Universidad Nacional Autónoma de México

Facultad de Ingeniería

Diseño de un Sistema Paramétrico para la Generación de Ejes de Transmisión en AML

TESIS

que para obtener el título de:

Ingeniero en Computación

Presenta:

Colín Arciniega Juan José

Director de Tesis: M.I. Álvaro Ayala Ruiz

México, D.F. 2004

<u>Al Distinguido Jurado:</u>

Presidente: *M.I. Lauro Santiago Cruz.*

Vocal: *M.I. Álvaro Ayala Ruiz.*

Secretario: *Ing. Víctor Damián Pinilla Morán.*

1er Suplente: *Ing. Rosario Barragán Paz.*

2do Suplente: *Dr. Vicente Borja Ramírez.*

Con gran respeto y admiración por su ayuda profesional.

Dedico este trabajo a las personas mas importantes de mi vida:

A Dios
Por haberme puesto en este maravilloso camino, darme fuerza, voluntad y decisión para seguir adelante, aun en los tiempos difíciles. Gracias por crear este mundo, que aun con defectos, es un lugar hermoso, maravilloso e ideal para vivir. Gracias por haberme permitido cumplir este sueño.

A mi madre, Guadalupe Arciniega Fernández.
Gracias por todos los años de apoyo, comprensión y cariño. Gracias por todo lo que has sacrificado para que yo pudiera llegar hasta este momento, por creer en mi, en mis sueños. Se que muchas veces te he decepcionado, pero quiero que este momento sea un nuevo comienzo para nosotros. Por favor no olvides que siempre te traigo en mi corazón.

A mi hermana, Guadalupe Oriente Colin Arciniega.
Gracias por todo el apoyo y cariño que me has dado en estos años. Te deseo todo el éxito del mundo y le doy gracias a Dios que por fin, estamos mas juntos que nunca.

A mis padrinos Jose G. García Estrada y Rosalina Caballero de García.
Gracias por la amistad y el cariño, que ni el tiempo ni las circunstancias han podido romper. Su ayuda en los tiempos difíciles permitió que pudiéramos salir adelante. Nunca lo olvidare.

A Banco de México y en especial a Alma Rosa Suaste Guzman.
Gracias por el apoyo y la experiencia adquirida durante todos estos años. Sin todo esto, no hubiera podido haber llegado a este momento importante en mi vida.

A Víctor Manuel Fernández Patiño.
Gracias por tu amistad, por todo lo que me has enseñado y por todo lo demás. Le doy gracias a Dios por haberte puesto en mi camino. Gracias por existir.

A la Lic. Elsa Amalia Castellanos López.
Gracias por haber sido la mejor jefa, guía y amiga de mi mamá. Por haber compartido su experiencia, sus vivencias, consejos, sabiduría durante el tiempo que trabajaron juntas. Gracias a usted, mi madre pudo entender muchas cosas. Un abrazo con mucho respeto y admiración.

Al Lic. Lauro Garfias.
Gracias por tu amistad y sabiduría. Un abrazo con gran admiración y respeto.

A Francisco Velásquez.
Gracias por tu amistad.. Un abrazo con gran admiración y respeto.

A mis tías Maria, Francisca , a mi primo Pedro y a mis primas Maribel, *Lilían y Jazmín.*
Gracias por el apoyo y la ayuda que nos han dado a mi familia y a mi todos estos años, sobretodo en los tiempos difíciles.

Y con gran admiración, menciono a los personajes que han tenido una gran influencia en mi vida y han sido mi inspiración, gracias a su trabajo y dedicación:
Ing. Salvador Rosas, Ing. Alberto Fuentes Maya, Ing. Rafael Flores, Ing. Betzabe, Isaac Newton, Albert Einstein, Leonardo DaVinci, Ing. Carlos Slim, Edward DeVere, J.R.R. Tolkien, Stan Lee, Jack Kirby, Steve Dikto, John L. Byrne, Mel Blanc, George Lucas, Steven Spielberg, Christopher Reeve, John Williams, Paul David "Bono" Hewson, David "The Edge" Evans, Cherilyn "Cher" Sarkisian LaPiere.

TEMARIO

INTRODUCCIÓN

El diseño de producto, manufactura, planeación de procesos, inspección, modelado y análisis de elemento finito han sido realizados independientemente como parte de un proceso requerido para el ciclo de diseño-producción de partes. En los últimos 15 años, los sistemas de computadora han sido usados en la automatización de diseño de producto, mientras que el diseño y planeación de procesos de manufactura ha permanecido como un esfuerzo manual con poca ó casi nada automatizados. En el área de modelado de elemento finito (FEA , Finite Element Analisis (Análisis de Elemento Finito)), un gran número de aplicaciones han sido desarrolladas para ayudar con éstos problemas. Como resultado, existen soluciones independientes para el diseño (CAD), análisis (FEA), y manufactura (principalmente maquinado). Estas aplicaciones han sido desarrolladas y están disponibles, aunque existen esfuerzos adicionales que se han enfocado en integrar y automatizar completamente el ciclo diseño-manufactura en un ambiente de Ingeniería Concurrente.

Debido a la falta de integración de las actuales técnicas, no proveen la interpretación directa de las geometrías de las partes. Para superar esta deficiencia, varias herramientas se han enfocado en ayudar al usuario en remodelar la parte para crear diferentes representaciones como se requiera para cada proceso. Las aplicaciones de manufactura e inspección requieren del usuario para rediseñar en términos de características de manufactura e inspección para automatización de procesos, en lugar de directamente interpretar el diseño para inspección y manufacturabilidad. Por lo que, los sistemas CAD / CAM / CAE actuales son herramientas de diseño aisladas. Ya que para realizar los diferentes procesos de diseño, manufactura y producción, es necesario manipular la geometría manualmente en cada aplicación CAD / CAM / CAE[I].

Para proveer soporte computacional a las actividades particulares del ciclo de vida de producción, es necesario contar con un conjunto de aplicaciones de ingeniería y herramientas de software, basándose en los modelos del producto y de manufactura, que trabajen en forma concurrente y utilice el paradigma de programación orientada a objetos, como AML[II] en el manejo de partes [Technosoft, 2003].

El trabajo propuesto tiene como objetivo: Desarrollar un sistema que asista al proceso de diseño utilizando el paradigma de programación orientada a objetos utilizando el lenguaje de programación AML. AML es un lenguaje de programación orientado a objetos que cuenta con soporte CAD. Con este lenguaje, se propone el desarrollo de una interfaz gráfica capaz de modelar piezas de revolución con información proveniente de una base de datos orientada a objetos (OODB[III]) administrada por el sistema SADET[IV] *(ver sección 3.2.)*.

El sistema desarrollado permitirá instanciar modelos de ejes de transmisión, cambiar de posición elementos del eje, adicionar características secundarias a los objetos primitivos que lo conforman, y generar archivos de salida con nueva información que podrán ser alimentados al sistema SADET. Se busca demostrar que utilizando AML es posible realizar aplicaciones de propósito particular, que simplifique el proceso de diseño mecánico.

Para cumplir con el objetivo de este trabajo de tesis, en el capítulo 1 se estudiará sobre la teoría de los sistemas CAD/CAM y se mencionarán los procesos de diseño y manufactura, así como el concepto de Ingeniería Concurrente. También se definirán las entidades geométricas con los que cuentan los sistemas CAD, así como su arquitectura, como manipulan y almacenan datos.

En el capítulo 2 se obtendrán antecedentes de ingeniería de software, para programación secuencial y para programación orientada a objetos, así como antecedentes de el paradigma de programación orientada a

[I] CAD / CAM / CAE: Diseño, Manufactura e Ingeniería Asistida por Computadora.
[II] AML: Lenguaje de Modelado Adaptativo (Adaptative Modeling Language).
[III] OODB: Object Oriented Data Base.
[IV] SADET: Sistema Auxiliar para el Diseño de Ejes de Transmisión.

objetos. Todos éstos antecedentes servirán para poder realizar un diseño de software con calidad y apegado a los estándares internacionales. También se describirá el lenguaje de programación AML.

En el capítulo 3 se definirán los requerimientos y especificaciones del sistema a desarrollar en el lenguaje de programación AML. Entre éstos se definen las clases, métodos y las relaciones de las clases que conformarán el sistema. También se hará uso de los diagramas de secuencia y de casos de uso para analizar el comportamiento del sistema ante el usuario.

En el capítulo 4 se describirá el proceso para la implementación del sistema utilizando el lenguaje de programación AML. Se analizara partes de código que conforma la interfaz gráfica y la aplicación. También se presentara el manual de operación para usuarios.

En el capítulo 5 se definirá el caso de estudio que se utilizará para demostrar la funcionalidad de la herramienta desarrollada en AML. Como caso de estudio, se utilizarán los datos obtenidos de la base de datos del sistema SADET *(ver sección 3.2.)*, para generar un eje de transmisión, será manipulado, modelado, se le aplicaran características secundarias y se generara un archivo de salida con datos de la geometría modelada.

Capítulo 1.- Antecedentes CAD - CAM

1.1 Introducción.

En este capítulo se obtendrá una vista general de los sistemas CAD / CAM y de la teoría en la cual éstos sistemas se basan. Se analizará la teoría para representar figuras de 2 ó 3 dimensiones y la representación paramétrica de la geometría. Se estudiara la geometría de sólidos. Se definirán las entidades geométricas manejadas por los sistemas CAD/CAM.

También analizaremos los sistemas coordenados, los modelos de alambre, la teoría de modelado de sólidos y definiremos a los sólidos simples primitivos y las operaciones booleanas que se pueden realizar entre ellos. Detallaremos la arquitectura de un sistema CAD, la forma en que almacenan y manipulan los datos, así como la estructura de los datos.

Además se podrán apreciar diferencias entre conceptos como "modelos de un diseño de procesos" y "modelos de diseño". Se definirán los conceptos básicos en los que la "Ingeniería Concurrente" se basa para optimizar ó reducir los tiempos de diseño de productos.

1.2 Diseño asistido por computadora.

Modelado y Comunicación.

El concepto de un diseñador trabajando con *modelos de diseño* es fundamental para trabajar con ***diseño asistido por computadora*** **(Computer-Aided Design, CAD)**. Es importante distinguir entre modelos de un diseño de procesos y modelos de diseño. A través del proceso de diseño, diseño es un abstracto, ya que el artefacto físico no existe hasta que es construido ó manufacturado. Por lo cual es necesario que existan modelos del diseño involucrados en el artefacto para evaluarlos, manipularlos y refinarlos. Tomiyama y sus colaboradores, en 1989, sugieren que dichos modelos pueden existir como diferentes representaciones [Chris McMahon, 1998]. La geometría modelada de un componente de ingeniería puede ser representado en diferentes formas. Si el diseño es muy simple, puede haber ideas volando en la cabeza del diseñador, por lo cual, puede ser necesario una representación más formal.

Los modelos de diseño tienen una gran cantidad de usos, pero el más básico, es que son usados por el diseñador para grabar y manipular ideas, y proveer la base para la evaluación del diseño. El proceso del diseño muy rara vez es tomado por un sólo diseñador, por lo cual, los modelos desempeñan un papel importante en la ***comunicación*** del diseño entre los participantes del proceso, manufactura, desarrollo y subsiguiente uso del producto.

Modelando Usando CAD.

Para apoyar el desarrollo del diseño, los diseñadores construyen una serie de modelos con varios aspectos del modelo usando un número de técnicas de representación. Secuencial o concurrentemente, otros involucrados en la evaluación del diseño y en la manufactura del producto extraen información de éstos modelos y, en el proceso, forman nuevos modelos para ayudarse en su trabajo. La **figura 1.3** muestra el desarrollo de un proceso en Ingeniería Concurrente:

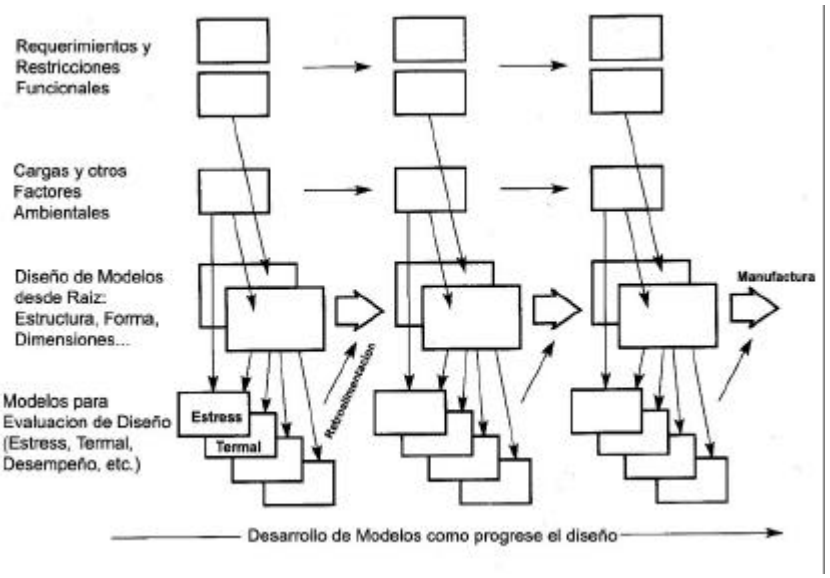


Figura 1. 1.- El uso de modelos en diseño.

La **figura 1.1** muestra modelos de la información requerida para la manufactura del producto. En paralelo, la siguiente información esta siendo desarrollada:

- Modelos de lo funcional y otros requerimientos del cliente para el diseño, porque pueden cambiar durante el proceso de diseño.
- Modelos de restricciones impuestas en el diseño, por ejemplo por la disponibilidad de materiales y procesos de manufactura.
- Modelos de cargas impuestas en el diseño.
- Modelos usados para evaluar el desempeño del diseño

El objetivo de CAD es aplicar las computadoras al modelado y a la comunicación de diseños. Esto se ha realizado en dos formas:

- En un nivel básico, se usan las computadoras para automatizar o asistir en tales tareas como la producción de dibujos ó diagramas y la generación de listas de partes en el diseño.
- En un nivel avanzado, se provee de nuevas técnicas que dan al diseñador amplias facilidades para asistir en el proceso de diseño.

1.3 Manufactura asistida por computadora.

Aplicación de Modelos de Diseño.

La información recibida gracias a los modelos, puede ser clasificada en dos grupos:

1) **Acciones de Evaluación:** Llevadas a comprobar las propiedades del diseño.
2) **Acciones Generativas:** Generan información desde el modelo, usualmente para progresar en su manufactura.

En cualquiera de los dos casos de acción involucrados en la extracción de información de la representación del diseño, y la combinación de esta con nueva información, pueden generar un nuevo modelo.

1.4 Ingeniería Concurrente.

Las descripciones de diseños, son usualmente llamados ***modelos de diseños de procesos***. En la **Figura 1.2**, el modelo mostrado es el propuesto por Pahl y Beitz en 1984 [Chris McMahon, 1998]. En este modelo el proceso de diseño es descrito por un diagrama de flujo que comprende cuatro fases principales que son:

- **Clarificación del Objetivo:** Involucra recolectar información acerca de los requerimientos del diseño y las restricciones del diseño, describiendo esto en una especificación.
- **Diseño Conceptual:** Involucra el establecimiento de las funciones a ser incluidas en el diseño, y la identificación y desarrollo de soluciones aceptables.
- **Construcción del Diseño:** En donde la solución conceptual es desarrollada en más detalle, los problemas son resueltos y los aspectos débiles son eliminados.
- **Diseño Detallado:** En donde las dimensiones, tolerancias, materiales y una forma de componentes individuales del diseño son especificados en detalle por manufactura subsiguiente.

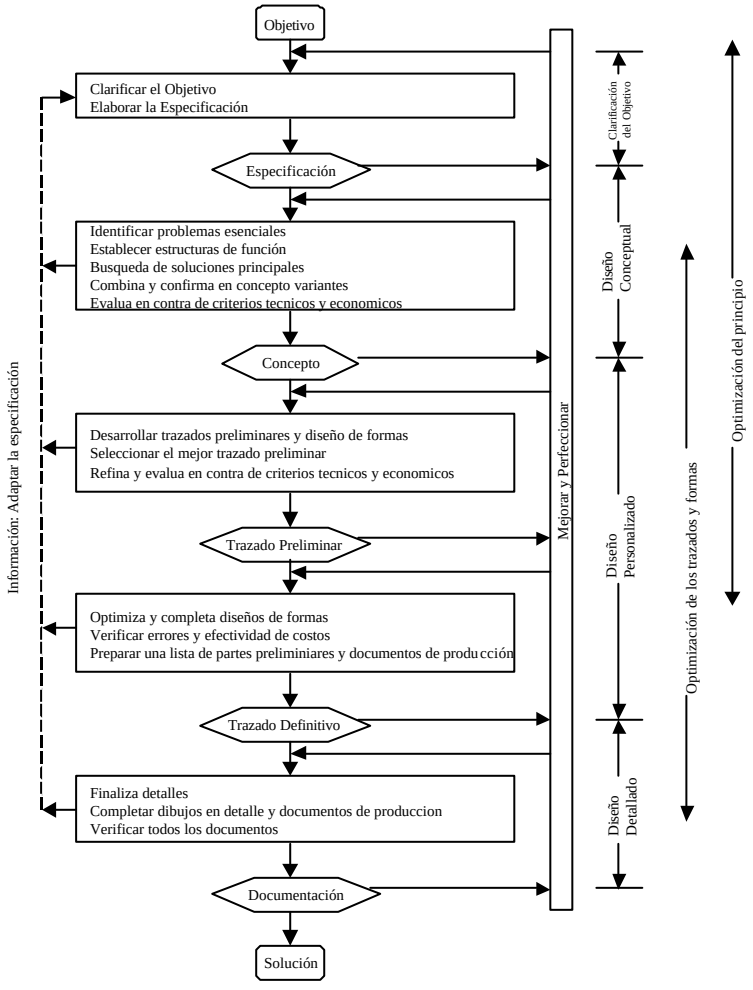


Figura 1. 2.- Pasos para el diseño de proceso de acuerdo con Pahl y Beitz (1984).

En la **figura 1.3**, el segundo modelo viene del trabajo de Ohsuga (1989). Ohsuga describe diseño como una serie de estados, en este caso, progresando desde requerimientos a través de diseño conceptual y diseño preliminar hasta el diseño detallado [Chris McMahon, 1998].

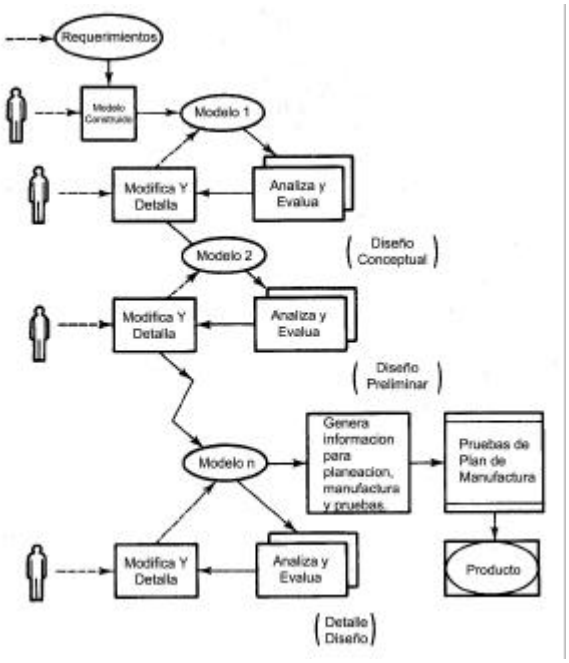


Figura 1. 3.- El proceso de diseño de acuerdo a Ohsuga.

