UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE CIENCIAS

INTEGRA++ PARA WINDOWS 95: UN ANALIZADOR DE SISTEMAS DINAMICOS

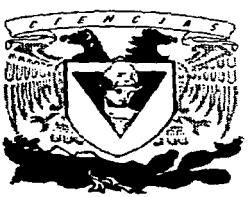
T E S I S

QUE PARA OBTENER EL TITULO DE

M A T E M A T I C O

P R E S E N T A :

FERNANDO CORNELIO TAPIA SALINAS



MEXICO, D.F. 1997

VNIVERSIDAD NACIONAL
AVPNOMA DE
MEXICO

M. en C. Virginia Abrin Batule
Jefe de la División de Estudios Profesionales de la
Facultad de Ciencias
P r e s e n t e

Comunicamos a usted que hemos revisado el trabajo de Tesis: "Integra ++ para Windows 95: Un analizador de sistemas dinámicos"

realizado por : Fernando Cornelio Tapia Salinas

con número de cuenta 9052489-9 **, pasante de la carrera de** Matemáticas

Dicho trabajo cuenta con nuestro voto aprobatorio.

Atentamente

Director de Tesis
Propietario Dr. Humberto Carrillo Calvet

Propietario M. en C. Ma. Guadalupe Elena Ibargüengoitia González

Propietario M. en C. Noe Sierra Romero

Suplente Dr. Guillermo Sienra Loera

Suplente Dr. José Angel L. Ortega Herrera

Consejo Departamental de Matemáticas

Mat. César Cuevara Bravo.

Universidad Nacional Autónoma de México
Facultad de Ciencias

INTEGRA++ para Windows 95: Un Analizador de Sistemas Dinámicos

Autor: Fernando C. Tapia Salinas

Tesis dirigida por: Dr. Humberto Carrillo Calvet

Noviembre de 1997

Dedicatorias

A la memoria de Lety. Por esa valentía y honestidad a toda prueba que siempre mostraste.

A mi Madre. Por ser la mejor de las madres y por que en gran medida este trabajo es tuyo también.

A mis hermanos Saúl, Yola, Angel, Rosalío y Fabi. Por su apoyo sin condiciones, especialmente por ayudarme en la adversidad.

A mis sobrinitos Pamela, Junior, Angelito, Keny y Andy. Por aprender de ustedes a reir de la misma forma que ustedes lo hacen.

Agradecimientos

Son muchas las personas con las que he contado con ayuda muy valiosa, antes y después de ingresar a la carrera, a todas ellas deseo expresarles mis agradecimientos.

Indudablemente que el contar con el apoyo incondicional de mi familia ha sido lo mejor que me ha sucedido para lograr mis objetivos. Sin ella posiblemente no estaría aquí. Quiero agradecer a mi madre y a mis hermanos por ese apoyo:

Primeramente, deseo agradecer a mi madre, quien a pesar de haberse quedado sola cuando todos sus hijos éramos pequeños, supo enfrentarse a la vida y lograr sacar fuerzas para impulsarnos hacia adelante y lograr que uno por uno lográramos los objetivos que nos habíamos trazado. Sin duda Mamá, eres la mejor de las madres, estoy orgulloso de ti.

A Saúl, con quien disfruté aprendiendo Matemáticas cuando estaba en el Bachillerato, de quien aprendí el valor del estudio y con quien aún disfruto platicando de cuantas cosas se le ocurren. Te agradezco que me hayas motivado a entrar a estudiar Matemáticas, es el mejor consejo que pudiste darme.

A Yola, por que aún me regaña, y esos regaños redituan en reflexiones para mí. Gracias por ser tan buena hermana.

A Angel, la persona mas trabajadora que haya conocido, porque he aprendido de ti el valor del trabajo. Por que nunca te faltan motivos para hacer las cosas bien y nunca tienes justificaciones de porque no hacer las cosas y además haces del trabajo tu pasatiempo. Admiro tu coraje para enfrentarte a la vida. Te agradezco que nunca me hayas dejado caer, gracias por estar conmigo.

A Rosalío, por aquellos tiempos que convivimos juntos en nuestra niñez y que aún añoro. Has sido un gran amigo. Y porque estando lejos nunca has perdido la brújula. Admiro tu determinación para hacer las cosas.

A Lety, con quien convivi toda una vida. Sobran palabras para ti, pero he de decirte que permanecerás en mi por siempre. Fuiste una excelente hermana.

A Fabi, porque me has apoyado cuando las cosas no han ido bien conmigo y has sabido entenderme.

A mis sobrinitos, Pamela, Junior, Angelito, Keny y Andy, porque con ellos logro volver a esa niñez perdida y por ese mar de risas que me hacen sacar cada vez que estoy con ellos.

Un trabajo de esta naturaleza hubiera sido muy difícil llevarlo a cabo si no hubiera contado con el apoyo de excelentes profesores.

Al Dr. Humberto Carrillo, por que me haya aceptado como su tesista y por ese apoyo moral y económico siempre que fue necesario. Le estoy infinitamente agradecido.

A Luis Nava por haberme invitado a participar en su proyecto. Quiero darte las gracias también por haberme ayudado a visualizar el sentido del proyecto y que me hayas motivado a trabajar en INTEGRA, además por esas largas discusiones que tuvimos acerca de mi trabajo de tesis.

A los profesores Noé Sierra Romero, Guadalupe Ibargüengoitia, Guillermo Sienra y José Angel Ortega, por haber aceptado revisar mi trabajo de tesis.

A Sais, por ese tiempo robado para ayudarme a escribir o formatear algún texto de la tesis, a pesar de que estabas llena de trabajo en la escuela (y yo te daba más!!!). Por esos fines de semana que no salimos juntos por estar trabajando. Gracias por entenderme y quererme siempre.

A los compañeros del Laboratorio de Dinámica No Lineal, por permitirme usar la computadora cuando era necesario.

A mis amigos, por soportarme y estimarme.

Y a todas las personas que hayan tenido algo que ver con este trabajo.

INDICE GENERAL

INDICE GENERAL

Capítulo 1

INTRODUCCION

Generalmente los sistemas dinámicos se modelan con ecuaciones diferenciales cuyas soluciones no pueden obtenerse analíticamente. Esto se debe al hecho típico de que las interacciones entre los diferentes componentes del sistema obedecen a una dinámica no lineal. Por esta dificultad es importante hacer simulaciones numéricas de los sistemas dinámicos que permitan, mediante la experimentación computacional, estudiar sus diferentes comportamientos. Para este fin, la computadora se ha convertido en el laboratorio ideal.

En base a lo anterior, el objetivo principal del presente trabajo es crear un prototipo de software desarrollado para asistir la enseñanza e investigación de los sistemas dinámicos, modelados con ecuaciones diferenciales o ecuaciones en diferencias, a este sistema de software le hemos llamado *INTEGRA++*. Así, bajo un ambiente sencillo de operar, *INTEGRA++* permite analizar sistemas de ecuaciones de una, dos y tres dimensiones.

Las principales características funcionales de *INTEGRA++* son:

1. Generar y manipular bibliotecas de sistemas de ecuaciones diferenciales ordinarias.
2. Seleccionar el sistema a estudiar
3. Modificar los valores de los parámetros del sistema en estudio.
4. Manipular la escala de los ejes coordenados.
5. Seleccionar los métodos numéricos de integración.
6. Seleccionar condiciones iniciales
7. Dibujar órbitas en el espacio bidimensional y tridimensional.
8. Seleccionar las órbitas a proyectar
9. Grabar la imagen de los retratos de fases
10. Análisis interactivos del sistema dinámico
11. Impresión del análisis del sistema dinámico

Las anteriores funciones son algunas entre otras disponibles por *INTEGRA++*.

Este trabajo está basado en una version anterior la cual fué desarrollada usando el lenguaje C bajo el paradigma de programación estructurado. En esta nueva versión, *INTEGRA++* se desarrolló en C++ y para la orientación a objetos.

El presente trabajo está dividido en cuatro capítulos y seis apéndices:

En el primer capítulo, se presentan los fundamentos de la tecnología orientada a objetos. Iniciamos con una breve historia de la orientación a objetos, yendo desde los inicios de este enfoque y sus bases en los lenguajes de programación que apoyan esta metodología, hasta los conocimientos y su presencia en nuestros días. Se presentan los beneficios que ofrece la construcción de sistemas de software con la orientación a objetos (OO). Enseguida se estudia el *modelo de objeto*, y se enuncian una serie de definiciones de la palabra clave en nuestro estudio: *objeto*. Se sigue la definición de objeto de Booch, en la cual dota a un objeto de estado, comportamiento e identidad, se dan sus definiciones, presentando una discusión acerca de estos conceptos y sus respectivas implementaciones en el lenguaje C++, que fué el que se utilizó en la construcción de *INTEGRA++*. A lo anterior se suman, las relaciones entre objetos que existen: enlaces, visibilidad, sincronización y agregación. Se realiza la misma presentación de conceptos, para el *modelo de clase* y las relaciones que han sido definidas entre clases: asociación, herencia, agregación, uso, instanciación y metaclases, y el concepto de polimorfismo entre clases, para cada una de estas relaciones se ofrece su respectiva implementación en el lenguaje C++. Finalmente, se examinan en detalle los conceptos fundamentales de la modelación objetual: abstracción, encapsulación, modularidad, jerarquización y tipificación.

En el segundo capítulo, se presentan los elementos de los sistemas dinámicos. Se presentan conceptos básicos de los sistemas dinámicos: ecuación diferencial, ecuaciones diferenciales lineales y no lineales y su relación con los sistemas dinámicos. Se muestran los sistemas dinámicos y la forma canónica de los sistemas de ecuaciones diferenciales. Para que se pueda trabajar con un sistema dinámico es necesario garantizar que el sistema tiene solución, esto se garantiza con el teorema de existencia y unicidad de soluciones en los sistemas de ecuaciones diferenciales. Se muestran los sistemas autónomos y algunos resultados de este tipo de sistemas. Se ofrecen los conceptos de espacio de fases y campo vectorial que son fundamentales en el estudio de los sistemas dinámicos. Una vez presentado este marco conceptual se presentan los métodos numéricos para resolver sistemas de ecuaciones diferenciales y sus soluciones discretas mediante los métodos de un paso: Euler, Euler Mejorado y Runge-Kutta (para sistemas de ecuaciones diferenciales de grado n). En adición a este capítulo esta el apéndice A, en donde se presenta el código fuente en C++ de los métodos numéricos anteriores.

El capítulo tres es dedicado a la programación en Windows. Primeramente se ofrecen los elementos básicos de Windows: ventanas y sus componentes, así como las características de una aplicación en Windows y el enfoque orientado a objetos

que contiene. Se presentan la manera natural de crear programas en Windows con C, analizando la cola de mensajes y el ciclo de mensajes que Microsoft recomienda copiar como patrón y la **OWL** (*Object Windows Library*) de Borland, que es una biblioteca de objetos para Windows, en donde se encapsula al **API**(*Application Programming Interface*) de Windows. Se muestran los detalles de programación de las diferentes componentes de Windows: ventanas normales y decoradas, cajas de diálogo y menús. Se analiza la **GDI** (*Graphics Device Interface*), que permite la creación y manipulación de objetos gráficos vectoriales y *raster*.

En el último capítulo se presenta un panorama general de *INTEGRA++*. Se dan los antecedentes de *INTEGRA++*, de la misma manera las motivaciones para construirlo. Se muestran las clases que componen *INTEGRA++*, adicionalmente como apéndice se ofrecen los diagramas de clases y de módulos de *INTEGRA++*, estos diagramas están construídos con la notación de Booch. También se presentan los diagramas de clases y de módulos de *INTERFAZ++*, usando la misma notación. Se ofrece un recorrido general del funcionamiento de *INTEGRA++* e *INTERFAZ++*, para conocer como capturar y manipular sistemas dinámicos. Como extensión a este capítulo y a la información presentada, se ha agregado el "Manual de Usuario" en el apéndice F.

Capítulo 2

FUNDAMENTOS DE LA TECNOLOGIA OBJETUAL

Breve historia de la orientación objetual

En un principio los sistemas de información no eran construidos con un método formal, esto se podía hacer porque los sistemas eran de pequeña o mediana escala, y los métodos informales de construcción eran suficientes para resolver los problemas, pero a medida que la complejidad de los mismos se incrementaba y estos llegaban a ser de gran escala, hubo la necesidad de buscar otros métodos de solución. Es decir, los sistemas orientados a objetos surgieron como un proceso evolutivo en la construcción de sistemas de software, este es evolutivo porque no rompe con las técnicas existentes, sino que se apoya en ellas y las mejora. El término objeto surge casi independientemente en varias áreas de la computación, esto como respuesta al análisis de problemas (de software) complejos; se atacó el problema de complejidad con métodos de descomposición modular y funcional, entre otros, de forma tal que el problema estriba en encontrar métodos de solución que modelaran el sistema en partes menos complejas, vistas desde el dominio del problema y también vistas desde el dominio de la solución. Dichos modelos descomponen el problema en unidades de abstracción de una complejidad manejable. Los primeros sistemas (programas) no tenían una complejidad alta, de hecho los programas estaban compuestos de unas centenas de líneas de código en ensamblador, aunque muy áspero este lenguaje, la complejidad de los programas era bastante manejable para el desarrollador y el usuario, que en muchos de los casos eran el mismo.

En nuestro tiempo este tipo de programas son realizados por programadores novatos, estudiantes, o programadores profesionales quienes son los únicos que utilizan el software que ellos mismo crean, de forma tal que las etapas de mantenimiento, evolución y expansión del mismo casi no existen. Sobra decir que este no es el tipo de sistemas sobre los que nosotros estamos preocupados por construir, sino por el contrario aquellos de una complejidad tal que por su propia naturaleza rebasan al individuo si él deseara resolver el problema en un entorno aislado, es decir, aquellos para los que se necesita todo un grupo de personas para resolver el problema. Las nuevas metodologías de desarrollo están enfocadas en el análisis de la construcción de sistemas a gran escala, por ejemplo automatizar una planta automotriz, controlar procesos de producción, simular un reactor nuclear,

construir controladores de vuelo, etc.

Los avances mas significativos en la tecnología orientada a objetos los dieron los lenguajes de programación. Sin duda el primer gran salto en la programación de alto nivel fue el desarrollo de FORTRAN. FORTRAN (a mediados de los cincuenta) introdujo diversos e importantes conceptos en los lenguajes de programación incluyendo variables, arreglos, estructuras de control (iteración y saltos condicionales) entre los mas importantes. Aún en nuestros tiempos FORTRAN es un lenguaje que tiene mucha popularidad entre la comunidad científica como un lenguaje orientado a la solución de problemas científicos y de ingeniería.

A finales de los años cincuenta, uno de los problemas cuando se desarrollaban programas grandes en FORTRAN fue que los nombres de las variables tenían conflicto en diferentes partes del programa. Los diseñadores de los lenguajes de programación decidieron proveer barreras de protección a los nombres de las variables dentro de los segmentos, y es así como nacieron los bloques Begin...End en Algol 60, de esta manera se solucionó el problema de los nombres de variables porque cada bloque conocía que variables le eran visibles. Las estructuras de bloques son ampliamente usadas en nuestros días.

A principios de los años sesenta, los diseñadores del lenguaje Simula-67 (Dahl y Nygaard, 1966,; Dahl, Myhrhaug, y Nygaard, 1970) tomaron el concepto de bloque de Algol en un primer paso e introdujeron el concepto de objeto. Aunque las raíces de Simula estuvieron en Algol, fue principalmente entendido como un lenguaje de simulación. Así, los objetos de Simula tenían una existencia propia, y podían (en algún sentido) comunicarse unos con otros durante la simulación.

Simula también incorporó la noción de clases, las cuales son usadas para describir la estructura y comportamiento de un conjunto de objetos. La herencia de clases también fue soportada por Simula. La herencia organiza las clases en jerarquías, permitiendo compartir la implementación y la estructura. Simula también distinguía entre dos tipos de igualdad, idéntica y figurada, reflejando la distinción entre la interpretación de objetos basada en la referencia (identidad) contra la de valor (contenido). Por lo tanto Simula puso los fundamentos de los lenguajes orientados a objetos.

A principios de los años setenta, el concepto de abstracción de datos fue perseguido por un buen número de diseñadores de lenguajes con el propósito de manejar programas grandes (Parnas, 1972). Ahí hay dos aspectos fundamentales de los tipos de datos abstractos. Uno es agrupar la estructura del tipo con las operaciones definidas sobre el tipo. El otro aspecto es el de ocultamiento de la información, donde los detalles de la implementación y representación de los objetos están ocultos y no pueden ser accesados a través de los usuarios de los objetos. Lenguajes tales como Alphard (Wolf, London y Shaw, 1976) y CLU (Liskov,1977) introdujeron abstracción de datos. En CLU, por ejemplo los tipos de datos abstractos fueron implementados a través de *clusters*, un nombre muy apropiado, porque se esta agrupando información. Con esos lenguajes fueron desarrollados los funda-

mentos y la teoría matemática de los tipos de datos abstractos [5]. Esto ayudó a establecer el concepto de tipos de datos abstractos, impulsando el desarrollo de la teoría matemática de la orientación a objetos (Goghen, Thatcher, Wegner, y Writhgt, 1975; Gutag 1977; Burshar y Goguen, 1977). La teoría fue entonces desarrollada mas allá de especificaciones para aplicaciones (Ehrig, Kreosky, y Padawiz, 1978), y para unos tipos de datos abstractos de un nivel más alto (Parsaye, 1982).

Uno de los lenguajes mas importantes que soporta tipos de datos abstractos es Ada (Booch, 1986). Por mucho tiempo se ha debatido si Ada es o no un lenguaje orientado a objetos. Ada soporta diversos conceptos de la orientación a objetos tales como tipos de datos abstractos, sobrecarga de funciones y operadores, polimorfismo paramétrico, y más aún, especialización de tipos definidos por el usuario. Pero Ada no soporta herencia completamente, en consecuencia según Booch, Ada es solo un lenguaje basado en objetos.

Durante los años 70s y 80s, los conceptos de lo orientado a objetos de Simula y otros de los primeros prototipos fueron materializados en uno de los lenguajes orientados a objetos más influyentes: Smalltalk (Goldberg y Robson, 1983), fue inicialmente un proyecto de investigación en Xerox Palo Alto Research Park (PARC). Durante los 70s un grupo de investigadores inventó y solidificó tecnologías ahora reconocidas como orientadas a objetos dentro del terreno de los lenguajes de programación y de interfaces de usuario. Smalltalk fue desarrollado en este tiempo. En el campo de las interfaces de usuario, dos de las estaciones de trabajo Star (también desarrollada en PARC) y su predecesor Alto, influenció el diseño y belleza de la Apple Macintosh, el software de escritorio para publicidad Aldus PageMaker, los ambientes de software Microsoft Windows y Methaphor DIS.

El lenguaje Smalltalk incorpora muchas de las características orientadas a objetos de Simula, incluyendo clases, herencia y soporte a la identidad de objetos.

En la década de los 70s y 80s, los conceptos orientados a objetos (tipos de datos abstractos, herencia, identidad, y concurrencia), comenzaron a unirse y a darle vida a nuevos lenguajes, extensiones y dialectos de Smalltalk y Simula. De Smalltalk encontramos Smalltalk/V de Digital. Como extensiones a los lenguajes convencionales (estructurados) que agregan características orientadas a objetos, algunos lenguajes que podemos listar: C++ que fue desarrollado por Bjarne Stroustrup en 1986, Objetive-C (Cox, 1987), los dos anteriores son un superconjunto de C, los dos proporcionan los tres elementos básicos de la orientación a objetos (clases, herencia y polimorfismo). Para Pascal encontramos Object Pascal para la Macintosh de Apple Computer y Turbo Pascal de Borland para PCs. Object Pascal (Schmucker, 1986) fue diseñado por Niklaus Wirth y un grupo de diseñadores de Apple Computer, este es una extensión a Pascal para soportar la noción de objeto y la definición de clase.

En 1982 se predijo que la programación orientada a objetos sería en los 80s lo que la programación estructurada fue en los 70s (Rentsch,1982). La década de

los ochenta fue conocida posiblemente como la década que lanzó la era orientada a objetos en la computación.

En esta década, los lenguajes orientados a objetos, bases de datos e interfaces de usuario están siendo utilizados en muchos desarrollos de software. Mucho del desarrollo de los sistemas de software es afectado por la orientación a objetos de una u otra forma.

Los beneficios de la tecnología objetual

Las ventajas fundamentales de las técnicas orientadas a objetos comparadas con las técnicas tradicionales estructuradas, están en la creación de un enfoque modular dirigido hacia el análisis, diseño e implementación de sistemas de software [4]. La modularidad ha sido siempre reconocida por su importancia en comunicación, ideas y en la reducción de la complejidad de los sistemas de software a un nivel comprensible.(Myers, 1978; Yourdon y Constantine, 1979).

En la etapa de análisis, el análisis orientado a objetos proporciona un solo enfoque unificado el cual se carece en el análisis estructurado de los sistemas, todas las técnicas de modelación objetual aplican los mismos elementos fundamentales del sistema, esto es, las clases de objetos, mientras que el análisis estructurado de sistemas ve aspectos diferentes de un sistema usando diferentes elementos fundamentales, las entidades de datos son las que se usan en el modelo entidad relación, y procesos en los diagramas de flujo de datos. Esto lleva a perder las vistas asociadas del sistema final el cual hace difícil el trabajo de delegación de módulos a desarrolladores de software individuales.

A través del uso del proceso de herencia, polimorfismo y enlace dinámico, la orientación a objetos proporciona mecanismos que soportan el reuso y extensibilidad de software como nunca antes. Esto ayuda a reducir los tiempos de desarrollo de software, le da mejor legibilidad a los cambios en los requerimientos y baja los costos de mantenimiento significativamente. El soporte a la extensibilidad hace a la orientación a objetos también particularmente útil en donde los requerimientos de software no estén bien entendidos. El ciclo de desarrollo incremental promueve la evaluación y reevaluación del análisis a través de la implementación en un ciclo corto, permitiendo que los requerimientos del producto se involucren con el producto bajo desarrollo.[4]

Otra de las ventajas es que al presente momento técnicas y herramientas tienen una gran madurez. En el mercado hay una gran cantidad de lenguajes que proporcionan un apoyo total a la orientación a objetos. En particular los equipos de desarrollo de lenguajes de programación de Borland y Microsoft para computadoras personales, están trabajando fuertemente en herramientas de desarrollo orientadas a objetos.

El modelo de objeto

La palabra objeto es a menudo usada vagamente en la literatura. Algunas veces objeto significa una sola cosa, otras veces se refiere a un grupo de cosas similares. Generalmente el *contexto* resuelve alguna ambigüedad.

La noción de objeto es inherente al ser humano, desde que nace, las primeras cosas con las que el niño se familiariza es con objetos reales: un juguete, una pelota, las personas, etc., además comienza a reconocer que acciones pueden hacer los objetos con los que tiene contacto, comienza a aprender que las personas caminan, hablan, ríen, gritan etc., en este proceso de aprendizaje el niño en una forma natural dota a los objetos de ciertas propiedades inherentes a él, de ésta manera el ser humano está en una interacción constante con ellos, nos sentamos en una silla, escribimos con un lápiz, golpeamos con un martillo, etcétera. El concepto de objeto existe y además hay una relación también natural entre el objeto y las acciones que puede realizar, es decir con un pico podemos cavar, con un martillo podemos extraer un clavo, con un serrucho cortar madera, y así con cada uno de los objetos del mundo físico, si sabemos cual es su funcionamiento sabremos cuales operaciones podemos hacer con él y cuales no, por ejemplo si sabemos como funciona un serrucho y para que fue hecho sabríamos inmediatamente que no deberíamos extraer un clavo o cavar un hoyo con el, primeramente porque dichas operaciones no son atribuibles a dicho objeto y segundo porque simplemente existen objetos con la funcionalidad que se le quiere atribuir. En otras palabras, no existe relación directa entre el serrucho y la acción de extraer un clavo o cavar un hoyo. Lo que estamos logrando al aislar el objeto en su funcionalidad y sus propiedades naturales, es construir una entidad con una serie de atributos y comportamientos inherentes al mismo.

La noción de objeto como tema de estudio data desde el siglo IV a.C., y fue el filósofo griego Aristóteles quien introdujo las nociones de entidad y atributo como partículas fundamentales de información en su paradigma de la información [6]. Aunque hubo muchas teorías que rivalizaron con su paradigma, la concepción Aristotélica permaneció firme.

No fue sino hasta finales del siglo pasado que el paradigma objetual surgió para retar dicha concepción. Gottlob Frege (1848-1925), en su trabajo publicado a finales del siglo pasado, rompió esta barrera y se movió a la concepción del paradigma objetual [5]. Frege vió el objeto en una forma psicológica mas que lógica, contrario a como los científicos de la computación lo ven. El se dió cuenta que las clases eran objetos y se refería a conceptos generales para clases de objetos [6].

Todos los objetos tienen identidad y son distinguibles. Dos manzanas con el mismo color , figura y textura son aún manzanas individuales; una persona puede comer una y comer la otra. Similarmente, dos gemelos idénticos son dos personas distintas, aun pensando que ellos parecieran igual. El término identidad significa que los objetos son distinguidos por propia existencia y no por la descripción que

ellos puedan tener.

La noción de objeto vino a ser una revolución conceptual en el mundo de la información, al establecer un nuevo paradigma de la información basado en la concepción intelectual del objeto como unidad atómica de la información. Dichas unidades básicas de información (objetos) en este nuevo paradigma, vinieron a constituirse en la construcción de sistemas de software, en los bloques mínimos de la arquitectura de los sistemas de información.

Los objetos sirven para dos propósitos: estimulan el entendimiento del mundo real y proporcionan bases prácticas para la implantación computacional. La descomposición de un problema en objetos depende del juicio y la naturaleza del problema. No existe ninguna representación completa.

Definiciones de Objeto

Definición de Booch. *Un objeto tiene estado, comportamiento, e identidad; la estructura y comportamiento de los objetos similares están definidas en su clase común; los términos* instancia *y* objeto *son intercambiables.* [7]

Definición de Martin. *Un objeto es cualquier cosa al cual un concepto es aplicado, y un concepto es una idea o noción que compartimos que es aplicado a ciertos objetos en nuestra conciencia.*

Definición de Rumbaugh. *Definimos un objeto como un concepto, abstracción o cosa con fronteras definidas y el significado para el problema en mano.*

Definición de Cox. *Cualquier cosa en un problema con definición de su frontera.*

Definición de Coad y Yourdon. *Una abstracción de algo en el dominio del problema o su implementación, reflejando las capacidades de un sistema para mantener información acerca de él, interactuar con él o ambos; es una encapsulación de los valores de los atributos y sus servicios exclusivos.*

Según la definición de Booch, podemos ver que consta de tres partes fundamentales, *estado, comportamiento, e identidad* [7]

Estado

Para entender el significado de estado en el modelo objetual, tomemos por ejemplo el cajero automático de un banco cualquiera, para ilustrarlo, un cajero automático, entre sus propiedades está la de aceptar dinero dentro de un sobre como depósito a alguna cuenta, o la de entregar descargándolo de alguna de la cual ya le hayan depositado al menos la cantidad solicitada, estas son dos de las muchas funciones que pudiera tener, ahora supongamos que insertamos nuestra tarjeta y suponiendo que tuviésemos dinero disponible, ¿ que sucede si solicitamos mas dinero del que tenemos disponible ?, ciertamente en la pantalla de dicho cajero aparecerá un mensaje que dirá algo parecido a los siguiente: "Cantidad excede el límite disponible", o simplemente nos regresará nuestra tarjeta. ¿Qué fué lo

que sucedió aquí?, la respuesta es simple: "se han violado los supuestos básicos de existencia del cajero". En otras palabras el cajero no esta hecho para dar mas dinero del que se tiene disponible, es decir que antes de intentar retirar dinero del cajero se debió haber hecho un deposito a la cuenta.

En los casos anteriores podemos ver que cada una de las etapas es influenciada de acuerdo a una regla de seguimiento o secuencial, es decir, que las respuestas del cajero variaran de acuerdo al tiempo en que hayan sido solicitadas, y por lo tanto son dependientes en tiempo y comportamiento. Se puede ver que el orden en que nosotros interactuemos con el objeto es muy importante, esto determina como se va a comportar el objeto para las operaciones solicitadas, este orden puede fijar el conjunto de supuestos de operación. La razón de esto es para conocer el estado dentro de un objeto bajo algún comportamiento.

Otra propiedad que tiene el cajero es la de conocer cual es la cantidad de dinero que tiene disponible para cada usuario, esta es una propiedad estática y además muy importante, o esencial, porque a cada nueva operación se modifica o no la cantidad de dinero que puede disponer. Dependiendo de esto se puede encender un estado de activa o inactiva, pero la cantidad de dinero que esta dando en ese momento es un valor dinámico de su propiedad de saber cual es la cantidad de dinero que está disponible.

Un atributo es una característica inherente o distintiva, cualitativa o rasgo que contribuye a hacer a un objeto único con respecto de otro. Todas las propiedades tienen un valor. Este valor puede ser una simple cantidad o puede denotar otro objeto [7].

Un atributo es dinámico si cambia de valor durante alguna circunstancia y es estática en caso contrario.

