23

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE INGENIERIA

PROGRAMACION ORIENTADA A OBJETOS, UN CASO PRACTICO: SISTEMA DE CAPTURA DE REDES DE AGUA POTABLE

T E S I S

QUE PARA OBTENER EL TITULO DE

INGENIERIA EN COMPUTACION

P R E S E N T A

MARTIN BRAVO AGUIRRE

DIRECTOR: ING. GABRIEL CASTILLO HERNANDEZ

MEXICO, D. F. 1996

TESIS CON
FALLA DE ORIGEN

Agradecimientos:

A mi Dios y Señor, roca y fortaleza mia.

A mis padres, por darme siempre su apoyo y alentarme a seguir adelante.

A mi esposa, por su comprensión y apoyo en todo momento.

A mis hermanos, por contar siempre con ellos.

A mis amigos, por el equipo de trabajo que hemos formado.

A la universidad, por darme la oportunidad de estudiar una carrera.

Programación Orientada a Objetos, un caso práctico: Sistema de Captura de Redes de Agua Potable

Índice

Introducción

1. Introducción

En la realidad un sistema en su parte más elemental esta compuesto por una serie de objetos. Cada objeto tiene una o varias responsabilidades que en conjunto hacen que el sistema funcione. El ver cada objeto de manera independiente nos permite implementar el sistema de una forma rápida y de fácil mantenimiento. La programación Orientada a Objetos (POO) nos permite precisamente esa abstracción de objetos y su implementación.

El propósito del presente trabajo es presentar las bases de la programación orientada a objetos y cómo aplicarlas utilizando Borland Pascal V7.0, de tal forma que el lector sea capaz de crear un sistema (orientado a objetos) que corra bajo ambiente Windows.

Además de esta introdución, el trabajo consta de cuatro capítulos más, cuyo contenido es:

En el segundo capítulo se explica el Diseño, el Análisis y la Programación Orientada a Objetos. Se definen las características que debe tener un programa orientado a objetos y cómo se puede hacer un análisis y diseño orientado a objetos.

En el tercer capítulo se ejemplifican los conceptos vistos en el capítulo anterior auxiliándose del planteamiento de un problema: La creación de un sistema para captura de redes de agua potable y se utiliza cómo marco de referencia Borland Pascal.

En el cuarto capítulo se presenta una guía para la creación de programas bajo ambiente Windows utilizando Borland Pascal. Este capítulo toma como base la teoría de la programación orientada a objetos y explica el uso de las clases definidas por el lenguaje y cómo se definen los objetos de usuario.

En el quinto capítulo se revisa la creación del sistema de Captura de Redes de Agua Potable (CRAP) y se da su estructura general (Clases, Objetos) y finalmente se presentan unos comentarios finales.

CRAP (Captura de Redes de Agua Potable) es una aplicación Windows que permite la captura de datos de los elementos que intervienen en una red de agua potable (Nodo, Pozo, Tanque y Tubo). La representación de la red se hace en forma gráfica y la interfaz con el usuario es mediante el uso de diálogos (ventanas) y el uso del mouse.

Programación Orientada a Objetos (POO)

2. Programación Orientada a Objetos (POO)

La Programación Orientada a Objetos nace como una necesidad de contar con una herramienta que permita modelar el mundo real, un mundo lleno de sistemas complejos. Si se parte de la idea de que el software es diseñado en su mayoría para modelar sistemas reales o resolver situaciones cotidianas, se llegará a la conclusión de que los sistemas informáticos son complejos por si mismos. Entendiendo como complejos aquellos sistemas que se integran por varios subsistemas.

La complejidad del software inicia al momento de plantear el problema a solucionar, frecuentemente el usuario no sabe a ciencia cierta que es lo que espera del sistema, el programador no entiende los requerimientos del usuario y ve el problema de forma distinta de como lo ve el usuario. Esto lleva a que el diseño del software es modificado durante la fase de programación. En ocasiones, una vez terminada la aplicación, no satisface las expectativas del usuario y muchas veces se tiene que efectuar el "mantenimiento" de la misma, que en realidad es un re-diseño.

Generalizando, se puede decir que un sistema tiene los siguientes atributos:

1. Frecuentemente, la complejidad toma una forma jerárquica, esto es, el sistema esta compuesto por una serie de subsistemas que se relacionan entre si, y a su vez, estos subsistemas tienen sus propios subsistemas, y se repite la estructura hasta encontrar en el nivel más bajo los componentes elementales.

2. El criterio mediante el cual se determina que los componentes son elementales es arbitraria y depende del punto de vista con que se vea el sistema.

3. Las relaciones intra-componentes son más fuertes que las relaciones inter-componentes. Lo que da la posibilidad de estudiar cada componente de manera independiente (relativamente)

4. Un sistema jerárquico normalmente está compuesto por unos cuantos tipos de subsistemas, los cuales se presentan en varias combinaciones y arreglos.

Al momento de iniciar el análisis de un sistema, se encuentra que existen una serie de partes que interactúan y una multitud de relaciones. A simple vista se pueden identificar cada parte del sistema y sus relaciones, pero si no se contemplan todas las variables a las que puede estar sujeto el proceso se llegará a un diseño deficiente. Aquí estriba la importancia de contar un mecanismo que permita, por un lado, descomponer el proceso (o problema) hasta sus partes más elementales, y por otro, incluir todos los elementos dentro del análisis. Mucho dependerá de la capacidad de abstracción con que se cuente y de la profundidad del conocimiento que se tenga sobre el tema.

Tradicionalmente se utiliza un método de descomposición del problema del tipo TOP-DOWN, esto es, la descomposición se hace de arriba a abajo, de lo general a lo particular.

En el presente trabajo se hablará del método llamado Orientado a Objetos (Object-Oriented), este método ve el problema como un conjunto autónomo de agentes que colaboran para ejecutar una función mayor. Se asocian los datos y las operaciones que actúan sobre ellos en elementos autónomos denominados objetos. Cada objeto tiene responsabilidades especificas.

Ahora la pregunta es: ¿Cuál de los dos métodos es el mejor?, una buena respuesta es que los dos puntos de vista son importantes, el primer método enfatiza el orden de los eventos y Orientación a Objetos (OO) enfatiza los agentes y sus causas de acción. Dicho de otra forma, OO enfatiza cuales son los sujetos que intervienen en el proceso y sus operaciones. Lo que es importante remarcar es que no se puede construir un sistema tratando de seguir los dos métodos simultáneamente, sin embargo, se puede iniciar el análisis con uno y el resultado usarse como marco de referencia para expresar el problema desde la otra perspectiva.

La tendencia es que se use inicialmente el método de Orientación a Objetos porque da una mejor aproximación y ayuda a organizar el sistema, es decir, permite describir la complejidad del proceso. La descomposición orientada a objetos tiene una serie de ventajas con respecto a otros métodos de descomposición. La descomposición orientada a objetos permite la generación de sistemas pequeños que pueden ser reutilizables, lo que lleva a un ahorro en la generación de código. Adicionalmente se reduce grandemente el riesgo de construir sistemas complicados (difíciles de mantener), pues el sistema es diseñado para integrar en forma incremental sistemas más pequeños.

La abstracción juega un papel importante en el diseño orientado a objetos. La abstracción debe permitir identificar cada objeto que interviene en el sistema así como sus mecanismos de operación.

Al igual que la abstracción, la jerarquización es importante. Se manejan 2 tipos de jerarquías, una de objetos y otra de clases. La jerarquía de objetos es importante porque ilustra cómo los diferentes objetos colaboran unos con otros a través de mecanismos de comunicación basados en mensajes. La jerarquía de clases es igualmente importante porque enfatiza las estructuras comunes y su funcionamiento en el sistema.

El análisis y diseño orientado a objetos permitirán crear software flexible al cambio y escrito con un ahorro de código (lo que implica un inversión menor de tiempo al momento de codificar el sistema).

Booch[1] define la Programación Orientada a Objetos (POO) de la siguiente forma:

> La programación Orientada a Objetos es un método de implementación en el cual los programas son organizados como una colección de objetos, donde los objetos representan instancias de alguna clase, y las clases son todas miembros de una jerarquía de clases unidas vía relaciones hereditarias.

Revisando la definición se puede distinguir que:

1. La POO utiliza objetos, no procedimientos.
2. Cada objeto es una instancia(ejemplo) de alguna clase.
3. Una clase esta relacionada con otra vía relaciones hereditarias (Parentesco).

Para que un programa sea orientado a objetos debe de presentar por lo menos éstas características, de lo contrario no lo es. Aquí hay que mencionar que si el programa cuenta con las 2 primeras características pero no maneja relaciones hereditarias se trata de un programa con tipos de datos abstractos.

En cuanto a los lenguajes de programación Cardelli y Weigner[2] dicen que:

> Un lenguaje (de programación) es orientado a objetos si y sólo si satisface las siguientes características:
>
> - ☑ Si soporta objetos que son abstracciones de datos con una interfase y un estado local oculto.
> - ☑ Los objetos tienen un tipo [clase] asociado.
> - ☑ Los tipos [clases] pueden heredar atributos de supertipos [superclases].

El énfasis de los métodos de programación es básicamente el uso apropiado y efectivo de un mecanismo propio de un lenguaje en particular. En contraste los métodos de diseño enfatizan la estructuración apropiada y efectiva de un sistema (principalmente si este es complejo). Siguiendo esta idea Booch sugiere que el diseño orientado a objetos se puede definir como:

> El diseño orientado a objetos es un método de diseño que encierra el proceso de descomposición orientada a objetos y una notación que describe los modelos lógico y físico del sistema bajo diseño.

Hay dos partes importante de la definición. En primer lugar el diseño orientado a objetos se encamina a una descomposición orientada a objetos. En segundo lugar usa diferentes notaciones para expresar el modelo lógico (estructuras de clases y objetos) y físico (Arquitectura de procesos y módulos).

El soporte para la descomposición orientada a objetos es lo que marca la diferencia entre el diseño orientado a objetos y el diseño estructurado:

- ☑ El uso de abstracciones de objetos y clases para estructurar lógicamente un sistema.
- ☑ El uso de algoritmos de abstracción.

Por otro lado se tiene que, el análisis orientado a objetos enfatiza la construcción de modelos del mundo real, por lo que Booch hace la siguiente definición:

> El análisis orientado a objetos es un método de análisis que examina los requerimientos desde una perspectiva de clases y objetos encontrados en el vocabulario del dominio del problema.

¿Cómo están relacionados el diseño y el análisis orientado a objetos? Básicamente el resultado del análisis orientado a objetos sirve como modelo desde el cual se puede iniciar el diseño. El resultado (del diseño) es usado como base para la implementación total del sistema utilizando métodos de programación orientada a objetos.

Hasta este momento se ha definido lo que es el análisis, diseño y programación orientada a objetos. Se han mencionado conceptos como jerarquías, abstracción y clases entre otros, sin embargo no se han definidos los conceptos en los que se funda la orientación a objetos.

Primero, se darán las definiciones de lo que es un objeto y una clase para posteriormente explicar los elementos de la orientación a objetos.

Booch define lo que es un objeto y una clase de la siguiente forma:

> Un objeto tiene estado, función e identidad; la estructura y funcionalidad de objetos similares es definida en su clase común.

donde:

- ☑ El estado de un objeto incluye todas las propiedades del objeto más los valores actuales de dichas propiedades.
- ☑ La función es cómo un objeto actúa y reacciona; en términos de cambios de estado y paso de mensajes.
- ☑ La identidad es la propiedad de un objeto que lo distingue de otros objetos.

Finalmente:

> Una clase es un conjunto de objetos que comparten una estructura común y una funcionalidad común

2.1. Elementos del modelo Orientado a Objetos

Existen 7 elementos del modelo Orientado a Objetos, de los cuales se identifican cómo elementos esenciales a:

- ☑ Abstracción.
- ☑ Encapsulado.
- ☑ Modularidad.
- ☑ Jerarquización.

Un modelo sin uno de estos elementos no es un modelo orientado a objetos.

Por otro lado, se tienen los elementos complementarios:

- ☑ Tipificación.
- ☑ Concurrencia.
- ☑ Persistencia.

Los cuales pueden o no aparecer en el modelo.

A continuación se explicarán cada uno de éstos elementos:

2.1.1. Abstracción

Booch define la abstracción como:

> Una abstracción denota las características esenciales de un objeto que lo distingue de otros tipos de objetos y ofrece de manera conceptual el límite del mismo, relativo a la perspectiva del observador .

La abstracción siempre depende del punto de vista con que se observe, por lo tanto, es importante considerar que el nivel de separación de las funciones y cualidades del objeto dependerá del enfoque del observador. Lo anterior lleva a una gama de tipos de abstracción que se pueden clasificar de la siguiente forma:

- ☑ ***Abstracción por entidad*** *:* Un objeto es representado como un patrón (ejemplo) útil del dominio del problema o una entidad del dominio de la solución

- ☑ ***Abstracción por acción:*** Un objeto provee un conjunto de operaciones generalizadas, las cuales ejecutan el mismo tipo de función.

- ☑ ***Abstracción por máquina virtual:*** Un objeto reúne una serie de operaciones las cuales son usadas por un nivel superior de control, y que a su vez, usan un conjunto de operaciones de un nivel inferior.

Booch trabaja las abstracciones del tipo entidad ya que llevan directamente al vocabulario que se utiliza en el problema.

En la abstracción se maneja:

Objeto Cliente: Es cualquier objeto que usa los recursos de otro objeto.

Objeto Servidor: Es aquel que provee los recursos al objeto cliente.

Se puede caracterizar el comportamiento de un objeto, considerando los servicios que provee a otros objetos, así cómo las operaciones que puede hacer sobre otros objetos. Este punto de vista forza a determinar un "contrato" entre los objetos, es decir, a definir las *responsabilidades* del objeto, especialmente la función por la cual ha sido escogido. El *protocolo* denota la forma en que un objeto puede actuar o reaccionar y por lo tanto constituye una abstracción desde el punto de vista estático y dinámico.

Todas las abstracciones tienen propiedades estáticas y dinámicas, por ejemplo el objeto Archivo ocupará un determinado espacio en disco, tendrá un nombre y contenido. Estas son propiedades estáticas. El valor que pueda tener cada una de estas es dinámico. En un estilo de programación estructurada (procedure-oriented) la actividad que cambia los valores dinámicos es la parte medular de los programas. Las cosas pasan cuando los subprogramas o rutinas son llamados y las instrucciones son ejecutadas. En un estilo de programación orientada a objetos, las cosas pasan siempre que se manda un mensaje al objeto. Lo anterior quiere decir que al invocar el objeto, este reaccionará de determinada forma. Las posibles reacciones del objeto dependerán de las responsabilidades de él.

Una vez que se han detectado que objetos se tienen es necesario definir cómo se van a implementar y es aquí donde la encapsulación permite separar las características del objeto y la forma de comunicarse con otros objetos.

2.1.2. Encapsulación

La abstracción y la encapsulación son dos conceptos complementarios. La abstracción enfatiza la función del objeto. La encapsulación se enfoca a la implementación que pueda dar como resultado dicha función. La encapsulación es conocida también como el proceso de ocultamiento de información. Generalmente la estructura de un objeto esta oculta, así como, la implementación de sus métodos.

En el ámbito de la orientación a objetos, los datos dentro de un objeto son accesados sólo a través de los métodos del objeto. Un objeto no puede accesar los datos de otro objeto, en su lugar, los objetos intercambian información vía mensajes.

