como verificar si un conjunto de selección llamado “MiConjunto” existe en el dibujo y de ser así, eliminarlo. Luego se crea un nuevo conjunto de selección con este nombre, se le agregan todos los elementos situados dentro

de la ventana definida por las esquinas (0, 0, 0) y (200, 100, 0), y se cambia el color a rojo. Finalmente se muestra un mensaje informando cuantas entidades fueron modificadas, se vacía el conjunto de selección y se elimina.

```
Sub CrearConjuntoSeleccion()
  ' Crea un conjunto de selección, le adiciona todos
  ' las entidades comprendidas en el rectángulo desde
  ' (0, 0, 0) a (200, 100, 0), les cambia el color a
  ' rojo y muestra un mensaje informado la cantidad de
  ' objetos modificados.
  Dim MiSeleccion As AcadSelectionSet
  Dim I As Integer
  Dim Punto1(0 To 2) As Double, Punto2(0 To 2) As Double
  For I = 0 To ThisDrawing.SelectionSets.Count - 1
    If ThisDrawing.SelectionSets.Item(I).Name Like _
       "NuevaSeleccion" Then
      ThisDrawing.SelectionSets.Item(I).Delete
    End If
  Next I
  Set MiSeleccion = _
        ThisDrawing.SelectionSets.Add("NuevaSeleccion")
  Punto1(0) = 0: Punto1(1) = 0: Punto1(2) = 0
  Punto2(0) = 200: Punto2(1) = 100: Punto2(2) = 0
  MiSeleccion.Select acSelectionSetWindow, Punto1, _
                     Punto2
  For I = 0 To MiSeleccion.Count - 1
    MiSeleccion.Item(I).color = acRed
  Next I
  MsgBox "Fueron modificados " & MiSeleccion.Count & _
         "objeto(s).", vbInformation, _
         "Conjunto '" & MiSeleccion.Name & "'"
  MiSeleccion.Clear
  MiSeleccion.Delete
End Sub
```

5.5 – Edición de entidades.

La modificación de objetos mediante programación, se realiza utilizando los métodos que cada clase de objeto proporciona. Luego ejecutar un método que afecte las propiedades visuales de un objeto, se debe llamar al método `Update`, para actualizarlo en pantalla.

Copiado de objetos.

Para copiar un único objeto, se utiliza el método `Copy` del mismo. Éste, a diferencia del comando de copiado de AutoCAD, no mueve el objeto copiado de su posición original, sino que lo deja en la misma posición del objeto fuente.

Movimiento de objetos.

Se lleva a cabo con el método `Move`. Es necesario especificarle el punto de origen y el punto de destino. En el siguiente ejemplo, se muestra el copiado y movimiento de una circunferencia.

```
Sub CopiarMover()
  ' Crea una circunferencia y luego la copia y la mueve
  Dim CirculoOriginal As AcadCircle
  Dim CirculoCopia As AcadCircle
  Dim Punto1(0 To 2) As Double, Punto2(0 To 2) As Double
  Punto1(0) = 150: Punto1(1) = 100: Punto1(2) = 0
  Punto2(0) = 200: Punto2(1) = 100: Punto2(2) = 0
  Set CirculoOriginal = _
        ThisDrawing.ModelSpace.AddCircle(Punto1, 50)
  Set CirculoCopia = CirculoOriginal.Copy
  CirculoCopia.Move Punto1, Punto2
End Sub
```

Sobreimpresión de objetos.

La sobreimpresión de objetos permite reproducir las entidades que forman un objeto a una distancia determinada. Son susceptibles a sobreimpresión los siguientes objetos: arcos, círculos, elipses, líneas, polilíneas, splines y xlines.

La sobreimpresión se realiza con el método `Offset` que proporcionan estos objetos. El único parámetro que es necesario especificar en la distancia de sobre-impresión. Si la misma es positiva, el objeto será agrandado; si es negativa, entonces será achicado.