Cada uno de éstos dos modelos de diseño presentados siguen una visión tradicional en la cual, existen una secuencia de etapas de diseño, seguidas por manufactura. El incremento en la presión para reducir el tiempo de diseño del producto esta llevando a las compañías a la ***Ingeniería Simultanea*** ó ***Ingeniería Concurrente***. **(Ver Figura 1.4)**.

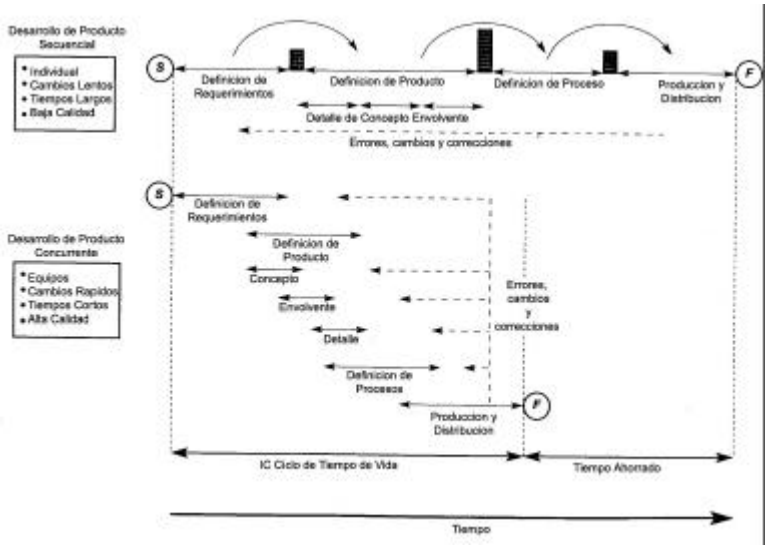


Figura 1. 4.- Desarrollo de producto, secuencial contra concurrente, del principio "S", hasta el final "F".

1.5 Integración de los sistemas CAD / CAM.

La Representación de una Figura usando Dibujos.

La técnica para representar figuras de tres dimensiones en un espacio de dos dimensiones por medio de dibujos de ingeniería (en papel ó en la pantalla de la computadora), es formalmente conocido como ***Geometría Descriptiva.*** Las figuras de tres dimensiones son representadas en dos dimensiones al trasladar puntos de un objeto hacia múltiples planos de proyección perpendiculares usando proyecciones paralelas que son normales a los planos de proyección.

Técnicas para Modelado Geométrico: La Representación Paramétrica de la Geometría.

Las representaciones paramétricas de la geometría involucran esencialmente relaciones de expresiones para los puntos coordenados *x*, *y* y (si es apropiado) *z* en una curva o superficie o en un sólido, no en términos de cada uno, sino de una o más variables independientes conocidas como ***parámetros***. Para una curva, sólo se usa un sólo parámetro: *x*, *y* y *z*, cada una expresada en términos de una sola variable, *u*. Para una superficie, dos parámetros, típicamente *u* y *v* son usados, y *x*, *y* y *z* son funciones. Para sólidos tres parámetros, *u*, *v* y *w* son usados **(Ver Figura 1.5)**.

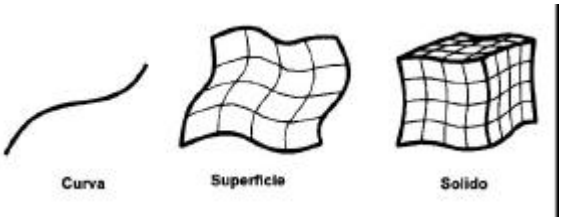


Figura 1. 5.- Curva parámetrica, superficie y sólido.

Curvas Polinomiales Cúbicas Paramétricas.

Al modelar en tres dimensiones una representación geométrica es necesario que esta describa curvas no-planas, pero además debe evitar dificultades computacionales y ondulaciones no deseadas que puedan ser introducidas por curvas polinómicas de orden superior. Éstos requerimientos son cumplidos por un polinomio cúbico (el orden más bajo del polinomio puede ser descrito como una curva no plana). Como dos puntos pueden ser únicos por una línea, y tres puntos por un arco ó un circulo, cuatro puntos pueden proporcionar las condiciones necesarias para un polinomio cúbico. La construcción de curvas a través de puntos es conocida como la Interpolación de Lagrange. Una curva cúbica también puede ser definida por dos puntos y dos vectores de condiciones en los puntos. Esto se conoce como la Interpolación de Hermite **(Ver Figura 1.6)**.

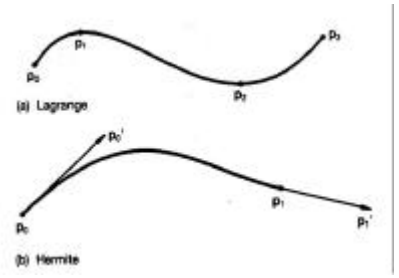


Figura 1. 6.- Curvas de interpolación de Lagrange e interpolación de Hermite.

Geometría de sólido Constructiva.

En la geometría de sólido constructiva (GSC), debemos recordar que involucra la construcción de un modelo por la combinación de primitivos geométricos como cilindros, bloques rectangulares y otros, un gráfico dirigido es otra vez usado para la estructura de datos para el modelo. En este caso, el gráfico es de un tipo en particular conocido como ***Árbol Binario***, en el cual, los nodos están conectados por ramas a un nodo raíz. Cada nodo sólo puede tener un nodo "padre" y dos nodos "hijos". El nodo raíz no tiene nodo padre y los nodos sin hijos son conocidos como nodos hojas. En un modelo GSC, las hojas representan primitivos geométricos, y el nodo raíz y los nodos intermedios componen las operaciones booleanas que construyen el modelo **(Ver Figura 1.7)**.

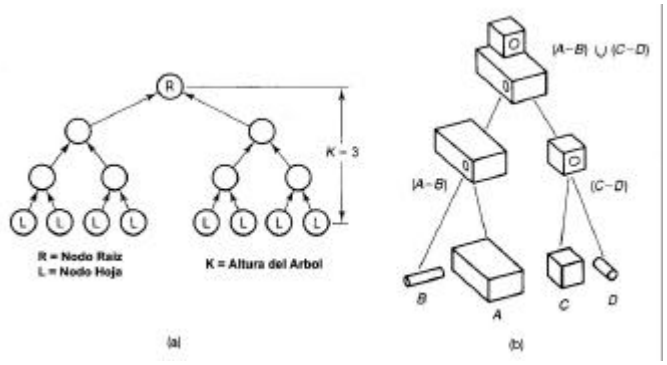


Figura 1. 7.- Árbol binario (a), Modelo GSC (b).

Los primitivos pueden ser definidos por un número de diferentes formas. En algunos sistemas pueden ser sólidos de fronteras, pero en otros casos pueden ser derivados de intersecciones de primitivos simples conocidos como ***Medios Espacios***.

Otras Técnicas de Modelado.

La técnica de ***Primitiva Pura Instanciada*** es la más simple de las técnicas, e involucra describir modelos al variar las dimensiones de primitivas sencillas invocadas desde una librería. Esta técnica puede ser aplicada a formas dentro de familias de partes que son geométricamente y topológicamente, pero no dimensionalmente. Esta es una herramienta de diseño con valor limitado.

En la técnica de ***Descomposición de Celdas***, el modelo es descrito por el arreglo de un número pequeño de formas elementales que son unidas sin interceptarse. Aunque esta técnica no es muy usada en el modelado geométrico, esta asociada a métodos de subdivisión de espacios de dos y tres dimensiones, conocidos respectivamente como subdivisión en ***cuadrantes*** y ***octantes***. Estas técnicas involucran subdivisiones sucesivas ó recursivas de una región en formas cuadradas (cuadrante) y cúbicas (octante) **(Ver Figura 1.8)**.

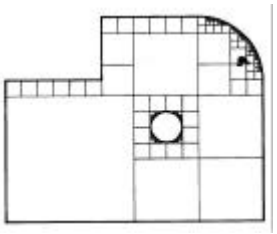


Figura 1. 8.- Ejemplo de subdivisión cuadrática.

1.6 Entidades Geométricas.

Para representar la geometría, el dibujo será una colección de puntos, líneas, arcos, secciones cónicas y otras curvas (elementos geométricos sencillos que frecuentemente serán llamados ***entidades***), arreglados en un plano bi-dimensional. Algunos ejemplos de los tipos de entidades geométricas que están disponibles en los sistemas CAD se muestran en la **figura 1.9**. Estas entidades normalmente estarán definidas por el sistema en términos de valores numéricos por su punto de coordenada ó otro dato.

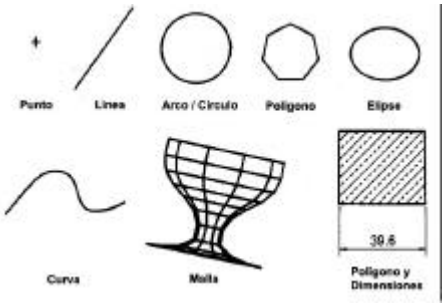


Figura 1. 9.-Entidades geométricas disponibles en un sistema CAD.

1.6.1 Sistemas Coordenados.

Se ha visto que los dibujos están construidos en un sistema de coordenadas de dos dimensiones. Los modelos en tres dimensiones son construidos en espacios de tres dimensiones (3-D), típicamente en un sistema de coordenadas cartesianas basado en la mano derecha **(Ver Figura 1.10)**. Normalmente existirá un sistema de coordenadas arreglado que será usado para la definición del modelo en general, que es llamado ***sistema de coordenadas global*** (SCG). Adicionalmente, un ***sistema de coordenadas de trabajo*** movible (SCT), que puede ser usado para ayudar en la construcción del modelo **(Ver Figura 1.11)**.

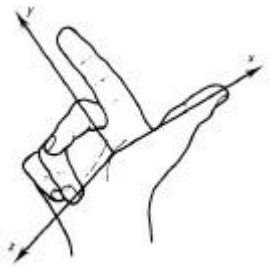


Figura 1. 10.- El sistema coordenado de la mano derecha.

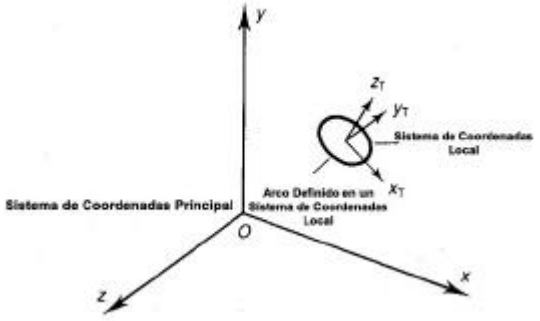


Figura 1. 11.- Uso de un sistema coordenado local.

Cuando se dibuja manualmente, el tamaño de la representación esta restringido por el tamaño físico de la hoja de dibujo, por lo que artefactos o tamaños diferentes son acomodados cambiando la escala del dibujo. Usando sistemas CAD, tales restricciones no existen.

El modelo es construido por una serie de procedimientos computacionales que generan curvas en un sistema de coordenadas de dos dimensiones que esta limitado sólo por restricciones del tamaño de números que pueden ser almacenados y manipulados por la computadora. Por consecuencia, en CAD, los dibujos podrían ser construidos a tamaño real.

1.6.2 Modelos de Alambre.

Un modelo de alambre de un objeto es el más simple modelo geométrico que puede ser usado para representar matemáticamente en la computadora. Muchas veces es referenciado como la representación de contornos de un objeto. Típicamente este modelo consiste enteramente de puntos, líneas, arcos, círculos, conos y curvas. Esta es la técnica más comúnmente empleada en todos los sistemas comerciales CAD/CAD **(Ver Figura 1.12)**.

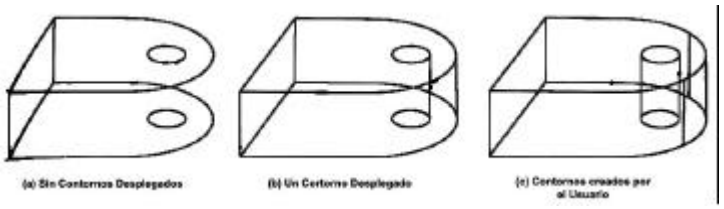


Figura 1.12.- Despliegue de orificios y curvas finales en un modelo de alambre.

1.6.3 Modelado de Sólidos.

El modelado de sólidos es una extensión natural de el uso esencial de entidades de "una dimensión" (curvas) ó entidades de "dos dimensiones" (superficies), para el modelado de figuras usando sólidos de tres dimensiones. **Woodwark,** en 1986 [Chris McMahon, 1998], propuso que las reglas exitosas para representar sólidos debería de ser:

- completo y sin ambigüedades;
- apropiado para el mundo de objetos de ingeniería;
- practico para usar con computadoras existentes.

Muchos métodos han sido propuéstos para el modelado de sólidos, en los cuales ninguno ha sido enteramente satisfactorio. Pero dos métodos han sido parcialmente exitosos, y han venido a dominar el desarrollo de sistemas prácticos. Éstos son las estrategias constructivas, en donde la más ampliamente aplicada variante es el método de ***Geometría de Sólido Constructiva*** (también conocido como método Booleano), y también, el método de ***Representación de Fronteras*** que domina las aplicaciones CAD hoy en día.

<u>Geometría de Sólido Constructiva.</u>

En el método de Geometría de Sólido Constructiva, los modelos son construidos usando combinaciones de sólidos simples ***primitivos***, como los cuboides, cilindros, esferas, conos y otros **(Ver Figura 1.13)**.

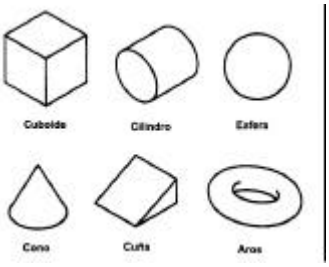


Figura 1. 13.- Primitivos disponibles en un sistema de modelado de sólidos.

Por ejemplo, dados dos objetos sólidos A y B, entonces la ***Unión*** comprende todos los puntos en el espacio que están contenidos en A OR B ($A \cup B$). El operador ***Intersección*** comprende todos los puntos que están contenidos en A AND B ($A \cap B$). Existen dos resultados de operaciones diferentes entre los dos objetos: los puntos contenidos en A AND NOT B ($A \cap \overline{B}$, también escrito $A - B$), y los puntos contenidos en B AND NOT A ($B \cap \overline{A}$, también escrito $B - A$) **(Ver Figura 1.14)**.

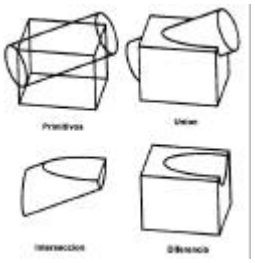


Figura 1. 14.- Operaciones booleanas en un bloque y cilindro.

Los resultados de estas operaciones son sólidos compuéstos, que pueden ser combinados con otros primitivos ó sólidos compuéstos para crear figuras adicionales. Para crear los modelos de la **figura 1.15**, cuatro figuras primitivas fueron requeridas (dos bloques rectangulares y dos cilindros).

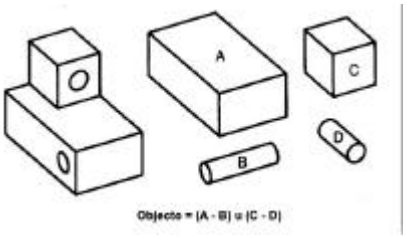


Figura 1. 15.- Construcción de un modelo sólido usando bloques y cilindros simples.

Éstos modelos tienen la ventaja de ser muy compactos y están garantizados para modelar sólidos validos sin ambigüedad.

Representación de Fronteras.

Los modelos de superficies no contienen información acerca de las conexiones entre superficies, ni que parte de un objeto es sólido. Si información es agregada acerca de la conectividad entre superficies (que serán llamadas ***caras***), además el lado del sólido de cualquier cara es identificado, entonces esto forma los elementos de la segunda estrategia del modelado del sólido principal, la estrategia de representación de fronteras **(Ver Figura 1.16)**. Los sistemas reales van más allá de esto, e incorporan métodos para verificar la consistencia topológica de los modelos, además verifica que los modelos no presenten anomalías geométricas.

La consistencia geométrica es alcanzada al asegurar que el modelo define la frontera de un objeto sólido "razonable", como lo nota Mäntylä en 1988 [Chris McMahon, 1998], donde:

- Las caras de los modelos no se interceptan una con tras excepto en vértices comunes ó bordes.
- Las fronteras de las caras son simples ***ciclos*** de bordes que no se interceptan entre ellos.
- El juego de caras del modelo se cierran para completar la piel del modelo sin partes perdidas.

La tercera condición prohíbe las figuras abiertas. Las primeras dos prohíben la auto-intersección de objetos.

La forma más simple de modelo de frontera es uno que representa todas las caras como planos ó ***facetas***. Una superficie curveada, como un cilindro, es representada en un modelo como una serie de facetas que se aproximan a la superficie. Tal representación, conocida como modelo ***poligonal***, es computablemente relativo, y por lo tanto tiene ventajas de rendimiento, que lo hacen ampliamente usado en la visualización de paquetes, juegos de video, simuladores de vuelo y otros.

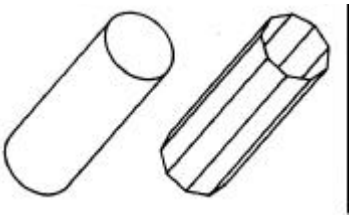


Figura 1. 16.- Representación de un cilindro mediante el uso de caras.

Esta estrategia es claramente limitada en la extensión de modelar figuras "reales", como componentes de ingeniería. En contraste, la información almacenada de modelos de frontera que hace referencia a caras y bordes, es almacenada en una forma ***evaluada***. Esto tiene ventajas de desempeño, porque la información para ciertas aplicaciones del modelo pueden ser extraída directamente de la estructura de datos. Tales aplicaciones incluyen la generación de imágenes del modelo con propósitos de despliegue, y el cálculo del área de la superficie de modelos, simplemente sumando las áreas de las superficies de cada cara. La desventaja de esta representación es que la cantidad de información almacenada es relativamente larga, por lo que éstos modelos tienden a requerir archivos de datos muy largos.

Modelos No -Múltiples.

El modelado de modelos no-múltiples, que es opuesto al modelado de representación de fronteras, es en el cual el espacio es dividido sin ambigüedad hacia un sólido y el espacio por las fronteras por sólidos no-múltiples. Como se muestra en la **figura 1.17**:

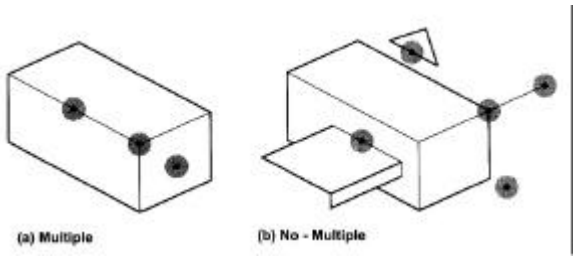


Figura 1. 17.- Modelos múltiples y no – múltiples.

En la **figura 1.17(a)** se muestra un modelo múltiple, en el cual cada punto de la superficie del objeto esta rodeada por un disco de superficie. En la **figura 1.17(b)**, se muestra un modelo no-múltiple, en el cual existen puntos con una cara vecina (en el contorno del triangulo separado), con tres caras vecinas (donde la placa en forma de "L" se junta con el bloque principal), y sin caras vecinas (en la línea desconectada conocida como contorno colgante).