Este intercambio de información basado en mensajes ofrece dos tipos importantes de protección:

1. Protege a las variables del objeto de ser corrompidas por otros objetos. Si un objeto tuviera acceso directo a las variables de otro objeto, se podría dar el caso de que este no manejara correctamente las variables y podría dañar el objeto. Un objeto se protege a sí mismo de este tipo de errores escondiendo sus variables y permitiendo su acceso sólo a través de sus métodos.

2. Un objeto protege a otros objetos de las complicaciones de depender en su estructura interna. Un objeto no depende de la estructura interna de otro objeto. Mediante la encapsulación, un objeto sólo necesita saber cómo pedir la información a otro objeto y cómo enviársela (interfase). No se preocupa de estar al tanto de cada nombre de variable, tipo de información que contiene o espacio que ocupa cada una (implementación).

Finalmente se define la encapsulación como:

> El proceso de empaquetar los elementos de una abstracción que constituyen su estructura y funcionalidad; la encapsulación sirve para separar de una abstracción la interfase y su implementación.

Dicho en otras palabras, la encapsulación "integra" los datos y procedimientos de un objeto y al mismo tiempo, separa la implementación de un objeto de las interfaces que pudiera tener con otros , es decir, la interfase entre objetos no depende de la implementación de los mismos.

Con base en la definición anterior se hace necesario que los programas orientados a objetos sean codificados en módulos para facilitar su construcción.

2.1.3. Modularidad

La modularización consiste en dividir un programa en módulos (subprogramas) los cuales pueden ser compilados por separado pero que tienen relaciones con otros módulos. En la programación orientada a objetos se agrupan lógicamente las clases y objetos relacionados. Los módulos sirven como contenedores físicos en los cuales se declaran las clases y objetos del diseño lógico.

El diseño estructurado tradicionalmente se interesa por la agrupación de los subprogramas utilizando los criterios de cohesión y acoplamiento. En el diseño orientado a objetos, el problema es substancialmente diferente: La tarea es decidir donde agrupar las clases y los objetos en el diseño lógico.

El propósito de descomponer en módulos un sistema es reducir el costo del software al permitir el diseño, la construcción y la revisión de los módulos en forma independiente. Cada módulo debe ser lo suficientemente sencillo como para poder ser entendido por separado. Al mismo tiempo debe ser posible cambiar la implementación de algún módulo sin afectar el funcionamiento de los demás módulos.

El resultado de la abstracción se pone en módulos para producir la arquitectura física del sistema. En aplicaciones grandes el uso de módulos es esencial para manejar la complejidad del sistema.

Decidir sobre cuantos y cuales módulos son los correctos para un sistema es casi igual de difícil que decidir sobre los niveles de abstracción que deben hacerse sobre el mismo sistema.

2.1.4. Jerarquía

A excepción de las aplicaciones triviales, casi siempre se encuentran más abstracciones de las que se pueden comprender al mismo tiempo. La encapsulación ayuda a manejar la complejidad del sistema a través de ocultar el punto de vista interno de las abstracciones. La modularidad ofrece un camino para agrupar lógicamente las abstracciones relacionadas. No obstante esto no es suficiente. Un conjunto de abstracciones comúnmente forma una jerarquía y si se identifican estas jerarquías en el diseño, se simplificará en gran manera el entendimiento del problema. Por lo tanto, se define la jerarquía como:

La jerarquía es un clasificación u ordenación de abstracciones.

Las dos jerarquías más importantes son la de estructura de clase y la de estructura de objetos.

En una jerarquía de clases la **herencia** es lo más importante y por lo tanto es un elemento esencial en un sistema orientado a objetos. Básicamente la herencia define la relación entre clases. Una clase comparte la estructura o función definida en una o más clases (llamadas herencia simple y herencia múltiple respectivamente). La herencia, por lo tanto representa una jerarquía de abstracciones. Por lo regular, una subclase aumenta o redefine la estructura y función de sus superclases.

Semánticamente la herencia se define cómo una relación del tipo "es un(a)". Por ejemplo un león "es un" mamífero, un sedan "es un automóvil". Por lo tanto la herencia implica una jerarquía de lo más general a lo más particular. Una subclase especializa la estructura general o función de su superclase.

La herencia es la parte más innovadora de la programación orientada a objetos, ya que no se encuentra implementada en los lenguajes tradicionales de programación. La herencia es una herramienta para la transmisión automática de código, es decir, un mecanismo por el cual una clase de objetos (subclase) es definida como un caso de una clase más general (superclase). La subclase hereda las definiciones y métodos de la superclase. Adicionalmente la subclase puede definir sus propias variables y métodos y de ser necesario sobreponerlos sobre los heredados.

La herencia es una manera de definir objetos cuyas propiedades son automáticamente transmitidas a aquellos objetos relacionados. La herencia enlaza los conceptos para formar un todo relacionado, de tal modo que, cuando cambia un concepto de alto nivel, el cambio se propaga automáticamente a los niveles inferiores.

La herencia simple o única es aquella donde una subclase solo tiene una superclase. La herencia múltiple es utilizada cuando una clase de objetos tiene que desempeñar varias funciones, y cada una de estas

funciones es caracterizada por diferentes clases. En la herencia múltiple una subclase tiene más de una superclase.

En conclusión, la herencia tiene como efecto la reducción de código al eliminar la necesidad de volver a programar funcionalidades comunes. La herencia es una herramienta para construir, organizar y emplear clases reutilizables. Sin herencia, cada clase sería una unidad independiente y se tendría que desarrollar desde cero. De no existir la herencia las distintas clases no tendrían relación entre sí.

Las clases pueden estar anidadas a cualquier nivel, y la herencia se acumulará automáticamente hacia abajo a través de todos sus niveles. Esta definición da como resultado una estructura de árbol conocida como jerarquía de clases.

2.1.5. Tipificación

El concepto de *tipo* se deriva principalmente de las teorías de tipos de datos abstractos. En ocasiones se usa tipo y clase como dos cosas similares. Aunque se puede decir que el tipo hace énfasis sobre el significado de la abstracción, Booch hace la siguiente definición:

> **La tipificación es la implementación de la clase de un objeto, lo que implica que objetos de diferentes tipos no deben de ser intercambiados.**

La tipificación permite expresar las abstracciones en un lenguaje de programación, en el cual se implementará el diseño. En los distintos lenguajes de programación existen varios modos de tipificación. La primer clasificación que se encuentra es tipificación débil y tipificación firme. La tipificación firme se refiere a que la consistencia entre tipos es manejada, cualquier intento de violación es detectado al momento de la compilación. En la tipificación débil el error surge al tiempo de ejecución.

Por otro lado se maneja la tipificación estática y dinámica. No se confunda la tipificación estática con la tipificación firme. La tipificación firme se refiere a la consistencia del tipo y la tipificación estática se refiere al tiempo en el que los tipos son nombrados. La tipificación estática significa que los tipos de la variables son fijados al momento de la compilación. En la tipificación dinámica los tipos de las variables no son conocidos sino hasta el tiempo de ejecución.

Cuando la tipificación dinámica y la herencia interactúan se obtiene como resultado el **polimorfismo**. El polimorfismo representa un concepto de la teoría de clases, en la cual, un nombre simple (como puede ser la declaración de una variable) puede denotar objetos de diferentes clases que están relacionadas por una superclase común. Dicho de otra forma, cualquier objeto con este nombre esta habilitado para responder a ciertas operaciones comunes de diferentes formas. El polimorfismo es la pieza más poderosa que pueden tener los lenguajes de programación orientados a objetos, ésta es la diferencia entre la programación orientada a objetos y la programación con tipos de datos abstractos.

2.1.6. Concurrencia

En algunos casos se presenta la necesidad de crear un sistema automático que maneje varios eventos simultáneamente. Regularmente este problema excede la capacidad del equipo (hablando de computadoras personales) que sólo cuentan con un procesador. En ocasiones, se considera la posibilidad de contar con una serie de computadoras que permitan implementar un proceso distribuido o la posibilidad de usar equipo capaz de hacer multiproceso. Por cada proceso se requiere una línea de control (thread of control), cada programa tiene por lo menos una línea de control. En un sistema que existe concurrencia no sucede lo mismo, puede existir más de una línea de control para un programa.

Algunas de estas líneas serán transitorias y otras estarán presentes todo el tiempo que dure la ejecución del sistema.

Muchos sistemas operativos contemporáneos soportan directamente la concurrencia, lo que representa una gran oportunidad para la creación de sistemas orientados a objetos concurrentes. El sistema operativo DOS no permite la concurrencia pero utilizando Windows se puede lograr. Adicionalmente Windows ofrece interfaces para la creación y manejo de procesos.

En la programación orientada a objetos se enfatiza la abstracción de datos, la encapsulación y la herencia. La concurrencia enfatiza la abstracción del proceso y la sincronización. En un diseño orientado a objetos se puede conceptualizar el mundo como una serie de objetos, de los cuales algunos están activos a un tiempo y otros no. Bajo este esquema Booch dice que:

La concurrencia es la propiedad de distinguir entre un objeto activo y otro que no lo esta.

2.1.7. Persistencia

Un objeto en software, tomará un cantidad de espacio y existirá por un determinado tiempo. La persistencia indica la existencia continua del objeto fuera de la ejecución de algún programa, por lo que se encuentran varios tipos de persistencia:

- ☑ Los datos existen entre las ejecuciones de un programa.
- ☑ Los datos existen entre varias versiones del programa.
- ☑ Los datos existen fuera del programa.

Estos tipos de persistencia son utilizados normalmente en la tecnología de base de datos.

Booch unifica la persistencia y el diseño orientado a objetos para concluir que:

> La persistencia es la propiedad de un objeto a través de la cual este existe trascendiendo en el tiempo y/o espacio.

Esto quiere decir que con la persistencia un objeto continúa existiendo a través del tiempo. El objeto existe aún después que su creador a dejado de existir.

2.2. Método Booch

Booch ofrece un método para el modelo de orientación a objetos, primero se presenta una técnica de clasificación de objetos ya que es la parte medular del análisis y posteriormente se explica el proceso.

2.2.1. Técnica de clasificación

La identificación de clases y objetos es la parte más difícil del análisis orientado a objetos. La clasificación permite identificar los patrones comunes de interacción entre objetos, de esta forma se podrán definir los mecanismos que servirán como base para la implementación. La clasificación también dará una guía para la toma de decisiones en la modularización. El problema fundamental de la clasificación es encontrar las similitudes entre los objetos para poder agruparlos de acuerdo a su funcionalidad o estructura común.

Se debe considerar que la clasificación depende del punto de vista del observador. Varios objetos pueden ser clasificados de distinta forma según el enfoque dado por el clasificador.

Booch recomienda una clasificación incremental e iterativa. Este proceso incremental e iterativo es utilizado en el desarrollo de interfaces gráficas, bases de datos y lenguajes de cuarta generación. El proceso se basa en una primera clasificación y posteriormente en la fase de diseño una revisión de dicha clasificación. En la medida que se adentre al diseño y se tenga más conocimiento del sistema, será posible mejorar la clasificación original.

2.2.2. Proceso

Se puede iniciar el análisis separando las clases y objetos que forman parte del vocabulario del dominio del problema, y en el diseño definir las abstracciones y mecanismos dados por los verbos del vocabulario.

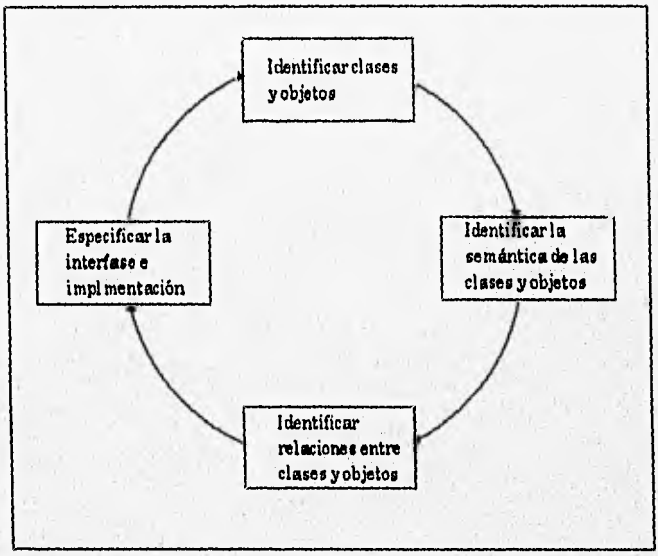
Existen varias fuentes de donde se pueden detectar las clases y los objetos, por ejemplo Shaler y Mellor sugieren que los objetos y clases pueden venir de una de las siguientes fuentes [3]:

- ☑ Cosas tangibles
- ☑ Roles
- ☑ Eventos
- ☑ Interacciones

Desde una perspectiva de modelado de bases de datos, Ross ofrece una lista similar [4]:

- ☑ Gente
- ☑ Lugares
- ☑ Cosas
- ☑ Organizaciones
- ☑ Conceptos
- ☑ Eventos

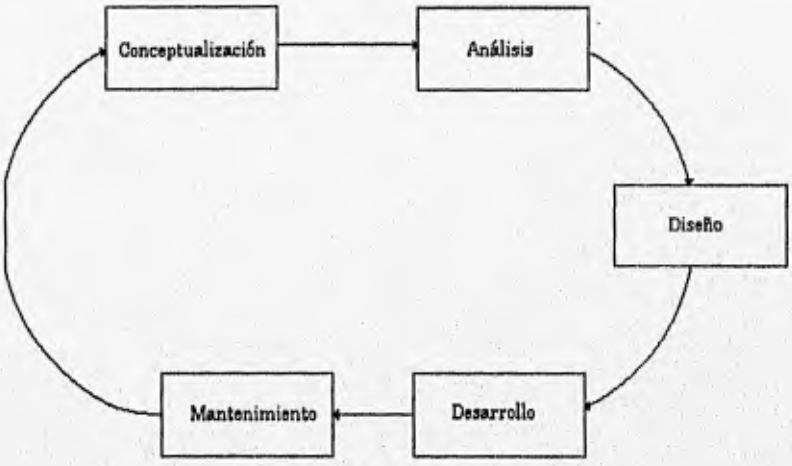
Anteriormente se mencionó que el proceso de clasificación puede ser incremental e iterativo, a continuación se presenta el proceso que recomienda Booch para ésta tarea:



La figura anterior muestra los pasos que deben de seguirse en la clasificación:

1. Identificar las clases y objetos a un nivel de abstracción adecuado.
2. Identificar la semántica de dichas clases y objetos.
3. Identificar las relaciones entre éstas clases y objetos.
4. Especificar las interfaces e implementaciones de las clases y objetos.

Por otro lado, Booch presenta un proceso completo para el desarrollo de aplicaciones orientadas a objetos:



Al igual que otros métodos, el proceso de construcción de una aplicación es iterativo:

1. Establecer los requerimientos centrales para el software (Conceptualización).
2. Desarrollar un modelo de la funcionalidad del sistema deseado (análisis).
3. Crear una arquitectura para la implementación (diseño).
4. Desarrollar la implementación a través de aproximaciones sucesivas (Desarrollo)
5. Dar mantenimiento a la implementación (Mantenimiento).

2.3. Beneficios

Es fácil reconocer que la programación orientada a objetos es fundamentalmente diferente a la programación estructurada, pero esto no significa que se abandonen los principios básicos y experiencias de los métodos de programación anteriores.

El análisis, diseño y programación Orientada a Objetos ofrece los siguientes beneficios:

- ☑ Explota el poder expresivo de los lenguajes de programación orientada a objetos.
- ☑ Fomenta la re-utilización de componentes de software (permitiendo ahorro de código).
- ☑ Los sistemas se hacen más flexibles al cambio y más sencillos de mantener.