El estado de un objeto abarca todas las propiedades (generalmente estáticas) del objeto mas los valores actuales (generalmente dinámicos) de cada una de esas propiedades [7].

De acuerdo con Rumbaugh [19] un estado es una abstracción de los valores de un atributo y la ligas de un objeto.

Implementación de Estado en C++

En C++, podemos pensar en un ejemplo simple de un objeto que representa una fecha. Nos gustaría preguntarle a este objeto datos como "¿ Cuál es tu mes ?","¿ Cuál es tu año ?". A la definición de la clase también se le denomina la interface de la clase [21]. En el caso del objeto fecha la interface lucirían así

```
class Fecha {
public:
    enum    Mes{Ene,Feb,Mar,Abr,May,Jun,Jul,Ago,Sep,Oct,Nov,
                Dic};
    Mes getMes() const;
```

```
    int getAño() const;
};
```

Esta interface no revela nada acerca de la implementación de las variables de estado. Las variables de estado de la clase Fecha pueden ser almacenadas como sigue:

```
class Fecha {
    private:
        Mes SuMes;
        int suAño;
    public:
        ...
};
```

La implementación de la información del estado es irrelevante a la interface. Los usuarios de los objetos **Fecha** conocen que estos objetos tienen estado, pero ellos pueden solamente pueden accesar ese estado a través de la parte pública de la clase. La interface oculta la implementación del estado de la información.

Comportamiento

Los objetos están en una constantes interaccion unos con otros, dicho de otra manera su existencia depende de que existan otros objetos, ningún objeto existe aisladamente. Para que exista esta interaccion los objetos tienen que operar unos con otros. Más aún éste es él comportamiento del objeto en su contexto. Booch define:

Definición: *El comportamiento es la forma en como un objeto actúa y reacciona, en términos de sus cambios de estado y paso de mensajes.*[7]

Dicho de otra forma, el comportamiento de un objeto representa sus actividades visibles y comprobables.

Tomemos como ejemplo el objeto **Archivo**, el paso siguiente sería abstraer sus propiedades esenciales, un archivo puede estar abierto o cerrado, y en caso de que este cerrado nosotros invocaríamos a su estado a apertura para conocer que operaciones se pueden realizar con este archivo. Una operación es una acción que ejecuta un objeto sobre otro[7]. En el caso de la apertura del archivo se invocaría a la operación **Abrir** con una serie de parámetros en donde establezcamos los estados de apertura. Otra operación que pudiéramos realizar es conocer su tamaño en bytes, en este caso el objeto archivo realizaría la operación de consultar por su longitud con **TamArch**.

En lenguajes orientados a objetos puros como Smalltalk a la ejecución de una acción (operación) se le llama *paso de mensajes* [7] , para nuestros propósitos se usara operación y mensaje de la misma manera. En muchos lenguajes orientados a objetos a las operaciones que realizan los objetos sobre otros objetos se les llama

métodos, y se declaran generalmente en su clase, en C++ a una operación se le llama función miembro.

Según Booch existen tres clase de operaciones:

1. Modificador. Una operación que altera el estado de un objeto
2. Selector. Una operación que accesa el estado de un objeto, pero no lo altera
3. Iterador. Una operación que permite a todas las partes de un objeto ser accesadas en un orden bien definido.

Además agrega dos operaciones que permiten que instancias de una clase sean creadas o destruidas.

1. Constructor. Una operación que crea un objeto y/o inicializa su estado
2. Destructor. Una operación que libera el estado de un objeto y/o destruye el objeto mismo.

El comportamiento de un objeto puede ser visto en función de sus cambios de estado aunque a veces sus cambios sean estáticos. Porque alguna de las operaciones pueden modificar el estado entonces la definición de estado también afecta su comportamiento.

Implementación de Comportamiento en C++

En el diseño orientado a objetos, los objetos son manipulados por medio de envío de mensajes a ellos. En C++ enviamos mensajes a otros objetos llamando a sus funciones miembro. Por ejemplo, dado un objeto `Graficador`, pudiéramos pensar en enviarle un mensaje diciéndole que comience a imprimir en la pantalla lo que tenga calculado en su buffer, iniciando en la coordenada 10,30, y dibujando con un color azul, mas o menos de la manera siguiente:

```
class Graficador{
public:
    void Dibujar(float buffer[],float x,float y,Color& color);
    ...
};

Graficador objetoGraf;
objetoGraf.Dibujar(buffer,10,30,Azul);
```

En términos generales, un mensaje es un verbo acompañado de modificadores opcionales. En el ejemplo del graficador, `Dibujar` es el verbo y `10,30` y `Azul` son los modificadores.

Identidad

Segun Koshafian y Copeland definen identidad como:

Definición: *"Identidad es la propiedad que tiene un objeto mediante la cual se distingue de todos los demás objetos"* [8]

La técnica comúnmente usada para identificar objeto en lenguajes de programación, bases de datos y sistemas operativos son los nombres definidos por el usuario para objetos. Existen limitaciones prácticas para el uso de nombres de variables sin el soporte de la identidad del objeto. Uno de los problemas es que un objeto puede ser accesado de diferentes maneras ; esto puede ser limitado para diferentes variables. Esas variables no tienen forma de saber si ellos se refieren al mismo objeto[8]. La identidad es una característica distinguible de un objeto que denota una existencia separada del objeto, aún pensando que el objeto pueda tener los mismos valores que otro objeto [19].

Cada objeto es una instancia de un clase. Una clase implementa un tipo. Este describe ambos la estructura y el comportamiento de sus instancias. La estructura es capturada en las variables de instancia y el comportamiento es capturado en los métodos que son aplicables a las instancias.

Los valores de las variables de instancia de un objeto constituyen el estado de un objeto. Cada valor de una variable de instancia es un objeto.

Implementación de Identidad en C++

La identidad de un objeto es generada cuando un objeto es creado. El estado de un objeto (los valores de sus variables de instancia) pueden ser cambiados arbitrariamente. En C++, el concepto de identidad se da en el momento de la instanciación. Tomemos la clase `Graficador` anterior. Supongamos que tenemos nuestra ventana particionada en cuatro divisiones, y que deseamos que diferentes objetos grafiquen sobre cada diferente sección de nuestra ventana, para mantener un control local del proceso de graficación en cada sección. Entonces podemos hacer la siguiente declaración en C++, para identificar cada graficador.

```
Graficador grafDiv01;
Graficador grafDiv02;
Graficador grafDiv03;
Graficador grafDiv04;
```

A cada una de las anteriores instancias se les asigna memoria del sistema y ocupan un lugar distinto en el espacio. Además aún cuando pudiesen contener el mismo estado no son los mismos objetos.

Relación entre objetos

Un objeto no vive aisladamente y por lo tanto no tiene importancia el mismo si no es mediante la interacción con otros objetos, estableciendo un sistema

cooperativo de objetos. Estas relaciones de colaboración son determinantes en el momento de definir el contexto de operación. La relación entre algunos objetos agrega las hipótesis de lo que hacen unos con otros, incluyendo que operaciones pueden ser ejecutadas y que comportamiento resultan.

Segun Booch se han encontrado dos clases de jerarquías entre los objetos que son de mucho interés, que el llama:

1. **Enlaces**
2. **Agregación**

Enlaces

El término enlace proviene de Rumbaugh, quien lo define como "una conexión conceptual o física entre los objetos"[9]. Un objeto colabora con otros objetos a través de sus enlaces con esos objetos. Dicho en otra forma, un enlace denota una asociación específica con el cual un objeto (el cliente) aplica el servicio de otro objeto (el servidor), o a través del cual un objeto puede comunicarse con otro. Un enlace es una instancia de una asociación. El paso de mensajes entre dos objetos es típicamente unidireccional, aunque este puede ocasionalmente ser bidireccional. Como participante en un enlace, un objeto puede jugar uno de los tres roles siguientes:

1. **Actor.** Un objeto que puede operar sobre otros objetos, pero nunca se opera sobre él por parte de otros objetos. En algunos contextos, los términos *objeto activo* y *actor* son intercambiables.
2. **Servidor.** Un objeto que nunca opera sobre otros objetos; solo otros objetos operan sobre él.
3. **Agente.** Un objeto que puede hacer tanto operar sobre otros objeto como ser operado por otros objetos; un agente es generalmente creado para hacer algún tipo de trabajo en nombre de un actor u otro agente.

Visibilidad

Sean dos objetos, A y B con un enlace entre ellos. Con el fin de que A envíe un mensaje a B, B debe de ser visible a A de alguna manera. Realmente hay cuatro formas en que un objeto puede ser visible a otro. En las etapas tempranas del análisis de nuestro problema, podemos ignorar los hechos de visibilidad, pero una vez que comenzamos a concebir implementaciones concretas, debemos considerar la visibilidad entre los enlaces, porque nuestras decisiones aquí dictan el ámbito y acceso de los objetos en cada lado de un enlace.

1. El objeto servidor es global para el cliente.

2. El objeto servidor es un parámetro de alguna operación del cliente.
3. El objeto servidor es parte del objeto cliente.
4. El objeto servidor es un objeto declarado localmente en alguna operación del cliente.

Como un objeto se hace visible a otro es un problema de táctica de diseño.

Sincronización

Siempre que un objeto pasa un mensaje a otro objeto a través de un enlace, se dice que los dos objetos están sincronizados. Para los objetos en una aplicación totalmente secuencial , esta sincronizacion es una simple invocación a un método. Sin embargo si existe la presencia de múltiples hilos de control, los objetos requieren un paso de mensajes mas sofisticado para tratar con problemas de exclusión mutua que pueden ocurrir en sistemas concurrentes. Aquí por el contexto se asegura que un objeto activo se engloba en su propio hilo de control. Sin embargo cuando un objeto activo tiene un enlace a uno pasivo, podemos escoger uno de los siguientes enfoques de sincronizacion:

1. **Secuencial.** Las semántica del objeto pasivo está garantizada solamente en presencia de un único objeto activo a la vez.
2. **Protegida.** La semántica del objeto pasivo está garantizada en presencia de múltiples hilos de control, pero los clientes activos deben colaborar para llevar a cabo la exclusión mutua.
3. **Sincróna.** La semántica del objeto pasivo está garantizada en presencia de múltiples hilos de control, y el servidor garantiza la exclusión mutua.

Agregación

Mientras que un enlace denota una relación igual-igual o cliente-servidor, la agregación denota una jerarquía todo-parte, con la capacidad para ir desde el todo (también llamado el *agregado*) a sus partes (también conocido como sus *componentes*). En este sentido, la agregación es una clase especializada de asociación. Supongamos el siguiente ejemplo, sea el objeto X con un enlace al objeto Y y también un atributo A de una clase C. El objeto X es de esta manera el total y A es una de sus partes. En otras palabras, A es una parte del estado del objeto X. Dado el objeto X, es posible encontrar su correspondiente atributo A. Dado un objeto tal como A, es posible comunicarse a su objeto global (también llamado su *contenedor*) si y solo si este conoce una parte del estado de X. En la figura ?? se muestra este ejemplo. La agregación puede o no denotar contención física. Por ejemplo una computadora esta compuesta de monitor, CPU, teclado, ratón,

Figura~2.1: Ejemplo de Agregación

etc; aquí estamos hablando de contención física. Por otro lado, la relación entre una persona y su dinero es una relación de agregación que no requiere contención física. La persona únicamente tiene dinero pero ello no significa que el dinero sea una parte física de la persona.

El modelo de clase

Los conceptos de objeto y de clase están estrechamente entremezclados, por lo que no podemos hablar de un objeto sin tomar en cuenta a su clase. Sin embargo, hay importantes diferencias entre esos dos términos. Mientras que un objeto es una entidad concreta que existe en el tiempo y espacio, una clase representa solamente una abstracción, la esencia de un objeto. Por ejemplo, podemos hablar de la clase mamífero, la cual representa las características comunes de todos los mamíferos. Booch [7] define como sigue:

Definición: *Una clase es un conjunto de objetos que comparten una estructura común y un comportamiento común.*

Un solo objeto es simplemente una instancia de una clase. ¿ Qué no es una clase? Un objeto no es una clase, aunque, curiosamente, una clase puede ser un objeto. Los objetos que no comparten ninguna estructura y comportamiento comunes no pueden ser agrupados en una clase, porque por definición ellos no están relacionados excepto por su naturaleza general como objetos.

Relación entre clases

Consideremos por el momento las similitudes y diferencias entre las siguientes clases de objetos: flores, margaritas, rosas rojas, rosas amarillas, pétalos. Podemos hacer las siguientes observaciones:

1. Un margarita es una clase flor
2. Un rosa también es una flor diferente a la margarita.
3. Las rosa rojas y las rosas amarillas son ambas clases de rosas.
4. Un pétalo es una parte de ambas clases de flores.

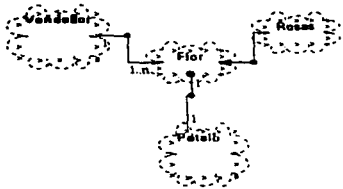


Figura 2.2: Muestra las relaciones entre clases

De este ejemplo concluimos que las clases, como los objetos no existen aisladamente. Establecemos relaciones entre dos clases por una de dos razones. Primero, una relación de clase podrá indicar algún orden de participación. Por ejemplo, las margaritas y las rosas son ambas clases de flores, significando que ambos tienen pétalos brillantemente coloreados, ambos emiten una fragancia y así sucesivamente. Segundo, la relación de clase podrá indicar alguna clase de conexión semántica. De esta manera, decimos que las rosas rojas y las rosas amarillas son más parecidas de lo que son las margaritas y las rosas, y las margaritas y las rosas están mas cercanamente relacionadas que los pétalos y las flores. Este ejemplo se puede ver más claro en la figura 2.2

Hay tres clases básicas de relaciones [9] . La primera de esas es la generalización-especialización, denotando una relación *es-una*. Por ejemplo una rosa es una clase de flor, significando que una rosa es una subclase especializada de la clase flor. La segunda es la relación *todo-parte*, la cual denota una relación *parte-de*. De esta forma, un pétalo no es una clase de flor; es una parte de una flor. La tercera es la asociación, la cual denota alguna dependencia semántica entre otras clases no relacionadas, tales como insectos y flores.

Algunos enfoques comunes han sido involucrados en los lenguajes de programación para capturar las relaciones de generalización-especialización, todo-parte y asociación. Muchos lenguajes de programación orientados a objetos proporcionan mecanismos para soportar alguna combinación de las siguientes relaciones:

1. Asociación
2. Herencia
3. Agregación
4. Uso

5. Instanciación
6. Metaclases

De las seis clases de relaciones anteriores, las asociaciones son las más generales pero las más débiles semánticamente. La herencia es quizás la más interesante, semánticamente hablando, y existen las relaciones de generalización-especialización para expresarlo. También se necesitan las relaciones de agregación, las cuales proporcionan las relaciones todo-parte manifestadas en las instancias de las clases. Adicionalmente nosotros necesitamos las relaciones de uso para establecer los enlaces entre las instancias de las clases.

Asociación

Una asociación solamente denota una dependencia semántica y no dice la dirección de esta dependencia (a menos que se diga que una asociación implica comunicación bidireccional), ni dice el camino exacto en el cual una clase se relaciona con otra (podemos solamente implicar esa semántica nombrando el rol que cada clase juega en relación con la otra) [7]. Sin embargo, esas semánticas son suficientes durante el análisis del problema, en ese momento solamente necesitamos identificar tales dependencias.

En una relación de asociación también es importante especificar la regla de correspondencia de la asociación, en otras palabras la cardinalidad de la misma. En la práctica hay tres clases de cardinalidad en una asociación :

1. Uno a uno
2. Uno a muchos
3. Muchos a muchos

Una relación uno a uno denota una relación estrecha. Por ejemplo en un alumno de una escuela tiene uno y solo un número de cuenta. Una relación uno a muchos es muy común, para ejemplificarlo el mismo alumno puede tener uno o varios cursos. Ejemplificando la relación muchos a muchos con la relación alumno-profesor, el alumno puede tener muchos profesores y el profesor muchos alumnos a su vez. En la práctica existe la variante de acotar la cardinalidad de la asociación a un número.

Herencia

La orientación a objetos intenta modelar aplicaciones del mundo real tan cercanamente a este como sea posible. La orientación a objetos también intenta llevar a cabo la reusabilidad y extensibilidad del software. El poderoso concepto orientado a objetos que proporciona *todas* esas capacidades es la *herencia* [8] .

A través de la herencia los diseñadores pueden construir nuevos módulos de software (tales como clases) sobre una jerarquía existente de módulos jerárquicos. Esto evita rediseñar y recodificar todo. Las nuevas clases pueden heredar tanto el comportamiento (operaciones, métodos) como la representación (variables de instancia, atributos) de las clases existentes.

Heredar el comportamiento habilita compartir el código (y por lo tanto permite la reusabilidad) entre los módulos. Heredar la representación habilita compartir la estructura entre los datos de los objetos. La combinación de esos dos tipos de herencia proporciona una muy poderosa estrategia de modulación y desarrollo de software. La herencia también proporciona un mecanismo muy natural para organizar la información. Esta clasifica los objetos en jerarquías de herencia bien definidas.

La herencia es una poderosa técnica que organiza complejas bases de código. Permite la construcción de nuevas clases sobre otras ya existentes. A través de la herencia existen relaciones semánticas más ricas entre las entidades en el espacio del objeto que pueden ser expresadas directa y naturalmente. De esta manera, en adición para compartir el código, la herencia también organiza los espacios del objeto en el dominio de la aplicación [8].

Dicho en una forma simple, la herencia es una relación entre las clases en donde una clase comparte las estructura y/o el comportamiento definido en una *(herencias simple)* o mas de una clase *(herencia múltiple)*. Llamamos a la clase que hereda, *superclase*. Similarmente llamamos a una clase que hereda de una o mas clases, *subclase*. La herencia por lo tanto define una jerarquía **es-una** entre clases, en la cual una subclase hereda de una o mas superclases [7]. Este es de hecho la prueba del tornasol para herencia, dadas las clases A y B, si A **no es-una** clase B, entonces A no deberá ser una subclase de B. La capacidad de un lenguaje para soportar esta clase de herencia distingue a los lenguajes orientados a objetos de los basados en objetos [7].

Una subclase típicamente aumenta o restringe la estructura y comportamiento existentes en sus superclases. Una subclase que aumenta sus superclases se dice que usa herencia por extensión [7]. Por ejemplo, la subclase X podrá extender el comportamiento de su superclase Y agregando operaciones extra que crean instancias de esta clase salvo en múltiples hilos de control. En contraste, una subclase comprime el comportamiento de su superclase se dice que usa la herencia por restricción [7]. Por ejemplo, la subclase X podrá comprimir el comportamiento de su superclase Y, prohibiendo a los clientes que usen solo algunos de sus comportamientos. Toda clase típicamente tiene dos tipos de clientes [10]:

1. Instancias
2. Subclases

Es a menudo útil definir interfaces diferentes para esos dos tipos de clientes. En particular, desearnos exponer solamente los comportamientos visibles exter-

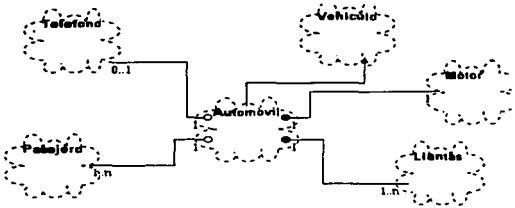


Figura 2.3: Ejemplo de Herencia

namente para instancias de los clientes, pero necesitamos exponer funciones de ayuda y representaciones solamente para clientes de subclases. Esta es precisamente la motivación para las partes públicas, privadas y protegidas de la definición de clase en C++ [7]. Un diseñador puede elegir cuales miembros son accesibles a instancias, subclases o a ambos clientes. La herencia significa que las subclases heredan la estructura de su superclase y también heredan el comportamiento de sus superclases [7].

En la figura 2.3 se muestra un ejemplo de herencia.

Ejemplo de Herencia en C++

Consideremos la siguiente clase en C++, que define una clase de archivos.

```
class Archivo{
    public:
        Archivo(const String& fileName);
        virtual int Open();
        virtual int Close();
        virtual int Read(void* theBuffer,int theLenght);
        virtual int Write(void* theBuffer,int theLenght);
    private:
        String itsFileName;
};
```

Esta clase tiene únicamente una variable privada y cuatro métodos que pueden ser llamados. Ahora consideremos la clase `ArchivoBinario`, la cual hereda la clase `Archivo`.

```
class ArchivoBinario:  public Archivo{
```

Figura 2.4: Diagrama de clases del Ejemplo de Herencia

```
    public:
        ArchivoBinario(const String& fileName);
        ...
};
```

ArchivoBinario hereda la interface y la implementación de los cuatro métodos y la variable de instancia de **Archivo**. **Archivo** es la clase base o superclase y **ArchivoBinario** la clase derivada o subclase. Decimos que **ArchivoBinario Es Un Archivo** porque este puede ser usado en cualquier parte que **Archivo** sea usado. Sin embargo **ArchivoBinario** puede ser extendido. Podemos agregar una nueva variable privada y métodos a este, y sobrecargar el comportamiento de los métodos existentes.

```
class ArchivoBinario{
    public:
        ArchivoBinario(const String& fileName);
        virtual int Read(void* theBuffer,int theLength);
        virtual int Write(void* theBuffer,int theLength);
        unsigned long getSeekPosition() const;
        void Seek(unsigned long theSeekPosition);
    private:
        unsigned long itsSeekPosition;
};
```

Ahora **ArchivoBinario** hereda la implementación solamente de **Open** y **Close**. Los métodos para **Read** y **Write** van a ser sobrecargados. Aún con los cambios en la interface de **ArchivoBinario** éste aún sigue teniendo la característica de que **ArchivoBinario Es Un Archivo**. En la figura ?? se muestra el diagrama de clases para este ejemplo.

Agregación

Las relaciones de agregación entre clases tienen un paralelismo directo a las relaciones de agregación entre objetos correspondientes a esas clases. Al hablar de tiempos de existencia de los objetos de alguna clase a la cual se le agregan objetos, nosotros contamos con que existe una dependencia directa de existencia de la instancia de la clase a la cual se le agrega una instancia de otra clase, en otras palabras, si se crea la instancia de agregación entonces en ese momento también se crea la instancia de la clase agregada, sucede lo mismo al momento de la destrucción de la instancia. La prueba del tornasol para la agregación es el siguiente: podemos tener una relación de agregación entre sus clases correspondientes si y solo si existe una relación *todo-parte* entre dos objetos [7] .

Ejemplo de Agregación en C++

Una clase puede contener otra clase. Típicamente, esto significa que una instancia de la clase contenida es mantenida en una de sus variables privadas o de instancia de la clase contenedor. Tales clases están relacionadas bajo una relación "contiene" o "tiene"; decimos que la clase **A** "contiene" o "tiene" una clase **B**. Por ejemplo tomemos el caso de una computadora personal, una clase que la describa podría contener clase para CD-ROM, monitor, teclado, procesador, tarjetas, etcétera. Una clase que la describa puede ser

```
class ComputadoraPersonal{
    private:
        CDRom suCDRom;
        Monitor suMonitor;
        Teclado suTeclado;
        Procesador suProcesador;
        Tarjetas susTarjetas;
};
```

La clase `ComputadoraPersonal` muestra como en C++, se puede implementar la agregación. La relación "contiene" también implica que el objeto contenedor tienè acceso intrínseco sobre el objeto contenido.

Uso

La relaciones de "uso" entre clases son paralelas a los enlaces uno a uno entre las instancias correspondientes de esas clases. Mientras que una asociación denota una conexión semántica bidireccional , una relación de "uso" es un posible refinamiento de una asociación, por lo que se establece cual abstracción es el cliente y cual es el servidor de ciertos servicios. La relación cliente-servidor ilustra muy bien la relación de "uso".

Las relaciones de "uso" estrictas son ocasionalmente muy limitadas porque

permiten al cliente accesar solamente a la interfaz pública del servidor. Algunas veces, por razones tácticas, debemos romper la encapsulación de esas abstracciones, lo cual es el propósito de el concepto *friend* en C++ [7].

Ejemplo de Uso en C++

Típicamente la relación de "uso" significa que la clase que usa envía un mensaje a la clase usada, la clase usada no es almacenada es una de las variables de instancia de la clase que usa. En lugar de esto la clase usada es pasada a la clase que usa por algún tercero, como un argumento de uno de los métodos de la clase que usa.

Por ejemplo, nuestra clase `ComputadoraPersonal` contiene un `CDRom`, este usa la clase de todos los objetos `CDRom`. Si nuestra computadora personal quisiera tocar su unidad de CDROM entonces tendría que pedir al objeto `CDRom` que tocara de la manera siguiente:

```
class ComputadoraPersonal{
    public:
        TocarCD(const CDRom& theCD); // Usa objetos CDRom
};

CDRom ObjCD;          // Definir una instancia CDRom
ComputadoraPersonal ObjPC; // Definir una instancia
                                  // ComputadoraPersonal
ObjPC.TocarCD(objCD); // PC ''usa'' la clase CDRom
```

Parametrización

Básicamente la parametrización se refiere al proceso de crear un ejemplar de una clase parametrizada o no parametrizada para producir una clase de la cual uno pueda crear instancias [7]

Hay cuatro formas básicas de construir clases parametrizadas [7] :

1. Primero se pueden usar macros. Este es el estilo que se ha tenido en las versiones anteriores de C++, pero como Stroustrup observa, este "enfoque no funciona bien excepto en pequeña escala" [11] porque mantener macros es torpe y fuera de la semántica del lenguaje; mas aún, cada instanciación resulta una nueva copia del código.

2. Segundo, podemos tomar el enfoque usado por Smalltalk y confiar en la herencia y en el enlace dinámico [12]. Con este enfoque podemos construir solamente clases contenedoras, porque no hay forma de asegurar la clase específica de los elementos del contenedor [7] todo elemento es tratado como si fuera una instancia de alguna clase base distinta.

3. Tercero, podemos tomar el enfoque comúnmente usado por lenguajes como Object Pascal, el cual es fuertemente tipificado, soporta herencia pero no soporta alguna forma de clases parametrizadas.

4. Cuarto, podemos tomar el enfoque introducido primero por CLU y proporcionar un mecanismo directo para clases parametrizadas. Una clase parametrizada (también conocida como clase genérica) es aquella que sirve como un template para otras clases - un template que puede ser parametrizado por, otras clases, objetos y operaciones -. Una clase parametrizada debe ser instanciada (esto es, los parámetros pueden deben ser llenados) antes de que los objeto sean creados [7].

Ejemplo de Parametrización en C++

El ejemplo siguiente muestra la implementación de un arreglo genérico, es decir este arreglo puede contener objeto de cualquier tipo que se le defina.

```
template <class T>
class Array{
    private:
         T* a;
        int size;
    public:
        Array(int);
        T& operator [] (int index);
};

template<class T>
Array<T>::Array(int sz):  a(new T[size = sz]){
    for (int i = 0; i < size;i++)
        a[i] = 0;
}
template<class T>
T& Array<T>::operator[] (int index)
{
     if (index >= 0 && index < size)
            return     a[index];
    cout     << ''Array::operator[] indice fuera de los limites'';
    return     * new T; // Usar un constructor por default del heap
}

void     main()
{
```

```
    Array     <int> I(10);          // ''Instanciación del template''

  for (int i = 0; i <= 10; i++)
       I[i] = 2*i;
}
```

El template para la clase **Array** puede contener cualquier tipo. El constructor asigna un número especificado de apuntadores T y los inicializa en 0. El operador sobrecargado [] regresa una referencia a un tipo arbitrario.

Metaclases

Hemos dicho que todo objeto es instancia de alguna clase. Lo anterior implica que podemos tratar a una clase como un objeto que puede ser manipulado. Una metaclase es una clase cuyas instancias son clases. Los lenguajes tales como Smalltalk y CLOS soportan el concepto de metaclase directamente; C++ no lo soporta. Es decir, la idea de una metaclase toma la idea del modelo objetual a su realización natural en lenguajes de programación orientados a objetos puros [7]. En lenguajes tales como Smalltalk, el propósito primario de una metaclase en proporcionar variables de clase (las cuales son compartidas por todas las variables de la clase) y operaciones para inicialización de variables de clase y para la creación de variables simples de metaclases [13] . A través de el uso de metaclases , uno puede redefinir mucho de la semántica de los elementos tales como la precedencia de la clase, funciones genéricas y métodos. El beneficio primario es que esto permite la experimentación con los paradigmas de programación orientada a objetos y las facilidades de construcción de herramientas de desarrollo de software [7].

Polimorfismo

Básicamente el polimorfismo es un concepto en la teoría de tipos, en donde un nombre puede denotar instancias de muchas clases diferentes, siempre y cuando que ellos estén relacionados por alguna superclase común. Cualquier objeto denotado por este nombre es, por lo tanto, capaz de responder a algún conjunto común de operaciones en formas diferentes [7]. El polimorfismo generalmente representa lá cualidad o estado de estar apto para asumir diferentes formas. Cuando se aplica en los lenguajes de programación la misma construcción del lenguaje puede asumir diferentes tipos o manejar objetos de diferentes tipos [7].