1.7 Arquitectura de un sistema CAD.

Hasta ahora, un sistema CAD ha sido descrito en términos generales. Más específicamente, puede estar formado por:

- **Hardware:** La computadora y el equipo periférico asociado;
- **Software:** Los programas de computadora ejecutándose en el hardware;
- **Datos:** La estructura de datos creada y manipulada por el software;
- **Conocimiento Humano y Actividades.**

Los sistemas CAD no son más que programas de computadora. Este software normalmente comprende un número de elementos diferentes ó funciones que procesan los datos almacenados en diferentes maneras. Los sistemas CAD incluyen elementos para:

- **Definición de Modelo:** Por ejemplo, adicionar elementos geométricos al modelo de la forma de un componente;
- **Manipulación del Modelo:** Para mover, copiar, borrar, editar ó de otra manera, modificar elementos en el modelo del diseño;
- **Generación de Imagen:** Para generar imágenes del modelo del diseño en la pantalla de una computadora ó en algún dispositivo de almacenamiento;
- **Interacción con el usuario:** Para manejar entrada de comandos por el usuario y para presentar salida de información hacia el usuario acerca de la operación del sistema;
- **Administración de Bases de Datos:** Para la administración de los archivos que conforman la base de datos;
- **Aplicaciones:** Éstos elementos del software no modifican el modelo del diseño, pero lo usan para generar información para evaluación, análisis ó manufactura;
- **Utilidades:** Partes del software que no afectan directamente el modelo del diseño, pero modifican la operación del sistema en alguna forma (ejemplo: la selección del color usado para desplegar, ó las unidades usadas para la construcción de una parte del modelo).

1.8 Manipulación de Entidades y Almacenamiento de Datos.

La especificación de una estructura de datos para apoyar el modelado interactivo es muy demandante. Hay que considerar algunos de los requerimientos impuéstos por un sistema típico, por lo que la estructura debería de:

- Permitir manipulación interactiva – adición, modificación y eliminación – de información;
- Poder soportar tipos múltiples de elementos de datos – geométrico, textual, dimensiones, etiquetas, rutas de herramientas, elementos finitos, etc.;
- Permita propiedades como número de pluma, estilo de línea, color y demás a ser asociados con elementos geométricos;
- Permita la asociación entre elementos de información donde sea importante para el modelo;
- Proveer facilidades para la extracción de partes de la estructura de datos liberada por eliminación ó modificaciones (algunas veces llamados ***colección de basura***);
- Tal vez proveer la facilidad para almacenar geometría comúnmente usada, con referencias a la geometría almacenada como ***instancias***;

- Ser compacto – para minimizar el almacenamiento en disco y los requerimientos de memoria principal;
- Permitir modelos de varios tamaños, y componer varias combinaciones de entidades, a ser definidas;
- Proveer un acceso eficiente a la información, como sea posible.

Éstos requerimientos presentan restricciones importantes en el diseño de la estructura de datos. Por ejemplo, cada tipo de entidad generalmente requerirá diferentes cantidades de información: un punto puede ser definido por tres números de punto flotante representando sus coordenadas *x*, *y* y *z*; por otro lado, una línea requerirá muchos elementos de datos.

Una Estructura de Datos Simple.

Una estructura de datos simple que permite almacenar cantidades arbitrarias de datos por cada entidad, y combinaciones arbitrarias de entidades, conforma una lista o tabla de entidades, que hace referencias cruzadas (ó ***apuntadores***) de esta lista para separar arreglos de puntos flotantes, enteros y otros datos específicos para las entidades. Esto es mostrado esquemáticamente en la **figura 1.18**:

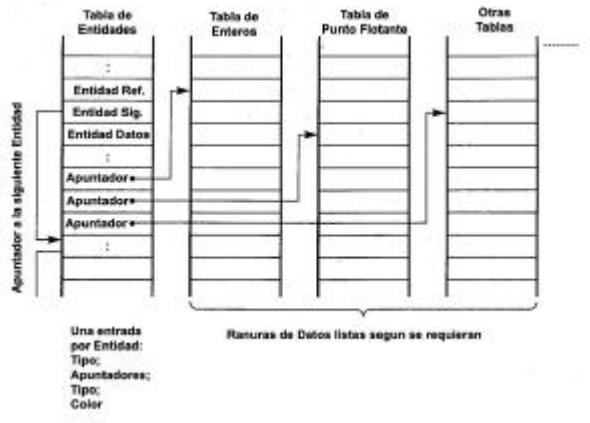


Figura 1. 18.- La entrada de un archivo de despliegue.

Nos referiremos a las tablas de esta estructura como una ***Tabla Entidad***, una ***Tabla de Datos Reales*** para datos de punto flotante y una ***Tabla de Datos Enteros***. En la tabla entidad, existe una serie de ranuras, cada una conteniendo un número de elementos del arreglo usados por la tabla, son asignados uno por entidad. Estas contienen datos generales aplicables a cualquier entidad, como el tipo de entidad, estilo de línea ó color, todo esto con apuntadores hacia datos más específicos de la entidad en la tabla de datos.

La tabla de entidades contiene apuntadores adicionales en cada ranura. Éstos apuntan desde la ranura de la entidad hacia las entidades siguientes y tal vez a las que procedan en la lista, para que los datos puedan ser adicionados, borrados o desplazados arbitrariamente en la lista y así la secuencia permanezca siendo la misma. Este tipo de estructura de datos es conocido como ***listas ligadas***. Si existen apuntadores en dos direcciones por cada entrada en la lista, entonces la estructura es conocida como una ***doble lista ligada***. El borrado de una entidad perteneciente a tal estructura es directo. La **figura 1.19** ilustra este proceso de borrado de entidades:

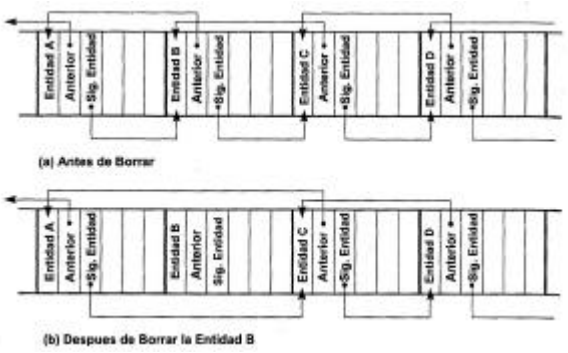


Figura 1. 19.- Entidades en una lista ligada.

Una operación de ***deshacer***, para restaurar entidades borradas erróneamente, se puede implementar al almacenar los detalles de las ligas que han sido modificadas, y recrear estas ligas si es requerido, siempre y cuando las ranuras no hayan sido reusadas por nuevas entidades. Cabe mencionar que mientras la lista de ligas logran hacer directo el reuso de espacio en la tabla de entidades, espacio liberado por el borrado de entidades, el reuso de espacio liberado en otras tablas es más difícil. Algunos sistemas requieren que el usuario solicite explícitamente la ***compresión*** de la estructura de datos para liberar el espacio del modelo usado por entidades borradas.

Datos Generales de Entidades.

Cabe mencionar que la tabla de entidades esta conformada por una serie de ranuras, una por cada entidad y contiene datos generales aplicables a casi todas las entidades. En la **tabla 1.1**, analizamos esto con más detalle:

Datos Generales de Entidad	
Datos	Notas
Tipo de Entidad	Por ejemplo, 1 = punto, 2 = línea, 3 = arco, etc.
Numero de Secuencia de Entidad	Son creadas como entidades alojadas.
Numero de Entidad.	Posición de la entidad en la tabla.
Apuntador a Datos Enteros Apuntador a Datos Reales	En un rango dependiente del tamaño máximo del modelo
Definición de Vista ó Sistema de Coordenadas	Sistema de Coordenadas para entidades planas
Numero de Pluma	Por ejemplo, en el rango de 0-7
Tipo de Curva ó Estilo	Por ejemplo, en el rango de 0-7, representando valores como completo, punteado y línea central.
Color de Entidad	Rango de 0-255
¿La entidad esta en blanco?	Si ó No
¿La entidad esta agrupada?	Si ó No
Numero de niveles ó capas	El numero de niveles debe de estar en el rango 0-255 ó 0-1023, aunque en algunos casos puede estar limitado por los limites de almacenamiento de números enteros.

Tabla 1. 1.- Datos generales de entidad.

Una explicación es requerida acerca de la **tabla 1.1**. Un ***número de entidad de secuencia*** puede ser colocado en secuencia de cada entidad como es definida. Esto permite que la secuencia de construcción puede ser almacenada aun cuando las entidades fueron colocadas en ranuras fuera de secuencia. Un número de entidad basado en la posición absoluta en la tabla permite referencia directa de una entidad a otra. El tipo de curva representa el estilo a ser usado para el dibujo de líneas y curvas. Algunos sistemas permiten variar el grosor en las entidades. Un ***grupo*** ó ***bloque*** es una entidad que representa una colección de otras entidades, agrupadas juntas para que puedan ser tratadas como un sólo elemento para propósitos de selección y manipulación. El ***nivel*** ó ***capa*** es un número almacenado a entidades para ayudarlos a particionar el dibujo/modelo.

Geometría Asociada y Atributos.

La herramienta conocida como ***Geometría Asociativa***, es de gran ayuda parta actualizar las dimensiones para reflejar cambios en un dibujo, como se muestra en la **figura 1.20**:

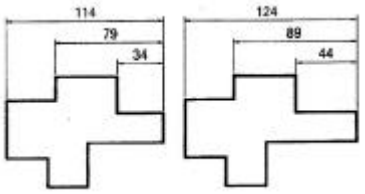


Figura 1.20.- La actualización de dimensiones asociadas.

En un sistema completamente asociativo, si el modelo del sólido es alterado, la geometría mostrada en el dibujo, y las dimensiones asociadas a la geometría, serán alteradas para reflejar el cambio en el modelo. La asociatividad es uno de los mecanismos por el cual la integración es lograda en CAD / CAM. La asociación puede ser extendida a cualquier tipo de datos derivados de un modelo CAD, por ejemplo, las representaciones de análisis de estrés. Con limites, la asociatividad permite diseñar cambios a ser propagados a través de procesos que dependen del modelo del diseño.

Atributos.

Además de entidades asociadas unas con otras, también podemos asociar datos no geométricos con la geometría del modelo a través del uso de ***atributos***. Éstos son típicamente nombres y valores, donde el nombre es una cadena de caracteres alfanuméricos, y el valor puede ser una cadena ó un número. están ligados al elementos del modelo del diseño a través de apuntadores desde los elementos hasta los atributos. Cada elemento, que puede ser una entidad en un dibujo, ó un cuerpo en un modelo sólido, puede ser asociado a un número de atributos, y cada atributo con un número de elementos del modelo, en lo que puede ser conocido como un arreglo "muchos a muchos" **(Ver Figura 1.21)**. El uso clásico de atributos es para la preparación de una lista de partes o "lista de materiales" desde el dibujo en computadora ó un modelo. La lista de materiales es una estructura, descomposición jerárquica de las partes, juntas con la información de cantidades requeridas, número de partes, información del proveedor y referencias al dibujo ó al modelo en CAD **(Ver Tabla 1.2)**.

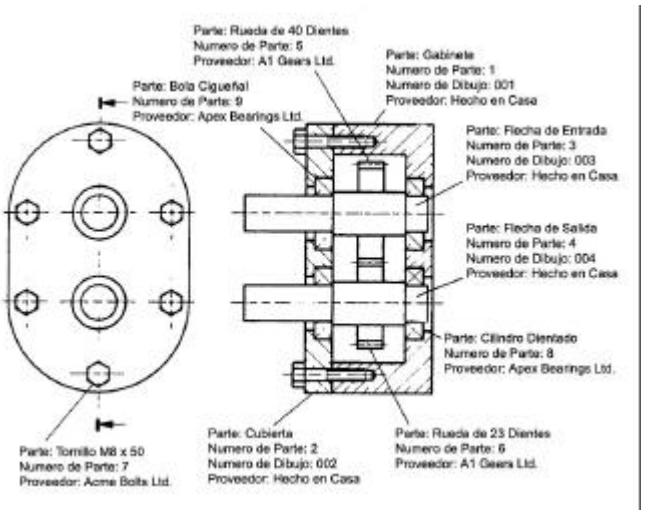


Figura 1. 21.- Ensamblado usando datos de atributos.

Nombre de Parte	Número de Parte	Número de Dibujo	Número Requerido	Nombre Proveedor
Gabinete	1	001	1	Hecho en Casa
Cubierta	2	002	1	Hecho en Casa
Flecha de Entrada	3	003	1	Hecho en Casa
Flecha de Salida	4	004	1	Hecho en Casa
Rueda de 40 Dientes	5		1	A1 Gears Ltd.
Rueda de 23 Dientes	6		1	A1 Gears Ltd.
Tornillo M8 x 50	7		6	ACME Bolts Ltd.
Cilindro Dentado	8		2	Apex Bearings Ltd.
Bola Cigüeñal	9		2	Apex Bearings Ltd.

Tabla 1. 2.- Lista de Materiales.

Capítulo 2.- Diseño del sistema paramétrico.

2.1 Introducción.

En este capítulo, se analizará la teoría de elementos y programación orientada a objetos. Se comprenderá los conceptos de objeto, propiedades, herencia, abstracción., encapsulamiento, modularidad, jerarquía, polimorfismo y otros.

Además se obtendrá un panorama general de la Ingeniería de Software, que ayudará a definir las necesidades del sistema, así como realizar la planeación y diseño del mismo mediante el uso de diagramas de clases, diagramas de casos de uso y diagramas de secuencia. Todos éstos diagramas serán representados usando el lenguaje de modelado unificado UML (Unified Modelling Language), que es un lenguaje de modelado visual de propósito general que es usado para especificar, visualizar, construir, y documentar los artefactos de un software. Se usa para entender, diseñar, administrar, mantener, y controlar información acerca de los sistemas.

También se obtendrá un panorama general sobre el "Lenguaje de Modelado Adaptativo" AML (Adaptative Modeling Language), que sí algo aporta a la teoría de orientado a objetos, es la adición de subobjetos, que es un nuevo concepto. También se comprenderá el manejo y el uso de este lenguaje, la creación y ejecución de código, así como la creación de modelos y objetos.

2.2 Ingeniería de Software.

2.2.1 Ingeniería de Sistemas.

Un sistema de alta tecnología comprende varios componentes: hardware, software, personas, bases de datos, documentación y procedimientos. La Ingeniería de Sistemas ayuda a traducir las necesidades del cliente en un modelo de sistema que utiliza uno o más de éstos componentes.

La Ingeniería de Sistemas comienza tomando una "visión global". Se analiza el dominio del negocio ó producto para establecer todos los requisitos básicos. El enfoque se enfoca en una "visión de dominio", donde cada uno de los elementos del sistema se analiza individualmente. Cada elemento es asignado a uno o más componentes de ingeniería que son estudiados por la disciplina de ingeniería correspondiente.

La ingeniería de procesos de negocio es un enfoque de la ingeniería de sistemas que se usa para definir arquitecturas que permitan a un negocio utilizar la información eficazmente. La intención de la ingeniería de procesos de negocio es crear minuciosas arquitecturas de datos, una arquitectura de aplicación y una infraestructura tecnológica que satisfaga las necesidades de la estrategia de negocios y los objetivos de cada área de negocio. La ingeniería de procesos de negocio comprende una *Planificación de la Estrategia de la Información (PEI)*, un A*nálisis del Área de Negocio (AAN)* y un análisis especifico de aplicación, que de hecho forman parte de la ingeniería de software.

La ingeniería de productos es un enfoque de la ingeniería de sistemas que empieza con el análisis de sistema. El ingeniero de sistemas identifica las necesidades del cliente, determina la viabilidad económica, y asigna funciones y rendimientos al software, hardware, personas y bases de datos; los componentes claves de la ingeniería.

La ingeniería de sistemas demanda una intensa comunicación entre el cliente y el ingeniero de sistema. Esto se logra a través de un conjunto de actividades bajo la denominación de ingeniería de requisitos – identificación, análisis y negociación, especificación, modelización, validación y gestión.

Una vez que los requisitos hayan sido aislados, el modelo del sistema puede ser realizado, y las representaciones de los subsistemas principales pueden ser desarrolladas. La tarea de la ingeniería del sistema finaliza con la elaboración de una *Especificación del sistema*, un documento que sirve para las tareas de ingeniería que se realizaran posteriormente.

2.2.2 Conceptos y Principios del Análisis.

El análisis de requisitos permite al ingeniero de sistemas especificar las características operacionales del software (función, datos y rendimientos), indica la interfaz del software con otros elementos del sistema y establece las restricciones que debe cumplir el software. El análisis de requisitos puede dividirse en cinco actividades: (1) reconocimiento del problema, (2) evaluación y síntesis, (3) modelado, (4) especificación y (5) revisión. Inicialmente, el analista estudia la *Especificación de Sistema* (si existe) y el *Plan del Proyecto de Software*.

Según lo anterior, antes de que los requisitos puedan se aislados, modelados o especificados, éstos deben ser obtenidos. La técnica de obtención de requisitos más usada es llevar a cabo una reunión o entrevista preliminar entre el ingeniero de software y el cliente, esta es una reunión de preguntas. Estas preguntas ayudan a identificar a todos los participantes que tienen interés en el software que se desarrollara. Las preguntas identifican los beneficios medibles de una implementación correcta de posibles alternativas para un desarrollo normal del software. Para poder trabajar como un equipo, para identificar los requisitos, se ha desarrollado un enfoque orientado al equipo para las reuniones, denominadas *Técnicas para Facilitar las Especificaciones de la Aplicación (TFEA)*, que consiste básicamente en:

- La reunión se celebra en un lugar neutro y acuden tanto los clientes como los desarrolladores.
- Se establecen normas de preparación y participación.
- Se sugiere una agenda formal para cubrir los puntos importantes, pero lo bastante informal para alentar el libre flujo de ideas.
- Se designa un coordinador que controle las reuniones.
- Se usa un "mecanismo de definición" (hojas de trabajo, gráficos, carteles, etc.).
- El objetivo es identificar el problema, proponer soluciones, negociar enfoques y especificar un conjunto preliminar de requisitos que permitan alcanzar el objetivo.

El *Despliegue de la Función de Calidad (DFC)* es una técnica de gestión de calidad que traduce las necesidades del cliente en requisitos técnicos del software, la cual identifica tres tipos de requisitos:

- **Requisitos normales:** se declaran objetivos y metas para un producto ó sistema. Si éstos están cubiertos, el cliente estará satisfecho.
- **Requisitos esperados:** son implícitos al sistema y pueden ser tan fundamentales que el cliente no los declara explícitamente.
- **Requisitos innovadores:** estas características van más allá de las expectativas del cliente y suelen ser muy satisfactorias.

La *especificación de los requisitos del software* se produce al finalizar la tarea del análisis. La *introducción* a la especificación de los requisitos del software establece las metas y objetivos del software. La descripción de la información proporciona una descripción detallada del problema que el software va a resolver. Se documenta el contenido de la información y sus relaciones, flujo y estructura. En la *descripción funcional* se describen todas las funciones requeridas para solucionar el problema. En la *descripción del comportamiento* se examina el rendimiento del software en respuesta de acontecimientos externos. Los *criterios de validación* son la lista de criterios a cumplir para validar el software a desarrollar.

2.2.3 Modelo del Análisis.

El modelo de análisis debe lograr tres objetivos principales:

1) Describir lo que requiere el cliente;
2) Establecer una base para la creación de un diseño de software;
3) Definir un conjunto de requisitos que se puedan validar una vez que se construye el software.