Análisis y diseño de CRAP

3. Análisis y Diseño del Sistema de Captura de Redes de Agua Potable (CRAP)

El objetivo de este capítulo es ejemplificar los conceptos analizados en el capítulo anterior para posteriormente implementar un sistema con la técnica de la Programación Orientada a Objetos utilizando el lenguaje de programación Borland Pascal v7.0

3.1. Planteamiento del problema

Se requiere crear un sistema que permita bajo ambiente windows la captura de datos para una red de distribución de agua potable.

3.2. Análisis y diseño Orientado a Objetos del problema

De acuerdo a lo planteado en el capítulo anterior se tienen que encontrar los objetos y clases que integran el problema. Una vez identificados los objetos y clases se podrá hacer un diseño orientado a objetos que permita implementar la solución al problema. De acuerdo con el método de Booch, del vocabulario del dominio del problema se pueden identificar los objetos y sus características que intervienen en una red de distribución de agua potable:

Tubos

- Nombre
- Longitud
- Distancias y elevaciones (Un tubo no siempre tiene la misma elevación con respecto al nivel del mar en toda su longitud)
- Diámetro
- FDarcy
- Celeridad
- Posición (X,Y)

Nudos

- Nombre
- Posición (X,Y)

Tanques

- Nombre
- Elevación
- Nivel (Tirante)
- Posición (X,Y)

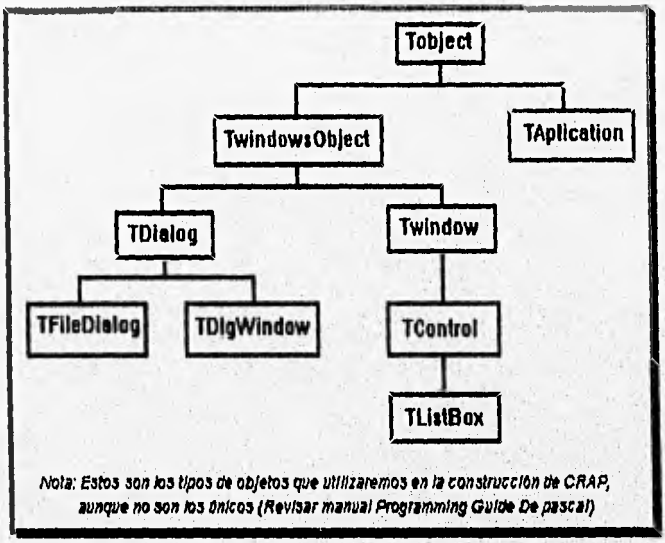
Pozos

- Nombre
- Elevación
- Polinomios (para modelado)
- Posición (X,Y)

De acuerdo a la información que se tiene se puede decir que existe una clase de objetos de la Red y se cuenta con cuatro objetos dentro de la clase (Tubo, Nudo, Tanque y Pozo). Cabe mencionar que la definición de la clase y la clasificación de los objetos ha sido relativamente fácil dado que los elementos que intervienen en el problema son pocos. Al momento de implementar la aplicación se tendrán que revisar que tipos de clases y objetos permite el lenguaje de programación utilizar, que para éste caso, será Borland Pascal v7.0. En la fase de diseño se deberá analizar que clases y objetos se podrán usar al momento de la implementación.

Se pretende construir un sistema que corra bajo ambiente windows que permita la captura de redes de agua potable. El lenguaje ha utilizar será Borland Pascal V7.0 ya que permite la programación orientada a objetos y la creación de aplicaciones que corren bajo Windows. A partir de este momento se hablará del Sistema de Captura de Redes de Agua Potable como **CRAP**. Antes de entrar a detalles del sistema se hablará un poco de Borland Pascal para conocer en que forma se diseñará el sistema.

Pascal cuenta con una serie de clases ya definidas y en base a ellas se puede construir cualquier aplicación, a continuación se muestra la jerarquía de clases que tiene Pascal. En este trabajo sólo se hablará de las facilidades que ofrece Pascal para una aplicación windows, dando por entendido que el lector posee un conocimiento previo sobre programación estructurada, Pascal y sus funciones.



Tobject es la base de cualquier objeto definido en Pascal. Es el ancestro común para todos los objetos windows que se definan. Cuenta básicamente con un constructor y un destructor.

TAplication. Cada aplicación orientada a objetos bajo windows debe ser definida como un tipo derivado de TAplication.

TwindowObject. Los objetos windows representan las ventanas tan familiares del sistema Windows así como los elementos de control (Botones, Diálogos, Scroll Bar, ...).

El sistema CRAP tendrá las siguientes características:

- Manejo de mouse para la edición de los elementos (Tubo, Nudo, Tanque y Pozo).
- Ventana para dibujar la red de agua potable (se dibujará previamente una malla de puntos por lo que el usuario sólo podrá dibujar un elemento en cada punto de la malla).
- Menú de opciones.
- Barra de iconos para agilizar el acceso a las opciones del menú.
- Ventanas de diálogos para el ingreso de los datos de cada elemento.
- Ventana para indicar las coordenadas de la posición del mouse (relativas a la malla de edición).

Con esta información se puede definir qué objetos y de qué tipo serán utilizados en la construcción de CRAP:

Objetos	Clase
CRAP	TAplication
Elemento	Tobject
Ventana para dibujar la red	Twindow
Dialogo para ingresar los datos de cada elemento	TDlgWindow
Dialogo para ingresar nombres de archivos	TFileDialog
Ventana para presentar los mensajes	TListBox
Ventana para presentar barra de iconos de ayuda	Twindow
Ventana para presentar la posición del cursor	Twindow

En esta clasificación no se mencionan el menú de opciones así como el manejo del mouse y esto se debe a que el menú es una característica del objeto CRAP. El manejo de mouse es un método de implementación del objeto ELEMENTO.

Es necesario indicar cómo intervienen las características de la programación orientada a objetos dentro del sistema que se esta diseñando:

Hasta este momento se han identificado (***abstracción*** del sistema) los elementos a manejar así como sus características. Con esta información se han determinado las clases y objetos que serán utilizados en la fase de construcción. Las otras características básicas de la programación orientada a objetos las proporciona Object Pascal de la siguiente forma:

Encapsulado: Al definir un objeto en Pascal se asocian los datos y procedimientos de tal forma que un objeto no puede corromper las variables de otro objeto, ej.:

```
TElemento = object(TObject)
  Icon: HIcon;                | Icono asociado al elemento }
  Cursor: HCursor;     { Cursor a ser desplegado cuando el elemento este en uso}

  { Creación y activación }
  constructor Init(IconName, CursorName: PChar);
  procedure Activa; virtual;
  procedure Desactiva; virtual;

  { Acciones iniciadas por el mouse }
  procedure Boton_Izquierdo_Abajo( X, Y: Integer); virtual;
  procedure Mouse_En_Movimiento( X, Y: Integer); virtual;
  procedure Boton_Izquierdo_Arriba; virtual;
  procedure Inicio_Dibujo(X, Y: Integer); virtual;
  procedure Fin_Dibujo(X, Y: Integer); virtual;
end;
```

TElemento encapsula las características y procedimientos comunes para los elementos de la red.

En el ejemplo anterior se estan utilizando clases abstractas. El objeto TElemento será el "padre" de otros objetos (un tubo por ejemplo). También se observa el uso de procedimientos virtuales, esto significa que si existe un procedimiento en el objeto "hijo" con el mismo nombre, éste código sustituye al del padre. El procedimiento Inicio_Dibujo puede ser definido dentro del objeto Tubo, lo que da la característica de ***polimorfismo*** ya que el mismo procedimiento puede ser codificado de distinta forma para otros objetos (Tubo, Pozo, Nudo y Tanque).

Modularidad: Pascal permite crear módulos llamados unidades (Unit). Al momento de construir el sistema. Las unidades permiten agrupar funciones y procedimientos que se pueden programar en forma independiente para después ser llamados desde el programa principal. Adicionalmente Pascal esta considerado como un lenguaje procedural.

Jerarquia: Anteriormente se presentó una parte de la jerarquía de clases que puede manejar Pascal, cada objeto que se define tiene una serie de procedimientos asociados que se **heredan** a los objetos de las clases inferiores ej.:

TObject tiene definidos los siguientes procedimientos

- Init
- Done
- Free

TWindowsObject hereda estos procedimientos y adicionalmente tiene definidos:

- Destroy
- Show
- SetupWindow
- GetId
- GetWindowClass
- etc.

Twindow Hereda las características de TObject y TWindowsObject y a su vez agrega más procedimientos, o reemplaza algunos de los ya definidos. Para un listado completo de los procedimientos y campos que contiene cada objeto consultar el manual **ObjectWindows Programming Guide** (Chapter 21 Object Windows Reference).

De la misma forma, las clases definidas por el usuario heredarán los métodos definidos en sus ancestros. Por ejemplo, se define *PTubo* cómo "hijo" de *TElemento*:

```
PTubo = Elemento;                    { Define el objeto Tubo }
Elemento = object(TElemento)
    { Acción del Mouse }
  procedure Boton_Derecho_Abajo( X, Y: Integer); virtual;
  procedure Inicio_Dibujo(X, Y: Integer); virtual;
  procedure Fin_Dibujo(X, Y: Integer); virtual;
end;
```

Esto significa que el objeto Tubo (PTubo) hereda los procedimientos y características de TElemento e incorpora otro procedimiento llamado Boton_Derecho_Abajo. En este caso los procedimiento Inicio_Dibujo y Fin_Dibujo serán codificados en este nivel de la jerarquía. El código de estos procedimientos reemplazarán a los que estén definidos para TElemento (Si es que existieran). Los otros procedimientos serán tomados de TElemento; una vez más se esta hablando de polimorfismo.

Concurrencia: La concurrencia de eventos no la da propiamente Object Pascal pero la proporciona Windows al permitir que varias aplicaciones corran en forma "simultánea".

En el capitulo siguiente se explicará como implementar y manejar cada objeto que será utilizado dentro de CRAP. Se analizarán también otras facilidades que ofrece Object Pascal para aplicaciones windows como pueden ser el manejo del mouse, menú, iconos y cursores, entre otros.

Borland Pascal v7.0

4. Borland Pascal v7.0

En los capitulos anteriores se han estudiado las bases que sustentan la Programación y el Diseño Orientado a Objetos y se ha hecho un análisis de lo que será el sistema de Captura de Redes

de Agua Potable (CRAP). En este capítulo se revisará cómo aplicar esos conceptos utilizando el lenguaje de programación Borland Pascal v7.0(en adelante BP). El objetivo de este capítulo es explicar las bases para poder crear una aplicación bajo ambiente Windows utilizando BP siguiendo la metodología de la Programación Orientada a Objetos. Al ir desarrollando el capítulo se irá construyendo CRAP.

El presente trabajo no pretende ser un manual de BP, por lo que es conveniente que el lector tenga experiencia en el manejo de Turbo Pascal (versión 4 o posterior). De esta forma se comenzará a dar algunas consideraciones importantes para generar una aplicación windows.

El elemento de Borland Pascal v7.0 que permite la creación y manejo de objetos windows es conocido como ObjectWindows. ObjectWindows incluye unidades para poder generar dichas aplicaciones. En este momento sólo se mencionarán las unidades básicas para cualquier aplicación Windows:

- ☑ OWindows: Contiene los objetos estándares usados por BP para aplicaciones windows.
- ☑ ODialog: Incluye ventanas de diálogos y los controles asociados a estas (Botones, Listas, etc.)

Todos las aplicaciones windows tienen una ventana principal que aparece cuando el usuario ejecuta la aplicación. El usuario da por terminada la aplicación cerrando la ventana principal.

4.1. TApplication

Cada Aplicación Windows debe estar definida como un objeto de la clase **TApplication**. Como se comento en el capitulo anterior TApplication forma parte de la estructura de clases ya definidas por Pascal. Por lo que la estructura básica de toda aplicación windows será la siguiente:

Ejemplo 01

```
{***********************************************}
{                                               }
{   Programa: CRAP_01.PAS                       }
{                                               }
{   Estructura básica de una aplicación Windows }
{                                               }
{***********************************************}

program Captura_de_Redes_de_Agua_Potable;

uses OWindows;

type
  CRAPApplication = object(TApplication)
    procedure InitMainWindow; virtual;
  end;

procedure CRAPApplication.InitMainWindow;
begin
  MainWindow := New(PWindow, Init(nil, 'CRAP'));
end;

var
  CRAP: CRAPApplication;

begin
  CRAP.Init('CRAP');
  CRAP.Run;
  CRAP.Done;
end.
```

Es importante notar que en el programa principal se hacen las llamadas a los siguientes procedimientos:

INIT: Es el creador (constructor) del objeto CRAP, en este punto se crea la ventana principal de la aplicación.

RUN: Pone en movimiento el objeto.

DONE: es el destructor del objeto CRAP.

Al crear una nueva aplicación es necesario redefinir por lo menos el método **InitMainWindow**. Es por esto que se declara como un procedimiento virtual al definir CRAPApplication, para poder codificar este procedimiento de acuerdo a las necesidades de la aplicación (en el ejemplo sólo se define una ventana con el título de 'CRAP').

CRAPApplication.InitMainWindow es llamado automáticamente por ObjectWindows (Interfaz de BP con Windows) para mostrar la ventana principal de la aplicación.

El ejemplo mostrado anteriormente no realiza ninguna función, sólo muestra una clásica ventana de windows (en blanco). El usuario utilizando el mouse puede mover de lugar la ventana, ajustarla de tamaño, maximizarla, minimizarla y cerrarla (propiedades básicas de cualquier aplicación windows).

4.2. TWindow

El ejemplo 01 consta de 2 objetos: Un objeto *Aplicación* y un objeto *ventana*. CRAP es el objeto aplicación y es una instancia de CRAPApplication, derivado de la clase TApplication. El campo MainWindow de CRAP contiene el objeto ventana, que es una instancia de **TWindow**.

En cualquier aplicación es necesario definir un objeto ventana propio para el campo MainWindow. AL definir la ventana principal se pueden incorporar las funciones especificas de la aplicación. En esta sección se analizará TWindow para poder definir la ventana principal propia de la aplicación.

Al igual que un objeto Aplicación, un objeto ventana encapsula sus propias responsabilidades:

- ☑ Ajuste de tamaño (resize)
- ☑ Cerrado
- ☑ Respuesta a eventos
  - Selección de una opción del menú
  - Movimiento del mouse
  - etc.
- ☑ Desplegar controles
  - Botones
  - Listas de selección
  - Campos de entrada
  - etc.

TWindow y su ancestro **TwindowsObject** proveen los métodos y campos para las funciones básicas de una ventana. Todos los objetos ventana (TWindowObject) tienen por lo menos 3 campos:

HWINDOW: Es un número único que asocia la ventana con sus objetos interfaces (otras ventanas, diálogos, controles, etc.).

PARENT: Contiene el apuntador de la ventana a su objeto 'padre'.

CHILDLIST: Contiene la lista de ventanas 'hijas'.