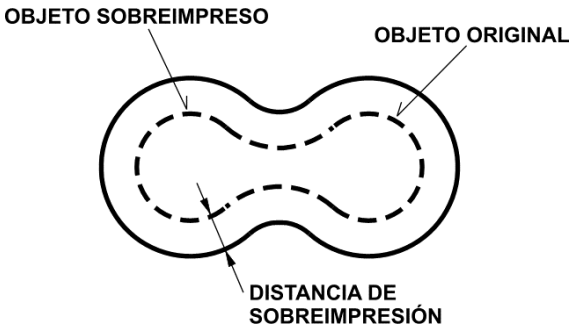


Fig. 5.2 – Sobreimpresión.

En el siguiente ejemplo, se muestra la creación de una polilínea cerrada y su posterior sobreimpresión a una distancia de 10 unidades.

```
Sub Sobreimpresion()
  ' Crea una polilínea cerrada y la sobreimprime
  Dim ObjOriginal As AcadPolyline
  Dim ObjNuevo As Variant
  Dim Puntos(0 To 20) As Double
  Puntos(0) = 100:  Puntos(1) = 100:  Puntos(2) = 0
  Puntos(3) = 200:  Puntos(4) = 100:  Puntos(5) = 0
```

```
  Puntos(6) = 200:  Puntos(7) = 150:  Puntos(8) = 0
  Puntos(9) = 175:  Puntos(10) = 150: Puntos(11) = 0
  Puntos(12) = 175: Puntos(13) = 125: Puntos(14) = 0
  Puntos(15) = 100: Puntos(16) = 125:   Puntos(17) = 0
  Puntos(18) = 100: Puntos(19) = 100: Puntos(20) = 0
  Set ObjOriginal = _
        ThisDrawing.ModelSpace.AddPolyline(Puntos)
  ObjNuevo = ObjOriginal.Offset(10)
  ObjNuevo(0).color = acRed
End Sub
```

Véase que el resultado del método `Offset` es una variable de tipo `Variant`. Esto se debe a que este método puede producir un conjunto de objetos en lugar de uno sólo.

Reflexión.

La reflexión, también llamada espejo, permite obtener la imagen de un objeto reflejada sobre una línea. Desde VBA se puede llevar a cabo con el método `Mirror`, del objeto que se desea reflejar, al cual es necesario pasarle, como parámetros, dos puntos que definan la línea de reflexión.

A diferencia del comando de AutoCAD, la reflexión desde VBA no permite eliminar los objetos fuentes, por lo que si se desea hacer, deberá emplearse el método `Delete`. A continuación se ofrece un ejemplo de la reflexión de una polilínea.

```
Sub Reflexion()
  ' Crea una polilínea y le aplica una reflexión
  Dim ObjOriginal As AcadPolyline
  Dim ObjNuevo As AcadPolyline
  Dim Puntos(0 To 20) As Double
  Dim Punto1(0 To 2) As Double, Punto2(0 To 2) As Double
  Puntos(0) = 100:  Puntos(1) = 200:  Puntos(2) = 0
  Puntos(3) = 80:   Puntos(4) = 180:  Puntos(5) = 0
  Puntos(6) = 80:   Puntos(7) = 160:  Puntos(8) = 0
  Puntos(9) = 90:   Puntos(10) = 150: Puntos(11) = 0
  Puntos(12) = 80:  Puntos(13) = 140: Puntos(14) = 0
  Puntos(15) = 80:  Puntos(16) = 120: Puntos(17) = 0
  Puntos(18) = 100: Puntos(19) = 100: Puntos(20) = 0
  Punto1(0) = 100:  Punto1(1) = 200:  Punto1(2) = 0
  Punto2(0) = 100:  Punto2(1) = 100:  Punto2(2) = 0
  Set ObjOriginal = _
        ThisDrawing.ModelSpace.AddPolyline(Puntos)
  Set ObjNuevo = ObjOriginal.Mirror(Punto1, Punto2)
  ObjNuevo.Color = acRed
End Sub
```

Arreglos.