Para lograr éstos objetivos, el modelo del análisis, extraído durante el análisis estructurado toma la forma ilustrada en la **figura 2.1**.

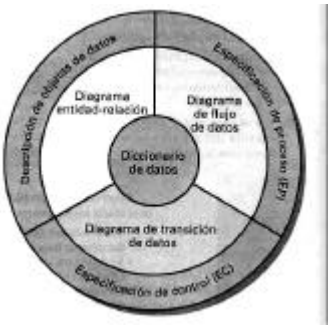


Figura 2. 1.- Estructura del Modelo de Análisis.

En el centro del modelo se encuentra el *diccionario de datos* –un almacén que contiene definiciones de todos los objetos de datos, consumidos y producidos por el software-. Tres diagra mas diferentes rodean al núcleo:

El *Diagrama de Entidad-Relación (DER)* representa las relaciones entre los objetos de datos **(Ver Figura 2.2)**. Los atributos de cada objeto de datos, señalados en el DER, se puede describir mediante una descripción de objetos de datos.

El *Diagrama de Flujo de Datos (DFD)* sirve para dos propósitos:
1) Muestra cómo se transforman los datos a medida del avance en el sistema;
2) Representa las funciones (y subfunciones) que transforman el flujo de datos.

El DFD proporciona información adicional que se usa durante el análisis del dominio de información y sirve como base para el modelado de función **(Ver Figura 2.3)**. En una *Especificación de Proceso (EP)* se encuentra una descripción de cada función presentada en el DFD.

El *Diagrama de Transición de Estados (DTE)* indica como se comporta el sistema. Para lograr esto, el DTE representa los diferentes modos de comportamiento (llamados estados) del sistema y la manera en que se hacen las transiciones de estado a estado. El DTE sirve como la base del modelado de comportamiento **(Ver Figura 2.4)**.

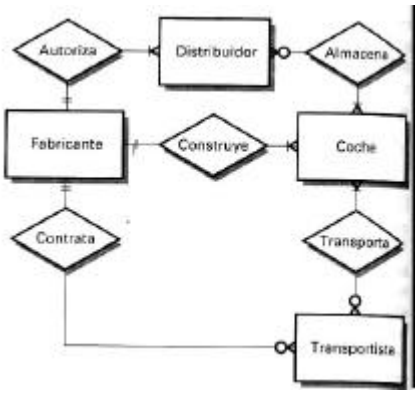


Figura 2. 2.- Ejemplo de diagrama entidad-relación.

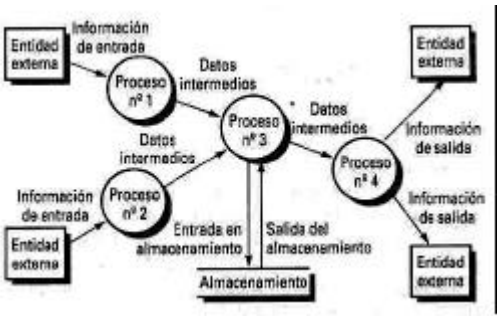


Figura 2. 3.- Ejemplo de diagrama de flujo de datos.

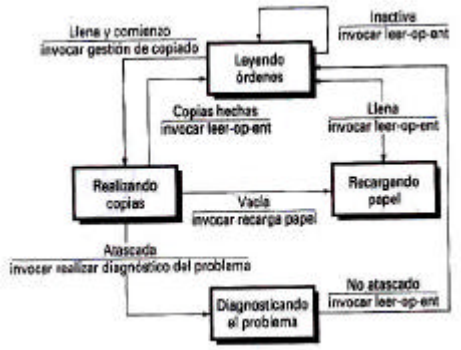
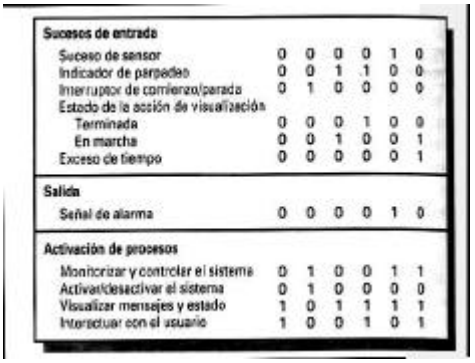


Figura 2. 4.- Ejemplo de diagrama de transición de estados.

La *Especificación de Control (EC)* representa el comportamiento del sistema de dos formas diferentes. La EC contiene un diagrama de transición de estados que es una especificación secuencial del comportamiento. También puede contener una *tabla de activación de procesos (TAP)*, que es una especificación combinatoria del comportamiento **(Ver Figura 2.5)**.



Sucesos de entrada						
Suceso de sensor	0	0	0	0	1	0
Indicador de parpadeo	0	0	1	1	0	0
Interruptor de comienzo/parada	0	1	0	0	0	0
Estado de la acción de visualización						
Terminada	0	0	0	1	0	0
En marcha	0	0	1	0	0	1
Exceso de tiempo	0	0	0	0	0	1
Salida						
Señal de alarma	0	0	0	0	1	0
Activación de procesos						
Monitorizar y controlar el sistema	0	1	0	0	1	1
Activar/desactivar el sistema	0	1	0	0	0	0
Visualizar mensajes y estado	1	0	1	1	1	1
Interactuar con el usuario	1	0	0	1	0	1

Figura 2. 5 .- Ejemplo de tabla de activación de procesos.

Se usa la *especificación de proceso (EP)* para describir todos los procesos del modelo que aparecen. El contenido puede incluir una narrativa textual, una descripción en *lenguaje de diseño de programas (LDP)* del algoritmo del proceso, ecuaciones matemáticas, tablas, diagramas ó gráficos.

El *diccionario de datos* es un listado organizado de todos los elementos de datos que pertenecen al sistema, con definiciones precisas y rigurosas que permiten que el usuario y al analista del sistema tengan una misma comprensión de las entradas, salidas, de los almacenes y de los cálculos intermedios. Actualmente se implementa el diccionario de datos como parte de una herramienta CASE de análisis y diseño estructurados. Aunque el formato varia entre las herramientas, la mayoría maneja la siguiente información: nombre del elemento de datos, alias, donde se usa, como se usa, descripción del contenido e información adicional como valores implícitos, restricciones, etc. Esto mejora la consistencia del modelo de análisis y reduce errores, ya que no permite la duplicidad de nombres entre elementos de datos.

2.2.4 Conceptos y Principios del Diseño.

Cada uno de los elementos del modelo de análisis proporciona la información necesaria para crear los cuatro modelos de diseño que se requieren para una especificación completa de diseño. Mediante uno de los muchos métodos de diseño, se produce un diseño de arquitectura de sistema, un diseño de interfaz y un diseño de componentes.

El ***diseño de datos*** transforma el modelo del dominio de información que se crea durante el análisis en las estructuras de datos que se necesitaran para implementar el software. Los objetos de datos y las relaciones definidas en el diagrama relación entidad y el contenido de datos detallado que se representa en el diccionario de datos proporcionan la base del diseño de datos. A medida que se van diseñando cada uno de los componentes del software, van apareciendo más detalles de diseño.

El ***diseño de arquitectura de datos*** define la relación entre los elementos estructurales principales del software, los patrones de diseño que se pueden utilizar para cumplir los requisitos que se han definido para el sistema, y las restricciones que afectan la manera en que se pueden aplicar los patrones de diseño de arquitectura de datos. La representación del diseño de arquitectura de datos puede derivarse de la especificación del sistema, del modelo de análisis y de la interacción del subsistema definido dentro del modelo de análisis.

El ***diseño de la interfaz*** describe la manera en que el software se comunica con sistemas que ínteroperan dentro de él y con las personas que lo utilizan. Una interfaz implica un flujo de información y un tipo especifico de comportamiento. Por tanto, los diagramas de flujo de control y de datos proporcionan gran parte de la información que se requiere para el diseño de la interfaz.

El ***diseño a nivel de componentes*** transforma los elementos estructurales de la arquitectura del software en una descripción de los procedimientos de los componentes del software.

La importancia del diseño del software se debe a que siempre se busca la mejor calidad. A lo largo de todo el proceso del diseño se evalúa con una serie de revisiones técnicas formales. Se sugieren tres características que sirven como guía para la evaluación de un buen diseño:

- El diseño deberá implementar todos los requisitos explícitos del modelo de análisis y deberá ajustarse a todos los requisitos implícitos.
- El diseño deberá de ser una guía legible y comprensible para aquellos que generan código, comprueban y dan soporte al software.
- El diseño deberá proporcionar una imagen completa del software.

Con el fin de evaluar la calidad de una representación de diseño, se deben establecer criterios técnicos para un buen diseño. Se presentan los siguientes:

1) Un diseño debe presentar una estructura de arquitectura de datos que (1) haya sido creada mediante patrones de diseño conocidos, (2) que este formada por componentes que muestren características de buen diseño y (3) que se puedan implementar de manera evolutiva, facilitando la implementación y comprobación.
2) Un diseño debe ser modular; esto es, el software debe estar dividido lógicamente en funciones y subfunciones especificas.
3) Un diseño debe contener diferentes representaciones de datos, arquitectura, interfaz y módulos.
4) Un diseño debe llevar a estructuras de datos adecuadas para los objetos que se van a implementar y que procedan de patrones de datos reconocibles.
5) Un diseño debe llevar a componentes que presenten características funcionales independientes.
6) Un diseño debe llevar a interfaz que reduzcan la complejidad de las conexiones entre módulos y el entorno externo.
7) Un diseño deberá derivarse mediante un método repetitivo y controlado por la información obtenida durante el análisis de requisitos de software.

Los conceptos principales de diseño de software proporcionan los criterios básicos para la calidad del diseño. La *modularidad* y el concepto de *abstracción* permiten que el diseñador simplifique y reutilice los componentes del software. El refinamiento proporciona un mecanismo para representar sucesivas capas de datos funcionales. El programa y la estructura de datos contribuyen a tener una visión global de la arquitectura de datos del software, mientras que el procedimiento proporciona el detalle necesario para la implementación de los algoritmos **(Ver Figura 2.6)**.

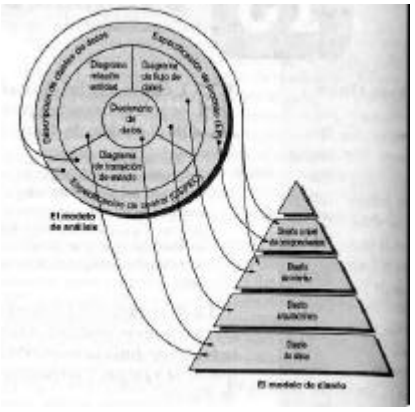


Figura 2. 6.- Conversión del modelo de análisis en un diseño de software.

2.2.5 Diseño de la Arquitectura de Datos.

La arquitectura de datos del software nos proporciona una visión global del sistema a construir. Describe la estructura y la organización de los componentes del software, sus propiedades y las conexiones entre ellos. Los componentes del software incluyen módulos de programas y varias representaciones de datos que son manipulados por el programa. Además, el diseño de datos es una parte integral para la derivación de la arquitectura de datos del software. La arquitectura de datos marca decisiones de diseño anticipadas y proporcionan el mecanismo para evaluar los beneficios en las estructuras de sistema alternativas.

Los objetos de datos definidos durante el análisis de requisitos de software son modelados utilizando diagramas de entidad-relación y el diccionario de datos. La actividad de diseño de datos traduce los elementos del modelo de requisitos en estructuras de datos a nivel de los componentes del software y a arquitectura de base de datos a nivel de aplicación. La arquitectura de base de datos esta clasificada de la siguiente manera:

Arquitectura de datos centralizada. En el centro de esta arquitectura se encuentra una base de datos central a la cual otros componentes acceden con frecuencia para actualizar, añadir, borrar o modificar los datos. El software de cliente accede a la base de datos central. En algunos casos, la base de datos es *pasiva*. Esto significa que el software de cliente accede a los datos independientemente de cualquier cambio en los datos o de las acciones de otro software de cliente **(Ver Figura 2.7)**.

Figura 2. 7.- Arquitectura de base de datos centralizada.

Arquitectura de flujo de datos. Esta arquitectura de datos se aplica cuando los datos de entrada son transformados a través de una serie de computadoras a lo largo de la ruta de salida de los datos de salida. Se utiliza un patrón tipo tubería y filtro **(figura 2.8)**, que tiene un grupo de componentes, llamados filtros, conectados por tuberías que transmiten datos de un componente al siguiente. Cada filtro trabaja independientemente de aquellos componentes que se encuentran en el flujo de entrada ó de salida; esta diseñado para recibir la entrada de datos de una cierta forma y producir una salida de datos de una forma especifica.

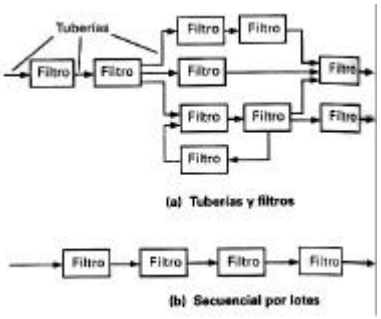


Figura 2. 8.- Estructura de arquitectura de flujo de datos.

Arquitecturas de llamada y retorno. Esta arquitectura de datos permite al diseñador del software construir una estructura de programa relativamente fácil de modificar y ajustar. Existen dos estilos dentro de esta categoría:

- ***Arquitectura de programa principal/subprograma:*** esta arquitectura clásica de programación descompone las funciones en una jerarquía de control donde un programa "principal" llama a un número de componentes del programa, los cuales también pueden llamar a otros componentes de programa. **(Figura 2.9)**.

- ***Arquitectura de llamada de procedimiento remoto*:** los componentes de una arquitectura de datos están formados por un programa principal/subprograma, y éstos están distribuidos entre varias computadoras en una red.

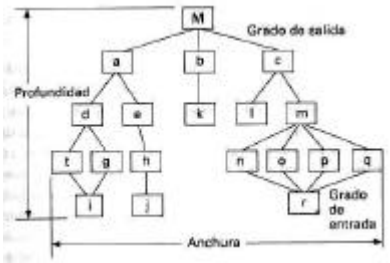


Figura 2. 9.- Terminologías de estructura para un estilo de arquitectura de datos de procedimiento de llamada y retorno.

Arquitectura de datos orientada a objetos. Los componentes de un sistema encapsulan los datos y las operaciones que se deben realizar para manipular los datos. La comunicación y la coordinación entre componentes se consigue a través del paso de mensajes.

Arquitecturas estratificadas. La estructura básica de una arquitectura de datos estratificada se representa en la **figura 2.10**. se crean diferentes capas y cada una realiza operaciones que progresivamente se aproximan más al cuadro de instrucciones de la maquina. En la capa externa, los componentes realizan las operaciones de interfaz de usuario. En la capa interna, los componentes realizan operaciones de interfaz del sistema. Las capas intermedias proporcionan servicios de utilidad y funciones de software de aplicaciones.

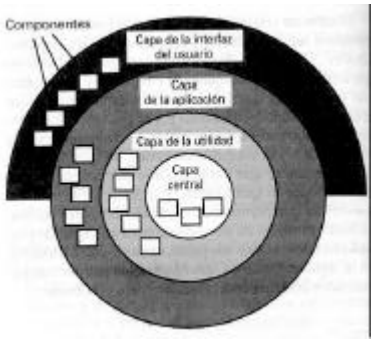


Figura 2. 10.- Estructura Arquitectura estratificada.

2.2.6 Diseño de la Interfaz de Usuario.

Se puede decir que la interfaz de usuario es el elemento más importante de un sistema en computadora. Si la interfaz tiene un diseño pobre, la capacidad que tiene el usuario de aprovechar la potencia de una aplicación disminuye gravemente. Una interfaz débil puede llevar al fracaso a una aplicación con una implementación sólida y un buen diseño.

Existen tres principios importantes que hacen del diseño de interfaz de usuario un proceso eficaz:

1. Poner el control en las manos del usuario.
2. Reducir la carga de memoria en la computadora del usuario.
3. Construir una interfaz con consistente.

La actividad del análisis inicial se concentra en el perfil de los usuarios que van a interactuar con el sistema, se identifican y describen las tareas que lleva a cabo el usuario para conseguir su objetivo. El objetivo del diseño de la interfaz es definir un conjunto de objetivos y acciones de interfaz (y sus representaciones en pantalla) que posibiliten al usuario llevar a cabo todas las tareas definidas de forma de que se cumplan todos los objetivos definidos por el sistema. La implementación comienza normalmente con la creación de un prototipo que permita evaluar todos los escenarios de operación posibles. La validación se centra en la habilidad de la interfaz para implementar correctamente todas las tareas del usuario, el grado de facilidad de utilización y aprendizaje de la interfaz y la aceptación de la interfaz por parte del usuario como una herramienta útil en su trabajo.

A medida que se desarrolla la interfaz de usuario, es común que los diseñadores no tomen en cuenta cualidades que debe de poseer la interfaz de usuario, como es el tiempo de respuesta del sistema, los servicios de ayuda al usuario, la manipulación de información de errores y otros. Estas características se toman en cuenta, por lo general, tiempo después, lo que provoca retrasos en el proyecto.

El tiempo de respuesta del sistema tiene dos características importantes: la duración y la variabilidad. Si la duración de la respuesta del sistema es demasiado larga, se obtiene la frustración y el estrés del usuario, sin embargo, si el tiempo de respuesta del sistema es breve puede obligar a que el usuario se precipite y cometa errores. La variabilidad se refiere a la desviación de respuesta promedio. Una variabilidad baja posibilita al usuario establecer un ritmo de interacción, aunque el tiempo de respuesta sea relativamente largo.

Los mensajes de error o de sugerencias deben tener las siguientes características:

- El mensaje deberá describir el problema en un contexto que el usuario pueda entender.
- El mensaje debe proporcionar consejos constructivos para recuperarse de un error.
- El mensaje debe indicar cualquier consecuencia negativa del error para que el usuario para que el usuario pueda comprobar la existencia de errores y si existen, corregirlos.
- El mensaje debe ir acompañado por una señal audible o visual.
- El mensaje nunca debe de culpar al usuario.

2.2.7 Diseño a Nivel de Componentes.

El diseño a nivel de componentes representa el software a un nivel de abstracción muy cercano al código fuente. A nivel de componentes, la ingeniería de software debe representar estructuras de datos, interfaz y algoritmos con detalle para servir de guía en la generación de código fuente de lenguajes de programación.

La programación estructurada es un método de diseño usando procedimientos, que restringe el número y tipo de construcciones lógicas que se utilizan para representar un algoritmo. El objetivo de la programación estructurada es ayudar al diseñador para que defina algoritmos menos complejos y más fáciles de leer, comprobar y mantener.

Los ciclos de procesamiento son secuénciales, condicionales y repetitivas. Un ciclo de procesamiento ***secuencial*** implementa los pasos del proceso esenciales para la especificación de cualquier algoritmo. Un ciclo de procesamiento ***condicional*** proporciona las funciones para procesos seleccionados a partir de una condición lógica y un ciclo de procesamiento ***repetitivo*** proporciona los bucles. Los tres ciclos de procesamiento son fundamentales para la programación estructurada.

La realización de un diagrama de flujo es bastante sencilla. Mediante el uso de una caja se indica el proceso, un rombo representa una condición lógica y las flechas indican el flujo de control.

El ***Lenguaje de Diseño de programas (LDP)***, también denominado ***lenguaje estructurado*** ó ***pseudocódigo***, es un lenguaje rudimentario en donde es utilizado el vocabulario de un lenguaje (por ejemplo, ingles o español), y la sintaxis global de un lenguaje estructurado de programación. Esta herramienta permite que el LPD se convierta en un esquema del lenguaje de programación real.