Twindow agrega otros campos como:

ATTR: Define los atributos de la ventana al momento de la creación, esta es la estructura de Attr:

```
Attr: TWindowAttr;

TWindowAttr = record
   Title: PChar; { Título de la ventana }
   Style: Longint; { Estilo de la ventana }
   ExStyle: Longint; { Estilo extendido de la ventana }
   X, Y, W, H: Integer; { Posición y tamaño de la
                          ventana }
   Param: Pointer;
   case Integer of
    0: (Menu: HMenu);  { menú principal o... }
    1: (Id: Integer);  { identificador de ventana hija}
end;
```

Inicialmente se usará la siguiente definición para usarla como ventana principal para CRAP.

```
Type
   PCrapWindow = ^TCrapWindow
   TCrapWindow = Object(TWindow)
   {Se definen las variables y procedimientos de la ventana}
   End;
```

Esta definición se sustituye en el ejemplo 01 dando:

Ejemplo 02

```
{*********************************************}
{                                             }
{   Programa: CRAP_02.PAS                     }
{                                             }
{   Estructura básica de una aplicación Windows. }
{                                             }
{   Define una ventana de 400*200 pixeles y de  }
{   tamaño no ajustable.                      }
{   Confirma la salida con un mensaje.        }
{*********************************************}

program Captura_de_Redes_de_Agua_Potable;

uses WinTypes, OWindows;

type

  PCrapWindow = ^TCrapWindow;
  TCrapWindow = Object(TWindow)
    Bandera    : Boolean;
    constructor Init(AParent: PWindowsObject; ATitle: PChar);
    funotion CanClose: Boolean; virtual;
  end;

  CRAPApplication = object(TApplication)
   procedure InitMainWindow; virtual;
  end;

 {* Se codifican los procedimientos       *}

  procedure CRAPApplication.InitMainWindow;

  begin
     MainWindow := New(PCrapWindow, Init(nil, 'CRAP'));
  end;

  constructor TCrapWindow.Init(AParent: PWindowsObject; ATitle: PChar);
     begin
        inherited Init(AParent, ATitle);
        Bandera := True;
        { WS_SysMenu = Menu del sistema (System menu Box)}
        { WS_MinimizeBox = Botón para minimizar la ventana}
        Attr.style:=WS_SysMenu or WS_MinimizeBox;
        { Se define el tamaño de la ventana}
        Attr.X:=0;
        Attr.Y:=0;
        Attr.H:=200;
        Attr.W:=400;
     end;

   function TCrapWindow.CanClose: Boolean;

      var
```

```
      Respuesta: Integer;
    begin
      CanClose := True;
      { Si Bandera = True se confirma la salida,
           si la respuesta es NO, la aplicación continúa}
      if Bandera then
        begin
          Respuesta := MessageBox(HWindow, '¿ Confirmando Salida ?',
           'Titulo del mensaje', mb_YesNo or mb_IconQuestion);
          if Respuesta = id_No then CanClose := False;
        end;
    end;


var
  CRAP: CRAPApplication;

begin
  CRAP.Init('CRAP');
  CRAP.Run;
  CRAP.Done;
end.
```

Ahora que ya se tiene la ventana principal se puede redefinir uno de los procedimientos o agregar nuevos. Se comenzará redefiniendo **INIT** y **CANCLOSE**.

4.2.1. INIT

Al momento de crear la ventana:

```
MainWindow := New(PCrapWindow, Init(nil, 'CRAP'));
```

Se invoca automáticamente el 'constructor' INIT (No confundir con INIT de TApplication) y al momento de cerrar la ventana se invoca la función CANCLOSE.

Si se redefine el código de INIT se pueden inicializar variables de la aplicación, definir atributos especiales de la ventana, abrir nuevas ventanas o inicializar otros objetos. En el ejemplo 02 se define una ventana de 400*200 pixeles, de tamaño fijo (no resizable) y sólo se puede minimizar.

Es obvio pensar que si se redefine INIT se tendrán que codificar aquellos procesos que se necesitan ejecutar al momento de inicializar una ventana y adicionar las definiciones de la aplicación. Para evitar esta recodificación se introduce como primera instrucción:

```
inherited Init(AParent, ATitle);
```

Inherited es una facilidad de BP. Invoca el método (Init) del ancestro y evita la recodificación de todo el método.

4.2.2. CANCLOSE

Una aplicación windows se da por terminada (por ejemplo) cuando el usuario selecciona la opción Cerrar (Close) del menú de control (pequeño recuadro en la esquina superior izquierda de la ventana). Cuando el usuario intenta cerrar la aplicación, Windows manda el mensaje *wm_Close* a la ventana principal. La ventana principal llama la función CANCLOSE, si esta función regresa un valor verdadero termina la aplicación.

Al modificar la función CANCLOSE se puede controlar si la aplicación termina o no. Por ejemplo, si estando editando una red de agua potable y el usuario intenta cerrar la aplicación (sin salvar la red), se puede desplegar un mensaje (en forma de ventana) para confirmar la operación o en su caso invocar el procedimiento de salvado antes de cerrar la aplicación.

En el ejemplo 02 se redefine INIT y CANCLOSE. INIT inicializa la variable BANDERA, se le da un tamaño y características especiales a la ventana (consultar manual ObjectWindows Reference para una lista detalla

de las características posibles). CANCLOSE consulta la variable BANDERA (en el ejemplo siempre es verdadera) y manda un mensaje para confirmar la salida, posteriormente se revisará a detalle el manejo de diálogos y ventanas de mensajes.

4.3. Manejo del Mouse

Cuando el mouse se mueve o algún botón es presionado dentro de la ventana, Windows manda un mensaje a la aplicación. El mensaje es interceptado por ObjectWindows y enviado al objeto ventana correspondiente.

El mensaje lleva información relacionada con el evento (ej. las coordenadas del mouse). Todos los mensajes (incluyendo las opciones del menú) son representados por números. Cada método que responde a un mensaje debe de tener un número único. Para facilitar el manejo de mensajes, ObjectWindows define constantes para algunos de ellos (se debe incluir la unidad Wintypes).

En este tema se revisarán los métodos y mensajes relacionados con el manejo del mouse. En otro tema se profundizará en el tratamiento de mensajes.

A continuación se presenta una tabla donde se relaciona el evento y el mensaje que se genera:

Evento	Mensaje
Movimiento del mouse.	wm_MouseMove
Botón izquierdo presionado.	wm_LButtonDown
Botón izquierdo liberado (estaba presionado y el usuario lo suelta).	wm_LButtonUp
Doble click con el botón izquierdo.	wm_LButtonDblclk
Botón intermedio presionado.	wm_MButtonDown

Botón intermedio liberado (estaba presionado y el usuario lo suelta).	wm_MButtonUp
Doble click con el botón intermedio.	wm_MButtonDblclk
Botón derecho presionado.	wm_RButtonDown
Botón derecho es liberado (estaba presionado y el usuario lo suelta).	wm_RButtonUp
Doble click con el botón derecho.	wm_RButtonDblclk

Esto facilita la programación del mouse, pues sólo se necesita codificar el evento que se desea controlar. En el ejemplo siguiente se despliega una caja con un letrero cuando el botón izquierdo o derecho es presionado.

Ejemplo 03

```
{*********************************************}
{                                             }
{   Programa: CRAP_03.PAS                     }
{                                             }
{   Manejo de Mouse                           }
{                                             }
{   Despliega un mensaje cuando se presiona   }
{   algún botón del mouse.                    }
{                                             }
{*********************************************}

program Captura_de_Redes_de_Agua_Potable;

uses WinTypes, OWindows;

type

  PCrapWindow = ^TCrapWindow;
  TCrapWindow = Object(TWindow)
    Bandera   : Boolean;
    constructor Init(AParent: PWindowsObject; ATitle: PChar);
    function CanClose: Boolean; virtual;
    procedure WMLButtonDown(var Msg: TMessage);
              virtual wm_First + wm_LButtonDown;
    procedure WMRButtonDown(var Msg: TMessage);
              virtual wm_First + wm_RButtonDown;
  end;

  CRAPApplication = object(TApplication)
```

```
  procedure InitMainWindow; virtual;
end;

{* Se codifican los procedimientos       *}
procedure CRAPApplication.InitMainWindow;
begin
   MainWindow := New(PCrapWindow, Init(nil, 'CRAP'));
end;

constructor TCrapWindow.Init(AParent: PWindowsObject; ATitle: PChar);
   begin
      inherited Init(AParent, ATitle);
      Bandera := False;
      { Se define el tamaño de la ventana}
      Attr.X:=0;
      Attr.Y:=0;
      Attr.H:=200;
      Attr.W:=400;
   end;

 function TCrapWindow.CanClose: Boolean;

    var
       Respuesta: Integer;
    begin
       CanClose := True;
       { Si Bandera = True se confirma la salida,
           si la respuesta es NO, la aplicación continúa}
       if Bandera then
          begin
             Respuesta := MessageBox(HWindow, '¿ Confirmando Salida ?',
              'Titulo del mensaje', mb_YesNo or mb_IconQuestion);
             if Respuesta = id_No then CanClose := False;
          end;
    end;

 procedure TCrapWindow.WMRButtonDown(var Msg: TMessage);

    begin
       MessageBox(HWindow, 'Botón Derecho presionado',
                           'Manejo de mouse', mb_OK);
    end;

  procedure TCrapWindow.WMLButtonDown(var Msg: TMessage);

    begin
       Bandera:=True;
       MessageBox(HWindow, 'Botón Izquierdo Presionado',
                           'Manejo de mouse', mb_OK);
    end;

var
  CRAP: CRAPApplication;

begin
  CRAP.Init('CRAP');
  CRAP.Run;
  CRAP.Done;
end.
```

MSG es un registro del tipo *TMessage*, este registro es el que contiene la información relacionada al evento. La estructura de MSG es siempre la misma pero La información que guarde MSG dependerá del evento. Esta es la definición de MSG y la información que guarda para los eventos relacionado con el mouse:

```
TMessage = record
   Receiver: HWnd;        <-- Ventana que recibe el msg.
   Message: Word;              <-- Identificador del msg.
   case Integer of
     0: (WParam: Word;    <-- Bandera para los otros botones.
         LParam: Longint;
         Result: Longint);
     1: (WParamLo: Byte;
         WParamHi: Byte;
         LParamLo: Word; <-- Posición horizontal del mouse (X)
         LParamHi: Word; <-- Posición vertical del mouse (Y)
         ResultLo: Word;
         ResultHi: Word);
   end;
```

WParam es utilizado para saber si otro botón, la tecla de Ctrl o Shift están siendo presionados simultáneamente (consultar manual ObjectWindows Reference para valores posibles).

Con LParamLo y LparamHi se identifica la posición del mouse.

Es importante notar que sólo los procedimientos WMLButtonDown y WMMouseMove son derivados de TWindow. Los otros procedimientos (ej. WMRButtonDown) se están anexando al objeto (TCrapWindow).

4.4. Ambiente Gráfico

Windows tiene un conjunto de funciones llamadas Graphics Device Interface (GDI). Estas funciones proveen para un ambiente gráfico el manejo de líneas, figuras, texto e imágenes (BitMap). Borland Pascal asocia un manejador al dispositivo (físico) con el que se esta trabajando.

Un dispositivo gráfico (DC = Device context) es una superficie virtual con atributos asociados como pueden ser una pluma (pen), brocha (brush), tipo de letra (font), color del fondo (background), color del texto y una dimensiones.

Cuando se llaman las funciones GDI para dibujar en un DC se hace una conversión de las instrucciones. El manejador asociado al dispositivo traduce las instrucciones en los comandos apropiados para dibujar. Estos comandos reproducen el dibujo lo más exacto posible en el dispositivo no importando las características físicas del dispositivo (EGA, VGA, SVGA, etc.).

El DC representa sólo la parte del dispositivo en el que se puede dibujar (ej. el área de una ventana). Un DC es un elemento administrado por windows y no un objeto de ObjectWindows.

A fin de poder dibujar dentro del DC, la aplicación primero deberá de obtener el DC de la ventana correspondiente. Lo anterior se debe a que la memoria que se requiere para el DC es grande, por lo que sólo se permite accesar 5 DC concurrentemente por cada sesión de Windows. Esto significa que cada ventana debe de obtener el DC sólo cuando lo necesite y liberarlo en cuando le sea posible. Por lo que el proceso para dibujar cualquier gráfico (incluyendo texto) es el siguiente:

- ☑ Obtener el Device Context.
- ☑ Dibujar.
- ☑ Liberar el Device Context.

Para ejemplificar lo anterior, se desea dibujar una serie de puntos en forma de malla para CRAP. Como los puntos (pixel) son muy pequeños se dibujarán líneas de 2 pixeles de largo para simular dichos puntos. Por el momento se programará que la malla se dibuje cada vez que se presione el botón izquierdo del mouse, quedando el programa:

Ejemplo 04
(sólo se muestra la parte del código que se esta revisando)

```
.
.
  PCrapWindow = ^TCrapWindow;
  TCrapWindow = Object(TWindow)
    DC_Actual : HDC;
    Bandera   : Boolean;
    Procedure Dibuja_Malla;
    constructor Init(AParent: PWindowsObject; ATitle: PChar);
    function CanClose: Boolean; virtual;
    procedure WMLButtonDown(var Msg: TMessage);
              virtual wm_First + wm_LButtonDown;
    procedure WMRButtonDown(var Msg: TMessage);
              virtual wm_First + wm_RButtonDown;
  end;
.
.
  Procedure TCrapWindow.Dibuja_Malla;

  Var i,j : Integer;

     Begin
        { Se obtiene el Device Context}
        DC_Actual:=GetDC(Hwindow);

        { Se dibuja la malla}
        For i:=0 to (400 Div 20) do
           for j:=0 to (200 div 20) do
              Begin
                 Moveto(DC_Actual,i*20,j*20);
                 Lineto(DC_Actual,i*20+1,j*20+1);
              End;

        { Se libera el Device Context}
        ReleaseDC(Hwindow,DC_Actual);
     End;

    procedure TCrapWindow.WMLButtonDown(var Msg: TMessage);

      begin
         Bandera:=True;
         Dibuja_Malla;
      end;
.
.
```

Lo ideal es que la malla se dibuje al iniciar la aplicación. Se puede pensar que el dibujo de la malla se puede hacer en INIT pero es un error. Recuerde que INIT sólo se utiliza para inicializar variables y fijar atributos de la ventana. La ventana puede ser utilizada para dibujar después que ha terminado el constructor INIT. En el siguiente tema se explicará cómo trazar un gráfico al momento de abrir la aplicación.

En el ejemplo anterior la malla se dibuja al momento de presionar el botón izquierdo del mouse. Si la aplicación se minimiza y después se restaura la malla se pierde pues la ventana no conserva los gráficos desplegados con anterioridad. Esto es importante tenerlo en mente al momento de diseñar una aplicación. Se debe de implementar un mecanismo para que la aplicación sea capaz de regenerar los gráficos que se van generando durante la ejecución de la aplicación.

4.5. Pintar y Dibujar en una ventana

La documentación de ObjectWindows maneja dos conceptos diferentes: "pintando" (painting) y "dibujando" (drawing). Estos dos conceptos hacen mención a la actividad de desplegar gráficos en una ventana pero tienen sus diferencias.

Pintar se refiere a desplegar automáticamente gráficos cuando la ventana aparece por primera vez o necesita ser actualizada. En cambio, dibujar se refiere a desplegar un gráfico específico en cualquier momento. El término gráfico se usa tanto para texto como para elementos gráficos (Bitmaps, Líneas, rectángulos, etc.).

4.5.1. Pintando

Pintar es el proceso de desplegar el contenido de una ventana. Una aplicación Windows es responsable de "pintar" su ventana cuando ésta es desplegada o cuando necesita ser actualizada. Por ejemplo al activar la aplicación desde un icono o cuando anteriormente fue cubierta por otra ventana.

Una ventana necesita ser "pintada" cuando ha sido invalidada, esto significa que el contenido de la ventana ya no es válido y necesita ser actualizado. Lo anterior puede ser resultado de varios eventos:

- ☑ La ventana se despliega por primera vez.
- ☑ La ventana es restaurada desde un icono.
- ☑ Se cambia de tamaño la ventana o se maximiza.
- ☑ Otra ventana (cualquier aplicación) es sobrepuesta parcial o completamente sobre la ventana activa (la ventana se invalida al momento que regresa el control a la aplicación).
- ☑ La ventana es invalidada por comando: (InvalidateRect(Hwindow,Nil,True).