Los arreglos permiten distribuir entidades sobre el espacio de trabajo en una forma regular determinada. Según esta distribución, los arreglos pueden ser rectangulares (cuando sus elementos están distribuidos en filas, columnas y capas) o polares (cuando están distribuidos en forma circular, alrededor de un punto central).

Para hacer un arreglo rectangular de un objeto, se utiliza el método `ArrayRectangular`, del propio objeto. Este método utiliza como parámetros la cantidad de filas, de columnas y de capas y la distancia entre ellas. Por supuesto, si el arreglo es bidimensional, entonces la cantidad de capas es 1.

A continuación se muestra un ejemplo de generación de un arreglo rectangular a partir de una circunferencia.

```
Sub ArregloRectangular()
  ' Crea un círculo y genera un arreglo
  ' rectangular a partir de él
  Dim Circulo As AcadCircle
  Dim CentroCirculo (0 To 2) As Double
  Dim CantFilas As Long, DistFilas As Double
  Dim CantCols As Long, DistCols As Double
  Dim CantCapas As Long, DistCapas As Double
  CantFilas = 6: DistFilas = 30
  CantCols = 3: DistCols = 60
  CantCapas = 1: DistCapas = 0
  CentroCirculo (0) = 100: CentroCirculo (1) = 100
  Set Circulo = ThisDrawing.ModelSpace.AddCircle( _
                  CentroCirculo, 10)
  Circulo.ArrayRectangular CantFilas, CantCols, _
              CantCapas, DistFilas, DistCols, DistCapas
End Sub
```

Por su parte, los arreglos polares se crean con el método `ArrayPolar`, al cual es necesario suministrarle, como parámetros, la cantidad de objetos que creará el arreglo (incluido el objeto original), el ángulo barrido y el punto central. A continuación se muestra el código para crear un arreglo polar a partir de una circunferencia.

```
Sub ArregloPolar()
  ' Crea un círculo y genera un arreglo
  ' polar que barre 180º, a partir de él
  Dim Circulo As AcadCircle
  Dim CentroCirculo(0 To 2) As Double
  Dim CentroArreglo(0 To 2) As Double
  Dim NoObjetos As Long
  Dim Angulo As Double
  NoObjetos = 6
```

```
  Angulo = 3.1416   ' 180º
  CentroCirculo(0) = 100: CentroCirculo(1) = 100
  CentroArreglo(0) = 100: CentroArreglo(1) = 150
  Set Circulo = ThisDrawing.ModelSpace.AddCircle( _
                 CentroCirculo, 10)
  Circulo.ArrayPolar NoObjetos, Angulo, CentroArreglo
End Sub
```

Rotación.

La rotación de un objeto se lleva a cabo con el método `Rotate`, el cual requiere, como argumentos, el punto alrededor del cual se va a rotar el objeto, y el ángulo de rotación. En el siguiente código, se muestra como rotar un rectángulo (formado por una polilínea) 30º alrededor de su esquina inferior izquierda.

```
Sub Rotacion()
  ' Crea una polilínea en forma de rectángulo
  ' y la rota 30º alrededor de la esquina
  ' inferior izquierda
  Dim Rectangulo As AcadPolyline
  Dim Puntos(0 To 14) As Double
  Dim PuntoBase(0 To 2) As Double
  Dim Angulo As Double
  Puntos(0) = 100:  Puntos(1) = 100:  Puntos(2) = 0
  Puntos(3) = 180:  Puntos(4) = 100:  Puntos(5) = 0
  Puntos(6) = 180:  Puntos(7) = 160:  Puntos(8) = 0
  Puntos(9) = 100:  Puntos(10) = 160: Puntos(11) = 0
  Puntos(12) = 100: Puntos(13) = 100: Puntos(14) = 0
  PuntoBase(0) = 100: PuntoBase(1) = 100
  Angulo = 3.1416 / 6
  Set Rectangulo = _
        ThisDrawing.ModelSpace.AddPolyline(Puntos)
  Rectangulo.Rotate PuntoBase, Angulo
End Sub
```

Escalado.