2.2.8 Técnicas de Prueba de Software.

El objetivo principal del diseño de casos de prueba es obtener un conjunto de pruebas que tengan la mayor probabilidad de descubrir los defectos del software. Para alcanzar este objetivo, su utilizan dos técnicas de diseño de casos de prueba: prueba de caja blanca y prueba de caja negra.

Las ***pruebas de caja blanca*** se centran en la estructura de control del programa. Se obtienen casos de prueba que aseguren que durante la prueba se han ejecutado, por lo menos una vez, todas las sentencias del programa y condiciones lógicas. La prueba de camino básico, que es una técnica de caja blanca, hace uso de diagramas de flujo de programa para obtener un conjunto de pruebas linealmente independientes que aseguren una cobertura total. Con éstos datos podemos calcular la complejidad ciclomática, que el número de caminos que se pueden recorrer en el diagrama de flujo del programa. La complejidad ciclomática se calcula de la siguiente manera:

La complejidad ciclomática , V(G), de un grafo de flujo G se define como:

$$V(G) = A - N + 2$$

donde A es el número de aristas del diagrama de flujo y N es el número de nodos del mismo.

Las pruebas de condiciones y de flujo de datos prueba aun más la lógica del programa y la prueba de bucles complementa otras técnicas de caja blanca, proporcionando un procedimiento para ejercitar bucles de diferentes grados de complejidad.

Las ***pruebas de caja negra*** o ***pruebas de comportamiento***, son diseñadas para validar los requisitos funcionales sin fijarse en el funcionamiento interno de un programa. La prueba de caja negra intenta encontrar errores como: funciones incorrectas o ausentes, errores de interfaz, errores de estructura de datos, errores de rendimiento y errores de inicialización.

El primer paso para el método de prueba basado en diagramas de flujo, es entender los objetos que se modelan en el software y las relaciones entre los objetos. El siguiente paso es definir una serie de pruebas que verifiquen que todos los objetos tienen entre ellos las relaciones esperadas. La prueba del software empieza creando un diagrama de flujo de objetos importantes y sus relaciones y después diseñar una serie de pruebas que cubran el diagrama de flujo de manera que se ejecuten todos los objetos y sus relaciones para descubrir los errores. La partición equivalente divide el campo de entrada en clases de datos que tienden a ejercer determinadas funciones del software. El análisis de *valores limite* prueba la habilidad del programa para manejar datos que se encuentran en los limites aceptables. La prueba de la tabla ortogonal provee un método sistemático y eficiente para probar sistemas con un número reducido de parámetros de entrada.

2.3 Ingeniería de Software Orientada a Objetos.

2.3.1 Conceptos y Principios.

La tecnología de objetos refleja una visión natural del mundo. Los objetos se clasifican en clases y las clases se organizan en jerarquías. Cada clase contiene un conjunto de atributos que la describen y un conjunto de operaciones que define su comportamiento. Los objetos modelan casi todos los aspectos identificables del dominio del problema: entidades externas, cosas, ocurrencias, roles, unidades organizacionales, lugares y estructuras pueden ser representadas como objetos. Es importante mencionar que los objetos (y las clases de las que se derivan), encapsulan datos y procesos. Las operaciones de procesamiento son parte del objeto y se inician al pasarle un mensaje al objeto. Una definición de clase, una vez definida, constituye la base para la reutilización en los niveles de modelado, diseño e implementación. Los objetos nuevos pueden ser instanciados a partir de una clase.

Tres conceptos importantes diferencian el enfoque OO de la ingeniería del software convencional. El ***encapsulamiento*** empaqueta datos y las operaciones que manejan esos datos. La ***herencia*** permite que los atributos y las operaciones de una clase puedan ser heredados por todas las clases y objetos que se instancias de ella. El ***polimorfismo*** permite que una cantidad de operaciones diferentes posean el mismo nombre, reduciendo la cantidad de líneas de código necesarias para implementar un sistema y facilita los cambios en caso de que se produzcan.

Los productos y sistemas orientados a objetos se producen usando un modelo evolutivo, a veces llamado recursivo/paralelo. El software orientado a objetos evoluciona iterativamente y debe dirigirse teniendo en cuenta que el producto final se desarrollara a partir de una serie de incrementos **(Ver Figura 2.11)**.

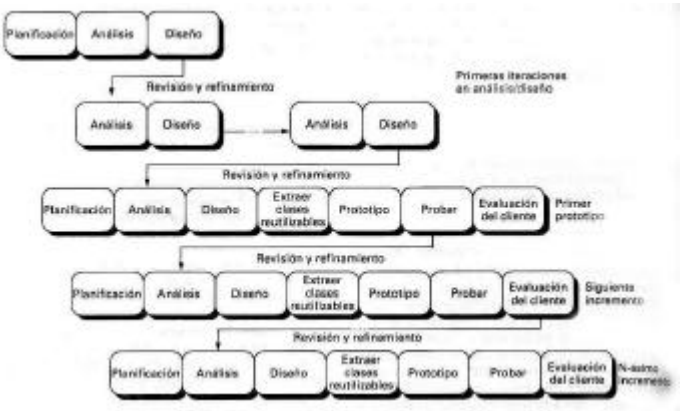


Figura 2. 11.- Secuencia típica de un proceso para un proyecto OO.

2.3.2 Análisis.

Loa métodos de análisis para OO permiten a un ingeniero de software modelar un problema representando las características tanto dinámicas como estáticas de las clases y sus relaciones como componentes principales del modelado. Como los métodos anteriores, el *Lenguaje Unificado de Modelado UML* construye un modelo de análisis con las siguientes características:

1) Representación de las clases y jerarquía de clases.
2) Creación de modelos objeto-relación.
3) Obtención de modelos objeto-comportamiento.

El proceso de análisis orientado a objetos comienza con la definición de casos de uso. La técnica de modelado de Clases-Responsabilidades-Colaboraciones (CRC) se aplica para documentar las clases y sus atributos y operaciones. También proporciona una vista inicial de las colaboraciones que ocurren entre los objetos. El siguiente paso en el análisis orientado a objetos es la clasificación de objetos y la creación de una jerarquía de clases. Los paquetes se pueden utilizar para encapsular objetos relacionados. El modelo ***objeto-relación*** proporciona información sobre las conexiones entre las clases, mientras que el modelo ***objeto-comportamiento*** representa el comportamiento de los objetos individualmente y global de todo el sistema.

2.3.3 Diseño.

El diseño orientado a objetos traduce el modelo de AOO del mundo real, a un modelo de implementación especifica, que se puede realizar en software. El proceso de DOO puede describirse como una pirámide compuesta por cuatro capas. La capa fundamental se centra en el diseño de subsistemas que implementan funciones principales del sistema. La capa de clases especifica la arquitectura de objetos global y la jerarquía de clases requerida para implementar un sistema. La capa de mensajes indica como debe ser realizada la

colaboración entre objetos, y la capa de responsabilidades identifica las operaciones y atributos que caracterizan a cada clase **(Ver Figura 2.12)**.

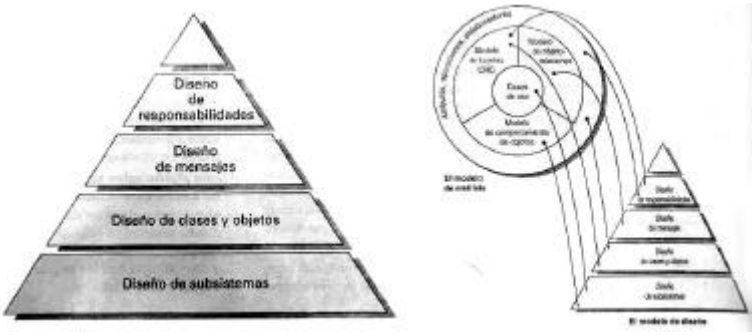


Figura 2. 12.- La pirámide de diseño OO.

Al igual que el AOO, existen diferentes métodos de DOO. UML es un intento de proporcionar una aproximación simple al DOO, que se aplica en los dominios de aplicación. UML aproxima el proceso de diseño mediante dos niveles de abstracción: diseño de subsistemas (arquitectura) y diseño de objetos individuales.

Durante el diseño del sistema, se desarrolla la arquitectura del sistema orientado a objetos. El proceso de diseño de objetos se concentra en la descripción de estructuras de datos, que usan los atributos de clase, los algoritmos que utilizan las operaciones y los mensajes que permiten las operaciones entre objetos relacionados **(Ver Figura 2.13)**.

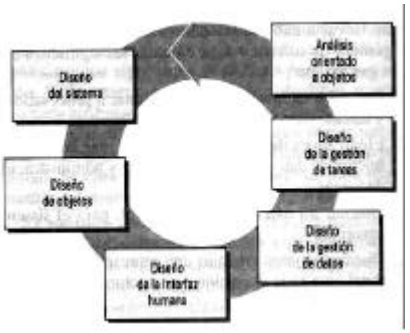


Figura 2. 13.- Flujo de proceso para DOO.

Los patrones de diseño permiten al diseñador crear la arquitectura de diseño integrando componentes reutilizables. Éstos patrones de diseño , por lo general, son desarrollados utilizando el lenguaje UML. La programación OO extiende el modelo de diseño al dominio de ejecución. El lenguaje de programación OO se usa para traducir las clases, atributos, operaciones y mensajes, para que puedan ser ejecutables.

2.3.4 Pruebas.

El objetivo general de la verificación OO, es idéntico al objetivo de prueba del software convencional, que es encontrar el máximo número de errores con un mínimo de esfuerzo. Pero la estrategia y técnicas para la prueba OO difiere en modo significativo, ya que la verificación se amplia para incluir la revisión de los modelos de diseño y de análisis.

El enfoque de prueba se aleja de los componentes de procedimientos, para enfocarse en la clase. Ya que los modelos de análisis y de diseño OO, y el código fuente resultante se acoplan, por eso, los métodos CRC, objeto-relación y objeto-comportamiento se pueden ver como una primera etapa de prueba.

Para evaluar el modelo de clases, se recomienda los siguientes pasos:

1. Revisar el modelo CRC y el modelo objeto-relación.
2. Inspeccionar la descripción de cada tarjeta CRC, para determinar si alguna responsabilidad delegada es parte de la definición del colaborador.
3. Invertir la conexión para asegurarse de que cada colaborador que solicita un servicio recibe las peticiones de una fuente razonable.
4. Utilizando las conexiones invertidas, ya examinadas en el paso 3, determinar si otras clases se requieren y si las responsabilidades se han repartido adecuadamente entre las clases.
5. Determinar si las responsabilidades muy solicitadas, deben combinarse en una sola responsabilidad.
6. Se aplican iterativamente los pasos 1 al 5 para cada clase, y durante cada evolución del modelo de AOO.

Una vez que la prueba OO ha sido concluida, las pruebas de unidad se aplican a cada clase. A diferencia de las pruebas de unidad para el software convencional que tienden a centrarse en el detalle del algoritmo de un modulo y de los datos que fluyen a través de la interfaz de modulo, la prueba de clases para el software OO se conduce mediante las operaciones encapsuladas por la clase y el comportamiento de la clase.

El diseño de pruebas para una clase utiliza varios métodos: pruebas basadas en errores, pruebas al azar y pruebas por partición. Cada uno de éstos métodos ejecuta las operaciones encapsuladas por la clase. Las secuencias de pruebas se diseñan para asegurarse de las operaciones relevantes se ejecutan. El estado de la clase representada por los valores de sus atributos se examina, para determinar si persisten errores.

Ya que el software orientado a objetos no tiene una estructura de control jerárquico, las estrategias convencionales de integración descendiente (top-down) y ascendente (botton-up) tienen muy poco significado. La prueba de integración puede llevarse a cabo utilizando una estrategia basada en el uso o *prueba basada en hilos*, la cual integra el conjunto de clases, que colaboran para responder a una entrada o suceso al sistema. Las pruebas basadas en el uso construyen el sistema en capas, comenzando con las clases que no usan clases servidoras.

Al nivel de sistema o de validación, los detalles de conexiones de clases desaparecen. Así como la validación convencional, la validación del software OO se centra en las acciones visibles al usuario y salidas reconocibles desde el sistema.

Los métodos de diseño de integración de casos de prueba pueden usar pruebas al azar y por partición. En resumen, las pruebas basadas en el escenario y las pruebas derivadas de los modelos de comportamiento pueden usarse para verificar una clase y sus colaboraciones. Una secuencia de pruebas registra el flujo de operaciones, a través de las colaboraciones de clases. La prueba de validación de sistemas OO esta orientada a caja negra y puede completarse aplicando los mismos métodos de prueba de caja negra para el software convencional.

2.4 Programación Orientada a Objetos.

La orientación a objetos se puede describir como el conjunto de disciplinas (ingeniería) que desarrollan y modelizan software que facilitan la construcción de sistemas complejos a partir de componentes. El atractivo intuitivo de la orientación a objetos es que proporcionan conceptos y herramientas con los cuales se modela y representa el mundo real tan fielmente como sea posible. Las ventajas de la orientación a objetos son muchas en programación y modelado de datos. Como mencionaban Ledbetter y Cox en 1985 [Roger S. Presuman, 2002] :

La programación orientada a objetos permite una representación más directa del modelo del mundo real en el código. El resultado es que la radicalmente transformación normalmente llevada a cabo de los requisitos del sistema (definido en términos de usuario) a la especificación del sistema (definido en términos de computadora) se reduce considerablemente.

La **figura 2.14** ilustra el problema. Utilizando técnicas convencionales, el código generado para un problema del mundo real consiste de una primera codificación del problema y la transformación del problema en términos de un lenguaje de computadora Von Newmann. Las disciplinas y técnicas orientadas a objetos manipulan la transformación automáticamente, de modo que el volumen de código codifica el problema y la transformación se minimiza. De hecho, cuando se compara con estilos de programación convencionales (*procedimentales, o por procedimientos*), la reducción de código van desde un 40 por 100 hasta un orden de magnitud elevado cuando se adopta un estilo de programación orientado a objetos.

Figura 2. 14.- Construcción de Software.

Las técnicas orientadas a objetos proporcionan mejoras y métodologías para construir sistemas de software complejos a partir de unidades de software modularizado y reutilizable. La orientación a objetos trata de cumplir las necesidades de los usuarios finales, así como las propias de los desarrolladores de productos de software. Estas tareas se realizan mediante la modelización del mundo real. El soporte fundamental es el ***modelo objeto***. Los cuatro elementos (propiedades) más importantes de este modelo son:

- Abstracción.
- Encapsulamiento.
- Modularidad.
- Jerarquía.

Como sugirió Booch [Roger S. Presuman, 2002], si algunos de éstos elementos no existe se dice que el modelo no es orientado a objetos.

2.4.1 Abstracción.

La abstracción es uno de los medios más importantes, mediante el cual se enfrenta con la complejidad inherente al software. La abstracción es la propiedad que permite representar las características esenciales de un objeto, sin preocuparse de las restantes características (no esenciales). Una abstracción se centra en la vista externa de un objeto, de modo que sirva para separar el comportamiento esencial de un objeto de su implementación. Definir una abstracción significa describir una entidad del mundo real, no importa lo compleja que pueda ser, y utilizar esta descripción en un programa. El elemento clave de la programación orientada a objetos es la clase. Una clase se puede definir como una descripción abstracta de un grupo de objetos, cada uno de los cuales se diferencia por su estado especifico y por la posibilidad de realizar una serie de operaciones. Por ejemplo, una pluma estilográfica es un objeto que tiene un estado (llena de tinta ó vacía) y sobre la cual se pueden realizar algunas operaciones (por ejemplo, escribir, poner o quitar el capuchón, llenar de tinta si esta vacía). La idea de escribir programas definiendo una serie de abstracciones no es nueva, pero el uso de clases para gestionar dichas abstracciones en lenguajes de programación ha facilitado considerablemente su aplicación.

2.4.2 Encapsulamiento.

El encapsulamiento ó encapsulación es la propiedad que permite asegurar que el contenido de la información de un objeto esta oculta al mundo exterior: el objeto A no conoce lo que hace el objeto B, y viceversa. La encapsulación (también se conoce como ocultación de la información), en esencia, es el proceso de ocultar todos los secretos de un objeto que no contribuyen a sus características esenciales. La encapsulación permite la división de un programa en módulos. Éstos módulos se implementan mediante clases, de forma que una clase representa la encapsulación de una abstracción. En la practica, esto significa que cada clase debe tener dos partes: un interfaz y una implementación. El interfaz de una clase captura sólo su vista externa y la implementación contienen la representación de la abstracción, así como los mecanismos que realizan el comportamiento deseado.

2.4.3 Modularidad.

La modularidad es la propiedad que permite subdividir una aplicación en partes más pequeñas (llamadas módulos), cada una de las cuales debe ser tan independiente como sea posible de la aplicación en si y de las restantes partes. La modularización consiste en dividir un programa en módulos que se puedan compilar por separado, pero que tienen conexiones con otros módulos. Al igual que la encapsulación, los lenguajes soportan la modularidad de diversas formas.

2.4.4 Jerarquía.

La jerarquía es una propiedad que permite una ordenación de las abstracciones. Las dos jerarquías más importantes de un sistema complejo son: ***estructura de clases*** (jerarquía <<es-un>>: generalización / especialización) y ***estructura de objetos*** (jerarquía <<parte-de>>: agregación).

No se debe confundir clases y objetos de la misma clase: un coche rojo y un coche azul no son objetos de clases diferentes, sino objetos de la misma clase con un atributo diferente.

Las jerarquías de generalización / especialización se conocen como ***herencia.*** Básicamente, la herencia define una relación entre clases, en donde una clase comparte la estructura ó comportamiento definido en una ó más clases (*herencia simple y herencia múltiple*, respectivamente). La agregación es el concepto que permite el agrupamiento físico de estructuras relacionadas lógicamente. Así, un camión se compone de ruedas, motor, sistema de transmisión y chasis; en consecuencia, camión es una *agregación*, y ruedas, motor, transmisión y chasis son agregados de camión.

2.4.5 Polimorfismo.

La quinta propiedad significativa de los lenguajes de programación orientados a objetos es el polimo rfismo. Esta propiedad no suele ser considerada como fundamental en los diferentes modelos de objetos propuestos, pero debido a su importancia, no tiene sentido considerar un *objeto modelo* que no soporte esta propiedad. Polimorfismo es la propiedad que indica, literalmente, la posibilidad de que una entidad tome muchas formas. En términos prácticos, el polimorfismo permite referirse a objetos de clases diferentes mediante el mismo elemento de programa y realizar la misma operación de diferentes formas, según sea el objeto que se referencia en ese momento. Por ejemplo, cuando se describe la clase mamíferos se puede observar que la operación comer es una operación fundamental en la vida de los mamíferos, de modo que cada tipo de mamífero debe poder realizar la operación ó función *comer*. Por otra parte, una vaca ó una cabra que pastan en un campo, un niño que se come un bombón ó caramelo y un león que devora a otro animal, son diferentes formas que utilizan los distintos mamíferos para realizar la misma función (comer). El polimorfismo implica la posibilidad de tomar un objeto de un tipo (mamífero, por ejemplo) e indicarle que ejecute *comer***;** esta acción se ejecutará de diferente forma, según sea el objeto mamífero sobre el que se aplica.