En cualquiera de estos casos, Windows manda el mensaje *wm_Paint* a la aplicación. Este mensaje invoca al método Paint. Este método automáticamente obtiene el DC **PaintDC** y al final lo libera, pero queda a responsabilidad de la aplicación pintar la ventana. Esto significa que el método Paint debe ser codificado de tal forma que tenga la capacidad de actualizar la ventana. Paint debe de 'recordar' todos los gráficos agregados y/o borrados desde que la ventana fue desplegada por primera vez.

Para ejemplificar lo anterior, se codificará en el método Paint el dibujo de la malla. De esta forma se evitará que la malla se pierda en caso de minimizar la aplicación o cuando otra ventana sea sobrepuesta. La malla aparecerá al momento de arrancar la aplicación:

Ejemplo 05 (Programa CRAP_04.PAS)

```
.
.
PCrapWindow = ^TCrapWindow;
  TCrapWindow = Object(TWindow)
    Bandera   : Boolean;
    constructor Init(AParent: PWindowsObject; ATitle: PChar);
    function CanClose: Boolean; virtual;
    Procedure Paint(PaintDC: HDC; var PaintInfo: TPaintStruct); virtual;
    procedure WMLButtonDown(var Msg: TMessage);
              virtual wm_First + wm_LButtonDown;
    procedure WMRButtonDown(var Msg: TMessage);
              virtual wm_First + wm_RButtonDown;
  end;
.
.
   Procedure TCrapWindow.Paint(PaintDC: HDC; var PaintInfo:
                               TPaintStruct);

      Var i,j : Integer;

      begin
          { Se dibuja la malla}
          For i:=0 to (400 Div 20) do
           For j:=0 to (200 div 20) do
              Begin
                 Moveto(PaintDC,i*20,j*20);
                 Lineto(PaintDC,i*20+1,j*20+1);
              End;
      end;
.
.
```

Note que en el método Paint no se captura y libera el DC PaintDC, pues como se mencionó anteriormente, esto lo hace en forma automática el método Paint.

4.6. Manejo de Recursos

Borland Pascal ofrece una herramienta llamada **Resource Workshop**. Resource Workshop (RW) permite crear y modificar una serie de recursos, los cuales pueden ser utilizados por ObjectWindows. Los recursos se pueden crear sin necesidad de usar RW, esto significa que se tiene que programar su definición. El utilizar Resource Workshop permite un ahorro en la codificación de la aplicación.

Los recursos son elementos de interfase con el usuario y la aplicación Windows. Los recursos se almacenan en forma de datos en un archivo ejecutable (dentro de la aplicación).

Los recursos que utiliza CRAP y que se pueden editar a través de Resource Workshop son:

- ☑ Menú Principal
- ☑ Ventanas de Diálogos
- ☑ Bitmaps
- ☑ Iconos
- ☑ Cursores

Estos no son los únicos recursos que puede manejar Resource Workshop. Este trabajo no pretende ser un manual sobre el uso de RW, por lo que sólo se explicará cómo utilizar los recursos ya creados y se darán algunos puntos importantes al momento de crear los recursos. Para una explicación completa del manejo de RW consultar el manual Resource Workshop User's guide.

4.6.1. Menú Principal

En una aplicación Windows, el menú no es un objeto separado, es un atributo de la ventana principal. Todos los objetos tipo ventana tienen un conjunto de atributos. Estos atributos se almacenan en el registro Attr (campo de Twindow).

Attr contiene un campo MENU en el cual se guarda el controlador del menú. Esta operación debe de hacerse al momento de construir la ventana, por lo que es necesario redefinir el método "Constructor".

Para incluir el menú principal en una aplicación se deben de seguir los siguientes pasos:

1. Diseñar el menú cómo un recurso tipo MENU (utilizando RW).
2. Definir las constantes del menú.
3. Incluir el archivo de recursos dentro del programa.
4. Cargar el menú dentro de la ventana principal.
5. Definir las responsabilidades de cada opción del menú.

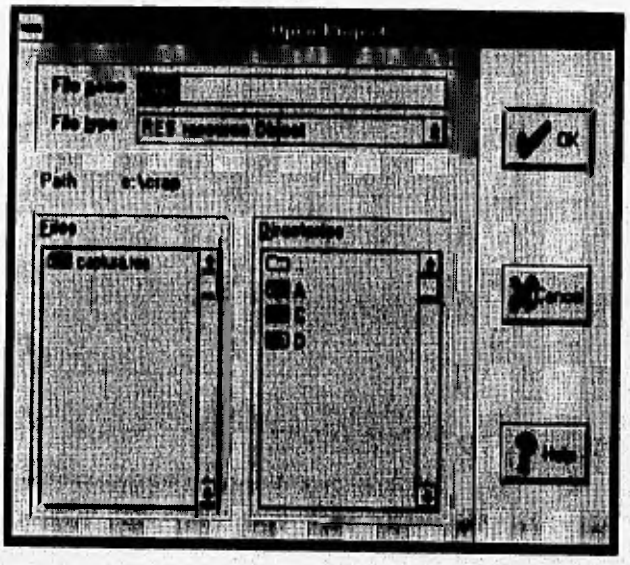
Ahora se explicarán cada uno de éstos puntos:

1. Diseñar el menú cómo un recurso tipo MENU.

Para entrar a Resource Workshop se puede hacer desde el menú principal de Borland Pascal (Tools) o desde el icono correspondiente a RW.

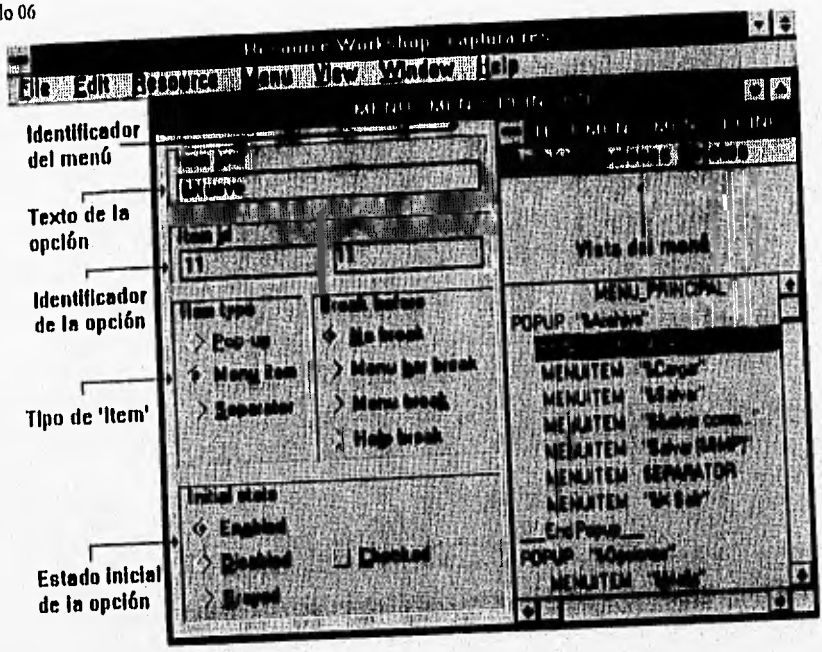
Ya dentro de Resource Workshop abrir un archivo (project) tipo RES (Resource Object).

Ejemplo 05



Una vez que se ha abierto el archivo de recursos se utiliza el menú de RW para crear los distintos objetos. Para éste caso se genera el menú principal:

Ejemplo 06



Aquí hay puntos importantes que mencionar:

- ☑ **Identificador del Menú** (MENU_PRINCIPAL). Es el nombre del recurso, éste nombre se asociará a la ventana principal.
- ☑ **Texto de la Opción.** Nombre de la opción, la letra después del símbolo **&** se desplegará subrayada. Ej. &Nuevo -> Nuevo.
- ☑ **Identificador de la opción.** Cada opción debe estar asociada con un número, este número formará parte del mensaje que será emitido al seleccionar dicha opción. Posteriormente se explicará cómo definir un procedimiento que responda a un mensaje determinado. De esta forma la aplicación aparentará llamar un procedimiento al seleccionar una opción. El número que se asocia a cada opción puede ser arbitrario. En ciertas opciones es recomendable utilizar las constantes ya definidas por ObjectWindows. Por ejemplo, ObjectWindows define el **24340** para terminar una aplicación.
- ☑ **Tipo de 'Item'**. existen 3 tipos de elementos:

1. Pop-Up. Equivale a un submenú
2. Menu item. Opción del submenú
3. Separator. Linea divisoria (Se utiliza para dar formato al menú, no requiere ningún parámetro).

☑ **Estado Inicial de la opción.** se define cómo se va a desplegar inicialmente la opción.

☑ **Vista del menú.** AL momento de agregar las opciones (items) se va desplegando la forma que tomará el menú.

2. Definir las constantes del menú.

Una vez diseñado el menú, cada opción tiene un identificador, se recomienda crear constantes con dicho valor para facilitar su manejo al momento de asignar las responsabilidades de cada opción. Para CRAP se han hecho las siguientes definiciones:

Ejemplo 07

```
const

     cm_Nuevo       = 11; {El mismo valor que se utilizo al diseñar el menú}
     cm_Cargar      = 12;
     cm_Salvar      = 13;
     cm_Salvar_Como = 15;
     cm_Salvar_BMP  = 14;
     cm_Malla       = 21;
     cm_200pc       = 220;
     cm_100pc       = 221;
     cm_50pc        = 222;
     cm_25pc        = 223;
     cm_Verifica    = 23;
     cm_Tubo        = 31;
     cm_Nodo        = 32;
     cm_Pozo        = 33;
     cm_Tanque      = 34;
```

3. Incluir el archivo de recursos dentro del programa.

Antes de definir las responsabilidades de cada opción del menú es necesario incluir el archivo generado por Resource Workshop en el código de la aplicación. Para esto se utiliza la directiva de compilación **$R** de la siguiente forma:

{$R Captura.Res}

Esta directiva de compilación agrega automáticamente el archivo de recursos al programa ejecutable.

4. Cargar el menú dentro de la ventana principal.

Para cargar el menú a la ventana es necesario redefinir el constructor INIT de la siguiente forma:

Ejemplo 08

```
constructor TCrapWindow.Init(AParent: PWindowsObject; ATitle: PChar);
  begin
    inherited Init(AParent, ATitle);
    Attr.Menu:=LoadMenu(Hinstance,'MENU_PRINCIPAL');
      .
      .
      .
  end;
```

Es importante mencionar que se llama al menú con el mismo nombre con que se identificó en Resource Workshop.

5. Definir las responsabilidades de cada opción del menú.

Al seleccionar una opción del menú se genera un mensaje. Esto significa que se necesita interceptar los mensajes emitidos por el menú y asignarles una responsabilidad.

Los mensajes que se generan tienen el siguiente valor:

cm_First + Identificador de la opción.

Cm_First es una constante de ObjectWindows para el manejo de mensajes tipo Comando de menú (Cm).

La asignación de las responsabilidades quedará de la siguiente forma:

Ejemplo 09 (Programa CRAP_05.PAS)

```
program Captura_de_Redes_de_Agua_Potable;

uses WinTypes, OWindows;

{$R Captura.Res}

const

     cm_Nuevo       = 11; {El mismo valor que se utilizo al diseñar el menú}
     cm_Cargar      = 12;
     cm_Salvar      = 13;
     cm_Salvar_Como = 15;
     cm_Salvar_BMP  = 14;
     cm_Malla       = 21;
     cm_200pc       = 220;
     cm_100pc       = 221;
     cm_50pc        = 222;
     cm_25pc        = 223;
     cm_Verifica    = 23;
     cm_Tubo        = 31;
     cm_Nodo        = 32;
     cm_Pozo        = 33;
     cm_Tanque      = 34;

type

PCrapWindow = ^TCrapWindow;
  TCrapWindow = Object(TWindow)
    Bandera   : Boolean;
    constructor Init(AParent: PWindowsObject; ATitle: PChar);
    function CanClose: Boolean; virtual;
    Procedure Paint(PaintDC: HDC; var PaintInfo: TPaintStruct); virtual;
    procedure WMLButtonDown(var Msg: TMessage);
            virtual wm_First + wm_LButtonDown;
    procedure WMRButtonDown(var Msg: TMessage);
            virtual wm_First + wm_RButtonDown;
    Procedure Activa_Salvado(Var Msg:Tmessage);
      Virtual cm_First + cm_Salvar;
    Procedure Activa_Salvado_Como(Var Msg:Tmessage);
      Virtual cm_First + cm_Salvar_Como;
    Procedure Activa_Salvar_BMP(Var Msg:Tmessage);
      Virtual cm_First + cm_Salvar_BMP;
    Procedure Activa_Nuevo(Var Msg:Tmessage);
      Virtual cm_First + cm_Nuevo;
```

```
          .
          .
          .
  end;
.
.
.
  constructor TCrapWindow.Init(AParent: PWindowsObject; ATitle: PChar);
     begin
        inherited Init(AParent, ATitle);
        Bandera := False;
        { Se define el tamaño de la ventana}
        Attr.X:=0;
        Attr.Y:=0;
        Attr.H:=200;
        Attr.W:=400;
        Attr.Menu:=LoadMenu(Hinstance,'MENU_PRINCIPAL');
     end;
```

Observe que al utilizar las constantes definidas en el punto 2 es más fácil identificar cada opción.

Una vez que se asocia un método con un mensaje sólo resta codificar el procedimiento (Ej. TCrapWindow.Activa_Nuevo) para que ejecute las operaciones correspondientes a su responsabilidad.

4.6.2. Ventanas de Diálogos

En ejemplos anteriores se han manejado algunos diálogos, por ejemplo:

1. `Respuesta := MessageBox(HWindow, '¿ Confirmando Salida ?', 'Título del mensaje', mb_YesNo or mb_IconQuestion);`

2. `MessageBox(HWindow, 'Botón Derecho presionado', 'Manejo de mouse', mb_OK);`

Este tipo de diálogos (MessageBox) son muy convenientes para avisos ó para confirmar algún evento. Sin embargo, para establecer una interfaz muy específica con el usuario (ej. Leer información) es necesario definir diálogos (dialog box) propios de la aplicación

Al igual que el menú principal, los diálogos se pueden diseñar desde Resource Workshop o vía comandos. ObjectWindows utiliza un objeto

para representar un diálogo, porque los programas tienen que interactuar con éste de diferentes formas, situación que no sucede con el menú por ejemplo.

Para crear un diálogo se recomiendan seguir los siguientes pasos:

1. Diseñar el diálogo como un recurso tipo Dialog Box.
2. Definir como constantes los identificadores de cada control.
3. Incluir el archivo de recursos dentro de programa.
4. Definir el buffer de datos, el diálogo y los controles.
5. Inicializar buffer de datos.
6. Crear el diálogo y los controles.
7. Asociar el buffer de datos y ejecutar el diálogo.