El escalado permite transformar las dimensiones lineales de un objeto, multiplicándolas por un factor de escala dado. Si utilizamos un factor de escala mayor que uno, el objeto obtenido será mayor que el original; si menor que uno, será menor. Por último, si usamos un factor de escala igual a uno, no se producirá ningún cambio (y podremos, razonablemente, poner en duda nuestra inteligencia).

Para ejecutar el escalado se utiliza el método `ScaleEntity` que proporciona el propio objeto. Éste requiere, como parámetros, que se le especifique el punto base (punto a partir del cual se recalcularán las nuevas coordenadas) y el factor de escala. Es bueno aclarar que el escalado se realiza en las tres coordenadas *x*, *y* y *z*.

En el siguiente ejemplo se muestra cómo escalar una entidad, en este caso, una polilínea que forma un triángulo rectángulo.

```
Sub Escalado()
  ' Crea una polilínea y la escala con factor 2
  Dim Points(0 To 11) As Double
  Dim PtoBase(0 To 2) As Double
  Dim Triangulo As AcadPolyline
  Points(0) = 50: Points(1) = 50: Points(2) = 0
  Points(3) = 90: Points(4) = 50: Points(5) = 0
  Points(6) = 90: Points(7) = 80: Points(8) = 0
  Points(9) = 50: Points(10) = 50: Points(11) = 0
  Set Triangulo = _
              ThisDrawing.ModelSpace.AddPolyline(Points)
  PtoBase(0) = 50: PtoBase(1) = 50: PtoBase(2) = 0
  Triangulo.ScaleEntity PtoBase, 2
End Sub
```

Extensión y tronzado.

Las acciones que se realizan en AutoCAD mediante extensión (`EXTEND`) y tronzado (`TRIM`), desde VBA se llevan a cabo a través de las propiedades del objeto. Por ejemplo, para modificar la longitud de una línea, tal como se muestra en el siguiente código, se cambian los valores del punto inicial (`StartPoint`) o del final (`EndPoint`).

```
Sub Redimensionado()
  ' Crea y redimensiona una línea
  Dim Punto1(0 To 2) As Double
  Dim Punto2(0 To 2) As Double
  Dim Linea As AcadLine
  Punto1(0) = 80: Punto1(1) = 50: Punto1(2) = 0
  Punto2(0) = 120: Punto2(1) = 70: Punto2(2) = 0
  Set Linea = ThisDrawing.ModelSpace.AddLine(Punto1, _
                                                Punto2)
  Punto2(1) = 90
  Linea.EndPoint = Punto2
  Linea.Update
End Sub
```

5.6 – Toma de datos del usuario.

Un tópico muy importante, dentro de la programación, lo constituye la toma de datos que establece el usuario. Para realizar estas tareas, VBA proporciona el objeto `Utility`, accesible a través de `ThisDrawing`. El objeto `Utility` contiene una variedad de

métodos para obtener información directamente desde la consola de comandos de AutoCAD; entre ellos se destacan:

- `GetReal`: Solicita un valor numérico real y lo devuelve como resultado. Requiere, como argumentos, el texto que será mostrado en la consola de commando para hacer la petición. Ejemplo:

```
Dim N As Double
N = ThisDrawing.Utility.GetReal( _
"Especifique un valor real: ")
MsgBox "El valor especificado es: " & N
```

- `GetInteger`: Solicita un valor numérico entero y lo devuelve como resultado. Requiere en texto que se mostrará en la consola. Ejemplo:

```
Dim M As Double
M = ThisDrawing.Utility.GetInteger( _
"Especifique un valor entero: ")
MsgBox "El valor especificado es: " & M
```

- `GetString`: Solicita una cadena de caracteres y la devuelve como resultado. Requiere dos parámetros: el primero es valor booleano que establece si el texto especificado puede o no contener espacios vacíos; el segundo es el texto mostrado en la consola de comandos. Ejemplo:

```
Dim S As String
S = ThisDrawing.Utility.GetString( _
True, "Especifique un texto: ")
MsgBox "El texto especificado es '" & S & "'"
```