El concepto de polimorfismo fue descrito en primer lugar por Strachey [23], que habló primero de un polimorfismo *ad hoc*, por el cual los símbolos como "+" podrían definirse para significar cosas distintas. Hoy en dia, en los lenguajes de programación modernos, se denomina a este concepto *sobrecarga*. Strachey habló también de *polimorfismo paramétrico*, que hoy se denomina sin más, *polimorfismo*.

Ejemplo de polimorfismo en C++

El siguiente programa muestra un ejemplo de polimorfismo, llamado enlace dinámico en C++.

```
class A{
    public:
        virtual void Display() {puts(''Soy A'');}
};
class B: public A{
    public:
        virtual void Display() {puts(''Soy B'');}
};
void Show(A* a)
{
    // Encuentra en tiempo de ejecución cual función usar
    a->Display();
}
void main()
{
    A* a = new A;
    B* b = new B;
    a->Display(); // usar A::Display()
    b->Display(); // usar B::Display()
    Show(a);      // usar A::Display()
    Show(b);      // usar B::Display()
}
```

Aquí la función miembro tiene un comportamiento polimórfico. Si la palabra reservada virtual no estuviese en la definición de la clase, entonces la salida sería así:

```
void main()
{
    A* a = new A;
    B* b = new B;

    Show(a);      // usar A::Display()
    Show(b);      // usar A::Display()
}
```

La segunda llamada a la función `Show()` causaría que `A::Display()` sea llamada, porque el argumento `B*`, es convertido a `A*`, y entonces es pasado a la función `Show(A*)`. La llamada `a->Display()` es acotada a la función `A::Display()`, resultando en un comportamiento fijo.

Ejemplo de polimorfisimo paramétrico en C++

Consideremos una función, `GetTime`, que regresa en su(s) parámetro(s) la hora actual, y supóngase que necesitamos dos variantes de esta función: una que proporcione la hora en segundos a partir de la medianoche, y la otra que la proporcione en horas, minutos y segundos. No hay razón para que estas dos variantes de la función tengan nombres distintos; después de todo hacen lo mismo.

El polimorfismo paramétrico en C++, es permitido mediante la sobrecarga de funciones. C++ permite que los nombres de las funciones se sobrecarguen , es decir, que la misma función tenga más de una definición.

Podemos definir la clase `Time`,esta clase tiene la función miembro `GetTime` sobrecargada, de la siguiente manera

```
class Time {
      // ...
    void GetTime(long& ticks);
    void GetTime(int& horas,int& minutos,int& segundos);
}
```

Cuando se llama a una función miembro de la clase `Time`, el compilador compara el número y tipo de los argumentos en la llamada con las definiciones de `GetTime` y escoge la que concuerde con la llamada.

Procesos básicos de la modelación objectual

Paradigmas de programación

Jenkins y Glasgow observan que "muchos programadores trabajan en un lenguaje y usan solamente un solo estilo de programación. Programan en un paradigma forzados por el lenguaje que usan. Frecuentemente, ellos no han expuesto caminos alternativos de solución al problema , y aquí tienen dificultades en ver la ventaja de elegir un estilo mas apropiado para el problema a la mano" [13]. Bobrow y Stefik definen un estilo de programación como "una forma de organizar programas sobre la base de algún modelo conceptual de programación y un lenguaje apropiado para crear programas en el estilo"[14]. Ellos sugieren que hay cinco principales clases de estilos de programación, aquí listados con las clases de abstracciones que ellos emplean:

Orientados a los procedimientos	Algoritmos
Orientados a objetos	Clases y objetos
Orientados a lógica	Objetivos, a menudos expresados como cálculo de predicados
Orientados a las reglas	Reglas Si-Entonces
Orientados a restricciones	Relaciones invariantes

No hay un solo estilo de programación que sea el mejor para todas las clases

de aplicaciones. Por ejemplo, la programación orientada a las reglas seria la mejor para el diseño de una base de conocimientos y la programación orientada a objetos sera la mas eficaz en cuanto al diseño de operaciones de cómputo intensas [7].

Cada uno de esos estilos esta basado sobre su propio marco conceptual. Cada uno requiere un razonamiento diferente, un camino diferente de pensar acerca de el problema. Por todo esto, en la orientación a objetos, el marco conceptual es el *modelo objetual*. Existen cuatro elementos *fundamentales* este modelo[7]:

1. Abstracción
2. Encapsulación
3. Modularidad
4. Jerarquización

Por *fundamental*, entendemos que un modelo sin uno de esos elementos no es orientado a objetos. Hay tres elementos *secundarias* del modelo objetual:

1. Tipificación
2. Concurrencia
3. Persistencia

Por *secundarios*, entendemos que cada uno de esos elementos es útil, pero no es esencial, es parte del modelo objetual solamente.

Sin el marco conceptual uno puede estar programando en un lenguaje de programación tal como Smalltalk, Object Pascal, C++, CLOS, o Ada, pero nuestro diseño se va a parecer como a una aplicación de FORTRAN, Pascal, o C [7].

Abstracción

La abstracción es la examinación selectiva de ciertos aspectos de un problema. El objetivo de la abstracción es aislar esos aspectos que son importantes para algunos propósitos y suprimir aquellos aspectos que no son importantes. La abstracción debe ser siempre para algún propósito, porque el propósito determina lo que es y lo que no es importante. Muchas abstracciones diferentes de la misma cosa son posibles, dependiendo del propósito para el cual fueron hechas [19]. Todas las abstracciones son incompletas e inseguras. Son algo que decimos sobre el problema, una descripción de él, y así sucesivamente, todo ello solo es un resumen del problema. Todos los lenguajes y palabras de los humanos son abstracciones -descripciones incompletas del mundo real- . Esto no destruye su utilidad. El propósito de una abstracción es limitar el universo de tal forma que podamos hacer cosas. En la construcción de modelos, por lo tanto, no debemos

buscar la verdad absoluta pero si debemos adecuarlo a nuestros propósitos. No hay un modelo único de una situación, solamente el adecuado o no adecuado [19].

Hablar de abstracción es tocar uno de los conceptos fundamentales del modelo objetual . La abstracción es uno de las formas fundamentales que tenemos como humanos para hacer frente a la complejidad. La abstracción surge de un reconocimiento de similitudes entre ciertos objetos, situaciones o procesos en el mundo real, y la decisión para concentrarse sobre esas similitudes y para ignorar por el momento la diferencias [15]. La abstracción es una descripción simplificada, o especificación, de un sistema que enfatiza algunos de los detalles del sistema o propiedades mientras suprime otros. Una buena abstracción es aquella que enfatiza los detalles que son significantes al lector o usuario y suprime aquellos que son, al menos por el momento, inmateriales o no interesantes [16].

Según Booch, *una abstracción denota las características esenciales de un objeto que lo distingue de todos los demás tipos de objetos y de esta manera, proporciona fronteras conceptuales definidas concisamente, relativas a la perspectiva del observador* [7].

Una abstracción se enfoca sobre la vista exterior de un objeto, y sirve para separar el comportamiento esencial de un objeto de su implementación. Existen diferentes clases de abstracción las cuales podemos adaptarlas a nuestro problemas, entre ellas están [7]:

1. Abstracción de entidades.. Un objeto que representa un modelo útil una entidad del dominio del problema o del dominio de la solución.

2. Abstracción de acciones. Un objeto que proporciona un conjunto generalizado de operaciones, de la cuales todas ejecutan la misma clase de función

3. Abstracción de máquinas virtuales. Un objeto que agrupa las operaciones que son todas usadas por algún nivel superior de control u operaciones que todas usan algún con junto de operaciones de nivel inferior.

4. Abstracción de coincedencias. Un objeto que almacena un conjunto de operaciones que no tienen relación una con otra.

La intención es construir abstracciones de entidad, porque ellas son directamente paralelas al vocabulario del dominio de un problema dado. Un cliente es algún objeto que usa los recursos de otro objeto (conocido este último, como el servidor). Podemos caracterizar el comportamiento de un objeto a partir de considerar los servicios que proporciona a los otros objetos, también las operaciones que este pudiera proporcionar sobre otros objetos. Este punto de vista nos forza a concentrarnos en la vista exterior de un objeto, y nos lleva al *modelo contractual* de programación de Meyer: la vista exterior de cada objeto define un contrato sobre el cual otro u otros objetos pueden depender, y en el cual deben ser llevados a cabo por la vista interior del objeto mismo (a menudo en colaboración con

otros objetos). Este contrato de esta manera establece todas las suposiciones que un objeto cliente puede hacer acerca del comportamiento de un objeto servidor [17]. Todas las abstracciones tienen propiedades dinámicas y estáticas [7]. Consideremos el objeto ventana, este solicita cierta cantidad de memoria al sistema operativo, contiene una serie de coordenadas en donde ser ubicada, los atributos de la misma (título, barra de estado, iconera, etc). Todas esas propiedades son estáticas. El valor de cada una de esas propiedades es dinámico porque es relativo al ciclo de vida del objeto y depende de las operaciones ejecutadas con el objeto: la ventana puede crecer o disminuir; el titulo de la misma puede cambiar su contenido, es decir, la barra de estado puede hacerlo, los deslizadores, la apariencia de la misma también, etc. En la programación estructurada esos cambios dinámicos suceden cuando son invocados los procesos que modifican esos valores. En la programación orientada a los eventos las cosas suceden cuando son disparados los eventos relacionados con las propiedades. En la programación orientada a objetos las cosas suceden cuando operamos un objeto, en otras palabras *enviamos un mensaje* a un objeto. De esta manera, invocar las operaciones sobre un objeto provoca alguna reacción del objeto mismo.

Encapsulación

La encapsulación proporciona barreras explícitas entre las diferentes abstracciones y de esta manera se dirige a una clara separación de asuntos. Por ejemplo, consideremos la estructura de una planta. Para entender como trabaja la fotosíntesis en un nivel alto de abstracción, podemos ignorar detalles como las responsabilidades de las raíces de la planta o la química de las paredes celulares. Similarmente, en el diseño de una aplicación de base de datos es una práctica estándar escribir programas en la cual ellos no tienen cuidado acerca de la representación física de los datos, pero dependen del esquema que denota la vista lógica de los datos. En ambos casos, los objetos en un nivel de abstracción son separados de sus detalles de implementación en los niveles bajos de abstracción [7].

Booch define encapsulación como sigue [7]:

Definición: *La encapsulación es el proceso de separar los elementos de una abstracción que constituyen su estructura y su comportamiento; la encapsulación sirve para separar la interface contractual de una abstracción y su implementación.*

Coad define encapsulación (ocultamiento de la información) [18]:

Definición: *Un principio utilizado cuando se desarrolla sobre todo un programa estructurado, que cada componente de un programa deberá encapsular u ocultar una decisión de diseño única. La interface para cada módulo está definida de modo tal en cuanto a revelar tan poco como sea posible acerca de su trabajo interior.*

La interface de una clase captura solamente su vista exterior, abarcando nues-

tra abstracción los comportamientos comunes a todas las instancias de la clase. La implementación de una clase comprime la representación de la abstracción y también los mecanismos que llevan a cabo el comportamiento deseado [7].

La abstracción y la encapsulación son conceptos complementarios: la abstracción se enfoca sobre el comportamiento observable de un objeto, mientras que la encapsulación se enfoca sobre la implementación que da salida su comportamiento. La encapsulación es mas a menudo llevada a cabo por medio del *ocultamiento de la información*, el cual es el proceso de ocultar todos los atributos de un objeto que no contribuye a sus características esenciales; típicamente, la estructura de un objeto es oculto, además de la implementación de sus métodos [7].

La mayoría de los lenguajes orientados a objetos proveen una interface bien definida a sus objetos a través de las clases. C++ tiene un propio mecanismo general de encapsulación y protección con miembros públicos, privados y protegidos. Los miembros públicos (datos miembro y funciones miembro) pueden ser accedidos desde cualquier parte; los métodos meter y sacar de una pila serán públicos. Los miembros privados son solamente accesibles desde el interior de una clase; la representación generalmente será privada. Los miembros protegidos son accesibles desde el interior de una clase y también desde el interior de una subclase (también llamadas clases derivadas); por ejemplo, la representación de una pila puede ser declarada protegida, permitiendo el acceso a una subclase.

Otro caso es la protección de objetos y clases. En la protección de clases es mas común donde los métodos de las clases pueden acceder cualquier objeto de esa clase y no solamente el receptor. Los métodos pueden solamente acceder el receptor en la protección de objetos.

Modularidad

Para todas las aplicaciones, el primer paso en el diseño del sistema es dividir el sistema en un número pequeño de componentes. Cada componente es llamado subsistema [19]. Cada subsistema abarca los aspectos del sistema que comparten alguna propiedad común - funcionalidad similar, la misma dirección física, o la ejecución de la misma clase de hardware. Por ejemplo, la computadora de una nave espacial puede incluir subsistema soporte de vida, navegación, control de máquinas y ejecución de experimentos científicos.

Típicamente un subsistema no es modelado como un objeto ni una función pero si es un paquete de clases, asociaciones, eventos, y restricciones que están interrelacionados y que tienen una razonablemente bien definida y (quizás) pequeña interface con otros subsistemas. Un subsistema es generalmente identificado por los servicios que proporciona. Un servicio es un grupo de funciones relacionadas que comparten propósitos comunes. Un subsistema define un camino coherente de ver aspectos del problema. Por ejemplo, el sistema de archivos de un sistema operativo es un subsistema; este comprende un conjunto de abstracciones

relaçionadas que son completamente independientes de las abstracciones en otros subsistemas, tales como el subsistema de manejo de memoria o el subsistema de control de procesos[19].

Cada subsistema tiene una interface bien definida del resto del sistema. La interface especifica la forma de todas las interacciones y el flujo que cruza las fronteras del subsistema, pero no especifica como el subsistema es implementado internamente. Cada subsistema puede ser diseñado independientemente sin afectar a los otros [19].

Cada subsistema deberá ser dividido en subsistemas mas pequeños. Los subsistemas del nivel mas bajo son llamados *módulos* [19].

Un módulo captura una perspectiva o vista de la situación. Un modelo objetual consiste de uno o más módulos. Los módulos nos habilitan para particionar un modelo objetual en piezas manejables. Los módulos proporcionan una unidad intermedia de empaquetamiento entre el modelo objetual entero y la construcción básica de los bloques de las clases y las asociaciones[19].

Miers, observa, "El hecho de particionar un programa en componentes individuales puede reducir su complejidad en algún grado. Aunque particionar un programa es útil por esta razón, una justificación mas poderosa, es que crea un número de fronteras bien definidas. Esas fronteras, o interfaces, son invaluables en la comprensión del programa" [20].

Liskow dice que "la modularidad consiste en dividir un programa en módulos los cuáles pueden ser compilados separadamente, pero que tienen conexión con otros módulos"[21]. La mayoría de los lenguajes que soportan el módulo como un concepto separado también distinguen entre la interface de un módulo y su implementación. De esta manera la modularidad y la encapsulación van de la mano [7].

Los módulos sirven como contenedores en los cuáles declaramos las clases y los objetos de nuestro diseño lógico. Parnas y Briton aseguran que "el objetivo principal de la descomposición en módulos es la reducción del costo del software por permitir que los módulos sean diseñados independientemente". Cada estructura de los módulos debe ser suficientemente simple para que esta pueda ser entendida completamente; esta debe permitir cambiar la implementación de los otros módulos sin conocimiento de la implementación de los otros módulos y sin afectar el comportamiento de los módulos; y la facilidad de hacer un cambio en el diseño deberá tener una relación razonable a la posibilidad de ser cambiado" [21]. Hay un problema pragmático en esos principios. En la práctica, el costo de recompilar el cuerpo de un módulo es relativamente pequeño: solamente una unidad necesita ser recompilada y la aplicación ligada. Sin embargo, el costo de recompilar la interface de un módulo es relativamente alto. Especialmente con lenguajes fuertemente tipificados, uno debe recompilar la interface del módulo, su cuerpo, todos los otros módulos que dependen de esta interface, los módulos que dependen de esa interface, y así sucesivamente.

Según Booch *"modularidad es la propiedad que tiene un sistema que ha sido descompuesto en un conjunto de módulos cohesivos y débilmente acoplados"* [7]. De esta manera, los principios de abstracción, encapsulación, y modularidad son sinergéticos. Un objeto proporciona una frontera clara alrededor de una simple abstracción, y ambos la modularidad y la encapsulación proporcionan barreras alrededor de esta abstracción.

Jerarquización

La abstracción es algo bueno, para todo, excepto para la mayoría de aplicaciones triviales, podemos encontrar muchas mas abstracciones diferentes de las que podemos entender a la vez. La encapsulación ayuda a manejar su complejidad ocultando la vista interior de nuestra abstracción. La modularidad ayuda también, dándonos la forma para agrupar lógicamente las abstracciones relacionadas. Sin embargo, esto no es suficiente. Un conjunto de abstracciones a menudo forman una jerarquía, e identificando esas jerarquías en nuestro diseño, simplificamos grandemente el entendimiento de nuestro problema, [7]

Booch define:

Definición: *Jerarquía es una categorización y ordenamiento de las abstracciones.*

Las dos jerarquías mas importantes en un sistema son sus estructura de clase (la jerarquía **es-una**) y su estructura de objeto (la jerarquía **parte-de**). [7].

Ejemplo de jerarquía. Herencia: La herencia es la jerarquía mas importante, y como ya se ha visto, es un elemento esencial de los sistemas orientados a objetos. Básicamente, la herencia define una relación entre clases, en donde una clase comparte su estructura o comportamiento definido a una o más clases. La herencia, de esta manera representa una jerarquía de abstracciones, en la cual una subclase hereda de una o más subclases. Típicamente una subclase aumenta o redefine la estructura existente y el comportamiento de sus superclases.

Ejemplo de jerarquía: Agregación: Considerando que la jerarquía **es-una**, denota una relación generalización-especialización, la jerarquía parte-de describe relaciones de agregación.

Cuando tratamos con jerarquías tales como esas, a menudo hablamos de niveles de abstracción, un concepto descrito primero por Dijkstra [21].

En términos de su jerarquía **es-una**, una abstracción de alto nivel, es generalizada y una abstracción de bajo nivel es especializada. En términos de su jerarquía **parte-de**, una clase esta en el nivel mas alto de abstracción que alguna de las clases que constituyen su implementación.

Tipificación

El concepto de *tipo* deriva primeramente de la teoría de tipos de datos abstractos. Un tipo es una caracterización precisa de propiedades estructurales o de comportamientos, que comparten una serie de entidades [22]. Aunque los conceptos de un tipo y una clase son similares, incluimos tipificación como un elemento separado del modelo objetual porque el concepto de un tipo pone un énfasis muy diferente sobre el significado de la abstracción.

Booch define tipificación como sigue:

Definición: *Tipificación es el reforzamiento de la clase de un objeto, tal que los objetos de tipos diferentes no pueden ser intercambiados o a lo mas, pueden ser intercambiados solamente en formas muy restringidas.*[7]

En la tipificación expresamos nuestras abstracciones, para que en el lenguaje de programación en el cual las implementamos pueden ser hechas, para cumplir con las decisiones de diseño.[7]

Ejemplo de tipificación. Tipificación fuerte y débil: Un lenguaje de programación dado puede ser fuertemente tipificado, débilmente tipificado y aún no tipificado y más aún ser llamado orientado a objetos. Por ejemplo Eiffel es fuertemente tipificado, significando conformidad (compatibilidad) con el tipo es estrictamente tipificado. En los lenguajes fuertemente tipificados, la violación de conformidad del tipo puede ser detectado en tiempo de compilación. Smalltalk, por otro lado, es un lenguaje no tipificado: un cliente puede enviar un mensaje a alguna clase (aunque una clase puede no saber como responder el mensaje). Las violaciones de conformidad de tipo no pueden ser conocidas hasta el momento de la ejecución y usualmente se manifiestan ellas mismas como errores de ejecución.

Los lenguajes tales como C++ son híbridos: ellos tienen tendencias hacia el tipeo fuerte, pero es posible ignorar o suprimir las reglas de tipificación.[7]

Ejemplo de tipificación. Enlace dinámico y estático: Los conceptos de tipificación dinámica y estática son completamente diferentes. La tipificación fuerte se refiere a la consistencia de tipos mientras que la tipificación estática, se refiere al momento cuando los nombres son ligados a los tipos. El enlace estático, significa que los tipos de todas las variables y expresiones son fijadas en tiempo de compilación. El enlace dinámico (también conocido como enlace retardado) significa que los tipos de todas las variables y expresiones no son conocidas hasta el tiempo de ejecución. Como la tipificación fuerte y el enlace son conceptos independientes, un lenguaje puede ser tanto tipificado fuertemente y estáticamente (Ada), como tipificado fuertemente y aun soportar enlace dinámico (Smalltalk). [7]

Capítulo 3

ELEMENTOS DE LOS SISTEMAS DINAMICOS

Ecuaciones diferenciales

En las ciencias naturales, se llama *sistema dinámico* a un sistema o porción del universo que esta caracterizado por una colección finita de magnitudes cambiantes en el tiempo y que puede representarse mediante un sistema de ecuaciones diferenciales . Como nuestro trabajo se va a centrar en la construcción de una herramienta de análisis de sistemas dinámicos, es preciso empezar dando un conjunto de definiciones acerca de las ecuaciones diferenciales, que nos servirán como base para investigar los sistemas dinámicos y además nos guiarán hacia la generación del conjunto de herramientas que un científico desearía tener para el análisis de los sistemas dinámicos.

A una ecuación que esta definida en función de sus derivadas o diferenciales se llama *ecuación diferencial*. Existen una gran cantidad de procesos físicos, químicos y biológicos que se pueden modelar con ecuaciones diferenciales. Para desarrollar sistemáticamente la teoría de las ecuaciones diferenciales, es útil clasificar los diferentes tipos de ecuaciones. Una de las clasificaciones mas obvias se basa en si la función desconocida depende de una o varias variables independientes. En el primer caso sólo aparecen derivadas ordinarias en la ecuación diferencial y se dice que es una *ecuación diferencial ordinaria*. En el segundo caso, las derivadas son parciales y la ecuación se llama *ecuación diferencial parcial*.

Ecuaciones diferenciales lineales y no lineales

Existe una clasificación importante de las ecuaciones diferenciales ordinarias, la cuál se basa en si éstas son lineales o no lineales. Se dice que la ecuación diferencial

$$F(x, y, y', y'', ..., y^{(n)}) = 0 \tag{3.1}$$

es *lineal* cuándo F es una función lineal en las variables $y, y', ..., y^{(n)}$. Por lo tanto una ecuación diferencial lineal ordinaria de orden n es una expresión de la forma:

$$a_0(x)\, y^{(n)} + a_1(x)\, y^{(n-1)} + ... + a_n(x)\, y = g(x) \tag{3.2}$$

Una ecuación diferencial de orden n que no es de la forma (3.2), es una ecuación *no lineal*. Un problema físico sencillo que da origen a una ecuación diferencial

no lineal es el péndulo simple. El ángulo θ que forma un péndulo oscilante de longitud l, respecto a la vertical, satisface la ecuación no lineal

$$\frac{d^2\theta}{dt^2} + \frac{g}{l}\operatorname{sen}\theta = 0 \tag{3.3}$$

La teoría matemática y las técnicas usadas para resolver ecuaciones diferenciales lineales son problemas resueltos. Por el contrario, para las ecuaciones no lineales, la situación no es tan satisfactoria, pues las técnicas generales para resolverlas no existen y la teoría de tales ecuaciones es más complicada.

Sistemas Dinámicos y Forma Canónica de los Sistemas de Ecuaciones Diferenciales Ordinarias

Las magnitudes que caracterizan un sistema dinámico, sus variables de estado, pueden ser agrupadas en un vector de estado $x = (x_1, x_2, ..., x_n) \in \Re^n$. Ese vector de estado evoluciona en un espacio n-dimensional llamado espacio de estados o espacio de fases. Supóngase que la evolución de dicho vector está determinada por una función vectorial f. Generalmente, un sistema dinámico puede modelarse mediante el sistema de ecuaciones diferenciales

$$\frac{dx}{dt} = \dot{x}(t) = f(t, x), \qquad \text{donde } t \in \Re \tag{3.4}$$

$f : G \subset \Re \times \Re^n \longrightarrow \Re^n$. El problema de valores iniciales consiste en encontrar la solución de (3.4) que satisface la condición inicial $x(t_0) = x_0$. La expresión $x(t; t_0, x_0)$ denota la solución de (3.4) que satisface la condición inicial $x(t_0) = x_0$.

Por una solución de (3.4) entendemos un conjunto de funciones $x_1(t), x_2(t), ..., x_n(t)$ tal que al ser sustituidas éstas en (3.4) , se verifica la igualdad. Así, podemos pensar que cada solución del sistema es una función $\phi : I \subset \Re \rightarrow \Re^n$, con la regla de correspondencia $\phi(t) = (x_1(t), x_2(t), ..., x_n(t))$, tal que

$$\dot{\phi}(t) = f(t, \phi(t))$$

para todo $t \in \Re$.

En general, una ecuación diferencial de orden m de la forma

$$x^{(m)} = f(t, x, x', x^{(2)}, \ldots, x^{(m-1)}) \tag{3.5}$$

puede ser expresada como un sistema de m ecuaciones de primer orden mediante un cambio adecuado de variables. Por ejemplo, si introducimos nuevas variables $x_1, x_2, \ldots, x_m$, tales que $x_1 = x, x_2 = \dot{x}, \ldots, x_m = x^{(m-1)}$, resulta el sistema equivalente:

$$\begin{aligned} \frac{dx_1}{dt} &= x_2, \\ \frac{dx_2}{dt} &= x_3, \\ &\cdots \\ \frac{dx_m}{dt} &= f(t, x_1, x_2 \ldots, x_m) \end{aligned} \tag{3.6}$$

Ejemplo *La ecuación del péndulo*

$$\frac{d^2\theta}{dt^2} + \omega^2 \operatorname{sen} \theta = 0$$

es equivalente al sistema de dos ecuaciones diferenciales de primer orden

$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= -\omega^2 \operatorname{sen}(x_1) \end{aligned}$$

introduciendo las nuevas variables $x_1 = \theta$ y $x_2 = \dfrac{d\theta}{dt}$.

Existencia y Unicidad de Soluciones

Dado un problema de valores iniciales, antes de buscar la solución, primero es necesario saber si ésta existe y de ser así, que se pueda garantizar que está bien determinada por sus condiciones iniciales.

Para discutir este tema, necesitamos algunas definiciones y algunos resultados de la teoría de ecuaciones diferenciales.

Definición: *Sea $D \subset \Re^n$ y considérese $f(t,x)$ definida en $G = [t_0 - a, t_0 + a] \times D$. Se dice que $f(t,x)$ satisface la condición de Lipschitz en G respecto a x si existe una constante L tal que*

$$\| f(t, x_1) - f(t, x_2) \| \leq L \| x_1 - x_2 \| \tag{3.7}$$

para todo $t, x_1, x_2 \in G$. A L se le conoce como constante de Lipschitz.

En lugar de decir que $f(t,x)$ satisface la condición de Lipschitz a menudo se usa la expresión: "$f(t,x)$ es Lipschitz continua en x".

Teorema de Existencia y Unicidad: *Sea $G = [t_0 - a, t_0 + a] \times D$ donde $D = \{x \in \Re^n$ tal que $\|x - x_0\| \leq d\}$. Si $f(t,x)$ es continua en G y es Lipschitz continua respecto a x en G, entonces el problema de valor inicial*

$$\dot{x} = f(t,x), \qquad x(t_0) = x_0$$

tiene una y solo una solución definida para $|t - t_0| \leq \inf\left(a, \dfrac{d}{M}\right)$ con

$$M = \sup_G \|f\|$$

Las condiciones suficientes (pero no necesarias) para la existencia y unicidad de una solución son que f sea continua en G y sea continuamente diferenciable en D con respecto a x.

Sistemas Autónomos

Los *sistemas autónomos* tienen la forma

$$\frac{dx}{dt} = \dot{x}(t) = f(x) \tag{3.8}$$

donde la función f que determina el miembro derecho de la ecuación no depende explícitamente de t.

Cualquier sistema $n-$dimensional de la forma (3.4) es equivalente a un sistema autónomo de dimensión $n+1$. La equivalencia se observa al introducir una nueva variable x_{n+1} que sustituya al tiempo en el lado derecho de la ecuación y agregando la ecuación $\frac{dx_{n+1}}{dt} = 1$. Los sistemas de la forma (3.4) son llamados *no autónomos*.

Espacio de Fases y Campo Vectorial

El conjunto abierto $D \subset \Re^n$ en el cuál evoluciona el vector de variables de estado x, se llama espacio de fases. Consideremos la ecuación (3.8)

$$\frac{dx}{dt} = \dot{x}(t) = f(x)$$

definida para $x = (x_1, x_2, ..., x_n) \in D \subset \Re^n$, entonces el conjunto D es el espacio de fases. Las imágenes de las soluciones constituyen curvas es este espacio llamadas *curvas integrales o trayectorias en el espacio de fases del sistema*. Nótese que la gráfica de las soluciones, entendiendo las soluciones como funciones de $\Re \longrightarrow \Re^n$, se encuentra en el espacio $\Re \times \Re^n$ (espacio producto entre el tiempo y el espacio de fases). El teorema de existencia y unicidad garantiza que por cada punto de este espacio producto pasa una única solución y por lo tanto, que sus gráficas no pueden cortarse. Lo anterior se muestra en el siguiente teorema.