Ahora se explicarán cada uno de éstos puntos:

1. Diseñar el diálogo como un recurso tipo Dialog Box.

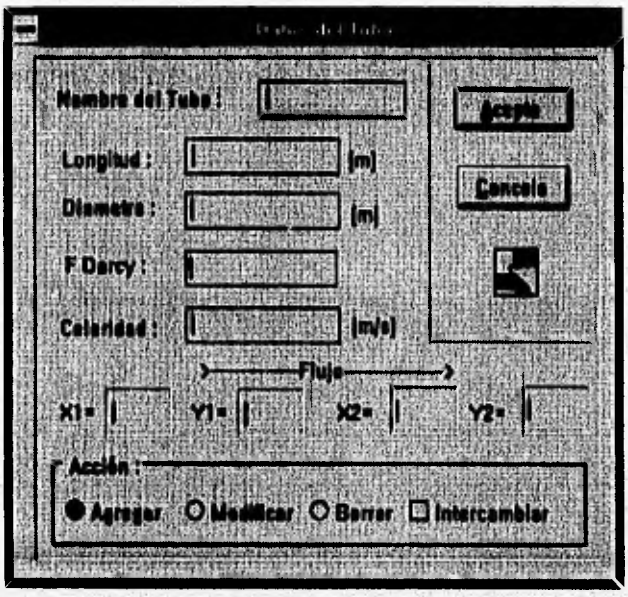
Un diálogo es una ventana especializada que contiene uno o más controles. Cuando se mencionan controles se esta hablando de botones, listas, iconos, etc. Al programar una aplicación Windows, el programador no tiene que preocuparse por la apariencia de los controles o su posición, lo único que tiene que conocer es de que tipo es el control y que identificador tiene asociado.

Así como cada opción del menú principal tiene un identificador asociado, cada control de un recurso tipo diálogo tiene su propio identificador. Este identificador es utilizado por la aplicación para distinguir sobre cual control se desea interactuar. Existen controles que no tienen interacción con el programa, por ejemplo una imágen (BitMap) o un texto descriptivo. Este tipo de controles no necesitan un identificador único, normalmente se les asigna el -1 a todos.

De la misma forma que en el menú principal, es recomendable crear un constante por cada identificador que se vaya a utilizar dentro de la aplicación.

El siguiente es un ejemplo de un diálogo creado usando Resource Workshop.

Ejemplo 10



El diálogo fue nombrado **DLG_TUBO** y se usaron los siguientes números de identificación (sólo se muestran algunos):

Control	Características	Identificador
Botón (Acepta)	Push Button	1
Botón (Cancela)	Push Button	2
Texto editable (Nombre del Tubo)	Edit Text	1001
Texto editable (X1)	Edit Text (Read Only)	1002
Botón (Agregar)	Auto Radio Button	1007
Botón (Intercambiar)	Auto Check Box	1011
Icon (Tubos)		-1

Aquí es importante mencionar que los dos primeros botones (Acepta y Cancela) tienen el identificador 1 y 2 respectivamente. Estos identificadores ObjectWindows los reconoce para terminar el diálogo (1 = OK, 2 = Cancel).

2. Definir como constantes los identificadores de cada control.

Para facilitar su manejo se define una constante por cada identificador:

Ejemplo 11

```
Const
     {Identificadores para el diálogo del tubo }

     Tubo_Nombre_Id      = 1001;
     Tubo_Longitud_Id    = 1009;
     Tubo_Diametro_Id    = 1019;
     Tubo_FDarcy_Id      = 1013;
     Tubo_Celeridad_Id   = 1003;
     Tubo_X1_Id          = 1002;
     Tubo_Y1_Id          = 1004;
     Tubo_X2_Id          = 1005;
     Tubo_Y2_Id          = 1006;
     Tubo_Agrega_Id      = 1007;
     Tubo_Borra_Id       = 1008;
     Tubo_Modifica_Id    = 1010;
     Tubo_Intercambia_Id = 1011;
```

Hay que notar que no se define un identificador para Acepta y Cancela, dado que éstos (1 y 2) ya son reconocidos por ObjectWindows.

3. Incluir el archivo de recursos dentro de programa.

Al igual que para la implementación del menú principal, se usa la directiva de compilación **$R** de la siguiente forma:

{$R Captura.Res}

Esta directiva sólo debe de incluirse una vez en el programa principal. Todos los recursos a utilizar en la aplicación deben estar definidos en el mismo archivo de recursos.

4. Definir el buffer de datos, el diálogo y los controles.

Al ejecutarse el diálogo, los datos pueden ser almacenados en un buffer de la aplicación. El buffer se construye como un registro, de esta forma, después que el diálogo ha terminado, se conservará la información tecleada por el usuario. Aquí hay dos puntos importantes que considerar. Primero, El buffer que maneja ObjectWindows sólo soporta arreglos de caracteres (not null string). Y segundo, no es la única forma de conservar los datos. Para CRAP, se define el siguiente buffer:

Ejemplo 12

```
TTuboData = record
  Nombre    : Array[0..5] of Char;
  Longitud,
  Diametro,
  FDarcy,
  Celeridad : Array[0..15] Of Char;
  X1,Y1,X2,Y2       : Array[0..3] Of Char;
  Agrega    : Bool;
  Borra     : Bool;
  Modifica  : Bool;
  Intercambia : Bool;
end;
```

Observe que todos los textos editables se definen como arreglos de caracteres. La longitud de los arreglos debe ser la longitud del campo +1. Por ejemplo si el nombre del tubo será de 5 caracteres, el arreglo Nombre se define de 6 posiciones (iniciando siempre en cero).

Una vez que tiene definido el buffer, se debe definir un objeto Pdialog (instancia de Tdialog) y uno por cada tipo de control (Ej. PEditor, PRadioButton).

Ejemplo 13

```
PCrapWindow = ^TCrapWindow;
  TCrapWindow = Object(TWindow)
    Dialogo:Pdialog;
    Editor:Pedit;
    Boton:PRadioButton;
    Tubo_Datos:TTuboData;
    ...
```

5. Inicializar buffer de datos.

Si se desea que el diálogo aparezca con valores iniciales, éstos deberán de fijarse en el buffer antes de ejecutar el diálogo. Hay que recordar que ObjectWindows maneja para el buffer de datos Arreglos de caracteres, por lo que para inicializar el buffer, se usa la instrucción StrpCopy para convertir un cadena a arreglo de caracteres.

Ejemplo 14

```
          {Inicializa los datos del buffer }

        Tubo_Datos.Agrega:=True;
        Tubo_Datos.Borra:=False;
        Tubo_Datos.Modifica:=False;
        Tubo_Datos.Intercambia:=False;
        StrpCopy(Tubo_Datos.Nombre,'Tubo1');
        StrpCopy(Tubo_Datos.Longitud,'0');
        StrpCopy(Tubo_Datos.Diametro,'0');
        StrpCopy(Tubo_Datos.FDarcy,'0.014');
        StrpCopy(Tubo_Datos.Celeridad,'1000');
        StrpCopy(Tubo_Datos.X1,'1');
        StrpCopy(Tubo_Datos.Y1,'2');
        StrpCopy(Tubo_Datos.X2,'3');
        StrpCopy(Tubo_Datos.Y2,'4');
```

6. Crear el diálogo y los controles.

Para crear el diálogo se usa la instrucción *NEW*:

```
Dialogo:=New(Pdialog,Init(@self,'Dlg_Tubo'));
```

Posteriormente, hay que definir los controles del diálogo:

Ejemplo 15

```
New(Editor,InitResource(Dialogo,Tubo_Nombre_Id,6));
New(Editor,InitResource(Dialogo,Tubo_Longitud_Id,16));
New(Editor,InitResource(Dialogo,Tubo_Diametro_Id,16));
New(Editor,InitResource(Dialogo,Tubo_FDarcy_Id,16));
New(Editor,InitResource(Dialogo,Tubo_Celeridad_Id,16));
New(Editor,InitResource(Dialogo,Tubo_X1_Id,4));
New(Editor,InitResource(Dialogo,Tubo_Y1_Id,4));
New(Editor,InitResource(Dialogo,Tubo_X2_Id,4));
New(Editor,InitResource(Dialogo,Tubo_Y2_Id,4));
New(Boton,InitResource(Dialogo,Tubo_Agrega_Id));
New(Boton,InitResource(Dialogo,Tubo_Modifica_Id));
New(Boton,InitResource(Dialogo,Tubo_Borra_Id));
New(Boton,InitResource(Dialogo,Tubo_Intercambia_Id));
```

Aquí hay dos puntos importantes que remarcar. Primero, los controles deben de definirse en el mismo orden en que se encuentra el buffer. Y segundo, la longitud de cada texto editable (Editor) debe ser igual al número de posiciones del arreglo correspondiente.

7. Asociar el buffer de datos y ejecutar el diálogo.

Finalmente, al campo TransferBuffer del objeto diálogo se le asigna la dirección del buffer de datos y se ejecuta el diálogo.

Ejemplo 16

```
    {Se define el buffer de datos }
Dialogo^.TransferBuffer := @Tubo_Datos;
    {Ejecuta Dialogo}
esultado:= Application^.ExecDialog(Dialogo);
```

En este momento, aparece el diálogo y el usuario puede teclear la información solicitada. El diálogo se dará por terminado hasta que el usuario presione el botón de Acepta o Cancela. Una vez terminado el

diálogo, el valor de la variable Resultado indicará de que forma terminó el diálogo el usuario. Los datos tecleados por el usuario quedarán en el registro Tubo_Datos.

Para resumir todo el proceso de implementación de un diálogo se presenta el siguiente programa:

Ejemplo 17 (CRAP_06.PAS)

```
...
{$R Captura.Res}
const
...
     {Identificadores para el diálogo del tubo }

     Tubo_Nombre_Id      = 1001;
     Tubo_Longitud_Id    = 1009;
     Tubo_Diametro_Id    = 1019;
     Tubo_FDarcy_Id      = 1013;
     Tubo_Celeridad_Id   = 1003;
     Tubo_X1_Id          = 1002;
     Tubo_Y1_Id          = 1004;
     Tubo_X2_Id          = 1005;
     Tubo_Y2_Id          = 1006;
     Tubo_Agrega_Id      = 1007;
     Tubo_Borra_Id       = 1008;
     Tubo_Modifica_Id    = 1010;
     Tubo_Intercambia_Id = 1011;

type

  TTuboData = record
    Nombre    : Array[0..5] of Char;
    Longitud,
    Diametro,
    FDarcy,
    Celeridad : Array[0..15] Of Char;
    X1,Y1,X2,Y2       : Array[0..3] Of Char;
    Agrega    : Bool;
    Borra     : Bool;
    Modifica  : Bool;
    Intercambia : Bool;
  end;

PCrapWindow = ^TCrapWindow;
  TCrapWindow = Object(TWindow)
    Bandera   : Boolean;
    Dialogo:Pdialog;
    Editor:Pedit;
    Boton:PRadioButton;
    Tubo_Datos:TTuboData;
```

```
    constructor Init(AParent: PWindowsObject; ATitle: PChar);
    function CanClose: Boolean; virtual;
    Procedure Paint(PaintDC: HDC; var PaintInfo: TPaintStruct); virtual;
    procedure WMLButtonDown(var Msg: TMessage);
              virtual wm_First + wm_LButtonDown;
    procedure WMRButtonDown(var Msg: TMessage);
              virtual wm_First + wm_RButtonDown;
   Procedure Obten_Datos_Tubo(Var Msg:Tmessage);
      Virtual cm_First + cm_Tubo;
  end;

...

    Procedure TCrapWindow.Obten_Datos_Tubo(Var Msg:Tmessage);

    Var
       Resultado : Integer;

       Begin

           {Inicializa los datos del buffer }

          Tubo_Datos.Agrega:=True;
          Tubo_Datos.Borra:=False;
          Tubo_Datos.Modifica:=False;
          Tubo_Datos.Intercambia:=False;

          StrpCopy(Tubo_Datos.Nombre,'Tubo1');
          StrpCopy(Tubo_Datos.Longitud,'0');
          StrpCopy(Tubo_Datos.Diametro,'0');
          StrpCopy(Tubo_Datos.FDarcy,'0.014');
          StrpCopy(Tubo_Datos.Celeridad,'1000');
          StrpCopy(Tubo_Datos.X1,'1');
          StrpCopy(Tubo_Datos.Y1,'2');
          StrpCopy(Tubo_Datos.X2,'3');
          StrpCopy(Tubo_Datos.Y2,'4');

          {Inicializa el diálogo }

          Dialogo:=New(Pdialog,Init(@self,'Dlg_Tubo'));

          {Relaciona el control e identificador con el diálogo.

             Los controles se deben de inicializar en el mismo
             orden del buffer Tubo_Datos. }

          New(Editor,InitResource(Dialogo,Tubo_Nombre_Id,6));
          New(Editor,InitResource(Dialogo,Tubo_Longitud_Id,16));
          New(Editor,InitResource(Dialogo,Tubo_Diametro_Id,16));
          New(Editor,InitResource(Dialogo,Tubo_FDarcy_Id,16));
          New(Editor,InitResource(Dialogo,Tubo_Celeridad_Id,16));
          New(Editor,InitResource(Dialogo,Tubo_X1_Id,4));
          New(Editor,InitResource(Dialogo,Tubo_Y1_Id,4));
          New(Editor,InitResource(Dialogo,Tubo_X2_Id,4));
          New(Editor,InitResource(Dialogo,Tubo_Y2_Id,4));
          New(Boton,InitResource(Dialogo,Tubo_Agrega_Id));
          New(Boton,InitResource(Dialogo,Tubo_Modifica_Id));
          New(Boton,InitResource(Dialogo,Tubo_Borra_Id));
          New(Boton,InitResource(Dialogo,Tubo_Intercambia_Id));
```

```
        {Se define el buffer de datos }

        Dialogo^.TransferBuffer := @Tubo_Datos;

        {Ejecuta Dialogo}

        Resultado:= Application^.ExecDialog(Dialogo);
        {Los datos quedan disponibles en el buffer Tubo_Datos}

        If Resultado = Id_Ok Then
             MessageBox(HWindow, 'Alta de Tubo',
                        'Resultado de la Operación', mb_OK);
        If Resultado = Id_Cancel Then
             MessageBox(HWindow, 'Operación Cancelada',
                        'Resultado de la Operación', mb_OK);
      End;  {Obten Datos del tubo }

...
```

4.6.3. Bitmaps

Un Bitmap representa los valores de un rango de memoria. Actualmente se utiliza el término bitmap para el manejo de gráficas bajo ambiente windows, finalmente la pantalla es un segmento de memoria. Resource Workshop maneja también bitmaps, aunque existen editores de bitmaps más poderosos.

La presentación de un bitmap desde una aplicación Windows es muy sencillo, a continuación se presenta la secuencia de pasos que deben de seguirse para desplegar un bitmap.

1. Crear el gráfico como un recurso Bitmap.
2. Cargar el bitmap en memoria.
3. desplegar el bitmap.

Ahora se explicarán cada uno de éstos puntos:

1. **Crear el gráfico como un recurso Bitmap.**

La creación del bitmap es muy sencilla, Resource Workshop ofrece un editor gráfico fácil de usar. Adicionalmente se pueden transportar imágenes de otro editor gráfico (formato BMP) vía clipboard (cut & paste).

Hay que recordar que se esta presentando un guía para manejo de bitmaps usando Resource Workshop, lo que implica que no es la única forma de manejar imágenes.

2. **Cargar el bitmap en memoria.**

Una vez que el bitmap se encuentra en el archivo de recursos, hay que incluir el archivo de la misma forma que se hizo para el menú principal o en los diálogos. Hay que recordar que sólo se puede incluir un archivo, por lo que todos los recursos a utilizar deberán estar en el mismo archivo.

El bitmap se carga en un DC (de paso) y posteriormente se despliega en el DC que se esta trabajando. Es obvio pensar que el DC de paso debe de tener las mismas características del DC actual. Anteriormente se revisó que para dibujar cualquier gráfico hay que obtener primero el DC, después desplegar el gráfico y finalmente liberar el DC. El tratamiento de bitmaps no es la excepción. Una vez que se obtiene el DC se crea el Device Context de paso con la instrucción de ObjectWindows *CreateCompatibleDC*.