Clases, herencia y polimorfismo son aspectos claves en la programación orientada a objetos, y se reconocen a éstos elementos como esenciales en la misma. El polimorfismo adquiere su máxima expresión en la *derivación* ó *extensión* de clases, es decir, cuando se obtiene una clase a partir de una clase ya existente, mediante la propiedad de derivación de clases ó herencia. así, por ejemplo, si se dispone de una figura que

represente figuras genéricas, se puede enviar cualquier mensaje, tanto a un tipo derivado (elipse, circulo, cuadrado, etc.) como al tipo base. Por ejemplo, una clase *figura* es un tipo de figura y puede recibir el mismo mensaje. Cuando se envía un mensaje, por ejemplo dibujar, esta tarea será distinta según que la clase sea un triangulo, un cuadrado ó una elipse. Esta propiedad es el polimorfismo, que permite que una misma función se comporte de diferente forma según sea la clase sobre la que se aplica. La función *dibujar* se aplica igualmente a un circulo, a un cuadrado ó a un triangulo, y el objeto ejecutara el código apropiado dependiendo del tipo especifico.

El polimorfismo requiere *ligadura tardía* ó *postergada* (también llamada *dinámica*), y esto sólo se puede producir en lenguajes de programación orientados a objetos. Los lenguajes no orientados a objetos soportan *ligadura temprana* ó *anterior*, esto significa que el compilador genera una llamada a un nombre especifico de función y el enlazador (*linker*) resuelve la llamada a la dirección absoluta del código que se ha de ejecutar. En programación orientada a objetos, el programa no puede determinar la dirección del código hasta el momento de la ejecución; para resolver este concepto, los lenguajes orientados a objetos utilizan el concepto de *ligadura tardía*. Cuando se envía un mensaje a un objeto, el código que se llama no se determina hasta el momento de la ejecución. El compilador asegura que la función existe y realiza verificación de tipos de los argumentos y del valor de retorno, pero no conoce el código exacto a ejecutar. Para realizar la ligadura tardía, el compilador inserta un segmento especial de código en lugar de la llamada absoluta. Este código calcula la dirección del cuerpo de la función para ejecutar en tiempo de ejecución utilizando información almacenada en el propio objeto. Por consiguiente, cada objeto se puede comportar de modo diferente de acuerdo al contenido de ese puntero. Cuando se envía un mensaje a un objeto, este sabe que ha de hacer con ese mensaje.

2.4.6 Otras Propiedades.

El modelo objeto ideal no sólo tiene las propiedades anteriormente citadas, sino que es conveniente que soporte, además, estas propiedades:

- Concurrencia (multitarea).
- Persistencia.
- Genericidad.
- Manejo de excepciones.

Muchos lenguajes soportan todas estas propiedades y otros sólo algunas de ellas. así, por ejemplo, Ada soporta concurrencia; Ada y C++ soportan genericidad y manejo de excepciones. La persistencia ó propiedad de que las variables – y por extensión a los objetos - existan entre las invocaciones de un programa es posiblemente la propiedad menos implantada en los lenguajes de programación orientados a objetos.

2.5 Programación en AML.

2.5.1 ¿Qué es A.M.L.?.

TechnoSoft Inc. fue fundada en 1992, como proveedor líder de software orientado a objetos para aplicaciones comerciales y de defensa. TechnoSoft ofrece un marco de trabajo avanzado en ingeniería con un paradigma de modelado orientado a objetos llamado “Lenguaje de Modelado Adaptativo” (Adaptative Modeling Language, AML), que permite el modelado y simulación de ciclos completos de desarrollo de productos; integrando y automatizando la configuración, visualización, diseño, análisis, inspección y estimación de costo del producto. Las oficinas de TechnoSoft Inc en la ciudad de Monterrey son las que están proporcionando a la Facultad de Ingeniería el software y las licencias de uso académico, para desarrollar aplicaciones en AML.

El Lenguaje de programación AML es un lenguaje de modelado que fue diseñado pensando en la Ingeniería Concurrente. AML provee un paradigma para el modelado y organización de conocimiento de ingeniería vital requerido para la integración y la automatización de ciclos enteros de ingeniería desde el diseño hasta la producción. AML esta basado en el concepto de programación orientada a objetos. En la programación

orientada a objetos, los bloques constructores de aplicaciones son objetos, que no son procedimientos ó funciones. Usando tecnología orientada a objetos, uno puede construir un modelo que refleja entidades reales y las operaciones pueden ser usadas para resolver un número de problemas relacionados. AML provee una base de conocimiento en ingeniería (Knowleged Based Engineering, KBE), un marco de trabajo que captura el conocimiento del dominio modelado y crea modelos paramétricos con ese conocimiento. AML es "adaptativo" en que puede ser usado para modelar un amplio rango de dominios que tienen componentes interactuando y comportamiento restringido entre ellos. AML puede ser adaptado a diversas aplicaciones de ingeniería.

El marco de trabajo de AML da soporte a una sola arquitectura orientada a objetos fundamental. El paradigma de diseño orientado a objetos de AML provee dos diferentes tipos de relaciones. Una relaciona clase-subclase que permite abstracción, encapsulamiento y compartir estructuras de datos y comportamiento, y polimorfismo. Las subclases pueden ser derivadas de cualquier clase de AML, o de clases definidas por el usuario. La herencia múltiple esta disponible para las clases. Una clase puede ser derivada de una clase existente, y nuevas propiedades pueden ser adicionadas, ó formulas y valores redefinidos para propiedades existentes como se muestra a continuación:

```
(define-class class-name
:inherit-from (object-list)
:properties (property-list)
:sub-objects (subobject-list)
)
```

AML provee un grupo expandido de clases que soportan un amplio espectro de aplicaciones. Tales clases son la base de varios módulos soportados dentro de AML. Cuando la propiedad de un objeto es cambiada, AML automáticamente notifica a todas las otras propiedades dependientes de esa propiedad para que sean consistentes con el nuevo valor para cuando vuelvan a ser demandadas.

Arquitectura.

El marco de trabajo en modelado en AML consiste en vario módulos (conjunto de clases y métodos), relacionando los diferentes dominios de conocimiento, cada uno concentrado en una funcionalidad diferente. Todos los módulos están escritos dentro de la arquitectura orientada a objetos de AML, aunque se comunican con programas externos a través de la Arquitectura de Capa Virtual. módulos adicionales pueden ser definidos y cargados en AML para adaptar el lenguaje para un propósito especifico. Debido a que AML es modular, solamente los sistemas necesarios necesitan ser cargados en AML. Si un problema requiere un marco de trabajo de modelado sin gráficos o geometría, solamente el kernel necesita ser cargado para esa aplicación. Aplicaciones que involucran diferentes aspectos de un sistema construido en AML utilizarán una interfaz de usuario común al sistema. Los módulos de diseño, análisis, manufactura e inspección utilizan una interfaz común en AML.

2.5.2 ¿Cómo funciona?

Básicamente se puede decir que AML es un lenguaje de programación orientado a objetos, con características CAD. AML cuenta con un editor de código "x-emacs", que permite editar el código de clases y métodos. La sintaxis de AML se basa en LISP.

AML tiene características de desempeño similares a otros lenguajes de programación orientados a objetos, como java. Cuando se desarrolla código en AML, desde el editor "x-emacs", se activa la maquina virtual de aml con el comando "run aml" 	La sintaxis del código que define clases y métodos puede ser revisada mediante el compilador de AML, mediante el uso del comando "compile form" 
Las nuevas clases desarrolladas se pueden anexar al árbol de clases de AML mediante el uso de la herramienta: "Load File" 	Para poder ver los resultados obtenidos con las clases y métodos desarrollados, es necesario activar el entorno gráfico AML-GUI, mediante el uso del comando "AML-GUI" 

Tabla 2. 1.- Comandos de AML.

Esto permite al código en AML tener una característica de exportación a cualquier plataforma similar a JAVA. Esto significa que AML no genera archivos autoejecutables .EXE para la plataforma Windows.

2.5.2.1 Métodología.

La métodología orientada a objetos se basa en los siguientes mecanismos ó principios:

- Objetos
- Clases
- Herencia
- Métodos

2.5.2.2 Objetos / Sub-objetos.

Las unidades básicas de construcción de la estrategia orientada a objetos son los objetos. Un objeto, en este paradigma, es una representación uniforme de una entidad de la vida real. En el método orientado a objeto, el programador piensa en términos de objetos físicos y las nuevas estructuras de datos (objetos) son definidos en términos de objetos reales. Los objetos pueden ser pensados como abstracciones de datos que contienen una colección de elementos de datos relacionados (propiedades y sub-objetos) y un grupo de procedimientos (métodos) que operan sobre los elementos de los objetos. Los objetos tienen las siguientes características esenciales:

- Los objetos interactúan uno con otro; esto es realizado a través de la transmisión de mensaje. La transmisión de mensajes en AML es básicamente una llamada al método que esta asociado con el objeto a ser comunicado con. En AML un objeto se puede comunicar con otro objeto al cambiar el valor de un atributo que esta asociado con el objeto a ser comunicado con.
- Cada objeto en el sistema tiene una identidad única. La identidad del objeto es la propiedad en el objeto que distingue al objeto de todos los otros objetos del sistema. En un sistema orientado a objetos la identidad del objeto es única e independiente del valor de los atributos del objeto.
- En una estrategia orientada a objetos, las unidades de encapsulamientos son objetos. Un objeto encapsula información de estado (datos) y comportamiento (operaciones). Las operaciones son sólo una forma de cambiar el estado de un objeto.
- Si un objeto es lógicamente relacionado con uno ó más objetos, entonces existe una asociación entre los objetos. Las asociaciones pueden ser implementadas usando atributos ó al usar un objeto para representar la asociación.

- Es posible construir objetos compuéstos. Un objeto compuesto es uno que consiste de partes que ellas mismas son objetos (relación objeto-subobjeto).

La **figura 2.15** es un ejemplo de un problema de descomposición donde construimos el modelo de un escritorio usando lenguaje de programación orientado a objetos. Fragmentamos el problema como lo muestra la siguiente figura, donde el escritorio tiene panel superior, panel derecho, panel izquierdo y un modesto panel. Después de que los objetos necesarios a ser modelados son identificados por el usuario, se puede trabajar en el modelo.

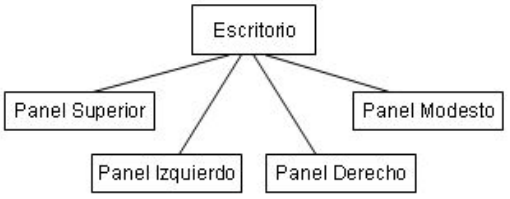


Figura 2. 15.- Ejemplo de Descomposición.

En el ejemplo, los paneles objetos son básicamente los mismos. Seria una perdida de tiempo y esfuerzo definir cada panel separadamente. Es más eficiente definir un panel genérico e instanciar el panel genérico para reflejar el estado de cada panel especifico.

2.5.2.3 Clases

En el paradigma orientado a objetos, la herramienta para crear nuevos tipos de datos es la clase. Una clase puede ser pensada como una plantilla que puede ser usada para crear objetos. Los objetos pertenecientes a una clase en particular se dice que son instancias de una clase. Las clases permiten a los objetos ser definidos en una manera eficiente. Los métodos y las variables para las clases son definidas una sola vez, en la definición de clase. Cada instancia de la clase contiene el valor actual de las variables. Los siguientes conceptos son esenciales para entender clases:

Una clase define la definición estructural de instancias de una clase. La clase define los nombres de atributos (estado) y métodos (comportamiento) de un objeto perteneciente a una clase. Las clases son usadas para crear objetos (instancias de una clase). Un objeto pertenece exactamente a una clase, mientras que una clase puede tener un cierto número de instancias.

La herencia es un mecanismo muy importante para la definición de clase. Una clase puede ser definida en términos de clases existentes estableciendo una relación superclase – subclase. Una subclase hereda los atributos y operaciones de la superclase(s) y puede adicionar atributos y operaciones. Una subclase puede ser pensada como la especialización de una superclase.

Por ejemplo, se asume que el usuario tiene que modelar un número de autos (sedan, coupe, etc.). Éstos autos tienen las mismas características básicas (4 neumáticos, parabrisas frontal, etc.). Para modelar el sistema, el usuario puede definir una superclase llamada AUTOMOVIL de la cual el resto de las clases heredan. La simple jerarquía es ilustrada en la **figura 2.16**.

Figura 2. 16.- Ejemplo de Jerarquía.

Una clase puede heredar de más de una clase. Esto es referido como herencia múltiple. Este es el caso cuando la clase de un objeto tiene que jugar múltiples papeles. Por ejemplo, la clase delfín puede heredar de la clase pez y de la clase mamífero. Cuando la herencia múltiple ocurre una estructura tipo árbol puede ser desarrollada para describir la jerarquía de clases **(Figura 2.17)**. Esta jerarquía de clases es importante para resolver conflictos, que ocurren porque las superclases pueden tener los mismos atributos y / ó mismos nombres de operaciones. Los conflictos tienen que ser resueltos antes de generar la definición de clase final.

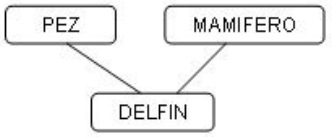


Figura 2. 17.- Ejemplo de herencia múltiple.

2.5.2.4 Métodos

En la técnica orientada a objetos, los bloques constructores básicos son los objetos. Los objetos que conforman un modelo necesitan comunicarse uno con otro al llamar a los métodos (operación) que están asociados al objeto a ser modificado. En la terminología orientada a objetos, los objetos se comunican unos con otros a través de mensajes. Un mensaje es simplemente el recibir objeto combinado con el nombre de uno de sus métodos. Una de las ventajas de usar métodos es que permiten el reuso de nombres (sobrecarga). En AML, una clase diferente representa cada tipo de forma geometría. A través de la sobrecarga, el usuario puede usar el mismo nombre para el método de dibujo en cada clase. En AML, para dibujar un objeto gráfico lo que el usuario tiene que hacer es invocar el dibujo del objeto a ser dibujado. Esto permite a las clases definidas por el usuario comportarse en la misma forma como las clases definidas por el sistema. Si la sobrecarga no estuviera disponible, entonces el usuario necesitaría dar un nombre diferente de operación para objetos dibujados de diferentes clases. Dependiendo de la clase objeto, el usuario tendrá que llamar la operación adecuada. El reuso de nombres (sobrecarga) permite el diseño de código simple.

2.5.3 ¿Cómo se utiliza?

Iniciando AML: AML se ejecuta en ambas plataformas, UNIX y Windows. Technosoft da soporte en AML para Hewlett Packard, Sun, Silicon Graphics y máquinas IBM UNIX, así como máquinas PC Intel. El editor x-emacs de AML, en la plataforma UNIX difiere ligeramente del editor de Windows NT.

Procedimiento general para el desarrollo de modelos.

El procedimiento general para el desarrollo de modelos involucra los siguientes pasos:

- Activar el editor x-emacs de AML.
- Iniciar AML.
- Editar los archivos a convertir en definiciones de clases, etc.
- Cargar los archivos en la memoria de AML.
- Instanciar las clases usando *create-model* ó *add-object*.
- Inspeccionar los resultados en el ambiente de AML.

Capítulo 3 .-Requerimientos y Especificaciones.

3.1 Introducción.

En este capítulo se definirán los requerimientos y especificaciones necesarias para la interfaz gráfica desarrollada en el lenguaje de programación AML y su operación conjunta con el proyecto SADET (ver sección 3.2.). Esto será necesario para que ambos sistemas puedan utilizar la misma información con un proceso sencillo.

Además, utilizando la teoría proporcionada por UML (The Unified Modeling Language), se modelará el sistema a desarrollar en AML, mediante la definición de clases, sus atributos y métodos. También definiremos las relaciones de estas clases, relaciones entre si mismas, así como las relaciones entre estas y las propias clases nativas de AML.

También haremos uso de los diagramas de casos de uso, que son descritos en la teoría de UML, que servirán para detallar el comportamiento del sistema a desarrollar en AML. También se describirá la interacción del usuario con el sistema, mediante los diagramas de caso de uso y los diagramas de secuencia de eventos.

3.2 Definición de requerimientos y especificaciones.

La principal tarea en el diseño de productos, es la determinación y evaluación de las características y propiedades del producto. Para poder desarrollar estas tareas el diseñador cuenta con una serie de técnicas metódicas y de soluciones organizacionales. Una de estas técnicas es el diseño para la manufactura CAD (Diseño Asistido por Computadora), CAM (Manufactura Asistida por Computadora), mas estas en la gran mayoría trabajan independientemente. Por lo que es necesario integrar las diferentes herramientas de computo que se usan durante la realización de productos para lograr así un soporte integral que asista eficazmente a equipos de diseño. Para alcanzar la integración, se plantea un sistema en donde se consultan y almacenan datos empleando un deposito de información, en donde la estructura y contenido de las bases de datos están determinados por los modelos de información.

La información necesaria para soportar la realización del diseño y funciones de manufactura en una empresa, puede ser capturada y representada en dos modelos de información [A. Ayala, et al, 2002]:

1. **Modelo del Producto**: Los modelos de los productos surgen del análisis de los datos necesarios para soportar el diseño y la manufactura de productos. Éstos modelos son implementados en bases de datos para proveer de información a las aplicaciones computacionales que asisten el diseño concurrente de productos.

2. **Modelo de Manufactura**: Un MM identifica, representa y captura los datos, información y conocimiento que describen los recursos, procesos y estrategias de manufactura de una compañía particular [Molina, 1995]. Un MM provee de la información de manufactura necesaria a aplicaciones para efectuar diseño para manufactura [A. Ayala, et al, 2002].

En la facultad de ingeniería de la UNAM, se lleva a cabo un proyecto (PAPIIT-IN110398), que tiene como objetivo el explorar la aplicación de modelos de información para asistir el rediseño de componentes. Se cuenta también con la colaboración de CONACYT (proyecto J-27755U) y empresas. Una parte fundamental del proyecto es el desarrollo del sistema SADET (Sistema Auxiliar para el Diseño de Ejes de Transmisión).

En el proyecto SADET (**figura 3.1**) se ha desarrollado un modelo de producto (MP), que almacena la estructura, la geometría, las dimensiones y tolerancias de ejes de transmisión, y en general de cualquier eje. Un modelo de manufactura (MM) que incluye la representación de talleres en términos de recursos y procesos de manufactura. Un modelador de ejes de transmisión (met), este programa permite crear y editar bases de datos que representan ejes de acuerdo a la especificación del MP. Modelador de celdas de manufactura (mcm), este programa se hizo en forma conjunta con el MM, y permite crear y editar bases de datos que representan celdas de manufactura de acuerdo a la especificación del MM.

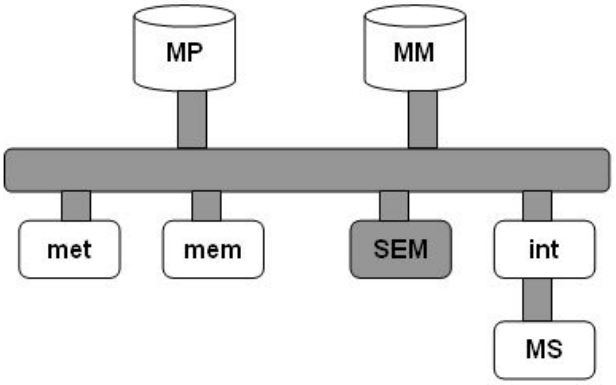


Figura 3.1.- Estructura del Proyecto SADET.