Teorema: *Las proyecciones en el espacio de fases de las gráficas de las diferentes soluciones del sistema autónomo son curvas que no se cortan en ningún punto.*

Demostración: Si $\varphi(t)$ es una solución y

$$F_\varphi = \{\varphi(t+c) \text{ tal que } c \in \Re\}$$

entonces a las gráficas de todas las soluciones que pertenecen a la familia F_φ les corresponde la misma proyección en el espacio de fases. Por otra parte si $\varphi_1(t)$ y $\varphi_2(t)$ son soluciones tales que $F_{\varphi_1} \neq F_{\varphi_2}$ entonces $\varphi_1(t)$ y $\varphi_2(t)$ no se cortan en ningún punto, pues de lo contrario: $\varphi_1(t_1) = \varphi_2(t_2) = (x_0, y_0)$ y entonces

considerando a $\varphi(t) = \varphi_2(t + t_2 - t_1)$ tendríamos que $\varphi_1(t_1) = \varphi(t_1)$ pero como φ es solución, por el teorema de existencia y unicidad $\varphi_1(t) = \varphi(t) \quad \forall t$ o lo que es lo mismo $\varphi_1(t) = \varphi_2(t + c)$ con $c = t_2 - t_1$, de donde $F_{\varphi_1} = F_{\varphi_2}$, contrario a lo que supusimos.■

Otra consecuencia que el teorema de existencia y unicidad tiene sobre los sistemas autónomos es:

Teorema: *La proyección en el espacio de fases de la gráfica de una solución del sistema autónomo, o no se corta o es cerrada.*

Demostración: Llamémosle $\varphi(t)$ a la solución en consideración y supongamos que $\varphi(t^*) = \varphi(t^* + c)$ para algún t^*, entonces la función $\varphi_1(t) = \varphi(t + c)$ es una solución que cumple que $\varphi_1(t^*) = \varphi(t^*)$ y por el teorema de existencia y unicidad $\varphi_1(t) = \varphi(t) \; \forall t$, de ahí que $\varphi(t) = \varphi(t + c) \; \forall t$.■

Desde el punto de vista de las aplicaciones, es conveniente pensar en las trayectorias del sistema en el espacio de fases porque si el sistema (3.8) representa algún sistema físico caracterizado por las variables de estado $x_1, x_2, ..., x_n$, entonces cada punto $(x_1, x_2, ..., x_n)$ del espacio de fases representa un estado del sistema y en cada tiempo t al correspondiente estado del sistema le corresponde un punto de este espacio. Si nos fijamos en el punto del espacio de fases que representa al sistema en distintos tiempos consecutivos, veremos que la evolución del sistema quedará representada por una curva, la correspondiente trayectoria del sistema.

Métodos numéricos para la solución de ecuaciones diferenciales

Cuando un científico se enfrenta con problemas que involucran resolver un sistema de ecuaciones diferenciales ordinarias, como ya hemos visto, en el caso lineal existen un buen número de métodos de solución, pero no así en el caso no lineal, para ello se han creado una serie de métodos que buscan una aproximación discreta de la solución, a dichos métodos se les llama *métodos de integración*.

Con la creación de las computadoras de alta velocidad, con las que se pueden hacer cientos o miles de cálculos por segundo, se facilita el uso de métodos numéricos para resolver los problemas con valores iniciales. Muchos problemas que eran difíciles de resolver analíticamente y que requerían de años para resolverse por cálculo manual, pueden hacerse ahora en cuestión de minutos, usando computadoras de alta velocidad. Por otra parte, no todas las preguntas que pueden hacerse sobre la solución de ecuaciones diferenciales pueden responderse usando métodos numéricos. Por ejemplo, puede ser difícil contestar preguntas tales como saber en que forma depende una solución de la condición inicial (lo cual a menudo es de interés) o cómo depende de otros parámetros del problema. Tales preguntas serían más sencillas si se conociera la solución analítica.

Los métodos de integración que se discutirán, son útiles para encontrar soluciones aproximadas de sistemas de ecuaciones diferenciales que tienen la forma

del sistema 3.4

Soluciones Discretas

La soluciones aproximadas que se construyen por métodos de integración numérica no son continuas, sino que constan de una sucesión discreta $\{x_i\}_{i=0}^{N}$ de aproximaciones de los valores de la solución en los puntos de otra sucesión $\{t_i\}_{i=0}^{N}$ llamados puntos de red, donde cada $t_i \in [a,b]$. A partir de la sucesión de aproximaciones, es posible obtener una aproximación continua de $x(t)$ utilizando alguna técnica de interpolación. Asociaremos a cada aproximación x_i el valor $x(t_i)$ que corresponde a la solución exacta de $x(t)$ en el punto t_i.

Asumiremos que los puntos de red están distribuidos uniformemente. Por ejemplo, la uniformidad de una sucesión de N puntos en el intervalo $[a,b]$ se tiene si hacemos $t_i = a + ih$ con $h = \dfrac{b-a}{N}$, $i = 0, \ldots, N$. La norma o distancia $h_i = \mid t_{i+1} - t_i \mid$ se llama *tamaño de paso* en la i-ésima iteración. Si los puntos de red están uniformemente distribuidos, el tamaño de paso es *fijo* y lo denotaremos por h. En otro caso, el tamaño de paso es *variable* y $h = \max\{h_i\}$.

Los métodos de integración que nos interesan están basados en una ecuación recursiva. Para calcular la aproximación x_{i+1}, dichas ecuaciones puede depender sólo del punto anterior (t_i, x_i). Es decir, se puede expresar en la forma

$$x_{i+1} = G(t_i, x_i)$$

dando lugar a un *método de un paso*.

El método general de un paso

Observación: Los métodos de un paso

$$x_{i+1} = G(t_i, x_i)$$

para aproximar las soluciones de la ecuación diferencial (3.4) se pueden escribir de la forma

$$x_{i+1} = x_i + h\psi(t_i, x_i, h) \tag{3.9}$$

donde ψ está determinada por f y está dada por $\psi = \dfrac{G(t_i, x_i) - x_i}{h}$

Definición: *Se dice que el método* (3.9) *tiene orden p si p es el mayor entero para el cual se cumple*:

$$\begin{aligned} d_{i+1}(h) &= x(t_{i+1}) - x_{i+1} \\ &= x(t_{i+1}) - x(t_i) - h\psi(t_i, x(t_i), h) \\ &= O\left(h^{p+1}\right), \ \forall i \end{aligned}$$

donde $x(t)$ es la solución exacta de la ecuación (3.4)

Formalmente, podemos definir la convergencia para métodos de un paso como:

Definición: *El método de un paso* (3.9) *es convergente si* $\max|x_i - x(t_i)| \to 0$ *cuando* $h = \max|t_{i+1} - t_i| \to 0$ *para alguna ecuación diferencial* $\dot{x} = f(t, x(t))$ *la cual satisface una condición de Lipschitz en* x.

Métodos de Taylor

Un ejemplo de métodos de un paso son los métodos de Taylor. De hecho constituyen la idea fundamental sobre la cual se construyen la mayoría de los métodos de un paso y multipaso. Dado el problema de valor inicial 3.4, para encontrar la solución $x(t)$, supongamos que ésta tiene derivadas al menos de orden $n+1$. Entonces, la solución se puede expandir en términos de su polinomio de Taylor de n-ésimo grado alrededor de t_i:

$$x(t_{i+1}) = x(t_i) + hx'(t_i) + \frac{h^2}{2}x''(t_i) + \cdots + \frac{h^n}{n!}x^n(t_i) + \frac{h^{n+1}}{(n+1)!}x^{(n+1)}(\xi_i) \quad (3.10)$$

donde ξ_i es algún punto del intervalo (t_i, t_{i+1}). Un método de Taylor de orden p se obtiene despreciando el último término que involucra a ξ_i. Entonces, el algoritmo que determina un método de Taylor de orden p queda expresado como sigue:

$$\begin{aligned} x_0 &= x(t_0) \\ x_{i+1} &= x_i + hx_i' + \frac{h^2}{2}x_i'' + \cdots + \frac{h^p}{p!}x_i^{(p)} \end{aligned} \quad (3.11)$$

donde

$$\begin{aligned} x_i' &= f(t_i, x_i) \\ x_i^{(p)} &= \frac{d^{p-1}f(t,x)}{dt^{p-1}}\Big|_{t=t_i} \end{aligned}$$

Por ejemplo, el método de Taylor de segundo orden estará dado por

$$x_{i+1} = x_i + hf(t_i, x_i) + \frac{h^2}{2}\left[f_t(t_i, x_i) + f_x(t_i, x_i)\, f(t_i, x_i)\right]$$

donde f_t y f_x denotan las derivadas parciales $\dfrac{\partial f}{\partial t}$ y $\dfrac{\partial f}{\partial x}$ respectivamente.

Método de Euler

El único método de Taylor de primer orden es el Método de Euler o Método de las Tangentes. A continuación consideraremos el caso de un sistema de dimensión uno para interpretar geométricamente el procedimiento de dicho método.

Como se conocen t_0 y $x(t_0)$, también se conoce la pendiente de la tangente en $t = t_0$, esto es $x' = f(t_0\,, x_0)$. Por lo tanto, podemos trazar la tangente a la gráfica de la solución en t_0 y obtener un valor aproximado x_1, de $x(t_1)$, moviéndose a lo largo de la tangente hacia t_1. Dado que $x_1 \approx x(t_1)$, entonces podemos decir que la tangente a la solución en el punto (t_1, x_1) tiene una pendiente aproximada de $x' = f(t_1, x_1)$. Así, procediendo como antes, es posible trazar la tangente a la solución en t_1 para obtener un valor aproximado x_2 para $x\,(t_2)$. Si repetimos este proceso para los siguientes puntos del intervalo en que se definió a t, obtenemos el método de Euler:

$$\begin{aligned} x_0 &= x\,(t_0) \\ x_{i+1} &= x_i + hf\,(t_i, x_i) \end{aligned} \tag{3.12}$$

Métodos de Runge-Kutta

Los Métodos de Runge-Kutta constituyen otro ejemplo de métodos de un paso. Se pensaron con el objetivo de contar con métodos que tuvieran orden p mayor o igual que 2 sin tener que conocer las derivadas de f. Pero, ¿Cómo garantizar que realmente tales métodos fueran de orden p tal como éste se definió? La respuesta es: proponiendo una forma para dichos métodos y después tratar de hacer coincidir tal expresión con la serie que define al método de Taylor de orden p. En lo que resta de ésta sección se profundiza en dicha idea.

Se propone que un método de *Runge-Kutta de orden 2* tenga la siguiente forma:

$$x_{i+1} = x_i + a_1 k_1 + a_2 k_2 \tag{3.13}$$

en donde,

$$\begin{aligned} k_1 &= hf\,(t_i, x_i) \\ k_2 &= hf\,(t_i + \alpha_1 h, x_i + \beta_1 k_1) \end{aligned}$$

Para que el orden de convergencia del método sea efectivamente dos, se escogen las constantes $a_1, a_2, \alpha_1, \beta_1$ de manera que la ecuación 3.13 coincida con la ecuación 3.11 hasta el término que contiene a h^2, es decir

$$x_i + a_i k_1 + a_2 k_2 = x_i + hf\,(t_i, x_i) + \frac{h^2}{2} f'\,(t_i, x_i)$$

Como el término x_i se encuentra en ambos lados puede omitirse. Tendremos entonces que ambas ecuaciones tienen el factor común h, por lo que el problema se reduce a resolver la igualdad

$$a_1 f(t,x) + a_2 f(t + \alpha_1 h, x + \beta_1 h f(t,x)) = f(t,x) + \frac{h}{2} f'(t_i, x_i) \tag{3.14}$$

Por la regla de la cadena,

$$f'(t,x) = \frac{d}{dt}(f(t,x)) = \frac{\partial f}{\partial t} + \frac{\partial f}{\partial x}(t,x)\frac{dx}{dt}$$

y como

$$\frac{dx}{dt} = f(x,t)$$

Sustituyendo en la ecuación 3.14,

$$\begin{aligned} & a_1 f(t,x) + a_2 f(t + \alpha_1 h, x + \beta_1 h f(t,x)) \\ = \; & f(t,x) + \frac{h}{2}\frac{\partial f}{\partial t}(t,x) + \frac{h}{2}\frac{\partial f}{\partial x}(t,x) f(t,x) \end{aligned} \tag{3.15}$$

Expandiendo el miembro izquierdo de la última ecuación en su polinomio de Taylor en dos variables de grado 1, alrededor de (t,x) tenemos:

$$\begin{aligned} & a_1 f(t,x) + a_2 f(t + \alpha_1 h, x + \beta_1 h f(t,x)) \\ = \; & a_1 f(t,x) + a_2 f(t,x) + a_2 \alpha_1 h \frac{\partial f}{\partial t}(t,x) + a_2 \beta_1 h \frac{\partial f}{\partial x}(t,x) + O(h^2) \end{aligned}$$

Sustiuyendo este valor en el primer miembro de la ecuación 3.15 y despreciando el término $O(h^2)$, nos queda

$$\begin{aligned} & a_1 f(t,x) + a_2 f(t,x) + a_2 \alpha_1 h \frac{\partial f}{\partial t}(t,x) + a_2 \beta_1 h \frac{\partial f}{\partial x}(t,x) \\ = \; & f(t,x) + \frac{h}{2}\frac{\partial f}{\partial t}(t,x) + \frac{h}{2}\frac{\partial f}{\partial x}(t,x) f(t,x). \end{aligned}$$

Dado que el polinomio de Taylor es único, necesariamente,

$$\begin{aligned} a_1 + a_2 &= 1 \\ a_2 \alpha 1 &= \frac{1}{2} \\ a_2 \beta 1 &= \frac{1}{2}. \end{aligned}$$

Estas ecuaciones tienen infinidad de soluciones , en particular cuando

$$\begin{aligned} a_1 &= a_2 = \frac{1}{2} \\ \alpha 1 &= \beta 1 = 1 \end{aligned}$$

tendremos el método de Euler Mejorado. De la misma forma cuando

$$\begin{aligned} a_1 &= 0 \\ a_2 &= 1 \\ \alpha 1 &= \beta 1 = \frac{1}{2} \end{aligned}$$

tendremos el método del Punto Medio y, con

$$\begin{aligned} a_1 &= \frac{1}{4} \\ a_2 &= \frac{3}{4} \\ \alpha 1 &= \beta 1 = \frac{2}{3} \end{aligned}$$

resulta el Método de Heun.

Análogamente, para obtener un método de Runge-Kutta de orden 3 se buscan valores para $a_1, a_2, a_2, \alpha_1, \alpha_2, \beta_1, \beta_2$ tales que

$$x_{i+1} = x_i + a_1 k_1 + a_2 k_2 + a_3 k_3$$

coincida la ecuación recursiva 3.11 hasta el término que contiene a h^3. Los métodos de cuarto orden son los más frecuentemente utilizados.

Deducir un método de Runge-Kutta de cuarto orden consiste en determinar $a_1, a_2, a_2, \alpha_1, \alpha_2, \beta_1, \beta_2$ tales que

$$x_{i+1} = x_i + a_1 k_1 + a_2 k_2 + a_3 k_3 + a_3 k_3 + a_4 k_4$$

donde,

$$\begin{aligned} k_1 &= hf(t_i, x_i) \\ k_2 &= hf(t_i + \alpha_1 h, x_i + \beta_1 k_1), \\ k_3 &= hf(t_i + \alpha_2 h, x_i + \beta_2 k_1 + \beta_3 k_2), \\ k_4 &= hf(t_i + \alpha_3 h, x_i + \beta_4 k_1 + \beta_5 k_2 + \beta_6 k_3) \end{aligned}$$

coincida con la ecuación recursiva 3.11 hasta el término que contiene a h^4. La deducción del método es totalmente análoga a la correspondiente para orden 2,

pero el álgebra necesaria es demasiada por lo que se omite y solamente se darán los 2 métodos de RK-4 más usados.
El primero está dado por las ecuaciones:

$$\begin{aligned} w_0 &= \alpha \\ k_1 &= hf(t_i, x_i) \\ k_2 &= hf(t_i + \frac{1}{2}h, x_i + \frac{1}{2}k_1), \\ k_3 &= hf(t_i + \frac{1}{2}h, x_i + \frac{1}{2}k_2), \\ k_4 &= hf(t_{i+1}, x_i + k_3), \\ w_{i+1} &= w_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \end{aligned}$$

El segundo:

$$\begin{aligned} w_0 &= \alpha \\ k_1 &= hf(t_i, x_i) \\ k_2 &= hf(t_i + \frac{1}{3}h, x_i + \frac{1}{3}k_1), \\ k_3 &= hf(t_i + \frac{2}{3}h, x_i + \frac{1}{3}k_1 + \frac{1}{3}k_2), \\ k_4 &= hf(t_{i+1}, x_i + k_1 - k_2 + k_3), \\ w_{i+1} &= w_i + \frac{1}{8}(k_1 + 3k_2 + 3k_3 + k_4) \end{aligned}$$

Método de Euler Mejorado

Con este método se busca mejorar la aproximación obtenida por el método de Euler. Para encontrar la aproximación x_{i+1}, utiliza la aproximación anterior x_i y obtiene una aproximación x_E de Euler para $x(t_{i+1})$, por lo que se pueden trazar las rectas tangentes a la solución en los puntos (t_i, x_i) y (t_{i+1}, x_E) y dibujar una recta más que pase por el punto donde se intersectan las otras dos pero cuya pendiente es el promedio de éstas, es decir, estará exactamente en medio. Finalmente, se traza una recta paralela a la recta promedio que pase por el punto (t_i, x_i) y se desplaza hacia t_{i+1}, obteniendo así, una mejor aproximación x_{i+1} de $x(t_{i+1})$.

De ahí que el método de Euler Mejorado queda expresado como sigue:

$$\begin{aligned} x(0) &= x(t_0) \\ x_{i+1} &= x_i + \frac{h}{2}(f(t_i, x_i) + f(t_{i+1}, x_E)) \end{aligned}$$

donde

$$x_E = x_i + hf(t_i, x_i)$$

Comentarios adicionales

En el apéndice A, se presenta el código fuente de los algoritmos de los tres métodos numéricos presentados, Euler, Euler Mejorado y Runge-Kutta.

Capítulo 4

PROGRAMACION EN WINDOWS

Breve introducción

En esta sección se describen los elementos del API (Application Programming Intetface) de Microsoft Windows que se usan para crear y usar ventanas; manejar relaciones entre ventanas; y dimensionar, mover y desplegar ventanas. Además se da una descripción del conjunto de rutinas de Object Windows Library (OWL) de Borland, las cuales permiten que la programación en Windows sea más fácil que si se estuviera programando en C estructurado. Igualmente se analiza como se programa el conjunto de características de Windows llamado GUI (Graphical User Interface) mas usada en computadoras personales, tales como ventanas, cajas de diálogo, íconos, menús, y parte de las características que proporciona Windows y que fueron útiles para el desarrollo de *INTEGRA++*.

Ventanas

Una ventana es una aplicación escrita para el sistema operativo Microsoft Windows, es un área rectangular de la pantalla donde la aplicación despliega salidas y recibe entradas del usuario. Una ventana comparte la pantalla con otras ventanas, incluyendo aquéllas que pertenecen a otras aplicaciones. Solamente una ventana a la vez puede recibir alguna entrada del usuario. El usuario puede usar el ratón, teclado o cualquier otro dispositivo de entrada para interactuar con esta ventana y la aplicación que le pertenece.

Las ventanas son los medios primarios que una aplicación gráfica basada en Windows tiene para interactuar con el usuario y realizar tareas, de esta manera una de la primeras tareas de una aplicación en Windows es la de crear una ventana.

Características de Windows

Windows tiene las siguientes ventajas

1. *Apariencia común.* Todas las aplicaciones tienen la misma apariencia básica. Una vez que se conoce una o dos aplicaciones de Windows, es fácil aprender otra.

2. *Independencia de los dispositivos.* Windows presenta una interface independiente de los dispositivos para aplicaciones. Una aplicación no esta limitada por los dispositivos de hardware tales como ratón, teclado o monitor. Windows se responsabiliza de los dispositivos. La aplicación solo tiene que comunicarse con el API de Windows para manipular los dispositivos.

3. *Multitarea.* Windows proporciona soporte a multitarea. Los usuarios pueden tener diversas aplicaciones en progreso a la vez. Cada aplicación puede estar activa en una ventana separada.

4. *Manejo de memoria.* Windows además proporciona manejo de memoria que rompe con el limite de los 640K de DOS. Una aplicación tiene la habilidad para usar la memoria extendida, compartir segmentos de datos e intercambiar segmentos no deseados a disco.

5. *Soporte para aplicaciones de DOS existentes.* Windows permite que la mayoría de las aplicaciones de DOS, se ejecuten en él directamente.

6. Interacción con diversos dispositivos, ratón, impresoras, plotters, módems, diversas tarjetas de video y de sonido.

7. *Un API (Application Programming Interface),* la cual es una interface de programación que encapsula un grupo de funciones que permiten a cualquier programador escribir aplicaciones para Windows, sin tener que accesar a un nivel muy bajo en la arquitectura de Windows. Con el API se logra que Windows sea un sistema operativo abierto. Cabe aclarar aquí, que dicha apertura es en relación a que el fabricante crea un conjunto de funciones las cuales permiten que las aplicaciones sobre Windows no sean solo las del fabricante o las de programadores expertos o mas aún en lenguaje ensamblador.

8. *Datos compartidos.* Windows permite transferir datos entre aplicaciones usando el "Clipboard". Cualquier tipo de datos puede ser transferido de una ventana a otra con el Clipboard, si la aplicación lo ha implementado.

9. Hay una similitud con el paradigma orientado a objetos, pero aclarando que sólo en la interfaz con el usuario, porque todas las funciones del API, fueron desarrolladas bajo el paradigma de la programación estructurada. Para adaptarlas a paradigma orientado a objetos, se tienen que encapsular en grupos de clases con la misma funcionalidad.

Aplicaciones para Windows

Toda aplicación basada en Windows crea al menos una ventana, llamada la ventana principal, que sirve como la ventana principal para la aplicación. Esta

ventana sirve como la interface primaria entre el usuario y la aplicación. La mayoría de las aplicaciones también crean otras ventanas, directa o indirectamente, para ejecutar tareas relacionadas a la ventana principal. Cada ventana participa en el desplegado de salida y en la entrada que se recibe del usuario.

Cuando iniciamos una aplicación, el sistema también asocia un botón de la barra de tareas con la aplicación. El botón de la barra de tareas contiene el icono del programa y su título.

Componentes de una Aplicación para Windows

Una ventana de la aplicación incluye elementos tales como

1. Barra de título
2. Barra de menú,
3. Menú de la ventana (anteriormente conocido como el menú del sistema),
4. Botón de minimizar, maximizar, restaurar, cerrar y tamaño.
5. Area cliente
6. Elevador horizontal
7. Elevador vertical

La ventana principal de una aplicación en Windows generalmente incluye todos esos componentes

Soporte a la orientación a objetos

Para crear objetos en la pantalla tal como ventanas, el desarrollador de la aplicación define una clase especificando las características necesarias. Instancias de la clase ventana pueden entonces ser creadas. Diversas aplicaciones pueden compartir las mismas definiciones de ventana simultáneamente. Para comunicarse con instancias de una clase ventana, los mensajes son enviados y recibidos por una función especial de la clase `TWindow`. Esa función de `TWindow` maneja todos los mensajes tales como redibujar la pantalla, desplegar íconos o menús y cambiar los contenidos del "área cliente" que indica el área de memoria en la que estamos trabajando. Como mencionamos anteriormente la comunicación entre ventanas se lleva a cabo mediante el paso de mensajes entre una ventana y otra, no como áreas de memoria sino como dos aplicaciones que "se entienden", en forma similar como lo harían dos objetos en la programación orientada a objetos.

Creando un Ciclo de Mensajes

Windows automáticamente crea una cola de mensajes para cada "thread" *(hilo de control)*. Si un "thread" crea una o mas ventanas, un ciclo de mensajes debe ser proporcionado; este ciclo de mensajes recupera mensajes desde la cola de mensaje del "thread" y los despacha a los procedimientos de la ventana apropiados.

Como Windows envía mensajes a ventanas individuales en una aplicación, un "thread" debe crear al menos una ventana antes de comenzar su ciclo de mensajes. La mayoría de las aplicaciones de Windows contienen un solo "thread" que crea las ventanas. Una aplicación típica registra la clase de ventana para su ventana principal, crea y muestra la ventana principal, y entonces inicia su ciclo de mensajes, todo en la función `WinMain`.

Creamos el ciclo de mensajes usando las funciones `GetMessage` y `DispatchMessage`. Si nuestra aplicación debe obtener entrada de caracteres del usuario, debemos incluir la función `TranslateMessage` en el ciclo. `TranslateMessage` traduce mensajes de teclas virtuales en mensajes de carácter. El siguiente ejemplo muestra el ciclo de mensajes en la función `WinMain` de una aplicación de Windows simple.

```
HINSTANCE hinst;
HWND hwndMain;

int PASCAL WinMain(HINSTANCE hInstance, HINSTANCE
                        hPrevInstance, LPSTR lpszCmdLine, int
                        nCmdShow)
{
 MSG msg;
 WNDCLASS wc;
 UNREFERENCED_PARAMETER(lpszCmdLine);

 // Registrar la clase de la ventana para la ventana principal

 if (!hPrevInstance)
 {
     wc.style = 0;
     wc.lpfnWndProc = (WNDPROC) WndProc;
     wc.cbClsExtra = 0;
     wc.cbWndExtra = 0;
     wc.hInstance = hInstance;
     wc.hIcon = LoadIcon((HINSTANCE) NULL, IDI_APPLICATION);
     wc.hCursor = LoadCursor((HINSTANCE) NULL, IDC_ARROW);
     wc.hbrBackground = GetStockObject(WHITE_BRUSH);
     wc.lpszMenuName = ''MainMenu'';
     wc.lpszClassName = ''MainWndClass'';
```

```
        if (!RegisterClass(&wc))
            return FALSE;
    }

        hinst = hInstance; // salvar el manejador de la instancia

        // Crear la ventana principal

    hwndMain = CreateWindow(''MainWndClass'', ''Sample'',
     WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, CW_USEDEFAULT,
     CW_USEDEFAULT, CW_USEDEFAULT, (HWND) NULL,
     HMENU) NULL, hinst, (LPVOID) NULL);

     // Si la ventana no puede crearse
     // finalizar la aplicación.

        if (!hwndMain)
            return FALSE;

        // Mostrar la ventana y pintar sus contenidos.

        ShowWindow(hwndMain, nCmdShow);
        UpdateWindow(hwndMain);

     // Iniciar el ciclo de mensajes

        while (GetMessage(&msg, (HWND) NULL, 0, 0))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }

     // Regresar el código de salida a Windows.

        return msg.wParam;
    }
```

Examinando la Cola de Mensajes

Ocasionalmente, una aplicación necesita examinar los contenidos de la cola de mensajes del "thread" desde fuera de su ciclo de mensajes. Por ejemplo, si

un procedimiento de la aplicación de la ventana ejecuta una operación de cálculo pesada, se quisiera que el usuario sea capaz de interrumpir la operación. A menos de que periódicamente la aplicación examine la cola de mensajes durante la operación por mensajes de ratón o teclado, ésta no responderá a entradas del usuario después de que la operación se haya completado. La razón de esto es que la función **DispatchMessage** en el ciclo de mensajes del "thread" no regresa hasta que el procedimiento finaliza el procesamiento de un mensaje.

Podemos usar la función **PeekMessage** para examinar un mensaje de la cola durante una operación. **PeekMessage** es similar a la función **GetMessage**; ambos checan un mensaje de la cola para un mensaje que coincida con un criterio de filtro y entonces copia el mensaje en un estructura **MSG**. La principal diferencia entre las dos funciones es que **GetMessage** no regresa hasta que un mensaje que coincida con un filtro sea puesto en la cola, mientras que **PeekMessage** regresa inmediatamente de que hay un mensaje en la cola.

El siguiente ejemplo muestra como usar **PeekMessage** para examinar un mensaje de la cola, para el ratón y teclado durante una operación.

```
HWND hwnd;
BOOL fDone;
MSG msg;

fDone = FALSE;
while (!fDone){
 fDone = DoOperation();

 while (PeekMessage(&msg, hwnd, 0, 0, PM_REMOVE)){
      switch(msg.message){
           case WM_LBUTTONDOWN:
           case WM_RBUTTONDOWN:
           case WM_KEYDOWN:
                fDone = TRUE;
          ...
          // Puede continuar toda la serie de mensajes que tiene
          // Windows.
      }
 }
}
```

Un ciclo como el anterior se encuentra en la mayoría de las aplicaciones para Windows y es el que se recomienda usar para examinar la cola de mensaje.

Object Windows Library(OWL)

OWL es una biblioteca de clases para Windows. Encapsula la funcionalidad de Windows en un conjunto de clases. Hay que hacer notar que el API de Windows esta escrito en C estructurado por lo que OWL lo que realiza es agrupar el API estructurado en un conjunto de clases.

Las clases que describiremos son las básicas para crear un programa en Windows. Además se esbozan brevemente las clases que se utilizaron en *INTEGRA++*.