- `GetPoint`: Solicita las coordenadas de un punto, que también puede establecerse directamente sobre el área de trabajo. Devuelve una variable `Variant`, que toma formato de arreglo de tres valores de tipo `Double` (como los utilizados para especificar los puntos extremos de las líneas o los centros de las circunferencias). Requiere, como parámetros, un punto que se tomará como referencia (es opcional) y el texto que se mostrará en la consola de comandos. Ejemplo:

```
Dim P As Variant
P = ThisDrawing.Utility.GetPoint( _
, "Especifique un punto: ")
MsgBox "Las coordenadas del punto son: (" & _
P(0) & ", " & P(1) & ", " & P(2) & ")"
```

- `GetDistance`: Solicita un valor de distancia especificada a través de dos puntos. Devuelve un valor de tipo Double. Requiere (opcionalmente) un punto base y el texto que se mostrará en la consola de comandos. Si no se establece ningún punto base, el usuario deberá especificar dos puntos. Ejemplo:

```
Dim D As Double
D = ThisDrawing.Utility.GetDistance( _
, "Especifique una distancia: ")
MsgBox "La distancia entre los puntos es " & D
```

- `GetEntity`: Solicita del usuario la selección de un objeto en el entorno de trabajo. Es necesario pasarle una variable de tipo `AcadObject` (donde se almacenará el objeto seleccionado), una de tipo `Variant` (para almacenar las coordenadas del punto de selección) y un texto para mostrar la petición en la consola. Ejemplo:

```
Dim Obj As AcadObject
Dim P1 As Variant
ThisDrawing.Utility.GetEntity Obj, P1, _
"Seleccione un objeto"
Obj.Color = acRed
MsgBox "Un objeto de tipo '" & Obj.ObjectName & _
"' ha sido cambiado a color rojo"
MsgBox "El punto de selección es: (" & _
P1(0) & ", " & P1(1) & ", " & P1(2) & ")"
```

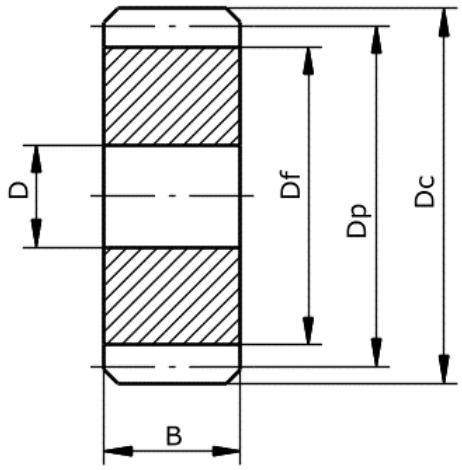
- `Prompt`: No solicita ningún dato; sólo muestra el texto que se le pasa como parámetro, en la consola. Ejemplo:

```
ThisDrawing.Utility.Prompt "Fin del ejemplo"
```

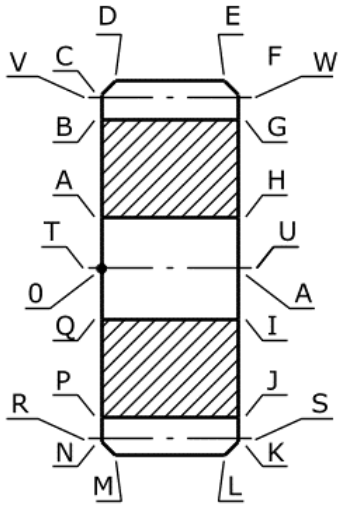
El objeto `Utility` tiene otros métodos muy interesantes, como son `PolarPoint`, que permite calcular las coordenadas cartesianas de un punto, a partir de sus coordenadas polares; `DistanceToReal`, que permite convertir valores de distancia en formato de texto a números reales; `RealToString`, que convierte un número real en un texto con un formato dado; entre otros.

Ejemplo Resuelto.