Nuestra necesidad consiste en la creación de una interfaz gráfica que nos permita modelar piezas de revolución obtenidas desde la base de datos que conforma el modelo del producto, poder realizar modificaciones en los modelos y poder grabar éstos cambios en la base de datos del modelo del producto de SADET. Para poder solucionar este problema, haremos uso de el lenguaje de programación AML. Por lo tanto, el planteamiento del problema se define como:

- **Problemática:** Se cuenta con un sistema que permite el diseño de piezas de revolución. Sin embargo, no se cuenta con una interfaz gráfica que permita mostrar el diseño.
- **Necesidad:** se requiere modelar piezas de revolución, las cuales provienen de una base de datos orientada a objetos (OODB).
- **Solución:** Realizar una interfaz gráfica utilizando AML.

Utilizaremos el lenguaje de programación AML, porque permite un acceso abierto a la representación de partes que es un requerimiento importante. AML permite al diseñador interactuar dinámicamente con el modelo de partes unificado y probar escenarios de prueba diferentes para evaluar diseños y procesos alternativos. Sobre muchos sistemas CAD / CAM / CAE, AML utiliza una arquitectura orientada a objetos única y sencilla, que da soporte a capas topológicas virtuales que aseguran la integración de los diferentes procesos involucrados en los ciclos de ingeniería de partes manufacturadas. Los diseños de partes (geometría, características, materiales, función, etc.) y procesos (manufactura, inspección y análisis) son concurrentemente generados desde, y almacenados en, en un modelo de partes simple.

La característica más importante de AML sobre los sistemas CAD / CAM / CAE, es que AML es un lenguaje de programación orientado a objetos que permite no solamente diseñar y modelar (posee las cualidades de los sistemas CAD / CAM / CAE), sino también crear herramientas hechas a la medida y necesidades de cualquier proceso, mientras que los sistemas CAD / CAM / CAE han sido creados como herramientas generales para diseño y modelado con manejo de archivos y resultados en formatos exclusivos para la herramienta usada, que es lo que hace difícil automatizar los procesos.

3.3 Diseño del sistema en UML.

3.3.1 Interfaz gráfica nativa de AML.

Existen clases que construyen elementos gráficos que permiten visualizar los modelos programados, asi como manipular sus vistas y sus características, pero lo más importante, es que estas clases permiten al usuario comunicarse con AML de una manera gráfica. Estas clases y herramientas estan definidas por las clases *ui-base-classes* y *ui-advanced-classes* del marco de trabajo de AML y estas son usadas para construir la interfaz gráfica de usuario (AML-GUI) para aplicaciones en AML. Para el diseño de la interfaz, se utilizan las siguientes clases.

La clase *ui-resizable-pane-mixin* proporciona un panel dividido de dimensiones variables, en los cuales se pueden instanciar herramientas en cada panel **(Ver Figura 3.3)**. Una de las herramientas que contiene es el canvas, o el área donde serán editados los objetos, esta generado por la clase *ui-canvas-class*. Todos los cambios que sean realizados sobre el modelo, se verán reflejados en el área "negra" del canvas, ya que en las propiedades de la clase antes mencionada, se puede definir que modelo del arbol de clases debe mostrar. La otra herramienta que contendrá el panel será generada por la clase *ui-graphic-control-form-class*, que es una herramienta de control de gráficos, que contienen herramientas para modificar la vista del modelo, así como controles de control de color y de luz. Todos los cambios que genera este control de gráficos se ve reflejado en el canvas **(Ver Figura 3.2)**.

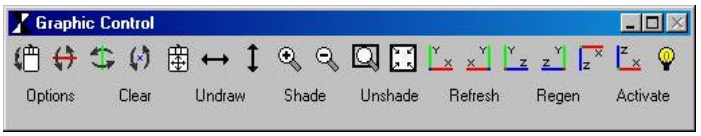


Figura 3. 2.- Ejemplo de la barra de herramientas de control de gráfico.

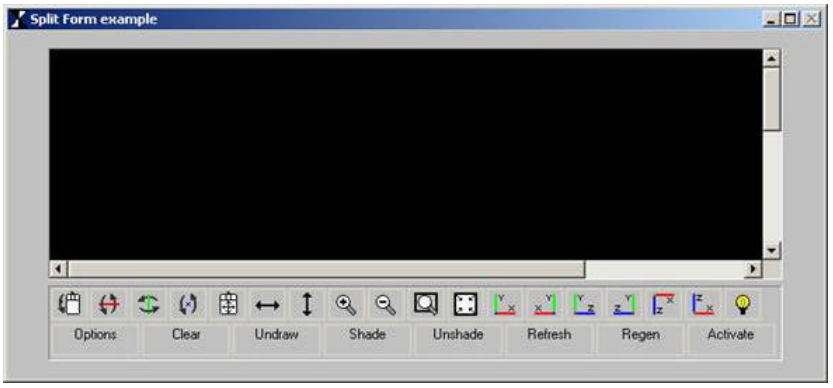


Figura 3. 3.- Ejemplo de Canvas y Herramientas de control de gráficos, unidos en el panel.

Utilizando la propiedad inherente de la programación orientada a objetos, la herencia, por medio de la clase *ui-form-class*, creamos ventanas ó formas contenedoras. Estas clases permiten instanciar otros objetos que heredan de otras clases, como subobjetos. La siguiente lista esta conformada por las clases usadas, instanciadas sobre las formas y su papel que juegan en el sistema.

ui-action-button-class	Esta clase provee un botón visual con capacidad de llamada con clic izquierdo y clic derecho.
ui-label-class	Esta clase provee de una área de etiqueta simple para desplegar texto o imágenes.
ui-labeled-field-class	Esta clase provee un objeto compuesto de una etiqueta y un campo de edición de datos. Este es usado para la entrada de datos.

ui-option-menu-class — Esta clase provee un objeto de selección de opción con un botón que despliega un menú de opciones disponibles . sólo una opción puede ser seleccionada a la vez. A menos que el botón sea presionado, el objeto muestra solamente la opción seleccionada. En la plataforma WINDOWS, este objeto es referenciado como un "Combo Box" no editable. El menú de opciones es activado con un clic izquierdo del Mouse.

ui-radio-buttons-class — Esta clase provee de un grupo de botones exclusivos, donde solamente un botón puede ser seleccionado a la vez.

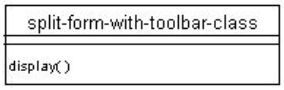
ui-toolbar-class — Esta clase permite la creación de varios puntos colocados adyacentemente, por lo que se conocerá como una barra de herramientas ó "toolbar". Los botones pueden ser colocados horizontalmente ó verticalmente

ui-typein-field-class — Esta clase provee un campo para la entrada de datos. Además, provee detección de errores y validación.

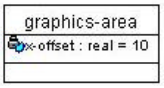
3.3.2 Definición de clases del sistema.

Para el sistema que nos permitirá modelar piezas de revolución obtenidas desde la base de datos que conforma el modelo del producto del sistema SADET *(ver sección 3.2)*, se proponen las siguientes clases, que servirán para desarrollar la interfaz en el lenguaje de programación AML.

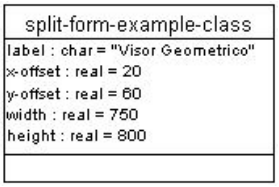
Para la definición de clases que formaran parte de la **interfaz gráfica**:



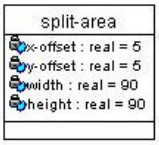
La clase "split-form-with-toolbar-class" será el objeto que agrupe todos los elementos que conformarán la interfaz gráfica. Deberá contar con el método display() que dibujara en pantalla la interfaz gráfica y activara todos los elementos gráficos contenidos por la clase.



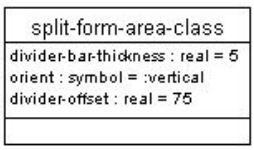
La clase "graphics-area" será una clase de asociación o subobjeto de la clase "split-form-with-toolbar-class". Este subobjeto tendrá la tarea de crear la zona donde será desplegado el canvas sobre la ventana. Esta clase tendrá asociación con la clase "split-form-example-class".



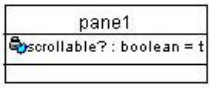
La clase "split-form-example-class" controlara las características generales de la interfaz gráfica, como el texto que aparecerá en la barra de título y la posición y tamaño en la pantalla de la interfaz gráfica.



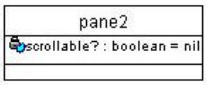
La clase "split-area" será una clase de asociación o subobjeto de la clase "split-form-example-class". Este subobjeto controlara lasa propiedades del área activa sobre la ventana. Esta clase tendrá asociación con la clase "split-form-area-class".



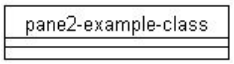
La clase "split-form-area-class" definirá las características de la división de la interfaz gráfica. Una zona contendrá la zona de dibujo ó canvas. La otra zona contendrá el control de gráficos. Esta clase definirá la orientación y grosor de la barra de división, así como el tamaño de un área de separación en la división.



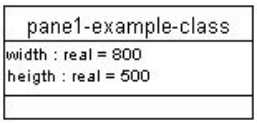
La clase “pane1” será una clase de asociación o subobjeto de la clase “split-form-area-class”. Esta clase controlara la posición en pantalla y contendrá los objetos generados por el canvas. Esta clase tendrá asociación con la clase “pane1-example-class”.



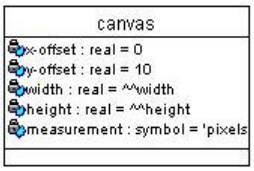
La clase “pane2” será una clase de asociación o subobjeto de la clase “split-form-area-class”. Esta clase controlara la posición en pantalla y contendrá los objetos generados por el control de gráficos. Esta clase tendrá asociación con la clase “pane2-example-class”.



La clase “pane2-example-class” definirá la zona que contendrá el control de gráficos.

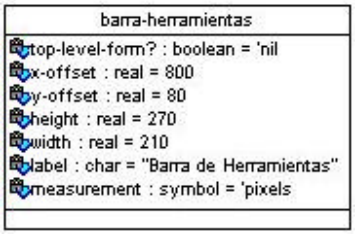


La clase “pane1-example-class” definirá la zona que contendrá la zona de dibujo ó canvas.

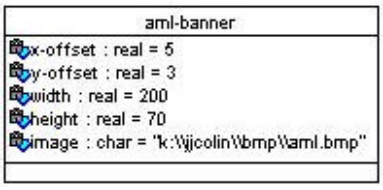


La clase “canvas” será una clase de asociación o subobjeto de la clase “pane1-example-class”. Esta clase controlara las características en pantalla del Canvas o Zona de Dibujo. Esta clase tendrá asociación con la clase “ui-canvas-class”.

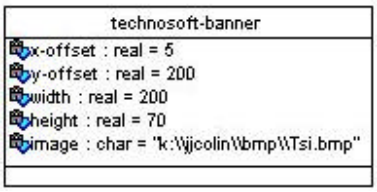
Para la definición de clases que formaran parte de la **Barra de Herramientas**:



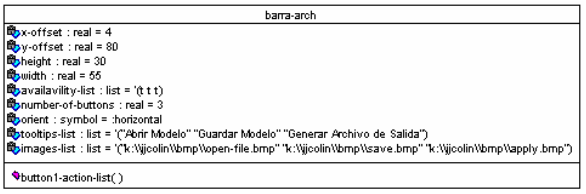
La clase “barra-herramientas" definirá una ventana que contendrá las herramientas adicionales que se desarrollaran para este sistema. Esta clase tendrá una herencia directa con la clase “ui-form-class”. Sus propiedades controlaran la posición, el tamaño y la etiqueta de la barra de herramientas.



La clase “aml-banner” será una clase de asociación o subobjeto de la clase “barra-herramientas”. Esta clase tendrá asociación con la clase “ui-label-class”. Esta clase controlara las características de la etiqueta que en esta ocasión, desplegara un gráfico.



La clase “technosoft-banner” será una clase de asociación o subobjeto de la clase “barra-herramientas”. Esta clase tendrá asociación con la clase “ui-label-class”. Esta clase controlara las características de la etiqueta que en esta ocasión, desplegara un gráfico.



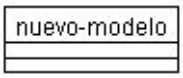
La clase "barra-arch" será una clase de asociación o subobjeto de la clase "barra-herramientas". Esta clase tendrá asociación con la clase "ui-toolbar-class". Esta clase controlara las características de los botones que conforman una barra de herramientas que tendrá las herramientas que activaran los eventos para *abrir modelos* y *guardar modelos*.

La clase "barra-geom-sec" será una clase de asociación o subobjeto de la clase "barra-herramientas". Esta clase tendrá asociación con la clase "ui-toolbar-class". Esta clase controlara las características de los botones que conforman una barra de herramientas que tendrá las herramientas que activaran los eventos para *insertar ranuras, sobrecortes, chaflanes y chaflanes angulares*.

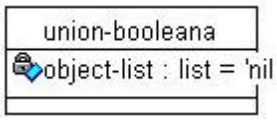
La clase "barra-obj" será una clase de asociación o subobjeto de la clase "barra-herramientas". Esta clase tendrá asociación con la clase "ui-toolbar-class". Esta clase controlara las características de los botones que conforman una barra de herramientas que tendrá las herramientas que activaran los eventos para *insertar una esfera, insertar un polígono rotacional, borrar un objeto, mover un objeto*.

La clase "barra-boolean" será una clase de asociación o subobjeto de la clase "barra-herramientas". Esta clase tendrá asociación con la clase "ui-toolbar-class". Esta clase controlara las características de los botones que conforman una barra de herramientas que tendrá las herramientas que activaran los eventos para *realizar operaciones booleanas (unión, diferencia, intersección)*.

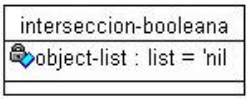
Para la definición de clases que formaran parte de la **Geometría**:



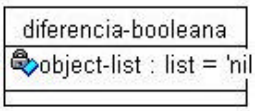
La clase "nuevo-modelo" tendrá una herencia directa con la clase "object", por lo cual será la clase que contendrá todos los objetos geométricos que sean adicionados o modificados en el modelo.



La clase "union-booleana" tiene una herencia directa con la clase "union-object". Al modificar la lista de la propiedad *object-list*, por la lista de objetos seleccionados, esos objetos podrán ser unidos en una operación booleana de unión.

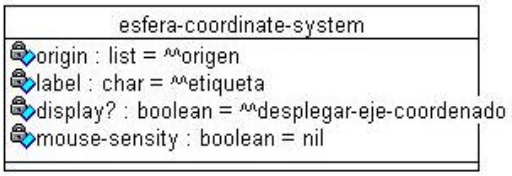


La clase "interseccion-booleana" tiene una herencia directa con la clase "intersection-object". Al modificar la lista de la propiedad *object-list*, por la lista de objetos seleccionados, esos objetos podrán ser intersectados en una operación booleana de intersección.



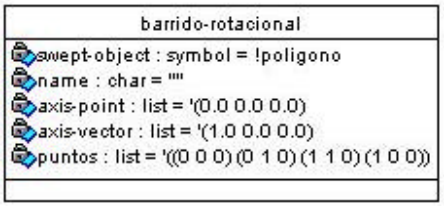
La clase “diferencia-booleana” tiene una herencia directa con la clase “difference-object”. Al modificar la lista de la propiedad *object-list*, por la lista de objetos seleccionados, esos objetos podrán ser diferenciados en una operación booleana de diferencia.

La clase “esfera-object” definirá las características de una esfera. Sus propiedades permitirán definir el nombre de la esfera, el punto de aplicación en el espacio donde será situada la esfera, el radio de la esfera, establecer si la esfera será o no un sólido, el color y tipo de color que presentara la esfera en la zona de dibujo, la etiqueta que presentara en la zona de dibujo y la propiedad de control que permitirá dibujar la esfera y su propio eje coordenado en la zona de dibujo. Esta clase tendrá una herencia directa con la clase “object”.

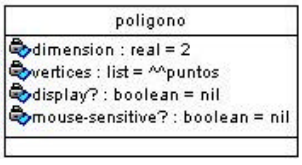


La clase “esfera-coordinate-system” será una clase de asociación o subobjeto de la clase “esfera-object”. Esta clase tendrá asociación con la clase “coordinate-system-class”. Esta clase controlara las propiedades del eje coordenado de la esfera, que será necesario para poder insertar una esfera en el espacio.

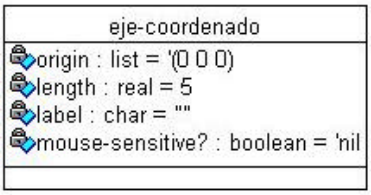
La clase “esfera” será una clase de asociación o subobjeto de la clase “esfera-object”. Esta clase tendrá asociación con la clase “sphere-object”. Esta clase controlara las propiedades de la esfera.



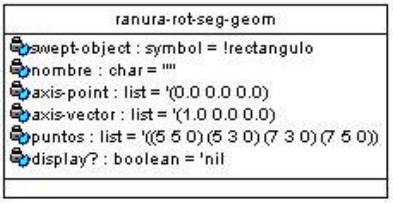
La clase “barrido rotacional” definirá los polígonos que serán barridos alrededor del eje simétrico. Sus propiedades permitirán guardar un apuntador hacia el polígono a rotar, el punto y la dirección del vector para el cual el polígono será rotado alrededor de, y una lista con los puntos en el espacio que conformarán el polígono a rotar.



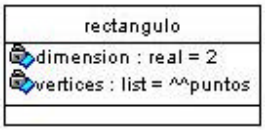
La clase “poligono” será una clase de asociación o subobjeto de la clase “barrido-rotacional”. Esta clase tendrá asociación con la clase “polygon-object”. Esta clase controlara las características del polígono que será rotado.



La clase “eje-coordenado” tendrá una herencia directa con la clase “coordinate-system-class”. Esta clase controlara las propiedades del sistema de ejes coordenados global.

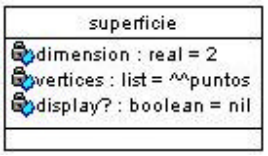


La clase "ranura-rot-seg-geom" tendrá una herencia directa con la clase "rotation-sweep-object". Esta clase definirá los polígonos que serán rotados alrededor del eje simétrico y que conformarán las ranuras.



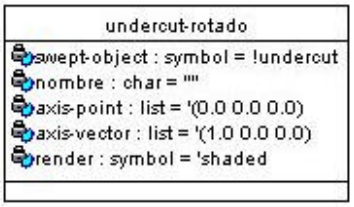
La clase "rectangulo" será una clase de asociación o subobjeto de la clase "ranura-rot-seg-geom". Esta clase tendrá asociación con la clase "polygon-object". Esta clase controlara las características del polígono que generará la ranura.

La clase "undercut-rot-seg-geom" tendrá una herencia directa con la clase "trimmed-surface-object". Esta clase definirá una superficie que será recortada por los objetos definidos como subobjetos, que servirá para crear sobrecortes.

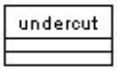


La clase "superficie" será una clase de asociación o subobjeto de la clase "undercut-rot-seg-geom". Esta clase tendrá asociación con la clase "polygon-object". Esta clase controlara las características del polígono que servirá de superficie.

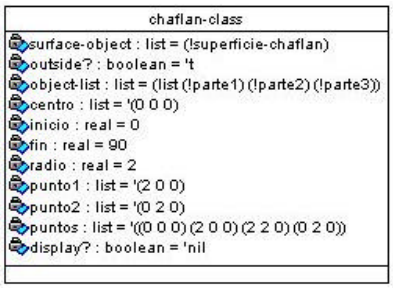
La clase "arco" será una clase de asociación o subobjeto de la clase "undercut-rot-seg-geom". Esta clase tendrá asociación con la clase "arc-ctr-radius-sta-enda". Esta clase definirá un arco al especificar su centro, su ángulo de inicio, su ángulo final y su radio.