El punto de entrada a una aplicación en OWL

OWL fue diseñado para hacer posible la creación de una aplicación en Windows escribiendo un mínimo de código. El programa más pequeño requiere solamente una función con una línea de código. Aquí un corto pero completo programa en OWL.

```
#include <owl/applicat.h>
int OwlMain(int argc, char **argv)
{
    return TApplication().Run();
}
```

La función `OwlMain()` es el equivalente de OWL de la función `main()` de DOS. Este es el punto de entrada en una aplicación OWL. OWL llama `OwlMain()` pasando dos parámetros: `argc` y `argv`. El primero es el número de argumentos en la línea de comando y el segundo un arreglo de cadenas con los argumentos de la línea de comando.

Aún cuando `OwlMain()`es el punto de entrada de un programa en OWL, se puede usar también la función del API de Windows `WinMain()`,realmente los programas en OWL derivan de la clase `TApplication` y agregan código para crear sus propias ventanas especializadas. El programa siguiente muestra como usar `WinMain()` en un programa de OWL.

```
#include <owl/applicat.h>
int PASCAL WinMain(HINSTANCE hInstance,HINSTANCE
                        hPrevInstance, LPTR cmdLine, int cmdShow )
{
    return TApplication(''Mi Applicación en Windows'',
                              hInstance, hPrevInstance, cmdLine,

                              cmdShow).Run();
}
```

El objeto `TApplication` muestra el nombre del programa en la línea de mensaje de la ventana principal.

Definición de la ventana principal

La mayoría de los programas tienen una ventana principal, OWL maneja muchos de los detalles de establecimiento de funciones para la ventana principal. Cuando corremos un programa OWL, debemos instanciar un objeto de una clase derivada de `TApplication`. La clase base `TApplication` entonces involucra la función virtual miembro `TApplication::InitMainWindow()`, la cuál es sobrecargada en nuestra clase aplicación para crear una instancia del objeto ventana que servirá como la ventana principal de nuestra aplicación. Enseguida se muestra un programa para definir la ventana principal.

```
// Programa que demuestra el funcionamiento de la definición
// de una ventana con SetMainWindow()
#include <owl/owlpch.h>
#include <owl/applicat.h>
#include <owl/framewin.h>

// Definición de la clase TIntegraApp, que herada
// de la clase base TApplication, los comportamientos
// para generar una aplicación en Windows
class TIntegraApp :  public TApplication{
 public:
            TIntegraApp() :  TApplication();
             void InitMainWindow();
            // Aqui es donde se define la ventana principal de
            // la aplicación
            SetMainWindow(new TFrameWindow(0, ''Ejemplo de un
            programa ObjectWindows''));
};


int OwlMain(int argc, char *argv [])
{
             return TIntegraApp().Run();
}
```

Creación de cajas de diálogo

Las cajas de diálogo son una parte estándard de toda aplicación en Windows, y OWL permite que el programador las cree con una gran facilidad. Solo se tiene que derivar de la clase TDialog, proporcionando el encapsulamiento con funciones miembro para trabajar con controles "hijo" y con estos enviar los mensajes que se necesitan para la creación de la ventana de diálogo.

Enseguida se muestra un programa para crear una caja de diálogo

```
#include <owl/framewin.h>
#include <owl/dialog.h>
#include <owl/applicat.h>


// Definición de la clase TIntegraAbout, la cual
// crea una caja de diálogo.  Hereda de la clase
// base TDialog los métodos y mecanismos
// de construcción de las cajas de diálogo.
class TIntegraAbout:  public TDialog{
public:
            TIntegraAbout():TDialog(NULL,''INT_ABOUT'') {}
};

// Definición de la clase TIntegraApp, la cual
// crea una aplicación en Windows.  Hereda de la clase
// base TApplication los métodos y mecanismos
// de construcción de una aplicación.
class TIntegraApp:  public TApplication{
          virtual void InitMainWindow();
};


// Crea la ventana principal de la aplicación
void TIntegraApp::InitMainWindow()
{
          MainWindow = new TFrameWindow(NULL,''Integra++
                                  About'',new TIntegraAbout,TRUE);
}


// Función principal, para la creación de la aplicación
```

```
int OwlMain(int,char **)
{
            return TIntegraApp().Run();
}
```

Un ejemplo de cajas de diálogo completo

Veamos como construir el diálogo de la figura 5.4.

```
#include <owl\owlpch.h>
#include <owl\applicat.h>
#include <owl\framewin.h>
#include <owl\static.h>
#include <owl\listbox.h>
#include <owl\inputdia.h>
#include <owl\validate.h>
#include <owl\edit.h>
#include <string.h>
#ifndef DGRAPH_H
#define DGRAPH_H
// Inicio de diálogo para Graphics
// Definición del buffer que utilizará el diálogo
struct TransfBuffGraphics{
 TransfBuffGraphics();
 BOOL graphic_3d;
 BOOL xy_projection;
 BOOL xz_projection;
 BOOL yz_projection;
 BOOL select_color_3d;
 BOOL select_color_xy;
 BOOL select_color_xz;
 BOOL select_color_yz;
 TColor color_3d;
 TColor color_xy;
 TColor color_xz;
 TColor color_yz;
};
// Clase TransfDialGraphics que heredará los
// comportamientosde TDialog, de esta se
// creará una instancia y es precisamente
// la caja de diálogo que se visualizará.
```

```
class TransfDialGraphics:  public TDialog{
public:
 TransfDialGraphics(TWindow* parent,int resId,
                    TransfBuffGraphics& ts);
 void EvDestroy(); // Sobrecargar
 void CloseWindow(int ret); // Sobrecargar
 void CmColor3d();
 void CmColorXY();
 void CmColorXZ();
 void CmColorYZ();
 TColor& getColor3d() {return color_3d;}
 TColor& getColorXY() {return color_xy;}
 TColor& getColorYZ() {return color_yz;}
 TColor& getColorXZ() {return color_xz;}
 BOOL get3d() {return graphic_3d;}
 BOOL getXY() {return xy_projection;}
 BOOL getXZ() {return xz_projection;}
 BOOL getYZ() {return yz_projection;}
private:
 BOOL graphic_3d;
 BOOL xy_projection;
 BOOL xz_projection;
 BOOL yz_projection;
 BOOL select_color_3d;
 BOOL select_color_xy;
 BOOL select_color_xz;
 BOOL select_color_yz;
 TColor color_3d;
 TColor color_xy;
 TColor color_xz;
 TColor color_yz;
 DECLARE_RESPONSE_TABLE(TransfDialGraphics);
};
#endif
// Definicion de la tabla de respuestas a los eventos
// de la caja de diálogo.
DEFINE_RESPONSE_TABLE1(TransfDialGraphics, TDialog)
 EV_WM_DESTROY,
 EV_COMMAND(IDC_SELECT_COLOR_3D,CmColor3d),
 EV_COMMAND(IDC_SELECT_COLOR_XY,CmColorXY),
 EV_COMMAND(IDC_SELECT_COLOR_XZ,CmColorXZ),
 EV_COMMAND(IDC_SELECT_COLOR_YZ,CmColorYZ),
```

```
END_RESPONSE_TABLE;
// Constructor del buffer, para inicialización
// de los datos que inicialmente requiere tener
// el diálogo
TransfBuffGraphics::TransfBuffGraphics()
{
 graphic_3d = 1;
 xy_projection = 0;
 xz_projection = 0;
 yz_projection = 0;
 select_color_3d = 0;
 select_color_xy = 0;
 select_color_yz = 0;
 select_color_xz = 0;
 color_3d = TColor::LtRed;
 color_xy = TColor::LtGreen;
 color_yz = TColor::LtBlue;
 color_xz = TColor::LtYellow;
}
// Constructor del diálogo.
// Es muy importante hacer notar que todos los controles se
//deben crear en el mismo orden que se definieron con el
//Resource Workshop
TransfDialGraphics::TransfDialGraphics(TWindow* parent,int
                                resId,TransfBuffGraphics& ts)
 :  TDialog(parent,resId),TWindow(parent)
 {
  new TCheckBox(this, IDC_3D_GRAPHIC, 0);
  new TCheckBox(this, IDC_XY_PROJECTION, 0);
  new TCheckBox(this, IDC_XZ_PROJECTION, 0);
  new TCheckBox(this, IDC_YZ_PROJECTION, 0);
  new TButton(this, IDC_SELECT_COLOR_3D, 0);
  new TButton(this, IDC_SELECT_COLOR_XY, 0);
  new TButton(this, IDC_SELECT_COLOR_XZ, 0);
  new TButton(this, IDC_SELECT_COLOR_YZ, 0);
  color_3d = TColor::LtRed;
  color_xy = TColor::LtGreen;
  color_yz = TColor::LtBlue;
  color_xz = TColor::LtYellow;
  SetTransferBuffer(&ts);
 }
 void
```

```
TransfDialGraphics::CmColor3d()
{
 TChooseColorDialog::TData colors;
 static TColor custColors[16] =
 {
 0x010101L, 0x101010L, 0x202020L, 0x303030L,
 0x404040L, 0x505050L, 0x606060L, 0x707070L,
 0x808080L, 0x909090L, 0xA0A0A0L, 0xB0B0B0L,
 0xC0C0C0L, 0xD0D0D0L, 0xE0E0E0L, 0xF0F0F0L
 };
 colors.Flags = CC_RGBINIT;
 colors.Color = color_3d;
 colors.CustColors = custColors;
 if (TChooseColorDialog(this, colors).Execute() == IDOK)
 color_3d = colors.Color;
}
void
TransfDialGraphics::CmColorXY()
{
 TChooseColorDialog::TData colors;
 static TColor custColors[16] =
 {
 0x010101L, 0x101010L, 0x202020L, 0x303030L,
 0x404040L, 0x505050L, 0x606060L, 0x707070L,
 0x808080L, 0x909090L, 0xA0A0A0L, 0xB0B0B0L,
 0xC0C0C0L, 0xD0D0D0L, 0xE0E0E0L, 0xF0F0F0L
 };
 colors.Flags = CC_RGBINIT;
 colors.Color = color_xy;
 colors.CustColors = custColors;
 if (TChooseColorDialog(this, colors).Execute() == IDOK)
 color_xy = colors.Color;
}
void
TransfDialGraphics::CmColorXZ()
{
 TChooseColorDialog::TData colors;
 static TColor custColors[16] =
 {
 0x010101L, 0x101010L, 0x202020L, 0x303030L,
 0x404040L, 0x505050L, 0x606060L, 0x707070L,
 0x808080L, 0x909090L, 0xA0A0A0L, 0xB0B0B0L,
```

```
    0xC0C0C0L, 0xD0D0D0L, 0xE0E0E0L, 0xF0F0F0L
    };
    colors.Flags = CC_RGBINIT;
    colors.Color = color_xz;
    colors.CustColors = custColors;
    if (TChooseColorDialog(this, colors).Execute() == IDOK)
    color_xz = colors.Color;
  }
  void
  TransfDialGraphics::CmColorYZ()
  {
   TChooseColorDialog::TData colors;
   static TColor custColors[16] =
  {
   0x010101L, 0x101010L, 0x202020L, 0x303030L,
   0x404040L, 0x505050L, 0x606060L, 0x707070L,
   0x808080L, 0x909090L, 0xA0A0A0L, 0xB0B0B0L,
   0xC0C0C0L, 0xD0D0D0L, 0xE0E0E0L, 0xF0F0F0L
   };
   colors.Flags = CC_RGBINIT;
   colors.Color = color_yz;
   colors.CustColors = custColors;
   if (TChooseColorDialog(this, colors).Execute() == IDOK)
   color_yz = colors.Color;
 }
 //
 // Transferir los datos aunque no se haya cerrado
 //
 void
 TransfDialGraphics::CloseWindow(int ret)
 {
  TransferData(tdGetData);
  TDialog::CloseWindow(ret);
}
void
TransfDialGraphics::EvDestroy()
{
 // Hay que decirle a la aplicacion que el diálogo se va a
// cerrar
 Parent->SendMessage(WM_DIALOG_CLOSED_GRAPHICS);
 TDialog::EvDestroy();
}
```

```
// Fin de diálogo para 3D graphics
```

La definición de la estructura `TransfBuffGraphics` es para guardar en un buffer los valores de las entradas de la caja de diálogo. Este buffer es necesario que lo conozca `TransfDialGraphics` porque de otra manera esta clase no tendría una área de memoria para guardar los datos que se van introduciendo. La forma de guardarse esta dentro de una caja negra que solo `TDialog` conoce, a `TransfDialGraphics` sólo le resta enviarle ese "*buffer*".

Windows es un sistema de manejo de ventanas por eventos, es por eso que después de definir el buffer y la clase, nosotros definimos una tabla de respuesta a eventos. Esta tabla será la que procesará cada mensaje que le llegue al Kernel de Windows de acuerdo a esta definición.

La forma de crear un diálogo se hará de acuerdo al siguiente mensaje

```
void
TIntegraWindow::CmUtilitiesGraphics()
{
     TransfDialogGraphics = new TransfDialGraphics(this,
                  DGraphics,TransfBufferGraphics);
     TransfDialogGraphics->Create();
     TransfDialogGraphics->ShowWindow(SW_SHOW);
}
```

Primeramente se crea la ventana padre en donde vivirá el diálogo y de acuerdo a un evento que se dispara desde la tabla de respuesta a eventos de esta ventana, se crea una instancia de la clasé `TransfDialGraphics`, en donde se genera la serie de eventos para procesar la caja de diálogo.

Generación de menús

Para generar un menú en una aplicación de Windows, la forma mas sencilla es utilizar el Resource Workshop (**RW**) de Borland. Con el Resource Workshop se captura el menú, utilizando el grupo de opciones de **RW**, el cual contiene una opción para la creación de menús. Una vez que se ha hecho esto, la parte siguiente es agregar dicho menú en nuestra aplicación. Esto se hace con una llamada a la función `AssignMenu()`, que recibe como parámetro el nombre del menú que definimos en **RW**. Esto se debe hacer en la llamada a la función miembro de `TWindow::SetMainWindow()` porque el menú tiene una asociación directa con la ventana que se esta creando. En otras palabras el menú va a ocupar espacio de la ventana principal. Aquí un ejemplo de como crear un menú suponiendo que se ha creado con RW un menú llamado `INTEGRA`.

```
#include <owl/framewin.h>
```

```
#include <owl/dialog.h>
#include <owl/applicat.h>


// Definición de la clase TIntegraAbout, la cual
// crea una caja de diálogo.  Hereda de la clase
// base TDialog los métodos y mecanismos
// de construcción de las cajas de diálogo.
class TIntegraAbout:  public TDialog{
public:
            TIntegraAbout():TDialog(NULL,''INT_ABOUT'') {}
};

// Definición de la clase TIntegraApp, la cual
// crea una aplicación en Windows.  Hereda de la clase
// base TApplication los métodos y mecanismos
// de construcción de una aplicación.
class TIntegraApp:  public TApplication{
            virtual void InitMainWindow();
};

// Crea la ventana principal de la aplicación
// además agrega un menú a la ventana principal
void TIntegraApp::InitMainWindow()
{
            MainWindow = new TFrameWindow(NULL,''Integra++
                        About'',new TIntegraAbout,TRUE);
            // Aqui es donde se asigna el menú, el
            // menú se llama INTEGRA.
            AssignMenu(''INTEGRA'');
}

// Programa principal, crea la aplicación y la ejecuta
int OwlMain(int,char **)
{
            return TIntegraApp().Run();
}
```

Funcionalidad de la Graphics Device Interface (GDI)

Bajo las operaciones gráficas de Windows, tan básicas como dibujar un punto,una línea, un circulo, polígonos, hasta las más especializadas como trazar una curva de Bézier (una clase de curva suave) o cargar en pantalla un bitmap, el API de Windows cuenta con la **Graphics Device Interface (GDI)**. En el corazón del sistema GDI está el **device context (contexto del dispositivo)**, comúnmente abreviado simplemente DC. Bajo Windows, antes que algo pueda ser dibujado sobre la pantalla, la aplicación debe comenzar obteniendo un manejador DC.

Preguntar a Windows por un DC, es el equivalente a preguntar por el permiso de uso de salida de tal DC, dada la característica de Windows de poder compartir los DC's. Después de obtener permiso, el manejador es incluido como un parámetro a las funciones de salida de los GDI, diciéndole a Windows cual DC se necesita.

Para el manejo de GDI, OWL ha encapsulado su funcionamiento en una clase llamada `TDC`, en la cual cada `TDC` hereda un manejador de `TGdiBase` y le hace una conversión de tipo a `Handle` a un (Handle Device Context) `HDC` usando el operador `HDC`. Las funciones Win API que toman un argumento `HDC` pueden por lo tanto ser llamadas por su correspondiente función miembro de `TDC` sin su manejador `DC` explícito.. Los objetos `DC` pueden ser creados directamente con constructores `TDC`, o vía los constructores de subclases especializadas (tales como `TWindowDC`, `TMemoryDC`, `TMetaFileDC`, `TDibDC`, y `TPrintDC`) para obtener un comportamiento especifico. Los objetos DC pueden ser construidos con un manejador `HDC` existente o uno prestado. Además puede ser creado supliendo la información del manejador de dispositivos, como con `::CreateDC`. La clase `TCreateDC` toma mucho del trabajo de creación y eliminación de `TDC`. `TDC` tiene cuatro manejadores como datos miembro protegidos:

1. `OrgBrush`,
2. `OrgPen`,
3. `OrgFont`,
4. `OrgPalette`

Esos manejadores mantienen el stock de objects GDI seleccionados en cada DC. Como los nuevos objetos GDI son seleccionados con `SelectObject` o `SelectPalette`, esos datos miembro guardan los objetos previos. Mas tarde pueden ser restablecidos individualmente con `RestoreBrush`, `RestorePen`, etcétera, o ellos pueden ser restablecidos con `RestoreObjects`. Cuando un objeto TDC es destruido (vía `TDC::~TDC`), todos los objetos seleccionados originalmente son restablecidos.. Los datos miembro `TDC::ShouldDelete` controlan la eliminación del objeto `TDC`.

Enseguida se presenta un programa ejemplo de como utilizar un objeto TDC. El funcionamiento del programa es mandar a imprimir en la ventana principal las coordenadas del punto en el que se encuentra el ratón cuando se presiona el botón izquierdo y elimina lo que se imprimió en pantalla cuándo se presiona el derecho:

```
   .
   #include <owl/owlpch.h>
   #include <owl/applicat.h>
   #include <owl/framewin.h>
   #include <owl/dc.h>
   #include <string.h>
// Definición de la clase TDibEjemTDCWindow,
   // la cual hereda de TWindow sus comportamientos
   // y sobrecarga algunas funciones de la clase TWindow.
   class TDibEjemTDCWindow :  public TWindow {
   public:
              TDibEjemTDCWindow(TWindow* parent = 0);
   protected:
              // Sobrecarga la funcion miembro de TWindow
              bool CanClose();
              // Funciones de respuesta a los mensajes
              // Respuesta al botón izquierdo
              void EvLButtonDown(uint, TPoint&);
              // Respuesta al botón derecho
              void EvRButtonDown(uint,TPoint&);
              DECLARE_RESPONSE_TABLE(TDibEjemTDCWindow);
   };


   // Tabla de respuesta a los mensajes de la ventana
   // TDibEjemTDCWindow
   DEFINE_RESPONSE_TABLE1(TDibEjemTDCWindow, TWindow)
   // Mensaje al botón izquierdo
    EV_WM_LBUTTONDOWN,
   // Mensaje al botón derecho
    EV_WM_RBUTTONDOWN,
   END_RESPONSE_TABLE;


   //Función que inicializa la ventana principal.  Constructor
   TDibEjemTDCWindow::TDibEjemTDCWindow(TWindow* parent)
   {
```

```
 Init(parent, 0, 0);
}

// Función sobrecargada para respuesta
// al cierre de la ventana
bool
TDibEjemTDCWindow::CanClose()
{
            return
            MessageBox(''Deseas salvar?'', ''Ventana modificada'',
            MB_YESNO | MB_ICONQUESTION) == IDNO;
}

// Función sobrecargada para respuesta
// al presionar el botón izquierdo
void
TDibEjemTDCWindow::EvLButtonDown(uint, TPoint& point)
{
            char s[16];
            TClientDC dc(*this);
            wsprintf(s, ''(%d,%d)'', point.x, point.y);
            dc.TextOut(point, s, strlen(s));
}

// Función sobrecargada para respuesta
// al presionar el botón derecho
void
TDibEjemTDCWindow::EvRButtonDown(uint, TPoint&)
{
            Invalidate();
}

// Definición de la clase TDibEjemTDCApp,
// la cual hereda de TApplication sus
// comportamientos y sobrecarga algunas
// funciones de la clase TApplication.
class TDibEjemTDCApp :  public TApplication{
public:
            TDibEjemTDCApp() :  TApplication() {}
            void InitMainWindow()
```

```
        {
        SetMainWindow(new TFrameWindow(0, ''Ejemplo de
        Uso de TDC ObjectWindows Program'',
        new TDibEjemTDCWindow));
        }
};

// Programa principal, crea una aplicación
// TDibEjemTDCApp y la ejecuta
int OwlMain(int , char* [])
{
        return TDibEjemTDCApp().Run();
}
```

Unas notas acerca del programa de arriba son pertinentes. Hay que darse cuenta de que en la clase `TDibEjemTDCWindow`, es donde se define la ventana principal y ahí se declara también una tabla de respuesta a los mensajes del botón izquierdo y derecho del ratón. En esta tabla se pueden declarar cualquier mensaje de Windows que pueda ser atrapado por la aplicación y que se desee una acción en particular.

En la función miembro de `TDibEjemTDCWindow::EvLButtonDown` contiene la siguiente declaración

```
TClientDC dc(*this);
```

el funcionamiento de la anterior declaración es definir un DC en la ventana principal ya que `this` trae como valor la referencia de esta ventana. Una vez creado el DC se puede mandar a llamar cualquier función miembro del GDI. En nuestro caso mandamos llamar a `TextOut()`, pero puede ser cualquiera, tal como `Ellipse()`, `Line()`, `Rectangle()`, cambiar la "brocha" con `SelectObject()`, etc. Todo acceso a las funciones gráficas del API de Windows, se tiene que hacer a través de la clase TDC.

Creación de ventanas decoradas

Para crear un menú decorado, es decir un menú con barras de herramientas y barra de estado, OWL hace uso de las clase `TDecoratedFrame`. `TDecoratedFrame` automáticamente se posiciona en su ventana cliente(debemos suplir una ventana cliente) esta tendrá el mismo tamaño que el rectángulo del cliente. Podemos agregar decoraciones adicionales tales como barras de herramientas y líneas de estado a una ventana. Podemos crear un objeto `TDecoratedFrame` sin barra de titulo borrando todos los bits en el dato miembro `style` de la estructura `TWindowAttr`.

El siguiente listado es un ejemplo de como crear un menú decorado.

```
// Ejemplo de una aplicación con un menú decorado
class TEjemDecoratedApp :  public TApplication {
public:
            TEjemDecoratedApp() :  TApplication() {}
            void InitMainWindow();
};


void TEjemDecoratedApp::InitMainWindow()
{
 // Construir la ventana del marco decorado
    TDecoratedFrame* frame = new TDecoratedFrame(0, ''Ejemplo
    de una Ventana Decorada'', new TEjemDecoratedWindow, true);
// Construir una barra de estado
TStatusBar* sb = new TStatusBar(frame, TGadget::Recessed);
// Construir una barra de control
TControlBar* cb = new TControlBar(frame);
cb->Insert(*new
TButtonGadget(CM_FILENEW,CM_FILENEW,TButtonGadget::Command));
cb->Insert(*new TButtonGadget(CM_FILEOPEN, CM_FILEOPEN,
           TButtonGadget::Command));
cb->Insert(*new TButtonGadget(CM_FILESAVE,
           CM_FILESAVE, TButtonGadget::Command));
cb->Insert(*new TButtonGadget(CM_FILESAVEAS,
           CM_FILESAVEAS, TButtonGadget::Command));
cb->Insert(*new TSeparatorGadget);
cb->Insert(*new TButtonGadget(CM_PENSIZE,
           CM_PENSIZE, TButtonGadget::Command));
cb->Insert(*new TButtonGadget(CM_PENCOLOR,
           CM_PENCOLOR, TButtonGadget::Command));
cb->Insert(*new TSeparatorGadget);
cb->Insert(*new TButtonGadget(CM_ABOUT,
           CM_ABOUT, TButtonGadget::Command));
// Insertar la barra de estado y la barra de control
// en la ventana
frame->Insert(*sb, TDecoratedFrame::Bottom);
frame->Insert(*cb, TDecoratedFrame::Top);
// Definir la ventana y su menu
SetMainWindow(frame);
GetMainWindow()->AssignMenu(''COMMANDS'');
```

```
}
int OwlMain(int argc, char * argv [])
{
        return TEjemDecoratedApp().Run();
}
```

Es importante saber que se pueden agregar o eliminar características de la ventana en forma dinámica.

Notas finales

El ambiente de programación de Windows es muy extenso, OWL contiene una gran cantidad de clases que incluyen casi toda la funcionalidad de Windows, clases para ventanas, elevadores, impresoras, etc. Se ha tratado de mencionar las partes que fueron más importantes en la construcción de *INTEGRA++*. Aunque se usaron casi todas las capacidades de programación de OWL, no se hizo énfasis en cuanto a la programación con OWL, ya que las versiones de Windows están cambiando, en consecuencia también OWL. Además que la funcionalidad arriba mostrada es en gran medida la parte mas importante que se necesita para comenzar a programar y crear aplicaciones bastante robustas en Windows.

Capítulo 5

INTEGRA++

Antecedentes

El diseño objetual de *INTEGRA++* fue hecho de acuerdo a las necesidades que se definieron al inicio de este proyecto, las cuáles fueron basadas en una versión anterior de *INTEGRA++*. Como ya existía lo que podríamos considerar un prototipo de *INTEGRA++*, no se tuvo que realizar un análisis detallado de las características del sistema, ello implicó que las etapas siguientes de desarrollo se facilitaran de alguna manera. Enseguida se muestran las diferencias más notables que hay entre esta versión y la anterior

1. Cambio de plataforma de DOS a Windows 95
2. Cambio de metodología de programación estructurada a orientada a objetos
3. Cambio de lenguaje de programación de C a C++.

Aunque podría pensarse que las diferencias anteriores son pocas en relación a una actualización de versión en cualquier sistema de software, esto no es verdad, a continuación, se analizaran las diferencias anteriores.

Primero, el cambio de platarforma de DOS a Windows 95 es muy importante ya que Windows a venido a reafirmarse como el estándar de los sistemas operativos en las computadoras personales, además de tener la interfaz gráfica mas usada a nivel mundial, razón por la cuál nuestras expectativas de dar a conocer nuestro producto crecen, agregando que la calidad de visualización e interfaz con el usuario son mejores que en DOS porque Windows ofrece herramientas para accesar su funcionalidad.

Segundo, la programación estructurada ha llegado a su fin (al menos para el desarrollo de sistemas de mediana y gran escala), el que la programación orientada a objetos se haya convertido en moda no quiere decir que sea la mejor ni la más correcta, pero si se considera que bajo este enfoque podemos construir sistemas complejos en una forma más natural, es decir, que podemos tener la contraparte de su funcionalidad del mundo real, si fuera el caso; el enfoque orientado a objetos no solo se considera útil en la construcción de software sino también en las etapas posteriores del ciclo de vida del software, es decir en el mantenimiento y la expandibilidad. Finalmente, el que una versión anterior a *INTEGRA++* se haya desarrollado con el lenguaje C facilitó el trabajo de programación de algunas funciones disponibles, ya que el lenguaje C es un subconjunto de C++. Es evidente

que el lenguaje de programación esta ligado a la metodología que apoya, en este caso C++ apoya la orientación a objetos, y como se realizó un cambio de metodología de desarrollo esto nos forzó a cambiar de lenguaje de programación, el cuál contiene muchas ventajas sobre el lenguaje C, como se analizó en el capítulo dos.

Clases de *INTEGRA++*

Una aplicación para Windows como hemos visto en el capítulo 4, en principio debe contener dos clases básicas, la clase para la aplicación y la clase para la ventana de trabajo, estas clases en OWL se llaman `TApplication` y `TWindow` o `TFrameWindow`, a partir de estas dos nosotros podemos añadirle las clases que hayamos definido para nuestra aplicación. Enseguida se muestran las clases que usa *INTEGRA++* y su funcionalidad e interacción entre ellas.

1. `GraphDC`. Esta clase encapsula la funcionalidad de un objeto gráfico. Realiza la funcionalidad de dibujar puntos, líneas, fijar los intervalos de la ventana de trabajo, realiza toda la parte gráfica. Además tiene una interfaz con los métodos del Integrador de funciones, donde encapsula el comportamiento de los métodos de integración que se usan para resolver los sistemas dinámicos. Los métodos que incluye son Runge-Kutta, Euler y Euler mejorado.

2. `FunctionHolder`. Esta realiza el manejo de las funciones de los sistemas dinámicos que van a ser analizados. Esta clase es muy importante porque permite la creación y evaluación de sistemas de orden n (por ahora la interfaz sólo permite introducir sistemas de orden 3). Por razones de diseño se planteó desde un principio sólo trabajar con sistemas de tres dimensiones.