En el siguiente ejemplo se obtiene el DC actual, se crea el DC de paso y se carga el Bitmap.

Ejemplo 18

```
Ventana:=GetDc(Hwindow);{Obtiene el Device Context actual }
Mapa_Memoria:=CreateCompatibleDC(Ventana);    {Crea el DC de paso }
    {Carga en memoria el Bitmap}
SelectObject(Mapa_Memoria,LoadBitmap(Hinstance, 'BitMap_Tanque'));
```

Hay que notar que el nombre del bitmap (BitMap_Tanque) es el mismo que tiene el recurso en el archivo de recursos incluido previamente.

3. desplegar el bitmap.

Una vez que se cargado el bitmap sólo resta desplegarlo. Aquí se presentan 2 comandos para hacerlo. El primero es el *BitBlt*, despliega el bitmap con las dimensiones con que fue creado. Y el segundo, *StretchBlt* que permite escalar el bitmap.

Para una explicación detallada de las instrucciones consultar los manuales de ObjectWindows. A continuación se presenta un ejemplo usando las 2 instrucciones:

Ejemplo 19 (CRAP_07.PAS)

```
...

PCrapWindow = ^TCrapWindow;
  TCrapWindow = Object(TWindow)
    Bandera   : Boolean;
    Ventana,
    Mapa_Memoria: HDC;
    ...

  procedure TCrapWindow.WMLButtonDown(var Msg: TMessage);

    Var
       X,Y : Integer;

      begin
         Bandera:=True;
         Ventana:=GetDc(Hwindow);{Obtiene el Device Context actual }
         Mapa_Memoria:=CreateCompatibleDC(Ventana);    {Crea el DC de paso }
             {Carga en memoria el Bitmap}
```

```
        SelectObject(Mapa_Memoria,LoadBitmap(Hinstance, 'BitMap_Tanque'));
        X:=Msg.Lparamlo;
        Y:=Msg.Lparamhi;
            {Despliega el Bitmap }
        BitBlt(Ventana,X,Y,20,15,Mapa_Memoria,0,0,SrcAnd);
        DeleteObject(Mapa_Memoria);  {Borra el DC de paso }
        ReleaseDC(Hwindow,Ventana);  {Libera el DC actual }
    end;

procedure TCrapWindow.WMRButtonDown(var Msg: TMessage);

Var
    X,Y : Integer;

    begin
        Bandera:=True;
        Ventana:=GetDc(Hwindow); {Obtiene el Device Context actual }
        Mapa_Memoria:=CreateCompatibleDC(Ventana);   {Crea el DC de paso }
            {Carga en memoria el Bitmap}
        SelectObject(Mapa_Memoria,LoadBitmap(Hinstance, 'BitMap_Tanque'));
        X:=Msg.Lparamlo;
        Y:=Msg.Lparamhi;
            {Despliega el Bitmap }
        StretchBlt(Ventana,X,Y,40,40,Mapa_Memoria,0,0,19,15,SrcAnd);
        DeleteObject(Mapa_Memoria);  {Borra el DC de paso }
        ReleaseDC(Hwindow,Ventana);  {Libera el DC actual }
    end;
```

En este ejemplo, se despliega un bitmap en la posición del mouse cuando se presiona un botón del mouse. El tamaño del bitmap depende del botón que se presione. Si el botón izquierdo es presionado, el bitmap se despliega en tamaño normal y si es el botón derecho se ajusta el bitmap de 20*16 pixeles a 41*41 pixeles.

4.6.4. Iconos

Un icono es un bitmap normalmente de 32*32 pixeles y representa una acción determinada. La imagen del icono debe sugerir la función de éste. Resource Workshop permite editar iconos de 32*32, 32*16 y 64*64 pixeles en 2, 8, 16 o 256 colores. ObjectWindows trata los iconos como una especie de bitmap, es decir, no ofrece una interfase para activar un proceso después que el usuario haga un click sobre el icono. Si se desea manejar iconos y que éstos 'respondan' se deben de programar las instrucciones necesarias para el efecto.

Para desplegar un icono se usa el comando *DrawIcon*. El icono se puede cargar con el mismo comando o mediante una variable tipo *HIcon*. Es importante recordar que antes de usar DrawIcon es necesario obtener el Device Context y liberarlo después que se despliega el icono.

Ejemplo 20

```
...
DrawIcon(Ventana,X,Y, LoadIcon(Hinstance,'Boton1'));
...
(Mediante una variable)
Icono:=LoadIcon(Hinstance,'Boton1'); (Var Icono:HIcon)
DrawIcon(Ventana,X,Y, Icono);
...
```

En la creación de Crap se usaron iconos para simular botones redondos. Al presionar un botón se 'prende' un led, esto se logra usando dos iconos. El primer icono muestra el botón en forma normal y el segundo botón en forma presionada:

Al saber que posición tiene el mouse al momento que se presiona el botón izquierdo, se obtiene que icono se debe de desplegar mientras el botón este presionado. Una vez que el botón se ha liberado y con la posición del mouse se sabrá que icono presionó el usuario y por lo tanto se podrá invocar el procedimiento o función correspondiente.

4.6.5. Cursores

Se le llama cursor a la figura que representa la posición del mouse dentro de la pantalla. El cursor (en un ambiente gráfico) es un bitmap de 32*32 pixeles, pero la posición del mouse es de sólo un pixel. Cada pixel puede tener tres atributos:

- ☑ Color Firme (Blanco ó Negro).
- ☑ Video Inverso.
- ☑ Transparente.

De esta forma el cursor no aparentará ser 'cuadrado' de 32*32 pixeles.

Resource Workshop permite editar un cursor. Al igual que los otros recursos se le asigna un nombre para poder llamarlo posteriormente. Adicionalmente a los cursores que pueda definir el usuario, ObjectWindows ofrece algunos ya definidos (flecha, Reloj de Arena, etc.). No hay que olvidar definir el pixel relativo al bitmap que indicará la posición del mouse (Hot Spot).

El uso de cursores es muy sencillo. Al igual que otros recursos es necesario incluir el archivo de recursos al inicio del programa. Para desplegar el nuevo cursor se utiliza el comando *SetCursor*, el cursor se puede cargar mediante una variable *HCursor* o directamente con SetCursor.

Ejemplo 21

```
...
{ Directamente }
SetCursor(LoadCursor(0,idc_Cross)); {Cursor definido por ObjectWIndows
...
SetCursor(LoadCursor(Hinstance, 'Cursor_Tanque')); {Definido por el usuario}
...
{Mediante una variable}
Cursor_Act:=LoadCursor(Hinstance, 'Cursor_Tanque');  {Var Icono:HIcon}
SetCursor(Cursor_Act);
...
```

Es importante mencionar que la instrucción SetCursor despliega la nueva forma del cursor siempre y cuando el cursor no cambie de posición. Si el cursor cambia de posición se desplegará nuevamente el cursor por defecto (ej. la flecha). Lo más recomendable es usar la instrucción SetCursor en el procedimiento de movimiento de mouse (WMMouseMove) siempre y cuando, durante la aplicación la forma del mouse cambie en base a algunas condiciones. De lo contrario se puede hacer uso del método *GetWindowClass* para definir el cursor por defecto para la aplicación.

La siguiente es una lista de instrucciones que permiten conocer las posición de la ventana dentro de la pantalla y fijar las coordenadas en las cuales el cursor se puede desplegar:

- ☑ GetClientRect.- Captura las coordenadas relativas de la ventana (0,0)-(MaxX,MaxY)
- ☑ GetWindowRect.- Captura las coordenadas de la ventana en la pantalla.
- ☑ Clipcursor.- Fija las coordenadas en las que se puede desplazar el cursor.

Se recomienda consultar estas instrucciones en el manual de ObjectWindows o en la ayuda en linea de Borland Pascal.

Finalmente, se presenta un ejemplo usando un cursor definido por el usuario (con la forma de tanque). El cursor **'Cursor_Tanque'** se despliega mientras el botón derecho del mouse está presionado.

Ejemplo 22 (CRAP_08.PAS)

```
...
PCrapWindow = ^TCrapWindow;
  TCrapWindow = Object(TWindow)
    Bandera,
    Boton_Presionado    : Boolean;
    Cursor_Act:HCursor;
    ...
    procedure WMLButtonUP(var Msg: TMessage);
            virtual wm_First + wm_LButtonUp;
    Procedure WMMouseMove(var Msg: TMessage);
            Virtual wm_First + wm_MouseMove;
```

```
    ...
  end;

  constructor TCrapWindow.Init(AParent: PWindowsObject; ATitle: PChar);
     begin
        inherited Init(AParent, ATitle);
        ...
        Boton_Presionado:=False;
          {Se carga el cursor}
        Cursor_Act:=LoadCursor(Hinstance, 'Cursor_Tanque');
     end;
...
   procedure TCrapWindow.WMLButtonDown(var Msg: TMessage);

      begin
         Bandera:=True;
         Boton_Presionado:=True;
         SetCursor(Cursor_Act); {Se despliega el nuevo cursor }
      end;


    procedure TCrapWindow.WMLButtonUp(var Msg: TMessage);

    Var
       X,Y : Integer;

      begin
         Boton_Presionado:=False;
         Bandera:=True;
         Ventana:=GetDc(Hwindow);{Obtiene el Device Context actual }
         Mapa_Memoria:=CreateCompatibleDC(Ventana);   {Crea el DC de paso }
             {Carga en memoria el Bitmap}
         SelectObject(Mapa_Memoria,LoadBitmap(Hinstance, 'BitMap_Tanque'));
         X:=Msg.Lparamlo;
         Y:=Msg.Lparamhi;
             {Despliega el Bitmap }
         BitBlt(Ventana,X,Y,20,15,Mapa_Memoria,0,0,SrcAnd);
         DeleteObject(Mapa_Memoria);  {Borra el DC de paso }
         ReleaseDC(HWindow,Ventana);  {Libera el DC actual }
      end;

    procedure TCrapWindow.WMMouseMove(var Msg: TMessage);

      begin
          If Boton_Presionado Then
             SetCursor(Cursor_Act);
      end;
...
```

En la construcción de CRAP se usó un mecanismo similar al anterior, en una variable se guarda el cursor del elemento activo. Cuando la posición del mouse coincide con un punto de la malla se despliega el cursor actual. Si se presiona el botón izquierdo del mouse se fijan las coordenadas del cursor para que no se pueda mover más alla de la ventana. Cuando el usuario libera el botón las coordenadas se restablecen.

4.7. GetWindowClass

Durante la inicialización de un objeto ventana se pueden fijar varios atributos, por ejemplo el estilo de la ventana, tamaño, menú principal, etc. Estos atributos se especifican a través del registro Attr en el método de INIT y son llamados atributos de creación.

Existen otros atributos propios de la ventana como:

- ☑ Color de fondo (background color).
- ☑ Icono representativo.
- ☑ Cursor.

Estos atributos se conocen cómo atributos de registro, porque se dan de alta al momento que la *clase de ventana* (window class) es registrada en Windows. La clase de ventana es una lista de atributos asociados a cada objeto ventana.

Los atributos de registro son almacenados en la variable WndClass. WndClass es un registro del tipo TWndClass que es definido y administrado por Windows.

El proceso de asociar una clase de ventana con su respectivo objeto ventana se llama *registrar* una clase de ventana. ObjectWindows realiza éste proceso en forma automática. Si se de desea modificar algún atributo de registro hay que redefinir el método GetWindowClass.

Estos son los campos de la variable WndClass y una breve explicación de cada uno de ellos:

- ☑ **Style.-** Estilo de la ventana. No confundir con el campo de Attr que establece características visuales, en cambio WndClass fija atributos funcionales (Ej. Aceptar doble click sobre la ventana).
- ☑ **hIcon.-** Establece el Icono que se usará para representar la ventana cuando ésta sea minimizada.
- ☑ **hCursor.-** Establece el cursor que se usara cuando el mouse este 'dentro' de la ventana.
- ☑ **hbrBackground.-** Establece el color del fondo de la ventana.
- ☑ **lpszMenuName.-** Establece el menú principal. En caso de querer eliminar la definición del menú en INIT.

Los atributos modificados en CRAP son: hbrBackground y Hicon. hCursor no se modificó porque el cursor que se utiliza cambia durante la aplicación (depende del elemento activo). Se presenta un ejemplo de la codificación de GetWindowClass:

Ejemplo 23 (CRAP_09.PAS)

```
...
PCrapWindow = ^TCrapWindow;
  TCrapWindow = Object(TWindow)
    ...
    procedure GetWindowClass(var WndClass: TWndClass); virtual;
  end;

...

    Procedure TCrapWindow.GetWindowClass(var WndClass: TWndClass);

     begin
        TWindow.GetWindowClass(WndClass);
        ( Se define el mismo color que se especifico en
          Windows (por el Panel de Control) )
        WndClass.hbrBackground := color_AppWorkspace + 1;
        {Los recursos estan definidos en Captura.Res }
        WndClass.hIcon := LoadIcon(HInstance, 'Logo' );
        WndClass.hCursor:= LoadCursor(Hinstance, 'Manita');
     end;
...
```

4.8. Manejo de Mensajes

Una aplicación Windows se dice que es 'manejada' por eventos, es decir, el flujo de instrucciones no es determinado por una secuencia. El control es dictado por una serie de eventos externos a la aplicación. Por lo tanto la aplicación debe 'responder' a los mensajes que envía Windows cada vez que se produce un evento.

Un mensaje se puede generar por:

- ☑ Intervenciones del usuario (manejo de mouse, teclado).
- ☑ Un programa manda explícitamente un mensaje.
- ☑ Una aplicación puede mandar un mensaje a otra aplicación para intercambiar información.
- ☑ Windows genera por sí mismo mensajes (ej. Cuando termina Windows).

Un mensaje es un registro (tipo *TMsg*). que contiene información relacionada con el evento. La estructura de los mensajes es la siguiente:

Ejemplo 24

```
Type
    TMsg = Record
       hwnd: Hwnd;
       Message: Word;
       wParam: Word;
       lParam: Longint;
       time: Longint;
       pt:Tpoint;
    End;
```

El mensaje se envía una vez que el evento a terminado y contiene la información del evento que lo generó, dónde sucedió y a que ventana va dirigido.

El campo de Message contiene un número de 16 bits que identifica al mensaje. Cada mensaje tiene un valor único, éste valor se representa también por un mnemónico (todos los mensajes enviados por Windows comienzan con wm_). Por ejemplo cuando se presiona el botón izquierdo del mouse el mensaje que se genera es el $0201 y su mnemónico es el wm_LButtondown.

Al hacer un programa, el programador no debe preocuparse por el manejo de los mensajes (cómo se transmiten los mensajes). La administración de los mensajes la proporciona Windows en combinación con ObjectWindows. Al producir un mensaje, el programador sólo debe poner atención a los parámetros que le debe de pasar y cómo debe de responder a los mensajes.

4.8.1. Declaración de un método para que responda a un mensaje

Básicamente para que un método (virtual) responda a un mensaje es que el método debe de tener un número entero asociado. El número puede ser una constante que identifique el mensaje.

Ejemplo 25

```
type

  PCrapWindow = ^TCrapWindow;
  TCrapWindow = Object(TWindow)
     procedure WMLButtonDown(var Msg: TMessage);
               virtual wm_First + wm_LButtonDown;
```

Hay que observar que la constante que se asocia esta compuesta por dos números (wm_First y wm_LButtonDown). El primero, wm_First sirve como un corrimiento (Offset) dado que ObjectWindow maneja varios rangos de mensajes y todos estan indexados a cero. El segundo valor corresponde al mensaje. Más adelante se darán que rangos de mensajes tiene ObjectWindows.