Se desea desarrollar una aplicación para dibujar la sección transversal de una rueda dentada cilíndrica, tal como se muestra en la Fig. 5.3 a).



a) Dimensiones de la rueda



b) Distribución de puntos

Fig. 5.3 – Sección transversal de la rueda dentada cilíndrica.

El primer paso, consiste en definir las variables necesaria para resolver el problema. Estas variables son:

- Los datos del problema (el punto de inserción, `p0`; módulo de la rueda, `M`; la cantidad de dientes, `Z`; el ancho de la rueda, `B`; el diámetro del cubo, `D`; y la longitud del cateto del bisel, `C`), que serán de tipo `Double` (excepto el punto de inserción que será `Variant` y la cantidad de dientes que será `Integer`). También se definen los diámetros de cabeza, `Dc`; primitivo, `Dp`; y de fondo de la rueda, `Df` (todos de tipo `Double`).

- Los puntos y líneas que definen conforman el dibujo. Los puntos serán arreglos de tres valores de tipo `Double`, denotándose como `pA`, `pB` ... `pW`. Las líneas son objetos de tipo `AcadLine`., los cuales se llamarán `lAB`, `lBC` ... , indicando así los puntos extremos de la línea.

- Los rayados inferior y superior y sus contornos. Los rayados serán objetos de tipo `AcadHatch`. Los contornos serán arreglos de cuatro valores de tipo `AcadEntity`, uno para cada línea que define el contorno de cada rayado.

Como siguiente paso, se toman los datos del usuario, mediante el objeto `Utility`, y se almacenan en las correspondientes variables.

Con los datos, se procede a calcular las coordenadas de todos los puntos. Para ello se utilizan las relaciones geométricas de la rueda, resultando las expresiones que se muestran a continuación:

Punto	X	Y
A	$X_A = X_0$	$Y_A = Y_0 + D/2$
B	$X_B = X_0$	$Y_B = Y_0 + Df/2$
C	$X_C = X_0$	$Y_C = Y_0 + Dc/2 - C$
D	$X_D = X_0 + C$	$Y_D = Y_0 + Dc/2$
E	$X_E = X_0 + B - C$	$Y_E = Y_D$
F	$X_F = X_0 + B$	$Y_F = Y_C$
G	$X_G = X_F$	$Y_G = Y_B$
H	$X_H = X_F$	$Y_H = Y_A$
I	$X_I = X_F$	$Y_I = Y_0 - D/2$
J	$X_J = X_F$	$Y_J = Y_0 - Df/2$
K	$X_K = X_F$	$Y_K = Y_0 - Dc/2 + C$
L	$X_L = X_E$	$Y_L = Y_0 - Dc/2$
M	$X_M = X_D$	$Y_M = Y_L$
N	$X_N = X_O$	$Y_N = Y_K$
P	$X_P = X_O$	$Y_P = Y_J$
Q	$X_Q = X_O$	$Y_Q = Y_I$
R	$X_R = X_0 - 2$	$Y_R = Y_0 - Dp/2$
S	$X_S = X_H + 2$	$Y_S = Y_R$
T	$X_T = X_R$	$Y_T = Y_R$
U	$X_U = X_S$	$Y_U = Y_0$
V	$X_V = X_R$	$Y_V = Y_0 + Dp/2$
W	$X_W = X_S$	$Y_W = Y_V$

Una vez calculados todos los puntos, se crean las líneas que conforman el dibujo, utilizando el método `AddLine`, del objeto `ModelSpace`.

Para elaborar los rayados, primero se le asignan a cada uno de los elementos del arreglo de contorno respectivo, las líneas que forman el borde. Luego de creado los rayados (con el método `AddHatch`), el contorno definido se le asigna con el método `AppendOuterLoop` y se evalúa, para obtener el área rayada.