3. `GeomGr`. Implementa la geometría que se utiliza: manejo de los ejes, es decir, la rotación y traslación, además de la definición de los límites de los ejes con los que se trabaja.

4. `TIntegraWindow`. Hereda de la clase `TWindow` sus comportamientos y se le agrega la funcionalidad deseada para *INTEGRA++*, en este caso se necesitó incluir los objetos anteriormente mencionados.

5. `TIntegraApp`. Tiene como clase base `TApplication` sus comportamientos y se le agrega la clase `TIntegraWindow` de forma que realiza el trabajo pertinente.

6. `TransfDialGraphics`. Encapsula las variables y comportamientos de la caja de diálogo de `Graphics`.

7. `TransfDialMoveTo`. Encapsula las variables y comportamientos de la caja de diálogo de `Move To`.

8. `TransfDialTmpCourses`. Encapsula las variables y comportamientos de la caja de diálogo de `Temporal Courses`.

9. `TransfDialVFParameters`. Encapsula las variables y comportamientos de la caja de diálogo de `Vector Field Parameters`.

10. `TransfDialScreenPars`. Encapsula las variables y comportamientos de la caja de diálogo de `Screen Parameters`.

11. `TransfDialNumerics`. Encapsula las variables y comportamientos de la caja de diálogo de `Numerics`.

12. `TransfDialChgSys`. Encapsula las variables y comportamientos de la caja de diálogo de `ChgSys`.

Los objetos descritos son los más importantes en el desarrollo de *INTEGRA++*, estas conforman el núcleo de la aplicación en relación al desarrollo de la misma. Es pertinente aclarar que todos los diálogos necesitan que sea declarada una clase por cada caja de diálogo como se vio en el capítulo 3. La implementación y diseño de las cajas de diálogo y menús necesitaron de mucho esfuerzo, esto es trivial en el diseño de la caja de diálogo, pero contiene una serie de trucos en la implementación, que hacen su programación muy peculiar y en cierta forma iterativa. De hacer notar algún punto malo en OWL es precisamente en la programación de las cajas de diálogo y su manejo de eventos.En el apéndice B aparece el diagrama de clases de *INTEGRA++* y en el apéndice C se muestra el diagrama de módulos de *INTEGRA++*. Se utiliza un diagrama de módulos para mostrar la asignación de clases y objetos a módulos en el diseño físico de un sistema. En el apéndice D aparece el diagrama de clases de *INTERFAZ++* y en el apéndice E se muestra el diagrama de módulos de *INTERFAZ++*. Una nota acerca de los diagramas de clases es que los nombres de las clases de los diálogos son muy largos, por eso se convino en eliminar el sufijo `TransfDial` de los nombres de las clases en los diagramas.

Recorrido a través de *INTEGRA++*

Cuándo se trabaja con *INTEGRA++*, lo primero que se tiene que conocer es el grupo de sistemas de ecuaciones con los que va trabajar, en esta primera versión se restringió a cuatro el número máximo de sistemas a introducir. Para la captura de los sistemas de ecuaciones se construyó un programa intermedio llamado *INTERFAZ++*. La función de *INTERFAZ++* es precisamente capturar el conjunto de sistemas de ecuaciones y dárselos a conocer a *INTEGRA++*, utilizando el compilador de Borland C++ 4.5. ¿Porque utilizar el compilador de Borland y no construir algún equivalente?. La respuesta a la pregunta anterior tiene que ver con la rapidez y la reusabilidad de código. Primero, si nosotros construyéramos un

intérprete, este no sería tan rápido como el código generado por el compilador de Borland. Segundo, la reusabilidad de sus bibliotecas de funciones matemáticas,es muy importante, por qué nos permite usar todas éstas sin implementarlas. En el apéndice F, dentro del manual del usuario de *INTEGRA++*, se incluye el manual del usuario de *INTERFAZ++*.

INTERFAZ++

Veamos ahora como capturar un conjunto de sistemas y estudiarlos. Primero hay que ejecutar desde la línea de comandos INTEGRA si se esta dentro de Windows o desde el Administrador de Programas. Enseguida aparecerá la ventana de trabajo que se muestra en la figura 5.1

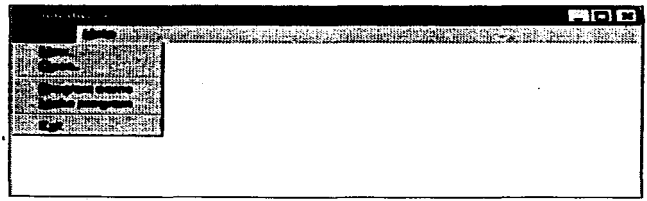


Figura 5.1: Ventana Principal de INTERFAZ++

Capturando un conjunto de sistemas de ecuaciones diferenciales

Como ejemplo tomemos el oscilador lineal armónico y el péndulo no amortiguado. Los cuales puede ser modelados con los siguientes sistemas de ecuaciones

$$\begin{aligned} \frac{dx_1}{dt} &= x_2 \\ \frac{dx_2}{dt} &= -x_1 \end{aligned}$$

para el oscilador y

$$\begin{aligned} \frac{dx}{dt} &= y \\ \frac{dy}{dt} &= -\left(\frac{g}{l}\right) \operatorname{sen} x \end{aligned}$$

para el péndulo.

Para capturar este conjunto de ecuaciones, haremos el siguiente proceso.

Desde el menú **Project** de INTERFAZ++ elegir la opción **New.** Esta opción ha sido generada para capturar un nuevo conjunto de sistemas de ecuaciones diferenciales. Enseguida aparece la caja de diálogo de la gráfica 5.2

La caja de diálogo de la figura 5.2 consta de cuatro secciones:

1. **Project**
2. **System**
3. **Equations**
4. **Parameters**

Enseguida se describe brevemente la función de cada uno de ellos.

Project

Aquí se captura el nombre del proyecto y una descripción de este. Está pensado para que el usuario, registre información acerca del grupo de sistemas con el que está trabajando.

System

En esta sección de la pantalla, está una lista de datos, que es en donde se van a almacenar los nombres de los sistemas. El sistema se guarda con el comando **Toggle** de esta sección. Es importante notar que el usuario tiene que apuntar con el ratón la región del sistema que desea capturar. Lo anterior es porque en la sección de **Equations** necesita conocer el sistema que se está capturando, para asociarle el grupo de ecuaciones.

Equations

Aquí se capturan las ecuaciones diferenciales del sistema dinámico que se vaya a analizar. Además permite que se introduzca información relacionada con dicho sistema. Como los sistemas que vamos a trabajar pueden ser tridimensionales, entonces necesitamos introducir los miembros derechos de las ecuaciones de x', y', z'.

Parameters

Algunos sistemas de ecuaciones contienen entre sus términos algunas constantes, consideraremos a estas como parámetros del mismo. Estos se capturan en una lista de datos, la primera sección de captura es para el nombre del parámetro

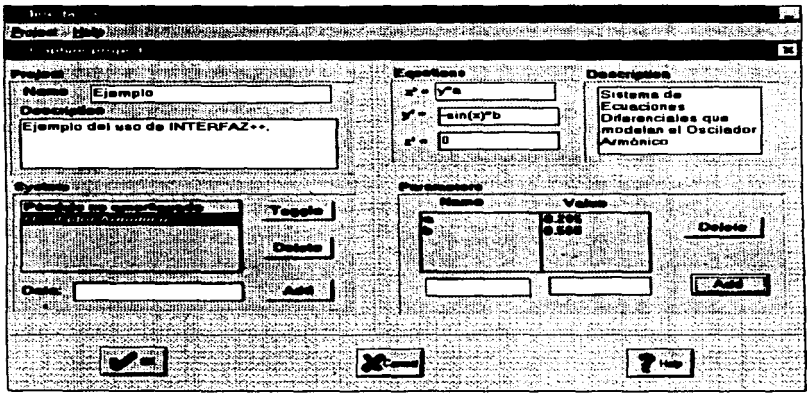


Figura 5.2: Caja de Diálogo para Capturar un Proyecto Nuevo

y la segunda para el valor del mismo. Los nombres de los parámetros en esta lista deben de tener su contraparte en la sección de ecuaciones.

Generando un sistema

Una vez que se ha capturado toda la información que *INTERFAZ++* solicita, se tiene que presionar el botón de <**OK**> para aceptar el sistema y ligarlo a *INTEGRA++*. El presionar dicho botón tiene la acción de crear un programa ejecutable con el nombre que se le haya indicado, en la caja de edición del nombre del proyecto. Si todo ha salido bien se podrá ejecutar *INTEGRA++* y comenzar el estudio de los sistemas dinámicos que se definieron con *INTERFAZ++*.

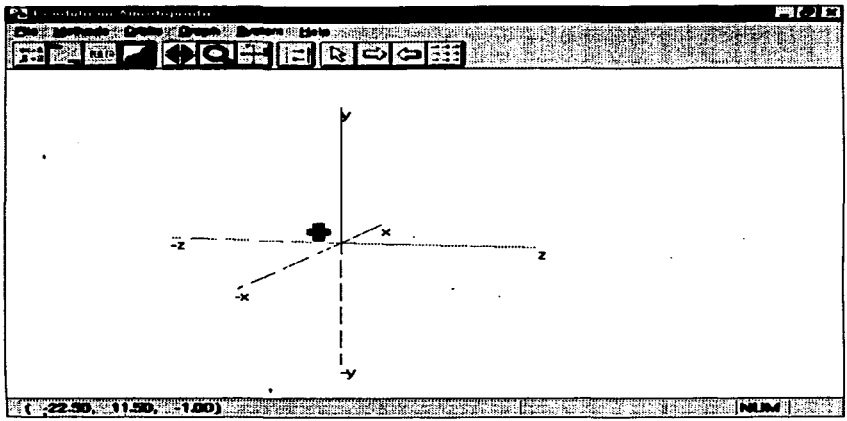


Figura 5.3: Ventana Principal de Trabajo de INTEGRA++

Funcionamiento general de *INTEGRA++*

Ventana de trabajo de *INTEGRA++*

La ventana principal de INTEGRA++ es el área de trabajo principal y es también nuestro espacio de visualización. Es aquí donde se realizará el análisis interactivo de los sistemas dinámicos. Esta ventana aparece como en la figura 5.3

Veamos que la ventana de trabajo por default dispone de un sistema de coordenadas tridimensional, el cual se puede modificar dinámicamente a través del la serie de menús que presenta *INTEGRA++*. Es importante saber que la parte punteada corresponde a su segmento negativo.

Menú de *INTEGRA++*

INTEGRA++ tiene los siguientes menús.

1. *File*. Establece las funciones de archivos.
   (a) *New*. Abre un nuevo archivo
   (b) *Open*. Abre un archivo existente
   (c) *Save*. Salva el archivo actual
   (d) *Save as*. Salva el archivo actual con otro nombre.
   (e) *Print*. Imprime el sistema actual.
   (f) *Page setup*. Establece las opciones de impresión del tipo de papel.
   (g) *Printer setup*. Establece las opciones de la impresora.
   (h) *Exit*. Terminar el programa

2. *Methods*. Establece qué tipo de método de integración se va a usar, para encontrar las soluciones del sistema dinámico. Solo un de los tres siguientes puede estar activado a la vez. Es decir son mutuamente excluyentes. Por default el método es Runge-Kutta.
   (a) *Runge-Kutta*. Fija como método de integración el de Runge-Kutta.
   (b) *Euler*. Fija como método de integración el de Euler.
   (c) *Improved Euler*. Fija como método de integración el de Euler Mejorado.

3. *Orbits*. Calcula y gráfica las órbitas del sistema dinámico en uso.
   (a) *Forward*. Calcula la órbita con tiempo positivo. las condiciones iniciales del sistema son las del cursor, en el momento que se manda a ejecutar esta opción.
   (b) *Backward*. Igual que la opción anterior sólo que la órbita se calcula con tiempo negativo.

4. *Graph*. Este es un grupo de herramientas de trabajo que *INTEGRA++* proporciona al usuario.
   (a) *Axis*. Establece que tipo de sistema coordenado se quiere para trabajar.
      i. *Default View*. Fija como sistema coordenado el espacio XYZ.
      ii. *Xy Plane View*. Fija como plano de visión XY.
      iii. *xZ Plane View*. Fija como plano de visión XZ.
      iv. *Yz Plane View*. Fija como plano de visión YZ.
      v. *Move Origin*. Mueve el origen del sistema de coordenadas actual.
      vi. *Grade Axis*. Graduar los ejes.
      vii. *Near/Far View*. Acercar o alejar al observador.

viii. *Rotations*. Rotaciones del sistema coordenado que se esté utilizando actualmente. Todo lo que se haga con el cursor y con las flechas de movimiento de aquí en adelante o hasta que se cambie de opción, será rotar los ejes de coordenadas.

(b) *Set Axis Colors*. Establece los colores para cada eje de coordenadas.

i. *X axis*. Fija el color para el eje X.

ii. *Y axis*. Fija el color para el eje Y.

iii. *Z axis*. Fija el color para el eje Z.

(c) *Cursor*. Opciones del cursor.

i. *Move to*. Mover el cursor hacia una posición especifica el espacio XYZ.

ii. *Origin*. Mover el cursor hacia el origen de coordenadas (0,0,0).

iii. *Step*. Tamaño de paso del cursor.

iv. *See coordinates*. Ver las coordenadas en la barra de estado.

(d) *Tmp courses*. Cursos temporales del sistema dinámico.

(e) *Vector Field*. Grafica el campo vectorial del campo vectorial actual.

i. *Parameters*. Parámetros para la graficación del campo vectorial.

ii. *Vector Field*. Grafica el campo vectorial.

iii. *Clear Screen*. Borra el contenido del la ventana principal. Esta operación también se puede conseguir presionando el botón derecho del ratón.

(f) *Text*. Define un texto para la ventana principal.

5. *System*. Establece las condiciones del sistema dinámico..

(a) *System Parameters*. Fija los valores de los parámetros del sistema, si este contiene alguno.

(b) *Screen Parameters*. Fija la dimensión de los ejes coordenados.

(c) *Numerics*. Fija el método numérico con el que se van a calcular las soluciones del sistema

(d) *Change System*. Cambia el sistema dinámico de trabajo, si existe otro en el proyecto.

(e) *Integra*. Caja de diálogo acerca de *INTEGRA++*.

6. *Help*. Ayuda para *INTEGRA++*.

(a) *About*. Acerca de *INTEGRA++*.

Trabajando con un proyecto

Debido a que se consideró que al usuario le es muy familiar la interfaz de Windows, solo se describen las opciones más usuales al analizar un sistema dinámico. Las demás se pueden inspeccionar a criterio del usuario.

Tomemos de ejemplo el péndulo y el oscilador armónico, que se capturaron con *INTERFAZ++* en la segunda sección. Al entrar a *INTEGRA++*, se inicia con la ventana de la figura 5.3

En dicha figura aparece, la ventana principal de *INTEGRA++*. Aquí es donde se va a realizar el análisis interactivo del sistema. Este será a través de los menús o de los íconos que se pueden ver en la barra de control. Cada ícono de la barra de control, es una orden para *INTEGRA++*, para saber que comando es, se despliega en la barra de estado de que operación se trata, al señalarla con el ratón.

Movimiento del cursor

El cursor se puede mover con las teclas de movimiento. En la barra de estado se pueden ver las coordenadas en (x, y, z). Los movimientos del cursor son los siguientes

$\rightarrow$: Mueve el cursor sobre el eje X positivamente.

$\leftarrow$: Mueve el cursor sobre el eje X negativamente.

$\uparrow$: Mueve el cursor sobre el eje Y positivamente .

$\downarrow$: Mueve el cursor sobre el eje Y negativamente.

$PgUp$: Mueve el cursor sobre el eje Z.positivamente

$PgDn$: Mueve el cursor sobre el eje Z negativamente.

Por default el cursor está en el origen.

Establecer los colores de los ejes de coordenadas

Para establecer los ejes de coordenadas hay que ir al menú *Graph* y seleccionar la opción *Axis* y después *Set Axis Colors* como se puede ver en la figura 5.4, dependiendo del eje que se desee establecer el color.

Fijándonos en la figura 5.4, vemos que tiene opciones para activar uno de los planos de visualización, XYZ, XY, XZ,o YZ. Estos se fijan haciendo "click" con el ratón sobre la opción deseada. Las demás opciones se activan de una manera similar.

Graficación del campo vectorial

El campo vectorial se puede graficar con una opción del menú *Graph*. Esta opción evidentemente se llama *Vector Field*. Vector Field contiene tres opciones:

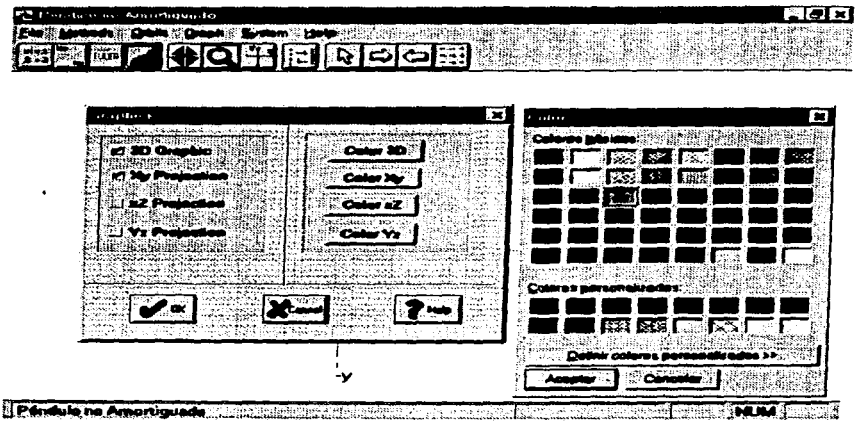


Figura 5.4: Caja de Diálogo para Establecer los Colores de los Ejes de Coordenadas

Parameters

Fija los parámetros del campo vectorial tal como se muestra en la figura 5.5. Los parámetros son el número de vectores en el eje X y en el Y. Así como fijar el tamaño del vector. *Fixed Field* en la opción que grafica el campo vectorial con una valor predeterminado por *INTEGRA++*.

Vector Field

Tiene la acción de graficar el campo vectorial una vez que se ha presionado el botón izquierdo del ratón.

Clear Screen

Elimina las gráficas que estén en la ventana principal.

Ejemplo

Un ejemplo de graficación del campo vectorial se puede ver en la figura 5.6, que muestra el campo vectorial del oscilador armónico.

Modificación de los parámetros

Si se desearan modificar los parámetros del sistema tenemos que ir al menú *System* y elegir la opción *System Parameters*. Si el sistema tuviera parámetros entonces aparecería una caja de diálogo como el de la figura 5.7.

Esta caja de diálogo contiene los parámetros que se introdujeron con *INTERFAZ++*. Recuérdese que las ecuaciones del oscilador armónico fueron capturadas como sigue

$$\begin{aligned} x' &= y * a \\ y' &= -\operatorname{sen}(x) * b \\ z' &= 0 \end{aligned}$$

INTEGRA++ toma a a y a b como parámetros. en consecuencia estos son los que aparecen en la lista de parámetros de la caja de diálogo *Parameters*. Los valores que fueron capturados en *INTERFAZ++*, pueden ser modificados ahí.

Límites de los ejes de coordenadas

Los límites de los ejes de coordenadas se pueden fijar en el submenú de *System*,en la opción *Screen Parameters*. Aquí podemos fijar los valores mínimos y

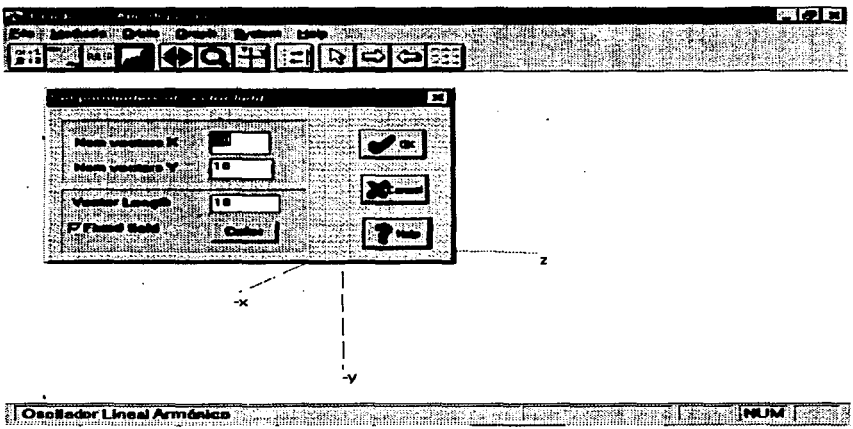


Figura 5.5: Caja de Diálogo para Fijar los Parámetros del Campo Vectorial

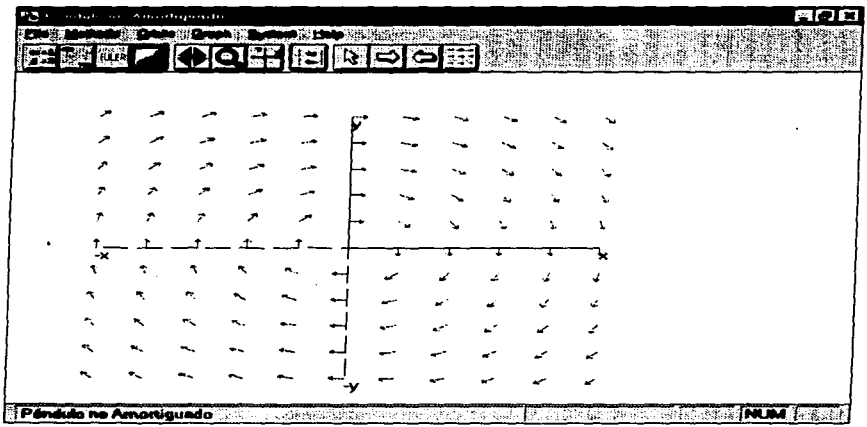


Figura 5.6: Graficación del Campo Vectorial

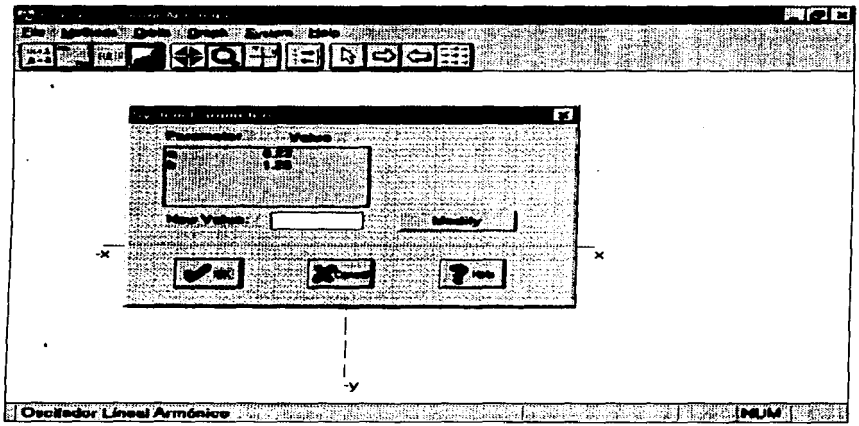


Figura 5.7: Caja de Diálogo para Modificar los Parámetros del Sistema

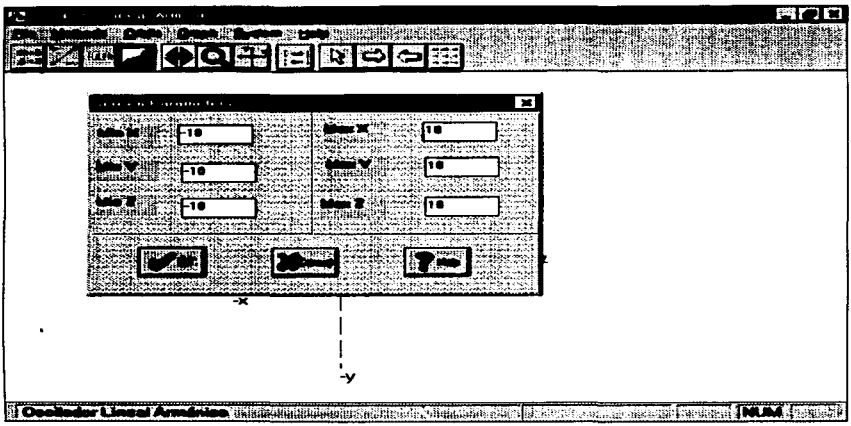


Figura 5.8: Caja de Diálogo para Establecer los Límites de los Ejes de Coordenadas

máximos de los ejes. Un ejemplo de esto está en la figura 5.8.

Parámetros de los métodos de integración

Los métodos de integración necesitan diversos parámetros. Estos se ya se trataron en el capitulo anterior. Los que *INTEGRA++* considera son el *tamaño del paso* y el *número de iteraciones* . Un ejemplo de ello se puede ver en la figura 5.9, en esta caja de diálogo también podemos capturar cada cuanto deseamos una cabeza de flecha sobre las órbitas, el método numérico y el color con que se grafica la órbita.

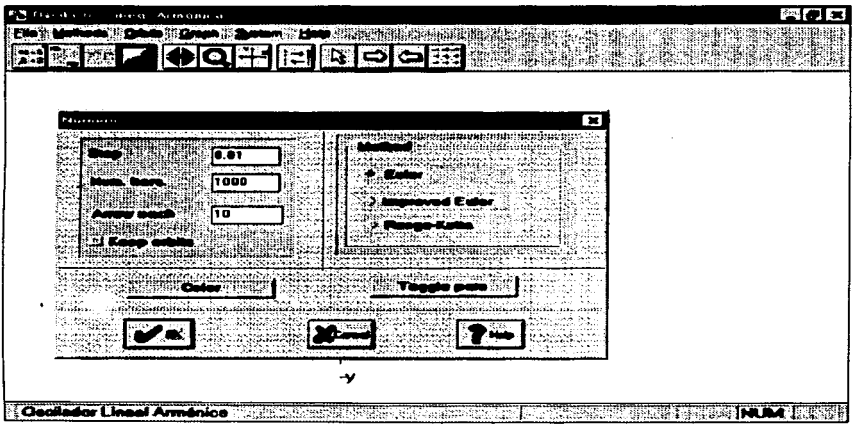


Figura 5.9: Caja de Diálogo para Fijar los Parámetros de los Métodos de Integración

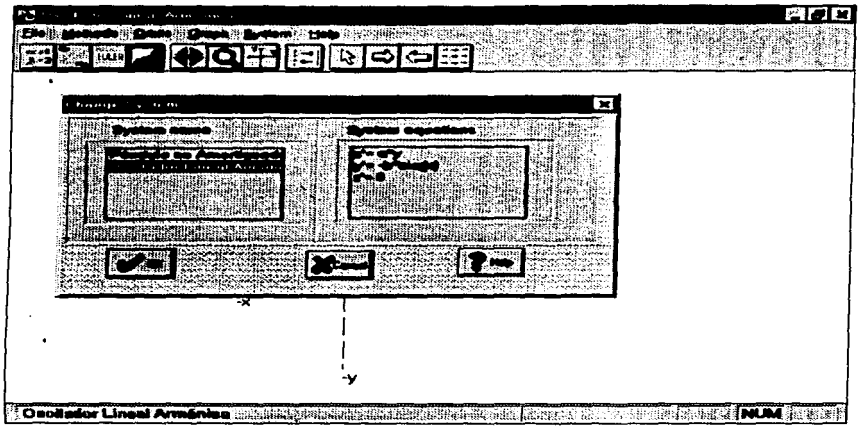


Figura 5.10: Caja de Diálogo para Cambiar de Sistema

Cambiando de sistema

Si nuestro proyecto contiene más de un sistema entonces lo que tenemos que hacer es seleccionar algún sistema ya que no podemos trabajar simultáneamente con todos. Para esto tenemos que entrar a la opción *Change System* del menú *System*. Al ingresar aquí si hay más de un sistema aparecerá una lista con los sistemas correspondientes al proyecto. En la figura 5.10 se muestra un ejemplo de esto.

Graficación de las órbitas

La graficación de las órbitas del sistema dinámico se hace activando el menú *Orbits*. *INTEGRA++* cuenta con la opción de que se grafique la órbita con tiempo positivo (*Forward*) o tiempo negativo (*Backward*). Un ejemplo de aparece en la figura 5.11 en la cual se grafican las órbitas del péndulo simple en R^2.

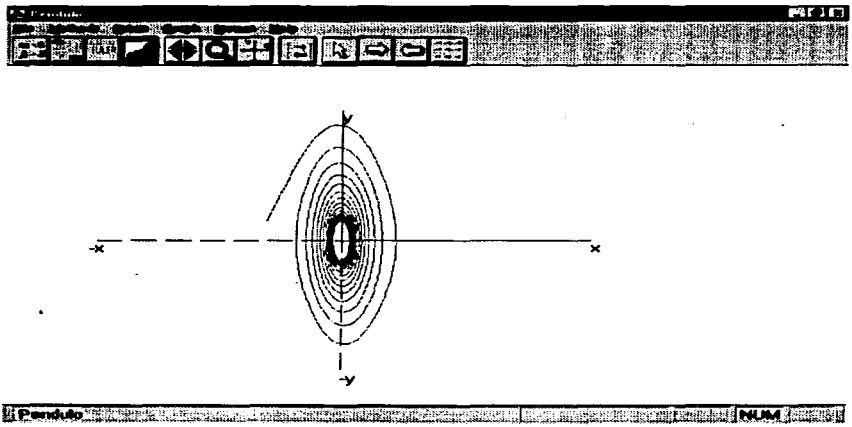


Figura 5.11: Graficación de las Orbitas

Comentarios adicionales

Para tener mayor información acerca de *INTEGRA++* e *INTERFAZ++*, se ha elaborado un manual de ayuda para el usuario que se encuentra en el apéndice F. Este manual se puede usar como complemento de este capítulo.