Para que un método se ejecute como respuesta a un evento, ObjectWindows provee un 'enrutador' de mensajes. ObjectWindows recibe el mensaje que envía Windows. Busca si existe un método asociado con dicho mensaje, si lo encuentra re-empaca el mensaje en una variable tipo TMessage y llama al método. En caso de que no encuentre un método asociado con el mensaje llama al procedimiento por defecto (siempre y cuando exista para ese evento). El proceso anterior se hace 'detrás' (background) de la aplicación por lo que es completamente transparente para el programador.

La Estructura de TMessage es la siguiente:

Ejemplo 26

```
TMessage = record
   Receiver: HWnd;     <-- Ventana que recibe el msg, equivale a Hwnd de TMsg
   Message: Word;           <-- Identificador del msg, igual que en TMsg.
   case Integer of
     0: (WParam: Word;
         LParam: Longint;
         Result: Longint);
     1: (WParamLo: Byte;
         WParamHi: Byte;
         LParamLo: Word;
         LParamHi: Word;
         ResultLo: Word;
         ResultHi: Word);
   end;
```

La información contenida en los campos WParam y LParam dependerá del evento que haya generado dicho mensaje. el campo Result se utiliza para controlar el código de retorno. En ocasiones el mensaje espera un código una vez que se ha ejecutado el método asociado.

4.8.2. Definición de mensajes de usuario

Windows ofrece la posibilidad de definir 31,744 mensajes de usuario. Todos los mensajes de usuario tendrán un corrimiento igual a wm_User. Lo más recomendable al definir un mensaje de usuario es definir una constante (basada en wm_User) que identifique al mensaje, en CRAP se utilizaron los siguientes mensajes de usuario:

Ejemplo 27

```
Const
     wm_ActivaMedidas  = Wm_user+201; {Mensajes de usuario}
     wm_ZoomProximo    = wm_User+202;
     wm_ZoomAnterior   = wm_User+203;
     wm_ActivaNodo     = wm_User+204;
     wm_ActivaTubo     = wm_User+205;
     wm_ActivaTanque   = wm_User+206;
     wm_ActivaPozo     = wm_User+207;
     wm_ActivaVerifica = wm_User+208;
     wm_ActivaSalvado  = wm_User+209;
```

Para responder a éstos mensajes es igual que cualquier otro mensaje:

```
Procedure WMActivaMedidas(Var Msg: TMessage);
    Virtual wm_first + wm_ActivaMedidas;
```

Una vez que se ha definido el mensaje y el método que responderá, sólo resta generar el mensaje. Windows ofrece dos funciones para la generación de mensajes: SendMessage y PostMessage. La diferencia entre éstas dos funciones es que SendMessage envía el mensaje y espera a que éste sea procesado antes de regresar el control. PostMessage sólo agrega el mensaje a la cola de mensajes y regresa el control.

Lo más recomendable es usar PostMessage, dado que el programador no se preocupa por saber el momento en que el mensaje se responde. Adicionalmente se evitan los bucles que SendMessage pudiera generar (al tener que esperar la respuesta de un mensaje). La forma en que se genera un mensaje es la siguiente:

```
PostMessage(HWND_BROADCAST,wm_ActivaMedidas,0,0);
```

Los parámetros de PostMessage son los siguientes:

- ☑ manejador de la ventana que recibe (HWND_BROADCAST)
- ☑ Número de mensaje (wm_ActivaMedidas)
- ☑ Parámetro Wparam (0)
- ☑ Parámetro Lparam (0)

4.8.3. Rango de mensajes

El rango de mensajes que maneja ObjectWindows es el siguiente:

Tipo de Mensaje	Valores	Corrimiento (Offset)	Observaciones
Reservados para windows	$0000-$03FF	wm_First	Generados por windows (mouse, ventanas, etc.).
Usuario	$0400-$7FFF	wm_User	Para ser generados por el usuario
Notificación de controles	$0800-$8EFF	id_First	
Reservado para ObjectWindows	$8F00-$8FFF	id_Internal	
Notificación de parents	$9000-$9EFF	nf_First	
Reservado para ObjectWindows	$9F00-$9FFF	nf_Internal	
Comandos	$A000-$FEFF	cm_First	Generados al seleccionar una opción del menú principal.
Reservado para ObjectWindows	$FF00-$FFFF	cm_Internal	

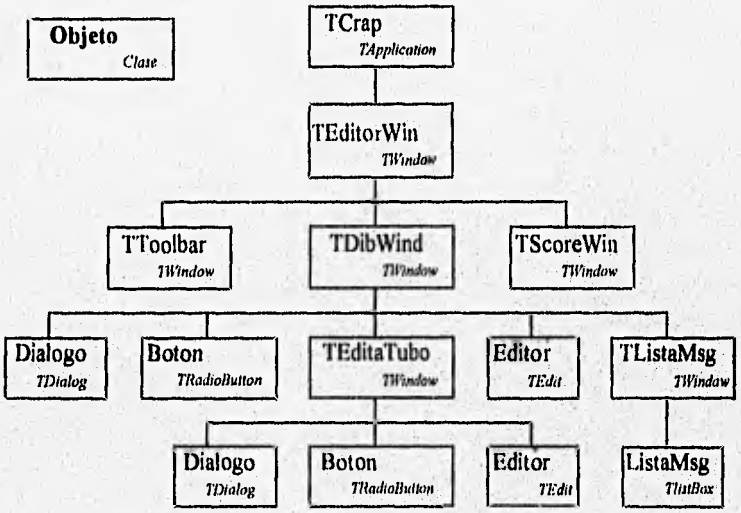
CRAP

5. CRAP

En los capítulos anteriores se han presentado las bases del Diseño y de la Programación Orientada a Objetos, así como una guía para el uso de Borland Pascal en la creación de aplicaciones Windows. En éste capitulo se revisarán como se han integrado éstos conocimientos dando como resultado la creación del sistema de Captura de Redes de Agua Potable (CRAP).

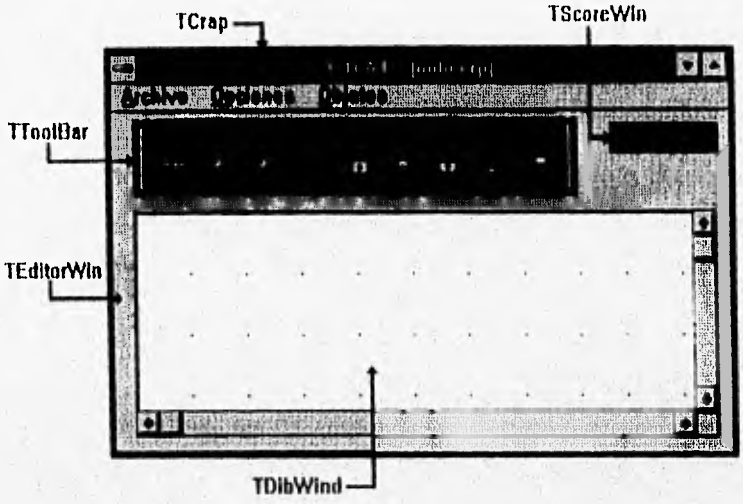
Este capítulo no pretende dar una explicación sobre el código de programación de CRAP, sino dar un panorama general de cómo fue diseñado, que objetos utiliza y cómo se relacionan éstos. CRAP cuenta con un manual de usuario donde se detalla su uso y facilidades.

A continuación se presenta la estructura jerárquica de las clases utilizadas en CRAP y qué objeto se definieron de cada una:



Al momento de correr se ve de la siguiente forma:

Ejemplo 29



Al observar la estructura de clases y la imagen anterior es fácil deducir que se requiere comunicación entre los objetos *TDibWind*, *TScoreWin* y *TToolBar*. Esta comunicación se establece de dos formas: Utilizando mensajes y un registro compartido.

El uso de mensajes (ver capítulo anterior) se hace cuando se desea activar un procedimiento en base a un evento determinado, por ejemplo al momento en que el cursor se posiciona en un punto relativo a la malla (*TDibWind*), es necesario refrescar la ventana (*TScoreWin*) que despliega la coordenada de la malla ó para activar el elemento seleccionado en la barra de herramientas (*TToolBar*).

Se utiliza un registro para conservar información generada en un objeto y que puede ser utilizada por otro. Compartir un registro es muy sencillo, se define la estructura del registro en el objeto padre y un apuntador con la misma estructura en cada uno de los objetos hijos. El

registro se inicializa en *TEditorWin* (por ser el padre de los otros) y al momento de inicializar cada uno de los objetos hijos se pasa como variable el apuntador del registro (dirección de memoria donde esta la información). De ésta forma, cada uno de los objetos podrá actualizar y leer la información de dicho registro. Para CRAP se hicieron las siguientes definiciones:

Ejemplo 30

```
PEstado = ^TEstado;      {Registro comun a todas las ventana }
TEstado = record
  Archivo_Actual:String;     { Nombre del archivo}
  Titulo_Ventana:Array[0..60] Of Char;
  Centro_Barra_X,
  Centro_Barra_Y,            { Posicion Central de la Barra de Scroll}
  EstadoX, EstadoY,          { Posicion del  cursor para Score }
  ZoomX, ZoomY,              { Offset del zoom }
  CuadriculadoOriginal,      { Numero de puntos en la malla Original}
  Elemento,                  { Elemento seleccionado -> Tubo, Nodo, etc. }
  CoordMaxX,
  CoordMaxY,                 { Coordenadas máximas donde hay un elemento}
  Modo : Integer;            { En que modo se encuentra el editor }
  Cursor_Act : Hicon;        { Cursor que se presenta cuando el mouse esta
                               en un punto_valido}
  CentroX,CentroY:Integer;   { Centro del dibujo visible}
end;
```

La forma en que se utilizan los mensaje y el registro *TEstado* se lleva a cabo de la siguiente manera (Ejemplificando): El usuario selecciona el elemento Pozo en la barra de herramientas (*TToolBar*). *ToolBar* en ese momento envía el mensaje *wm_ActivaPozo*. *TEditorWin* tiene el procedimiento *WMActivaPozo* asociado al mensaje *wm_ActivaPozo*. *WMActivaPozo* lo único que hace es llamar al procedimiento *TEditor.Activa_Pozo*. *Activa_Pozo* actualiza los campos *Elemento* y *Cursor_Act* de *TEstado*. Es importante mencionar que el Menú principal tiene asociado el procedimiento *Activa_Pozo* a la opción Objetos/Pozo, de tal forma que no importa si se selecciona el Pozo por el Menú principal ó por la barra de herramientas siempre se ejecutará el mismo procedimiento.

Se utiliza un mensaje por cada icono de la barra de herramientas y uno para activar el procedimiento que actualice la posición del cursor.

La información relativa a cada objeto se almacena en memoria mediante listas ligadas. Existen cuatro listas ligadas, una para cada elemento. Las listas ligadas se van generando conforme el usuario va agregando elementos o al momento de leer la malla de disco.

Al salvar la información de la malla se hace de la siguiente forma: Se escribe en un archivo secuencial (texto) los datos de cada elemento dado de alta, así como los datos para poder regenerar el dibujo.

Para Finalizar éste capítulo se dará una breve explicación de los objetos utilizados en CRAP:

- ☑ *TCrap.*- Aplicación CRAP corriendo bajo ambiente Windows.
- ☑ *TEditorWin.*- Ventana principal donde se despliegan tres ventanas (hijas): La malla (*TDibWind*), la barra de herramientas (*TToolBar*) y la posición del cursor relativa a la malla (*TScoreWin*).
- ☑ *TDibWind.*- Malla de trabajo, se dibujan los puntos donde se puede definir un objeto.
- ☑ *TToolBar.*- Barra de herramientas, se pueden activar las opciones más importantes del menú principal.
- ☑ *TScoreWin.*- Despliega la posición del cursor relativa a la malla (punto válidos de trabajo).
- ☑ *TListaMsg.*- Ventana para desplegar una lista (ListaMsg) con el resultado de la verificación de la malla.
- ☑ *TEditaTubo.*- Ventana que muestra las características (Distancia/Elevación) del tubo.
- ☑ *ListaMsg.*- Despliega en forma de lista el resultado de la verificación de la malla.
- ☑ *Dialogo.*- Se utiliza para leer/mostrar la información relativa a cada objeto, su estructura cambia dependiendo del elemento activo.
- ☑ *Editor* y *Boton.*- En combinación con *Dialogo* para interactuar con el usuario en la lectura de datos.

Comentarios finales

6. Comentarios Finales

En los capítulos anteriores se han presentado las bases de la Programación Orientada a Objetos y una guía para la generación de aplicaciones bajo ambiente Windows utilizando Borland Pascal v7.0. Los conceptos anteriores se han ejemplificado mediante la creación de CRAP. CRAP es un sistema gráfico de captura de datos, con base en éstos datos se puede realizar la simulación de la red. El desarrollo del simulador basado en las técnicas de la Programación Orientada a Objetos es materia de un nuevo proyecto.

La Programación Orientada a Objetos ofrece una nueva manera de analizar, diseñar e implementar sistemas. Se ha hablado de las ventajas que tiene su uso, sin embargo también se tienen desventajas. Aunque se dice que ésta tecnología tiene 25 años de existir, aun se encuentra en etapa de desarrollo, motivo por el cual se puede esperar que las desventajas tiendan a reducirse.

Finalmente se concluye satisfactoriamente la realización de este trabajo. La orientación principal del material escrito es que sirva como ayuda en el aprendizaje de la programación orientada a objetos, así como en la generación de aplicaciones bajo ambiente Windows utilizando Borland Pascal v7.0.

Referencias y bibliografía

7. Referencias

[1] Booch, Grady
Object-Oriented Analysis and Design with aplications
Benjamin/Cummings,

[2] Cardelli, L. and Wegner, P.
On Understanding Types, Data abstraction and Polymorphism.

[3] Shaler, s. and Melllor s.
Object-Oriented Systems Analysis: Modeling the world in data.[3]
Ross R.

[4] Entity Modeling: Techniques and Application.

8. Bibliografía

- Voss, Greg
  Programación Orientada a Objetos, Una introducción
  McGraw-Hill
- Weiskamp, K., Heiny L. and Flamig B.
  Object-Oriented Programming
  Wyley
- Petzold, Charles
  Programmig Window 3.1
  Microsoft
- Manuales de Borland Pascal V7.0:
  ObjectWindows, Programing Guide
  Tools and Utilities Guide
  Resource Workshop, User´s Guide
  User's Guide
  Language Guide
  Programmer´s Reference
- CRAP, Manual de Usuario
  Martín Bravo A.

7. Referencias

[1] Booch, Grady
Object-Oriented Analysis and Design with aplications
Benjamin/Cummings,

[2] Cardelli, L. and Wegner, P.
On Understanding Types, Data abstraction and Polymorphism.

[3] Shaler, s. and Melllor s.
Object-Oriented Systems Analysis: Modeling the world in data.[3]
Ross R.

[4] Entity Modeling: Techniques and Application.

8. Bibliografía

- Voss, Greg
  Programación Orientada a Objetos, Una introducción
  McGraw-Hill
- Weiskamp, K., Heiny L. and Flamig B.
  Object-Oriented Programming
  Wyley
- Petzold, Charles
  Programmig Window 3.1
  Microsoft
- Manuales de Borland Pascal V7.0:
  ObjectWindows, Programing Guide
  Tools and Utilities Guide
  Resource Workshop, User's Guide
  User's Guide
  Language Guide
  Programmer's Reference
- CRAP, Manual de Usuario
  Martín Bravo A.