A continuación se muestra el código del ejercicio resuelto.

```
Sub RuedaDentada()
  ' Declaración de las variables
  Dim p0 As Variant ' Punto de inserción
  Dim M As Double ' Módulo de la rueda
  Dim Z As Integer ' Número de dientes
  Dim B As Double ' Ancho de la rueda
  Dim D As Double ' Diámetro del cubo
```

```
Dim C As Double ' Cateto del bisel
Dim Dc As Double ' Diámetro de cabeza
Dim Dp As Double ' Diámetro primitivo
Dim Df As Double ' Diámetro de fondo
Dim pA(0 To 2) As Double, pB(0 To 2) As Double, _
    pC(0 To 2) As Double, pD(0 To 2) As Double, _
    pE(0 To 2) As Double, pF(0 To 2) As Double, _
    pG(0 To 2) As Double, pH(0 To 2) As Double, _
    pI(0 To 2) As Double, pJ(0 To 2) As Double, _
    pK(0 To 2) As Double, pL(0 To 2) As Double, _
    pM(0 To 2) As Double, pN(0 To 2) As Double, _
    pP(0 To 2) As Double, pQ(0 To 2) As Double, _
    pR(0 To 2) As Double, pS(0 To 2) As Double, _
    pT(0 To 2) As Double, pU(0 To 2) As Double, _
    pV(0 To 2) As Double, pW(0 To 2) As Double
Dim lAB As AcadLine, lBC As AcadLine, _
    lCD As AcadLine, lDE As AcadLine, _
    lEF As AcadLine, lFG As AcadLine, _
    lGH As AcadLine, lHI As AcadLine, _
    lIJ As AcadLine, lJK As AcadLine, _
    lKL As AcadLine, lLM As AcadLine, _
    lMN As AcadLine, lNP As AcadLine, _
    lPQ As AcadLine, lQA As AcadLine, _
    lAH As AcadLine, lBG As AcadLine, _
    lQI As AcadLine, lPJ As AcadLine, _
    lRS As AcadLine, lTU As AcadLine, _
    lVW As AcadLine
Dim RayadoSup As AcadHatch, _
    RayadoInf As AcadHatch
Dim ContornoSup(0 To 3) As AcadEntity, _
    ContornoInf(0 To 3) As AcadEntity
' Toma de datos
p0 = ThisDrawing.Utility.GetPoint(, _
                 "Especifique el punto de inserción: ")
M = ThisDrawing.Utility.GetReal( _
                 "Especifique el módulo de la rueda: ")
Z = ThisDrawing.Utility.GetInteger( _
                 "Especifique la cantidad de dientes: ")
B = ThisDrawing.Utility.GetReal( _
                 "Especifique el ancho de la rueda: ")
D = ThisDrawing.Utility.GetReal( _
                 "Especifique el diámetro del cubo: ")
C = ThisDrawing.Utility.GetReal( _
                 "Especifique el cateto del bisel: ")
' Cálculo de los diámetros de la rueda
Dc = (Z + 2) * M
Dp = Z * M
Df = (Z - 2.5) * M
```

```
' Cálculo de los puntos
pA(0) = p0(0): pA(1) = p0(1) + D / 2
pB(0) = p0(0): pB(1) = p0(1) + Df / 2
pC(0) = p0(0): pC(1) = p0(1) + Dc / 2 - C
pD(0) = p0(0) + C: pD(1) = p0(1) + Dc / 2
pE(0) = p0(0) + B - C: pE(1) = pD(1)
pF(0) = p0(0) + B: pF(1) = pC(1)
pG(0) = pF(0): pG(1) = pB(1)
pH(0) = pF(0): pH(1) = pA(1)
pI(0) = pF(0): pI(1) = p0(1) - D / 2
pJ(0) = pF(0): pJ(1) = p0(1) - Df / 2
pK(0) = pF(0): pK(1) = p0(1) - Dc / 2 + C
pL(0) = pE(0): pL(1) = p0(1) - Dc / 2
pM(0) = pD(0): pM(1) = pL(1)