Conclusiones

La construcción de sistemas de software utilizando la metodología orientada a objetos fué el área principal de aprendizaje en este trabajo, además de la programación en Windows con **OWL**; sin duda estos representan dos de los mejores beneficios obtenidos. *INTEGRA++* forma parte del esfuerzo del equipo de trabajo del Laboratorio de Dinámica No Lineal, para construir herramientas de software propias, orientadas para asistir el análisis y la enseñanza de los sistemas dinámicos que se modelan con ecuaciones diferenciales. Indudablemente vale la pena construir este tipo de herramientas debido la carencia de ellas en nuestro país.

Ciertamente existen un gran número de herramientas de análisis de sistemas dinámicos, por mencionar algunas Phaser, Dynamics, ODE, etc. Pero contar con una herramienta propia tiene como ventaja poder adicionarle funcionalidad de la manera que mejor convenga, o como lo desee el usuario y evitar la espera de que se libere alguna herramienta con las características que necesita el problema que se esté trabajando.

La rapidez con que crece la industria del software es tal, que apenas concluida la cónstrucción de *INTEGRA++*, se liberó toda una serie de productos para el desarrollo rápido de aplicaciones orientadas a objetos en Windows, usando C++, por tanto la perspectiva a corto plazo de *INTEGRA++* es extenderla con C++ Builder de Borland.

Apendices

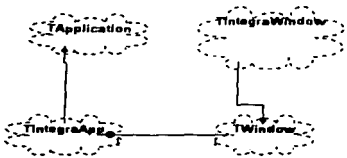
A. Código fuente de los métodos numéricos

```
// Método de Euler
void
GraphDC::Euler(float xe, float ye, float ze,
                    float n, float h,int sw)
{
 vector v;
 plane p;
 float dx,dy,dz,ax,ay,az,x1,y1,z1;
 char buffer[80];
 if( sw ){
      xe = oldx;
      ye = oldy;
      ze = oldz;
 }
 x1 = xe;
 y1 = ye;
 z1 = ze;
 primero = 1;
 while( n-- ){
      plot(xe,ye,ze,TColor::LtRed);
      primero = 0;
      dx = fnx.run(efx,xe,ye,ze );
      dy = fny.run(efy,xe,ye,ze );
      dz = fnz.run(efz,xe,ye,ze );
      ax = xe, ay = ye, az = ze;
      xe = ax+( dx*h );
      ye = ay+( dy*h );
      ze = az+( dz*h );
 }
      oldx = xe, oldy = ye, oldz = ze;
}
```

```
// Método de Euler Mejorado
void
GraphDC::improvedEuler(float xi, float yi, float zi,
                                  float n, float h , int sw)
{
     bool keep = FALSE;
     float dx,dy,dz,dx1,dy1,dz1,ax,ay,az;
     if( sw ){
          xi = oldx;
          yi = oldy;
          zi = oldz;
     }
     orbitArrow(OFF,0.0,0.0,0.0);
     primero = 1;
     while(n--){
          plot( xi, yi, zi,TColor::LtRed);
          orbitArrow(OFF, xi, yi, zi);
          dx = fnx.run(efx, xi,yi,zi );
          dy = fny.run(efy, xi,yi,zi );
          dz = fnz.run(efz, xi,yi,zi );
          dx1 = fnx.run(efx, xi+( dx*h ), yi+( dy*h ),
                                zi+(dz*h));
          dy1 = fny.run(efy, xi+( dx*h ), yi+( dy*h ),
                                zi+(dz*h));
          dz1 = fnz.run(efz, xi+( dx*h ), yi+( dy*h ),
                                zi+(dz*h));
          ax = xi, ay = yi, az = zi;
          xi = ax+( ( dx+dx1 )/2.  )  * h;
          yi = ay+( ( dy+dy1 )/2.  )  * h;
          zi = az+( ( dz+dz1 )/2.  )  * h;
          primero = 0;
     }
     oldx = xi, oldy = yi, oldz = zi;
}
// Método de Runge-Kutta
void
GraphDC::rungeKutta(float xk, float yk, float zk,
                         float n, float h, int sw)
{
     if( sw ){
          xk = oldx;
          yk = oldy;
```

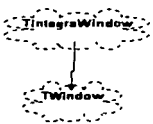
```
        zk = oldz;
    }
    orbitArrow(OFF,0.0,0.0,0.0);
    primero = 1;
    while(n--){
        plot( xk, yk, zk,TColor::LtRed);
        primero = 0;
        orbitArrow(OFF, xk, yk, zk);
        dx = fnx.run(efx, xk, yk, zk )*h;
        dy = fny.run(efy, xk, yk, zk )*h;
        dz = fnz.run(efz, xk, yk, zk )*h;
        dx1 = fnx.run(efx, xk+h/2., yk+dy/2., zk+h/2.  )*h;
        dy1 = fny.run(efy, xk+h/2., yk+dy/2., zk+h/2.  )*h;
        dz1 = fnz.run(efz, xk+h/2., yk+dy/2., zk+h/2.  )*h;
        dx2 = fnx.run(efx, xk+h/2., yk+dy1/2., zk+h/2.  )*h;
        dy2 = fny.run(efy, xk+h/2., yk+dy1/2., zk+h/2.  )*h;
        dz2 = fnz.run(efz, xk+h/2., yk+dy1/2., zk+h/2.  )*h;
        dx3 = fnx.run(efx, xk+h, yk+dy2, zk+h )*h;
        dy3 = fny.run(efy, xk+h, yk+dy2, zk+h )*h;
        dz3 = fnz.run(efz, xk+h, yk+dy2, zk+h )*h;
        ax = xk, ay = yk, az = zk;
        xk = ax + ( dx + 2.*dx1 + 2.*dx2 + dx3 )/ 6.;
        yk = ay + ( dy + 2.*dy1 + 2.*dy2 + dy3 )/ 6.;
        zk = az + ( dz + 2.*dz1 + 2.*dz2 + dz3 )/ 6.;
    }
    oldx = xk, oldy = yk, oldz = zk;
}
```

B. Diagramas de clases de *INTEGRA++*



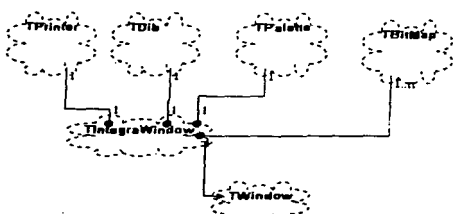
1. Diagrama de Clases de TIntegraApp

De la figura 3 a la 6, las relaciones que se describen por medio de la figura 2



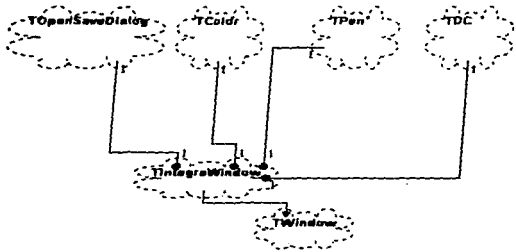
2. Diagrama base de TIntegraWindow

para la clase **TIntegraWindow**, son las mismas clases para todos los diagramas de clases. Se hizo por razones de espacio y claridad.

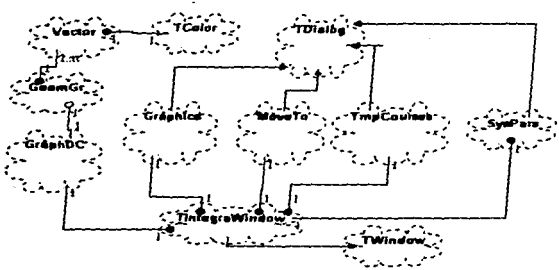


3. Diagrama de Clases de TIntegraWindow. Parte I

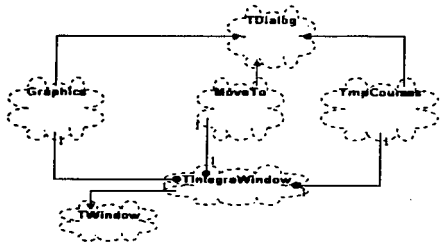
3. Diagrama de Clases de TIntegraWindow. Parte I



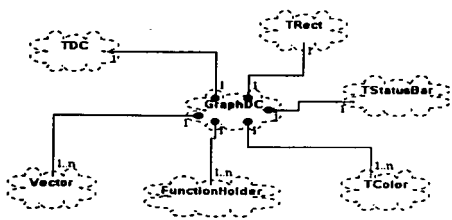
4. Diagrama de Clases de TIntegraWindow. Parte II



5. Diagrama de Clases de TIntegraWindow. Parte III



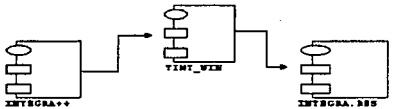
6. Diagrama de Clases de TIntegraWindow. Parte IV



7. Diagrama de Clases de GraphDC

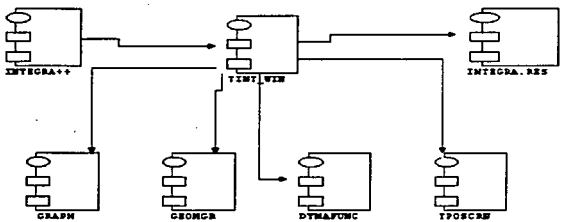
C. Diagrama de Módulos de *INTEGRA++*

En los diagramas de módulos de las figuras 2,3 y 4 se repite el siguiente diagrama

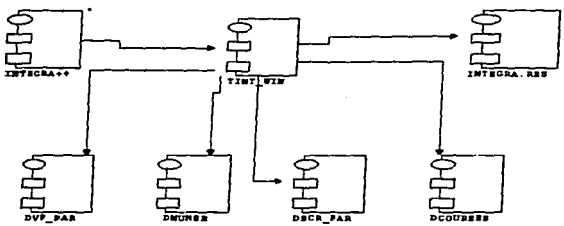


1. Diagrama de módulos base de INTEGRA++

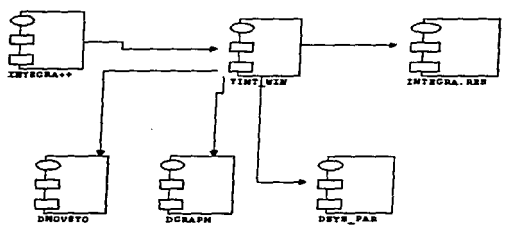
el significado es el mismo para los dos diagramas.



2. Diagrama de módulos de INTEGRA++. Parte I

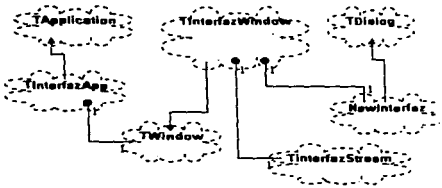


3. Diagrama de módulos de INTEGRA++. Parte II



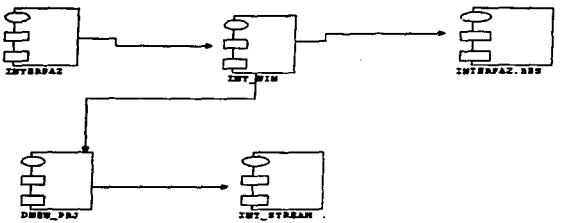
3. Diagrama de módulos de INTEGRA++. Parte III

D. Diagrama de clases de *INTERFAZ++*



1. Diagrama de clases de INTERFAZ++

E. Diagrama de módulos de *INTERFAZ++*



1. Diagrama de módulos de INTERFAZ++

F. MANUAL DEL USUARIO

INDICE DEL MANUAL DEL USUARIO

2. Control del Cursor
3. Selección de Condiciones iniciales.

II. Submenus y Cajas de Diálogo

1. Operación de una Caja de Diálogo
2. Caja de Diálogo de Systems
3. Caja de Diálogo de Parameters
4. Submenus y Caja de Diálogo de Cursor
   - 4.1. Move to
   - 4.2. Origin
   - 4.3. Step
   - 4.4. See Coordinates
5. Caja de Diálogo de Screen Parameters
6. Menus y Caja de Diálogo de los Ejes Coordenadas
   - 6.1. Projections
   - 6.2. Move Origin
   - 6.3. Grade Axis
   - 6.4. Zoom in/out
   - 6.5. Rotations
   - 6.6. Axis Name
   - 6.7. Change Color
7. Submenu y caja de dialogo de Vector Field
   - 7.1. Parameters (del campo vectorial)
   - 7.2. Vector Field
   - 7.3. Clear Screen
8. Caja de Diálogo de Numerics
   - 8.1. Method
   - 8.2. Step
   - 8.3. Num. Iters
   - 8.4. Keep Orbits
9. Acceso directo a algunas funciones de INTEGRA++.

Tercera Parte : INTERFAZ++

Primera parte: INTRODUCCIÓN

I. Componentes y Requerimientos de Software y Hardware

1. Programas que componen *INTEGRA++*

El sistema *INTEGRA++* está compuesto por los siguientes programas:

1.1. INTERFAZ.EXE

El programa *INTERFAZ++* es un módulo del sistema *INTEGRA++* que permite al usuario generar programas ejecutables que constituyen proyectos. Cada proyecto contiene una colección de sistemas de ecuaciones que pueden ser analizadas 80separada o comparativamente. Los archivos ejecutables que se generen con la *INTERFAZ++* son independientes de ella, estos, al ser ligados a los diversos módulos del sistema *INTEGRA++*, generan los programas ejecutables que pueden ser utilizados para el estudio de los sistemas dinámicos de interés.

El número máximo de sistemas de sistemas de ecuaciones diferenciales que pueden incluirse en un proyecto está limitado por la capacidad del manejador de memoria de Windows.

1.2. Biblioteca de Clases

La biblioteca de clases la conforman :

- **La biblioteca de métodos numéricos.** Es el conjunto de clases de métodos numéricos que manipulan internamente los sistemas de ecuaciones diferenciales. Esta parte del sistema no es visible al usuario por que es un encapsulamiento de la modelación de los sistemas dinámicos.

- **La biblioteca de interfaz gráfica de INTEGRA++.** Es el conjunto de funciones que permiten el manejo del escenario gráfico de INTEGRA++.

2. Requerimientos de Hardware y Software del Sistema

Esta versión de **INTEGRA++ Windows / C++ 1** ha sido desarrollada con el lenguaje de programación C++ 4.5 de *Borland*, tiene los siguientes requerimientos:

2.1. Requerimientos de hardware

- Un procesador 486 o superior con coprocesador matemático
- Monitor VGA o SVGA.
- 8 MB de RAM.
- 5 MB en disco fijo libres, para el programa y 30 MB para el compilador de Borland y sus bibliotecas de funciones.
- Mouse de dos botones

2.2. Requerimientos de software

- Sistema Operativo Windows 3.xx o 95
- Compilador de Borland C++ 4.5 o superior.

II. Descripción Panorámica de las Funciones de INTEGRA++

1. Menús, Submenús e Iconos

En *INTEGRA++* el usuario tiene acceso a un menú de opciones. Al activar alguno de los títulos del menú se despliega una caja de diálogo o, alternativamente, ofrecen un submenú. En las cajas de diálogo el usuario puede escoger opciones o introducir varios tipos de parámetros (activando subcajas de edición dentro de la caja de diálogo). Los submenús, a su vez, contienen opciones que pueden llevar a cabo alguna acción y otras que abren una caja de diálogo.

Para su comodidad el usuario de *INTEGRA++*, además de los menús, tiene acceso a íconos que permiten de una forma más directa accesar algunas de las cajas de diálogo de los menus o submenús. También exis-

ten algunos íconos que sirven para realizar una acción (de uso frecuente) de una forma inmediata.

Así tenemos dos tipos de íconos:

- Los que al presionarlos ejecutan un proceso (acción)
- Los que llaman a una caja de diálogo

1.1. Iconos de Acción


Integra en Tiempo Positivo

Al presionar este icono el sistema integrará, en tiempo positivo, tomando en cuenta el método y las condiciones iniciales actuales.


Integra en Tiempo Negativo

El sistema integrará, en tiempo negativo, tomando en cuenta el método y las condiciones iniciales actuales.


Limpia la Pantalla

Se limpiará la ventana de trabajo.


Selecciona la Proyección XY

Cambiará el escenario de visualización pasando al espacio bidimensional XY.

1.2. Iconos de Cajas de Diálogo

Al presionar este tipo de íconos se despliega una caja de diálogo (perteneciente a algún menú o submenú). Estas contienen opciones que se pueden seleccionar usando directamente el botón iz-

quierdo del mouse. En estas cajas también se pueden hacer cambios numéricos activando cajas de edición con el mouse. Una vez hechos los cambios deseados, para cerrar esta caja aceptándolos, se presiona el botón izquierdo del mouse sobre el ícono de aceptar  . Si no se desean aceptar estos cambios, presionamos el ícono de cancelar  .

Este tipo de iconos son los siguientes :

 **Seleccionar un Sistema de la Biblioteca de Sistemas**

La caja de diálogo permite escoger, en la biblioteca de sistemas, el sistema de ecuaciones con que se desea trabajar.

 **Cambiar los Parámetros del Sistema Seleccionado**

La caja de diálogo despliega y permite modificar los valores de los parámetros del sistema actual.

 **Cambiar las dimensiones de la Pantalla**

La caja permite modificar la escala de la pantalla de visualización.

 **Métodos Numéricos**

Aquí se selecciona el método numérico que se desea usar.

 **Fija las Condiciones Iniciales y controla el movimiento del cursor**

En esta caja de diálogo permite teclear (modificar numéricamente) las condiciones iniciales actuales. También permite trasladar y modificar el paso del cursor.

Escenarios Gráficos

Permite escoger el escenario gráfico de trabajo y el tipo de curvas que se desean graficar en él.

Campo Vectorial

Ofrece elementos que permiten dibujar el campo vectorial.

2. Los Menús de *INTEGRA++*

Los íconos sirven constituyen una vía de acceso directa para el usuario. *INTEGRA++* cuenta además con una colección de menús con títulos literales que sirven al usuario que no está familiarizado con el simbolismo de los íconos, para encontrar las funciones deseadas. Estos menús pueden ser accesados con el apuntador del mouse de la forma acostumbrada en Windows o, alternativamente, presionando la tecla [ALT] junto con la letra clave del título del menú. Al ser activados estos menús provocan el despliege de un submenú o de una caja de diálogo.

2.1. La Barra de Menús

Esta contiene los siguientes títulos:

- **Menú File**

  Con este menú se visualiza la descripción del proyecto, graba imagen actual, lee una imagen previamente grabada, edita archivo, visualiza archivo y termina el programa.

- **Menú Systems**

  Este permite seleccionar el sistema a trabajar y visualizar la descripción del sistema

- **Menú Parameters**

  Permite visualizar y cambiar los parámetros del sistema.

- **Menú de Auxiliary Functions**

  Permite generar, editar y gráficar las funciones auxiliares definidas por el usuario.

- **Menú Cursor**

  Ayuda a posicionar el cursor dentro de los ejes coordenados moviendo a una posición indicada por el valor X, Y, Z, mover el cursor al origen, modificar el tamaño del paso del cursor al utilizar las flechas, así como ver o ocultar las coordenadas dentro del área de mensajes.

- **Menú Screen**

  Modifica las dimensiones de la pantalla de trabajo, y redibuja la pantalla de trabajo.

- **Menú Graphs**

  Permite seleccionar el escenario de graficación, las proyecciones a graficar, mover el origen, graduar los ejes coordenados, acercar o alejar los ejes coordenados, rotaciones de los ejes y cambio de color de estos.

- **Menú Vector Field**

  Permite dibujar el campo vectorial en dos dimensiones, fijar la longitud del vector a visualizar dentro del campo vectorial, el número de vectores por cada eje en el campo vectorial, el plano en el que se visualizará el campo vectorial, color de la proyección a graficar y limpiar la pantalla de trabajo.

- **Menú Temporal Courses**

  Gráfica los cursos temporales de las funciones componentes del sistema.

- **Menú Numerics**

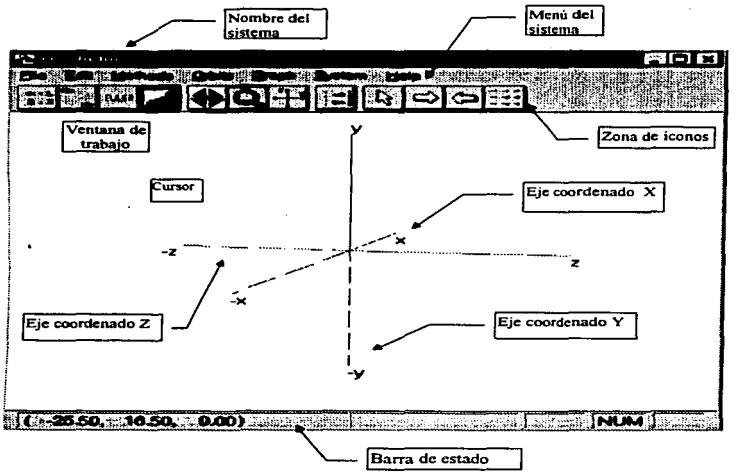
  Selecciona el método de integración, el paso de integración, el número de iteraciones por cada integración, y permite guardar en memoria el resultado del cálculo de las órbitas.

Segunda parte: OPERACIÓN DE *INTEGRA++*

I. El Escenario Gráfico de Entrada

1. La Pantalla de Trabajo

Al correr un programa ejecutable, generado con el programa INTERFAZ++, de entrada, aparecerá la siguiente pantalla:



Esta nos ubica en un escenario (espacio de estados) tridimensional. El sistema activo será el que aparece en primer lugar en la biblioteca de

sistemas. En este espacio se podrán visualizar órbitas o proyecciones del sistema dinámico.

En ésta pantalla se pueden distinguir los siguientes elementos:

1.1. Nombre del Sistema

Aparece el nombre del sistema activo aparece en la barra de título, encabezando la parte superior de la pantalla.

1.2. La barra de menús

Una colección de menús se muestra en la parte superior de la pantalla. Estos se puede accesar oprimiendo el botón izquierdo del mouse sobre ellos o usando las teclas **[Alt] + [Letra indicada]**.

1.3. La barra de iconos

Una colección de íconos se muestra tambbién en la parte superior de la pantalla, debajo de la barra de menús. Estos se activan al presionar el botón izquierdo del mouse sobre ellos. Al posicionar el apuntador del Mouse sobre el ícono, la descripción de su función aparecerá desplegada en el área de mensajes.

1.4. El área de mensajes

El área de mensajes está localizada en la parte inferior de la pantalla. Como hemos dicho, al posicionar el mouse sobre alguno de los íconos en esta área se deplegará una explicación de su función, también en ella aparecerá un reporte de actividad del sistema.

1.5. Los ejes coordenados

El escenario tridimensional tiene tres ejes de coordenadas; cada uno de ellos tiene un color diferente, el segmento punteado de ca-

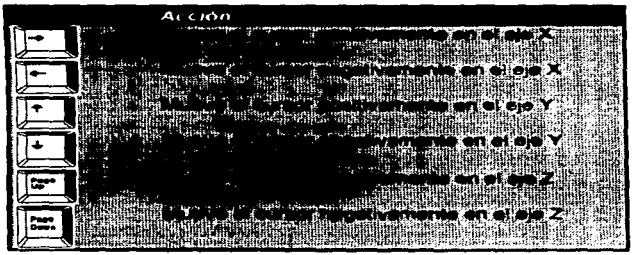
da uno corresponde a su parte negativa, también se visualizará el nombre (x,y o z) de cada uno de los tres ejes de coordenadas.

1.6. El cursor

Dentro de este espacio se encuentra una pequeña flecha de color magenta, que llamamos el cursor y con el cual se pueden escoger visualmente las condiciones iniciales para cada órbita que se desea graficar, desplazándolo con las teclas de movimiento (ver la siguiente sección).

2. Control del Cursor

El cursor puede ser desplazado en la pantalla con las siguientes teclas:



3. Selección de Condiciones Iniciales

Las condiciones iniciales pueden seleccionarse visualmente con el cursor. Habiéndolo desplazado a la posición deseada al realizar una integración, esta se llevará a cabo tomando como condiciones iniciales las que señala la

punta del cursor. Cada vez que el cursor se desplace el sistema actualiza las condiciones iniciales.

También éstas pueden seleccionarse de manera precisa (numéricamente hablando) posicionando al cursor en un punto de coordenadas (X, Y, Z). Para esto se recurre a la función que permite este tipo de desplazamiento del cursor (ver ícono de cursor).

II. Submenús y Cajas de Diálogo

Como explicamos anteriormente los títulos del menú de *INTEGRA++*, al activarse, abren un submenú o una caja de diálogo. A continuación explicaremos la forma de trabajar con cada uno de los correspondientes submenús o cajas de diálogo.

1. Operación de un Caja de Diálogo

Toda caja de diálogo tiene un ícono de aceptar. Para aceptar los cambios realizados, presionamos el botón izquierdo del mouse sobre este ícono.

Para cancelar los cambios se presiona el botón izquierdo del mouse sobre este ícono.

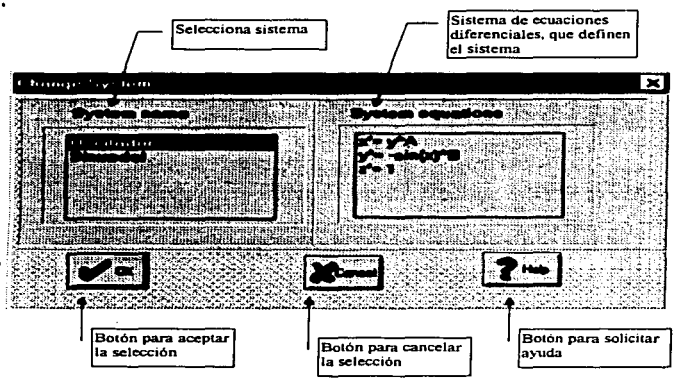
Icono de Ayuda

- En una caja de diálogo se puede uno desplazar a través de las diferentes opciones usando la tecla **[Tab]** , o alternativamente presionando el botón izquierdo del mouse sobre el área correspondiente. Si nos hemos posicionado con la tecla **[Tab]** sobre una opción que tiene caja de edición, inmediatamente se pueden alimentar los correspondientes datos. Si nos posicionamos sobre ella con el mouse entonces se activará al presionar sobre ella el boton izquierdo.

2. Caja de Diálogo de Systems.

La caja de diálogo que despliega **Systems** permite seleccionar un sistema dinamico de la Biblioteca de Sistemas y visualizar las ecuaciones que

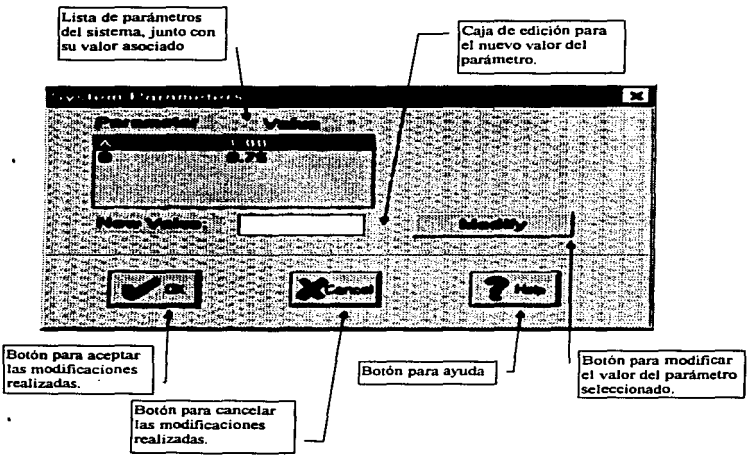
lo definen.



Para seleccionar un sistema dinámico señalamos con el botón izquierdo del mouse el área del título del sistema y presionamos sobre el ícono de aceptar, para realizar la selección.

3. Caja de Diálogo de Parameters

Esta permite leer y cambiar los parámetros del sistema.



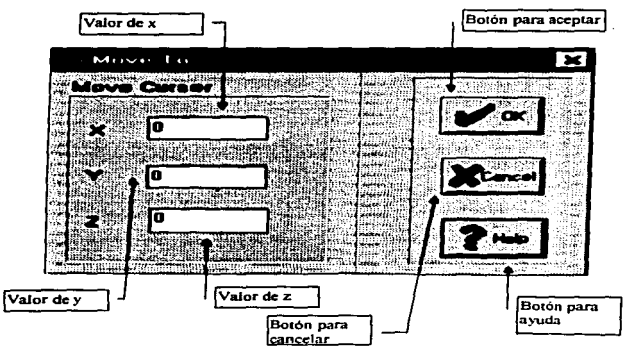
En esta caja de diálogo aparece la lista de parámetros definidos para el sistema dinámico activo.

4. Submenú y Caja de Diálogo de Cursor

Este título del menú abre el siguiente submenú.

4.1. Move to...

Al seleccionar esta opción aparece la siguiente caja:



En ella se muestran los valores de las coordenadas de la actual condición inicial, teclee los valores para X, Y, y Z.

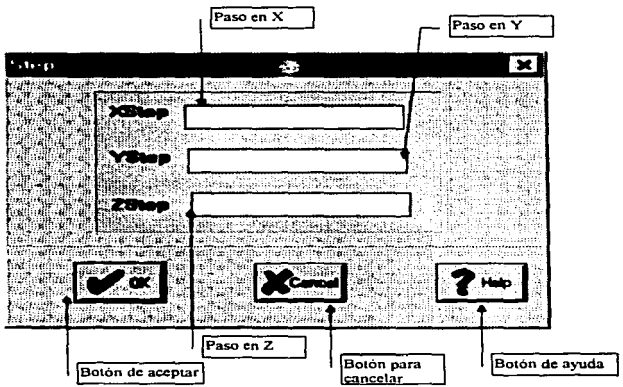
X es la caja de edición para el nuevo valor de la coordenada X.
Y es la caja de edición para el nuevo valor de la coordenada Y.
Z es la caja de edición para el nuevo valor de la coordenada Z

4.2. Origin

Al seleccionar esta opción manda el cursor de condiciones iniciales al origen de los ejes coordenados.

4.3. Step

Al seleccionar esta opción aparece la siguiente caja de diálogo.



En ella se muestran los valores del paso del cursor para los ejes X, Y, Z.

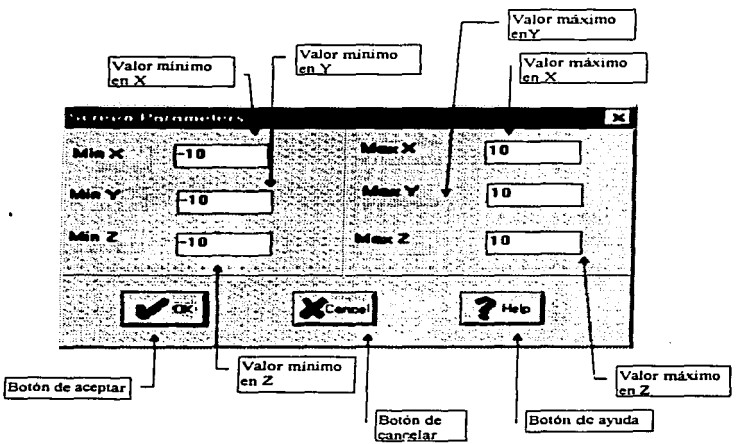
XStep: Caja de edición para el valor del paso de la coordenada X.
YStep: Caja de edición para el valor del paso de la coordenada Y.
ZStep: Caja de edición para el valor del paso de la coordenada Z.

4.4. See Coordinates

Por omisión se visualizaran las coordenadas del cursor en el área de mensajes, si no desea visualizar éstas, presione la tecla **[Enter]** o el botón izquierdo del Mouse sobre esta opción. Al hacer esto se desplegará la acción contraria en este renglón del submenú, reportando así el estado actual de esta opción.

5. Caja de Diálogo de Screen Parameters

Sirve para modificar las dimensiones de la pantalla de trabajo y redibujarla.



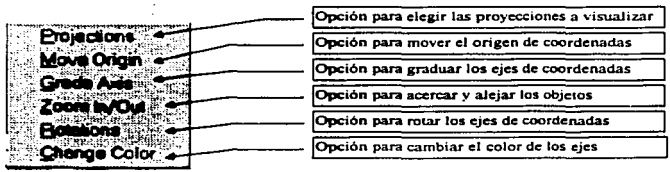
En ella se muestran los valores actuales (mínimo y máximo) de cada uno de los ejes coordenadas.

MinX. Valor mínimo del eje X
MaxX. Valor máximo del eje X
MinY. Valor mínimo del eje Y
MaxY. Valor máximo del eje Y
MinZ. Valor mínimo del eje Z
MaxZ. Valor máximo del eje Z

6. Menú y Caja de Diálogo Graphs

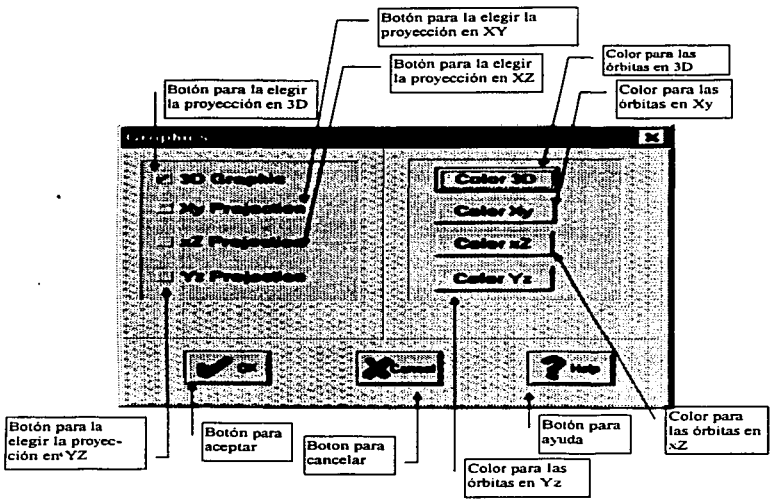
Permite seleccionar las proyecciones a visualizar (y escoger su color), mover el origen del eje coordenado, graduar los ejes coordenados, acercar o alejar los ejes coordenados, hacer rotaciones de los ejes y cambiar su color.

Para ello se ofrece el siguiente submenú:



6.1. Projections

Al seleccionar esta opción aparece la caja de diálogo:



En ella se seleccionan a los ejes coordenados a visualizar, así como también las proyecciones a visualizar al gráficar las órbitas al integrar, por omisión están activos los ejes tridimensionales y la proyección tridimensional.

6.2. Mover el origen

Al seleccionar esta opción mueve el origen de los ejes coordena-

dos usando las teclas :

Para terminar presione la tecla **[Esc]**.

6.3. Grade Axis

Al seleccionar esta opción define la graduación de los ejes coordenados, estos permanecerán graduados hasta que se indique lo contrario. Por omisión no estarán graduados los ejes coordenados.

6.4. Zoom In/Out

Al seleccionar esta opción permite acercar o aleja la posición del observador con respecto al origen del sistema coordenado, use la tecla **[+]** para acercar el origen y **[-]** para alejar de origen. Para terminar presione la tecla **[Esc]**.

6.5. Rotations

Al seleccionar esta opción permite rotar los ejes coordenados (utilizando coordenadas esféricas: *Teta, Fi, Ro*). Use las teclas :

para rotar el ángulo *Teta*;

para rotar el ángulo *Fi*;

para rotar el ángulo *Ro*.

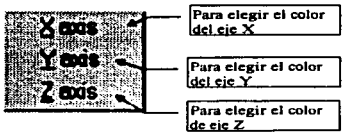
6.6. Axis Name

Por omisión se visualizaran el nombre de los ejes coordenados, si no desea visualizarlos presione la tecla **[Enter]** sobre esta opción o el botón izquierdo del Mouse sobre esta. La opción muestra también el estado de visualizar ejes coordenados.

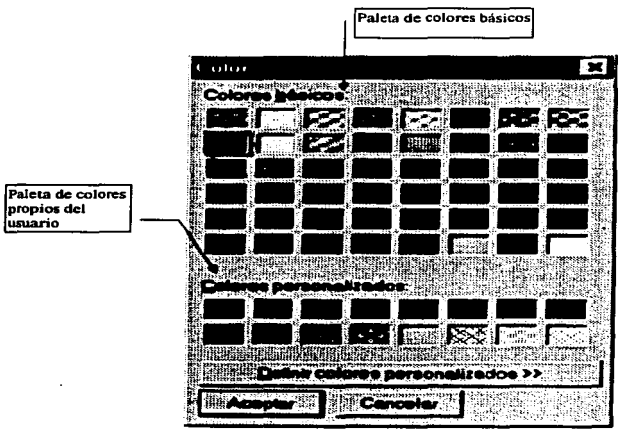
6.7. Change Color

Cambio de color de los ejes coordenados

Al seleccionar esta opción aparece un menú para elegir el eje al que se desea cambiar su color:



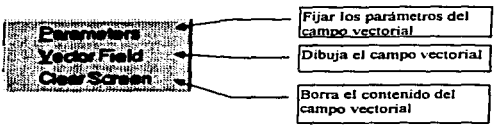
Al elegir una opción aparecerá la siguiente paleta de colores, para elegir el color deseado.



Seleccione presionado el botón izquierdo del mouse sobre el color que desea asignar al eje. También se puede generar un color propio (personalizado).

7. Submenú y Caja de Diálogo de Vector Field

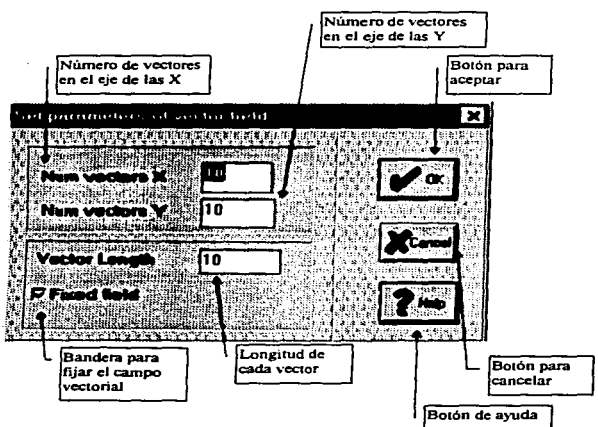
Permite dibujar el campo vectorial en dos dimensiones, así como el plano en que se visualizará, fijar la longitud y el número de los vectores a dibujar por cada eje y limpiar la pantalla de trabajo.



A continuación se describirá cada una de las opciones del menú.

7.1. Parameters (del campo vectorial)

Esta opción permite modificar los parámetros del campo vectorial. Al elegir ésta opción aparece la siguiente caja de diálogo :



- **Vector Lenght**
  Establece la longitud de los vectores a dibujar en el campo vectorial, teclee en nuevo valor y presione **[Enter]** o el ícono de aceptar.

- **Num Vectors X**
  Campo para establecer el número de vectores en el eje de las X, teclee en nuevo valor y presione **[Enter]** o el ícono de aceptar.

- **Num Vectors Y**
  Campo para establecer el número de vectores en el eje de las Y, teclee en nuevo valor y presione **[Enter]** o el ícono de aceptar.

- **Fixed Lenght**
  Bandera para establecer si todos los vectores del campo tienen una longitud fija.

7.2. Vector Field

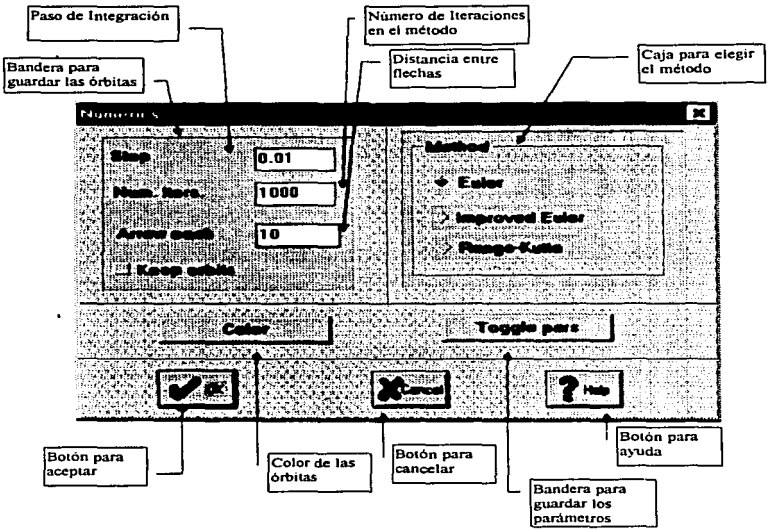
Dibuja el campo vectorial tomando en cuenta los par'amatros previamente definidos. En caso de que la función no esté definida en donde se evalúa el campo vectorial, marcará un error en el programa.

7.3. Clear Screen

Limpia y redibuja la pantalla de trabajo.

8. Caja de Diálogo de Numerics

Esta caja de diálogo permite seleccionar el método numérico para realizar la integración de las órbitas (en tiempo positivo o negativo), el número de iteraciones por integración, cambiar el paso de integración y seleccionar el guardar en memoria el resultado del cálculo de las órbitas.



8.1. Method

En esta opción se muestran los métodos de integración habilitados dentro de *INTEGRA++*, seleccione el que usted requiera.

8.2. Step

Este campo muestra el actual valor del paso de integración, modifique este valor tomando en cuenta el método numérico y la canti-

dad de iteraciones a realizar.

8.3. Num. Iters.

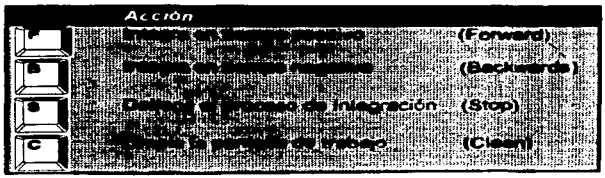
Este campo muestra una ventana con el valor actual de el numero de iteraciones por integración (positiva o negativa) que van a ser realizadas.

8.4. Keep Orbits

Por omisión no se guardarán en memoria las órbitas calculadas al integrar (en tiempo positivo a negativo), si desea guardar estás presione la tecla de la barra espaciadora o el botón izquierdo del Mouse sobre está opción. Se reporta el estado de la opción.

9. Acceso directo a algunas funciones de *INTEGRA++*

Por comodidad algunas funciones de uso frecuente pueden ser accesadas directamente oprimiendo una tecla especifica del teclado.



Tercera Parte : INTERFAZ++

I. ¿Qué es INTERFAZ++ ?

El sistema *INTEGRA++* permite al usuario crear programas que contengan una biblioteca de sistemas dinámicos, para trabajar con ellos individual o comparativamente. Cada uno de estos programas constituye lo que llamamos un *proyecto*. Así, cada proyecto contiene un conjunto de sistemas dinámicos.

INTERFAZ++ es un módulo del sistema *INTEGRA++* por medio del cual el usuario define y crea los proyectos de su interés. Para esto, la interfaz permite :

i) Capturar el conjunto de sistemas de ecuaciones diferenciales que definen el proyecto.
ii) Generar un programa ejecutable en el ambiente Windows.

INTERFAZ++ se encarga de enlazar otros módulos de INTEGRA++ (como la biblioteca de métodos numéricos y el módulo gráfico) con la librería de funciones matemáticas del compilador de Borland C y la interfaz de programación de Aplicaciones de Windows. Realizando esto, la *INTERFAZ++* genera código fuente en lenguaje C++, que es ejecutable en el ambiente Windows.

II. Componentes de INTERFAZ++

- **Interfaz de captura**

Esta parte de la interfaz que se encarga del despliege de una caja de diálogo para permitir la captura de todos los elementos que componen el grupo de sistemas dinámicos (proyecto).

- **Generador de código ejecutable para Windows.**

El generador de código es un analizador léxico que permite el reconocimiento de las funciones matemáticas. Además de la inclusión de las librerías pertinentes para que el programa se ejecute propiamente en Windows. Se encarga de optimiza el tamaño del código objeto generado y la velocidad de ejecución de las instrucciones.

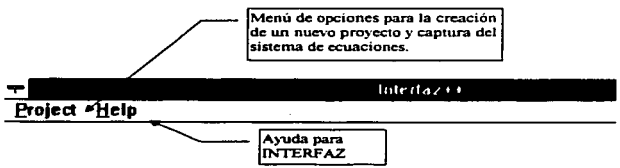
III. Como Ejecutar INTERFAZ++

Para ejecutar INTERFAZ++, realice los siguiente pasos :

1. Elija la ventana principal de Windows y posicionese en la opción de Ejecutar
2. Busque el programa INTERFAZ.EXE
3. Elíjalo y presione dos veces el botón izquierdo del mouse.

IV. Ventana de Entrada

Al entrar al programa nos aparece una pantalla que tiene el formato estándar de una ventana interactiva de Windows. Esta sirve para capturar el conjunto de ecuaciones y generar el código ejecutable (en el ambiente Windows).

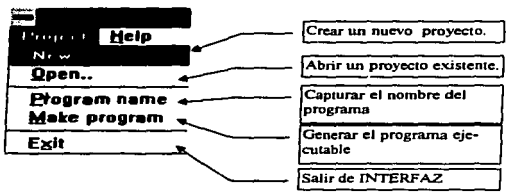


La barra superior de esta ventana tiene el aspecto que muestra la figura. En esta barra aparecen dos títulos de menús: **Project** y **Help**. Como en todo programa de Windows, al oprimir el botón izquierdo del mouse sobre el título se desplegará su correspondiente menú. Estos menús se manejan de la forma acostumbrada en Windows: cuando se presiona sobre el título de alguna opción del menú el botón izquierdo del mouse, se realiza inmediatamente una acción asignada para esa opción o alternativamente, se despliega un submenú o una caja de diálogo que requiere la alimentación de información adicional.

Como es costumbre, el título **Help** ofrece ayuda básica para el manejo de INTERFAZ++. Esta ayuda puede también obtenerse directamente oprimiendo las teclas **Alt+H**.

V. Project

Esta opción de la barra de la ventana de entrada abre el menú siguiente:



1. Descripción Panorámica de Project

New.. Esta opción permite la edición de un nuevo proyecto. Captura la descripción del proyecto, los sistemas de ecuaciones diferenciales, los parámetros y sus valores. Como una opción adicional genera el programa ejecutable.

Open.. Esta opción abre un proyecto existente y lo edita en la misma forma que la opción **New**.

Make program Esta opción genera el programa ejecutable. Una nota importante es que el usuario debe tener un subdirectorio en la unidad C : llamado c :/bc45, el cuál contendrá el compilador de Borland C++.

Program name Esta opción captura el nombre del proyecto. La extensión debe ser CPP.

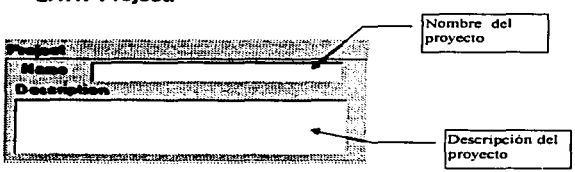
Esta opción finaliza la ejecución del programa INTERFAZ++.

2. Descripción Detallada de Project

2.1. New..

Al elegir esta opción, en seguida aparecerá una ventana de tipo caja de diálogo. Se trata de una caja grande que contiene a su vez otras cajas más pequeñas (subcajas). Estas subcajas tienen un título propio y contienen algunos campos de edición. A continuación explicaremos el uso de cada una de estas subcajas:

2.1.1. Project.



En esta ventana se capturarán el nombre del proyecto y su descripción.

- **Nombre del proyecto**

Este nombre tiene que teclearse necesariamente con la extensión **CPP** a razón de que el compilador de Borland C++ necesita como entrada un programa en C++ y cuya extensión es CPP. El nombre del programa ejecutable que se creará será el mismo que se intro-

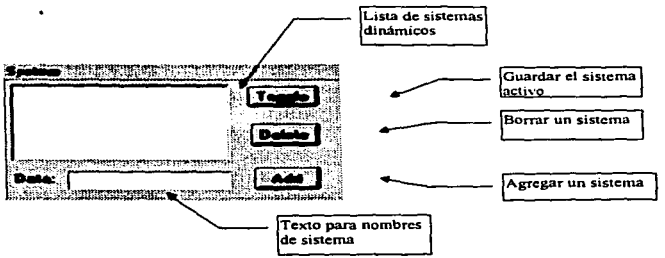
duzca aquí (excepto por la extensión **EXE**). Este ejecutable es código para Windows. El nombre del proyecto deberá tener una longitud máxima de 8 caracteres, empezando con una letra y sólo se permite usar los siguientes:

- ***Descripción del proyecto***

  Esta es la información que el usuario desee anexar al grupo de sistemas dinámicos que integran el proyecto. Por ejemplo notas, algunos resultados importantes acerca de estos sistemas, teoremas, bibliografía, etc..

2.1.2 System

Aquí se capturarán los nombres de los sistemas dinámicos. Estos aparecen en la forma de una lista de texto. Su ventana es la siguiente :



- **Lista de sistemas dinámicos**

  Esta es el conjunto de sistemas dinámicos que conforman el proyecto. El número de sistemas dinámicos en el proyecto sólo depende de la cantidad de memoria disponible que proporcione el manejador de memoria de Windows.

- **Caja de edición de texto**

  Esta caja de edición de texto, sirve para capturar el nombre de un sistema dentro del proyecto.

- **Guardar el sistema activo**

  Este botón sirve para guardar la información asociada con el sistema actual que se esta capturando.

- **Borrar un sistema**

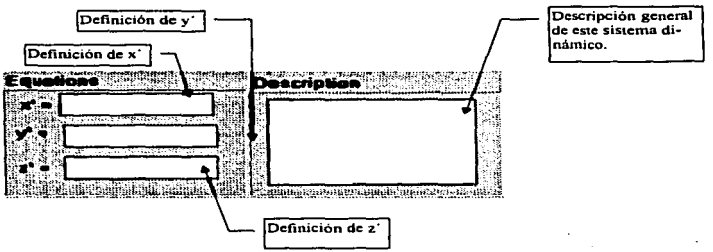
  Para borrar un sistema de la lista de sistemas dinámicos, se presiona este icono. El resultado de esta operación es que borra el nombre del sistema seleccionado y la información asociada a él.

- **Agregar un sistema**

  Para agregar un sistema a la lista de sistemas dinámicos, se presiona este icono. El resultado de esta operación es que agrega el nombre del sistema seleccionado y la información asociada a él.

2.1.3. Equations

Este grupo permite la captura del sistema de tres ecuaciones y su descripción. Estas ecuaciones son las que definen la dinámica del sistema que se desea estudiar. Su ventana es la siguiente :



- **Definición de x´.**

  Aquí se introduce el miembro derecho de la primera ecuación diferencial.

- **Definición de y´**

  Aquí se introduce el miembro derecho de la segunda ecuación diferencial.

- **Definición de z´**

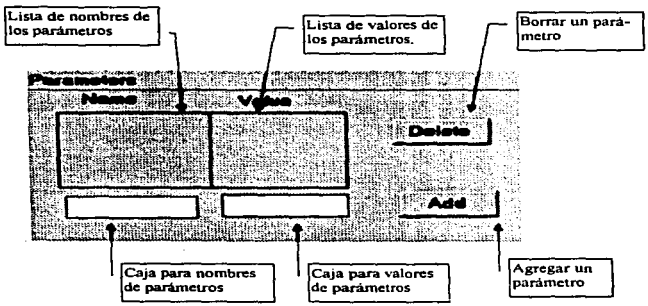
  Aquí se introduce el miembro derecho de la tercera ecuación diferencial.

- **Descripción general**

  En esta caja de edición de texto se captura información asociada a este sistema dinámico en particular.

2.1.4. Parameters

Si el sistema dinámico hacen uso de algunos parámetros estos también se deben capturar en el programa de INTERFAZ++, así como sus valores de entrada (default). Para esto tenemos la ventana siguiente :



- **Lista de Nombres de Parámetros.**

  Esta lista contiene los nombres de los parámetros. Estos parámetros serán reconocidos después por el Analizador Léxico de *INTEGRA++*.

- **Lista de Valores de Parámetros**

  Esta lista esta asociada con los nombres de los parámetros en una relación de uno a uno y en el mismo orden.

- **Caja de introducción de los Nombres de los Parámetros**

  En esta caja de edición de texto, se capturará el nombre de algún parámetro.

- **Caja de introducción de los Valores de Parámetros**

  En esta caja de edición de texto, se capturará el valor asociado con el parámetro que esta en la caja de nombres de parámetros.

- **Borrar Parámetro**

  Borra el parámetro seleccionado con el botón izquierdo del mouse, de la lista de parámetros y además elimina el valor asociado a él.

- **Agregar Parámetro**

  Agrega el parámetro actual en la caja de edición de nombres de parámetros y además agrega el valor asociado a él en la caja de edición de valores.

2.1.5. Generar el Archivo Ejecutable

Para generar el archivo ejecutable con las especificaciones dadas en todas las ventanas anteriores, solo tiene que presionarse el botón izquierdo del mouse sobre el icono  .

2.2 Open..

Este menú abre una caja de diálogo que contiene la biblioteca de proyectos existentes (con extensiones CPP) y permite:

- La creación de ejecutables a partir de ellos;
- La edición de un archivo de proyecto de la misma forma usando una caja de diálogo igual a la del menú **New**.

La caja de diálogo que abre es la siguiente :

La forma de abrir un archivo es la standard en cualquier otra aplicación de Windows.

Al abrir uno de estos archivos para ser editado, aparece la misma caja de dialogo de la opción **New** (previamente descrita).

2.3 **Program name** Edita el nombre del proyecto.

2.4 **Make program** Genera el proyecto.

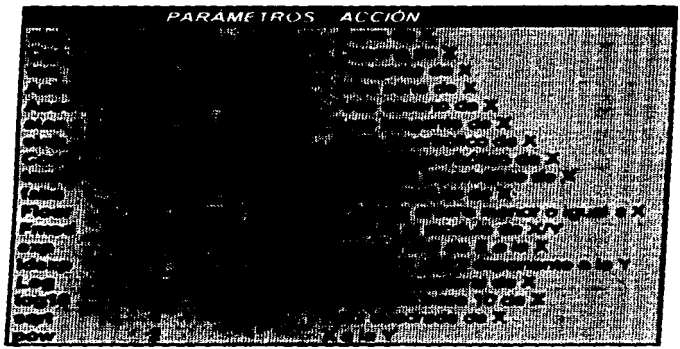
VI. Help

Este menú ofrece ayuda para el uso de INTERFAZ++.

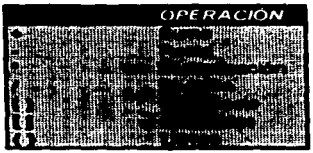
APÉNDICE DEL MANUAL DEL USUARIO

FUNCIONES MATEMÁTICAS

Las funciones matemáticas validas para definir las ecuaciones diferenciales y las funciones auxiliares son las siguientes:



Y las operaciones elementales son:



También se tienen definidas las siguientes constantes:

Bibliografía

[1] I. S. Lowry, *Short course in model design.* American Institute of Planners Journal. May 1965, pp. 158-66.

[2] A. Korzybski, *Science and Sanity.* Science Press, New York, 1941

[3] Richard Veriard, *Information Modelling.* Prentice Hall,,1992.

[4] George Wilkie, *Object-Oriented Software Engineering.* Addison-Wesley, 1992

[5] Setrag Khoshafian, Razmik Abnous, *Object Orientation: Concepts, Languages, Databases, User Interfaces.* John Wiley & Sons,1990.

[6] Chris Partridge, *Modelling the real world: Are classes abstractions or objects?.* Nov - Dic 1994, Pag 39-45, Journal Object Oriented Programming

[7] Grady Booch, *Object-Oriented Analysis and Design with Applications.* Second Edition, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, E.U.A., 1994.

[8] Koshafian, S. and Copeland, G. November 1986. *Object Identity.* SIGPLAN Notices Vol. 21 (11), p. 406

[9] Rumbaugh, J. April 1988. *Relational Database Design Using an Object-Oriented Methodology.* Communications of ACM vol. 31(4), p. 417.

[10] Micallef, J. April/May 1988. *Encapsulation, Reusability, and Extensibility in Ob ject-Oriented Programming Languages.* Journal of Object-Oriented Programming vol. 1(1), p.15.

[11] Stroupstrup, B. November 1987. *Possible Directions for C++.* Proceedings of de USENIX C++ Workshop. Santa Fe, NM, p.14.

[12] Stroupstrup, B. 1988. *Parameterized types for C++.* Proceedings of USENIX C++ Conference. Berkeley, CA: USENIX Association, p.1.

[13] Jenkins, M. and Glasgow, J. January 1986. *Programming Styles in Nial.* IEEE Soft ware vol. 3(1), p.48.

[14] Bobrow, D. and Stefik, M. February 1986. *Perspectives on Artificial Intelligence Programming Science.* vol. 231, p. 251.

[15] Dahl, O., Dijkstra, E., and Hoare, C. A. R. 1972. *Structured Programming. London.* England: Academic, Press, p. 83

[16] Shaw, M. October 1984. *Abstraction Techniques in Modern Programming Languages.* IEEE Software vol. 1(4) p.10

[17] Meyer, B. 1988. *Object-Oriented Software Construction.* New York, NY: Prentice Hall.

[18] Coad, P. Summer 1989. *OOA: Object-Oriented Analysis.* Austin Texas: Object In ternational

[19] Rumbaugh, J. et al 1991. *Object-Oriented Modeling and Design.* Prentice Hall

[20] Coad, P., Yourdon, E., 1991, *Object-Oriented Design,* Yourdon Press, Prentice Hall.

[21] Martin, R. C., 1995. *Designing Object-Oriented C++ Applications Using the Booch Method,* Prentice Hall.

[22] Zilles, S. 1984. *Types, Algebras and Modeling, in On Conceptual Modeling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*. New York, NY, Springer-Verlag, p. 442.

[23] As quoted in Harland, D., Szypiewsky, M., and Wainwright, J. October 1985. *An Alternative View of Polymorphism*. SIGPLAN Notices vol. 20(10).