pN(0) = p0(0): pN(1) = pK(1)
pP(0) = p0(0): pP(1) = pJ(1)
pQ(0) = p0(0): pQ(1) = pI(1)
pR(0) = p0(0) - 2: pR(1) = p0(1) - Dp / 2
pS(0) = pH(0) + 2: pS(1) = pR(1)
pT(0) = pR(0): pT(1) = p0(1)
pU(0) = pS(0): pU(1) = pT(1)
pV(0) = pR(0): pV(1) = p0(1) + Dp / 2
pW(0) = pS(0): pW(1) = pV(1)
' Trazado de las líneas
Set lAB = ThisDrawing.ModelSpace.AddLine(pA, pB)
Set lBC = ThisDrawing.ModelSpace.AddLine(pB, pC)
Set lCD = ThisDrawing.ModelSpace.AddLine(pC, pD)
Set lDE = ThisDrawing.ModelSpace.AddLine(pD, pE)
Set lEF = ThisDrawing.ModelSpace.AddLine(pE, pF)
Set lFG = ThisDrawing.ModelSpace.AddLine(pF, pG)
Set lGH = ThisDrawing.ModelSpace.AddLine(pG, pH)
Set lHI = ThisDrawing.ModelSpace.AddLine(pH, pI)
Set lIJ = ThisDrawing.ModelSpace.AddLine(pI, pJ)
Set lJK = ThisDrawing.ModelSpace.AddLine(pJ, pK)
Set lKL = ThisDrawing.ModelSpace.AddLine(pK, pL)
Set lLM = ThisDrawing.ModelSpace.AddLine(pL, pM)
Set lMN = ThisDrawing.ModelSpace.AddLine(pM, pN)
Set lNP = ThisDrawing.ModelSpace.AddLine(pN, pP)
Set lPQ = ThisDrawing.ModelSpace.AddLine(pP, pQ)
Set lQA = ThisDrawing.ModelSpace.AddLine(pQ, pA)
Set lAH = ThisDrawing.ModelSpace.AddLine(pA, pH)
Set lBG = ThisDrawing.ModelSpace.AddLine(pB, pG)
Set lQI = ThisDrawing.ModelSpace.AddLine(pQ, pI)
Set lPJ = ThisDrawing.ModelSpace.AddLine(pP, pJ)
Set lRS = ThisDrawing.ModelSpace.AddLine(pR, pS)
Set lTU = ThisDrawing.ModelSpace.AddLine(pT, pU)
Set lVW = ThisDrawing.ModelSpace.AddLine(pV, pW)
```

```
  ' Creación de los rayados
  Set RayadoSup = ThisDrawing.ModelSpace.AddHatch( _
                      acHatchPatternTypePreDefined, _
                      "Line", False)
  RayadoSup.PatternAngle = 3.1416 / 4
  RayadoSup.PatternScale = 0.5
  Set ContornoSup(0) = lAB
  Set ContornoSup(1) = lBG
  Set ContornoSup(2) = lGH
  Set ContornoSup(3) = lAH
  RayadoSup.AppendOuterLoop ContornoSup
  RayadoSup.Evaluate
  Set RayadoInf = ThisDrawing.ModelSpace.AddHatch( _
                      acHatchPatternTypePreDefined, _
                      "Line", False)
  RayadoInf.PatternAngle = 3.1416 / 4
  RayadoInf.PatternScale = 0.5
  Set ContornoInf(0) = lPQ
  Set ContornoInf(1) = lQI
  Set ContornoInf(2) = lIJ
  Set ContornoInf(3) = lPJ
  RayadoInf.AppendOuterLoop ContornoInf
  RayadoInf.Evaluate
End Sub
```

Preguntas y ejercicios propuestos

1. ¿Qué objeto proporciona los métodos básicos de creación de entidades en VBA?
2. ¿Cuáles son los pasos para crear un rayado desde programación?
3. Mencione las herramientas de edición vistas en el capítulo.
4. ¿Qué funcionalidades ofrece el objeto `Utility`?
5. Elabore un conjunto de macros para representar los símbolos usados en los esquemas de tuberías, de forma tal que siempre sus dimensiones sean proporcionales a una dimensión básica especificada por el usuario.