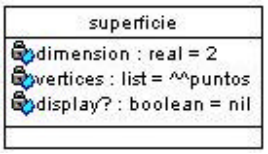
La clase "undercut-rotado" tendrá una herencia directa con la clase "rotation-sweep-object". Esta clase definirá los polígonos que serán rotados alrededor del eje simétrico y que conformarán los sobrecortes.



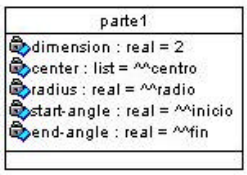
La clase "undercut" será una clase de asociación o subobjeto de la clase "undercut-rotado". Esta clase tendrá asociación con la clase "undercut-rot-seg-geom". Esta clase controlara las características del polígono que generará los sobrecortes.



La clase "chaflan-class" tendrá una herencia directa con la clase "trimmed-surface-object". Esta clase definirá una superficie que será recortada por los objetos definidos como subobjetos, que servirá para crear chaflanes.



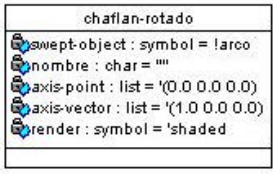
La clase "superficie" será una clase de asociación o subobjeto de la clase "chaflan-class". Esta clase tendrá asociación con la clase "polygon-object". Esta clase controlara las características del polígono que servirá de superficie.



La clase "parte1" será una clase de asociación o subobjeto de la clase "chaflan-class". Esta clase tendrá asociación con la clase "arc-ctr-radius-sta-enda". Esta clase definira un arco al especificar su centro, su ángulo de inicio, su ángulo final y su radio.

La clase "parte2" será una clase de asociación o subobjeto de la clase "chaflan-class". Esta clase tendrá asociación con la clase "line-object". Esta clase conecta dos puntos en un espacio de tres dimensiones.

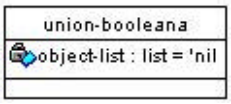
La clase "parte3" será una clase de asociación o subobjeto de la clase "chaflan-class". Esta clase tendrá asociación con la clase "line-object". Esta clase conecta dos puntos en un espacio de tres dimensiones.



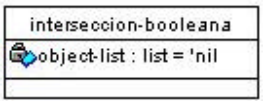
La clase "chaflan-rotado" tendrá una herencia directa con la clase "rotation-sweep-object". Esta clase definirá los polígonos que serán rotados alrededor del eje simétrico y que conformarán los chaflanes.

La clase "arco" será una clase de asociación o subobjeto de la clase "chaflan-rotado". Esta clase tendrá asociación con la clase "chaflan-class". Esta clase controlara las características del polígono que generará los chaflanes.

Para la definición de clases que formaran parte de las **Operaciones Booleanas**:



La clase "union-booleana" realizara la operación booleana de unión de objetos. Tendrá una herencia directa con la clase "union-object". Su propiedad *object-list* guardara una lista de ubicación de objetos a ser unidos.



La clase "interseccion-booleana" realizara la operación booleana de intersección de objetos. Tendrá una herencia directa con la clase "intersection-object". Su propiedad *object-list* guardara una lista de ubicación de objetos a ser intersectados.

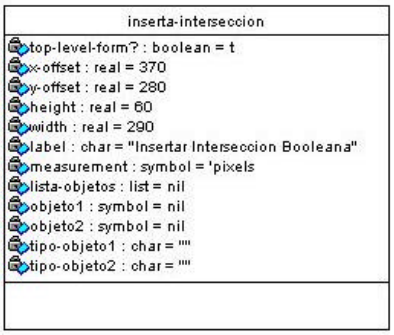
La clase "diferencia-booleana" realizara la operación booleana de diferencia de objetos. Tendrá una herencia directa con la clase "differemce-object". Su propiedad *object-list* guardara una lista de ubicación de objetos a ser diferenciados ó restados.

Para la definición de clases que formaran parte de los **Cuadros de Díalogo**:

La clase "inserta-esfera" definirá un cuadro de díalogo que nos permitirá modificar las características de las esferas que se insertaran. Esta clase tendrá una herencia directa con la clase "ui-form-class". Sus propiedades controlaran la posición, el tamaño y la etiqueta del cuadro de díalogo. Sus subobjetos constaran de etiquetas con campos de edición de texto (generados por una herencia directa de la clase "ui-labeled-field-class"), etiquetas (generados por una herencia directa de la clase "ui-label-class"), campos de opciones (generados por una herencia directa de la clase "ui-option-menu-class") y los botones "Aceptar" y "Cancelar" (generados por una herencia directa de la clase "ui-action-button-class") que en sus eventos llamados "button1-action" contienen el código que permite insertar esferas.

La clase "inserta-union" definirá un cuadro de díalogo que nos permitirá modificar las características de las operaciones boolenanas de unión que se insertaran. Esta clase tendrá una herencia directa con la clase "ui-form-class". Sus propiedades controlaran la posición, el tamaño y la etiqueta del cuadro de díalogo. Sus subobjetos constaran de etiquetas con campos de edición de texto (generados por una herencia directa de la clase "ui-labeled-field-class") y los botones "Aceptar" y "Cancelar" (generados por una herencia directa de la clase "ui-action-button-class") que en sus eventos llamados "button1-action" contienen el código que permite insertar uniones.

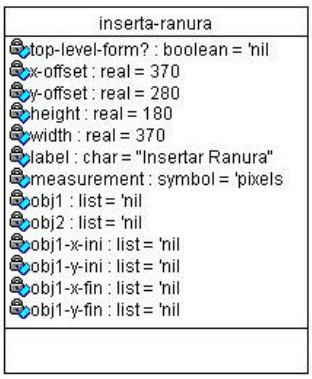
La clase "inserta-diferencia" definirá un cuadro de díalogo que nos permitirá modificar las características de las operaciones booleanas de diferencia que se insertaran. Esta clase tendrá una herencia directa con la clase "ui-form-class". Sus propiedades controlaran la posición, el tamaño y la etiqueta del cuadro de díalogo. Sus subobjetos constaran de etiquetas con campos de edición de texto (generados por una herencia directa de la clase "ui-labeled-field-class") y los botones "Aceptar" y "Cancelar" (generados por una herencia directa de la clase "ui-action-button-class") que en sus eventos llamados "button1-action" contienen el código que permite hacer diferencias de objetos.



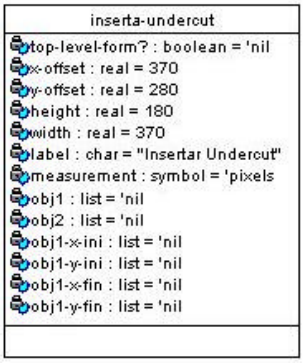
La clase "inserta-interseccion" definirá un cuadro de díalogo que nos permitirá modificar las características de las operaciones booleanas de intersección que se insertaran. Esta clase tendrá una herencia directa con la clase "ui-form-class". Sus propiedades controlaran la posición, el tamaño y la etiqueta del cuadro de díalogo. Sus subobjetos constaran de etiquetas con campos de edición de texto (generados por una herencia directa de la clase "ui-labeled-field-class") y los botones "Aceptar" y "Cancelar" (generados por una herencia directa de la clase "ui-action-button-class") que en sus eventos llamados "button1-action" contienen el código que permite hacer intersección entre objetos.

La clase "inserta-barrido" definirá un cuadro de díalogo que nos permitirá modificar las características de los polígonos barridos que se insertaran. Esta clase tendrá una herencia directa con la clase "ui-form-class". Sus propiedades controlaran la posición, el tamaño y la etiqueta del cuadro de díalogo. Sus subobjetos constaran de etiquetas con campos de edición de texto (generados por una herencia directa de la clase "ui-labeled-field-class"), etiquetas (generados por una herencia directa de la clase "ui-label-class"), botones de selección de opción (generados por una herencia directa de la clase "ui-radio-button-class"), campos de entrada de datos (generados por una herencia directa de la clase "ui-typein-field-class") y los botones "Aceptar" y "Cancelar" (generados por una herencia directa de la clase "ui-action-button-class") que en sus eventos llamados "button1-action" contienen el código que permite insertar poligonos barridos.

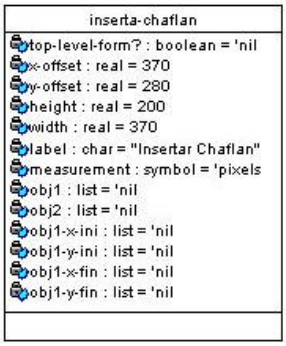
La clase "mover-objeto-window" definirá un cuadro de díalogo que nos permitirá modificar la posición de los objetos sobre el eje simétrico. Esta clase tendrá una herencia directa con la clase "ui-form-class". Sus propiedades controlaran la posición, el tamaño y la etiqueta del cuadro de díalogo. Sus subobjetos constaran de etiquetas (generados por una herencia directa de la clase "ui-label-class"), botones de selección de opción (generados por una herencia directa de la clase "ui-radio-button-class") y los botones "Aceptar" y "Cancelar" (generados por una herencia directa de la clase "ui-action-button-class") que en sus eventos llamados "button1-action" contienen el código que permite cambiar la posición de objetos.



La clase "inserta-ranura" definirá un cuadro de díalogo que nos permitirá modificar las características de las ranuras que se insertaran. Esta clase tendrá una herencia directa con la clase "ui-form-class". Sus propiedades controlaran la posición, el tamaño y la etiqueta del cuadro de díalogo. Sus subobjetos constaran de etiquetas con campos de edición de texto (generados por una herencia directa de la clase "ui-labeled-field-class"), etiquetas (generados por una herencia directa de la clase "ui-label-class") y los botones "Aceptar" y "Cancelar" (generados por una herencia directa de la clase "ui-action-button-class") que en sus eventos llamados "button1-action" contienen el código que permite insertar ranuras.



La clase “inserta-undercut" definirá un cuadro de díalogo que nos permitirá modificar las características de los sobrecortes que se insertaran. Esta clase tendrá una herencia directa con la clase “ui-form-class”. Sus propiedades controlaran la posición, el tamaño y la etiqueta del cuadro de díalogo. Sus subobjetos constaran de etiquetas con campos de edición de texto (generados por una herencia directa de la clase “ui-labeled-field-class”), etiquetas (generados por una herencia directa de la clase “ui-label-class”) y los botones “Aceptar” y “Cancelar” (generados por una herencia directa de la clase “ui-action-button-class”) que en sus eventos llamados “button1-action” contienen el código que permite insertar sobrecortes.



La clase “inserta-chaflan" definirá un cuadro de díalogo que nos permitirá modificar las características de los chaflanes que se insertaran. Esta clase tendrá una herencia directa con la clase “ui-form-class”. Sus propiedades controlaran la posición, el tamaño y la etiqueta del cuadro de díalogo. Sus subobjetos constaran de etiquetas con campos de edición de texto (generados por una herencia directa de la clase “ui-labeled-field-class”), etiquetas (generados por una herencia directa de la clase “ui-label-class”), botones de selección de opción (generados por una herencia directa de la clase “ui-radio-button-class”) y los botones “Aceptar” y “Cancelar” (generados por una herencia directa de la clase “ui-action-button-class”) que en sus eventos llamados “button1-action” contienen el código que permite insertar chaflanes.

La clase “inserta-chaflan-angular" definirá un cuadro de díalogo que nos permitirá modificar las características de los chaflanes angulares que se insertaran. Esta clase tendrá una herencia directa con la clase “ui-form-class”. Sus propiedades controlaran la posición, el tamaño y la etiqueta del cuadro de díalogo. Sus subobjetos constaran de etiquetas con campos de edición de texto (generados por una herencia directa de la clase “ui-labeled-field-class”), etiquetas (generados por una herencia directa de la clase “ui-label-class”), botones de selección de opción (generados por una herencia directa de la clase “ui-radio-button-class”) y los botones “Aceptar” y “Cancelar” (generados por una herencia directa de la clase “ui-action-button-class”) que en sus eventos llamados “button1-action” contienen el código que permite insertar chaflanes angulares.

La clase “gen-arch-sal" definirá un cuadro de díalogo que nos permitirá definir las características del archivo de salida a generar para usarse en conjunto con el proyecto SADET *(ver sección 3.2)*. Esta clase tendrá una herencia directa con la clase “ui-form-class”. Sus propiedades controlaran la posición, el tamaño y la etiqueta del cuadro de díalogo. Sus subobjetos constaran de etiquetas (generados por una herencia directa de la clase “ui-label-class”), campos de entrada de datos (generados por una herencia directa de la clase “ui-typein-field-class”), botones de selección de opción (generados por una herencia directa de la clase “ui-radio-button-class”) y los botones “Aceptar” y “Cancelar” (generados por una herencia directa de la clase “ui-action-button-class”) que en sus eventos llamados “button1-action” contienen el código que genera el archivo de salida.

3.3.3 Clases de AML y del Sistema Paramétrico.

La definición de clases permite observar las características de las clases que conformarán el sistema. Las relaciones entre las clases permiten observan como se relacionaran estas entre si, así como las propiedades heredadas y asociadas por las clases. El funcionamiento del lenguaje AML, de la unidad gráfica y del sistema paramétrico, se ejemplifican mejor en los diagramas de clases.

3.3.3.1 Clases Nativas de AML.

Las clases nativas de AML son las clases padres que definen al lenguaje. Estas clases no pueden ser instanciadas por el programador ni por el usuario, en algunos casos, se pueden instanciar clases que hereden de estas clases.

En la **figura 3.4.**, se muestra el diagrama de clases general del lenguaje AML. Como se puede observar, la clase "object" es la clase de la cual las clases *layer, position, sensitivity y solid* heredan. Estas clases controlan las capas, posición, sensitividad y la propiedad de ser sólido de cualquier elemento geométrico instanciado en AML. La clase *graphic-object* tiene una herencia con *layer-class y position-object*, lo cual le da las propiedades necesarias para controlar objetos geométricos. Estas clases no deben ser modificadas por el usuario.

Cuando la propiedad *solid?* De cualquier elemento geométrico es verdadera, este será un sólido. De lo contrario, la geometría será creada como un cascarón.

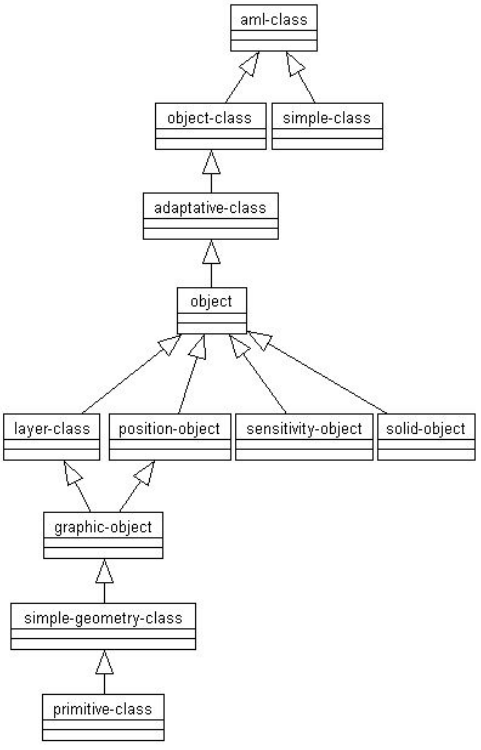


Figura 3. 4.- Diagrama de clases del lenguaje AML.

El diagrama mostrado en la **figura 3.5.** muestra cómo están relacionadas las clases que son utilizadas para instanciar elementos geometricos. Las clases que heredan de la clase *geom-object* , son clases que manejan las transformaciones sobre los elementos geométricos instanciados. Todas las clases que instancian elementos geometricos heredan de la clase *primitive-class*, pero los elementos geométricos que pueden estar representados en tres dimensiones presentan una herencia con la clase *solid-object*, estas clases presentando, una herencia múltiple. Estas clases no deben ser modificadas por el usuario.

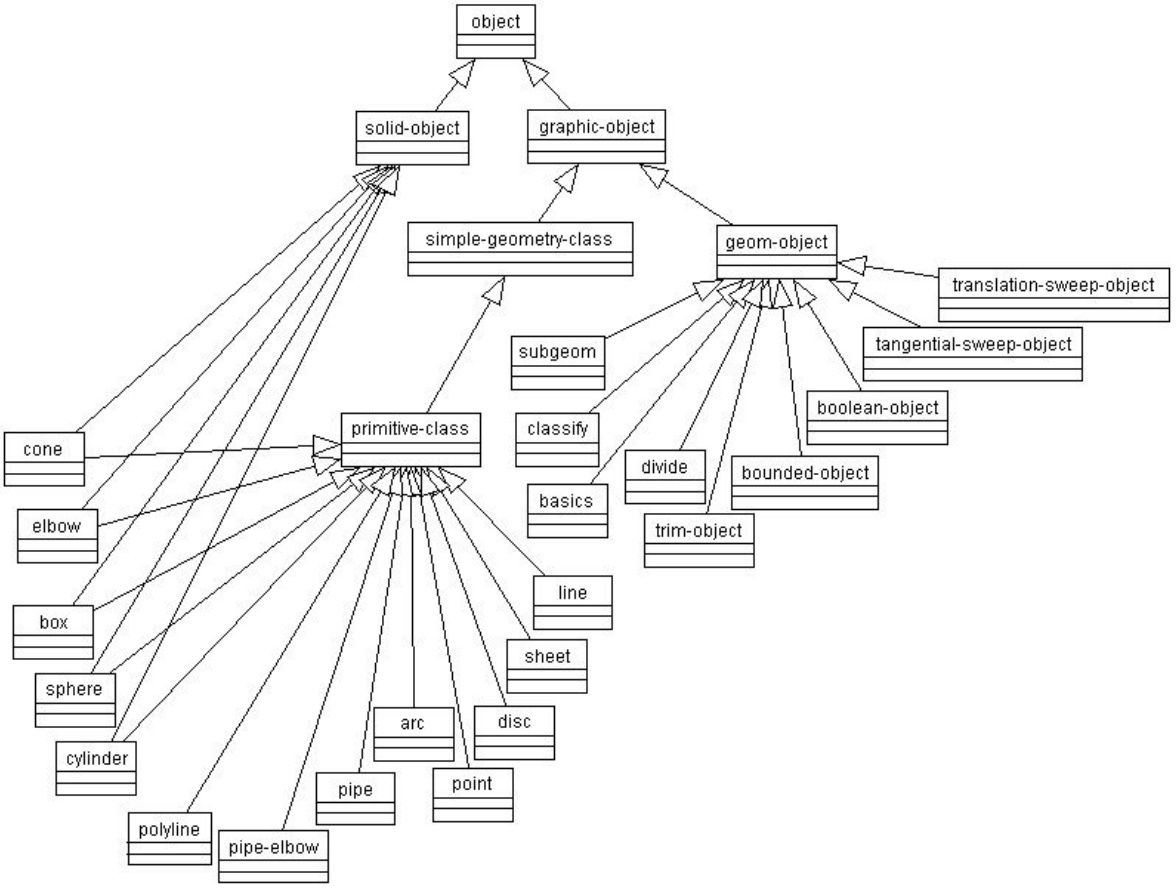


Figura 3. 5.- Diagrama de clases de Geometría en AML.

El siguiente diagrama mostrado en la **figura 3.6.** muestra como estarán relacionadas las clases que conforman los elementos de las interfaz gráfica de usuario GUI de AML. Estas clases permiten instanciar objetos gráficos para interactuar con el sistema. Entre estas clases se encuentra *ui-combo-box*, que permite instanciar un cuadro de opciones desplegables, *ui-fields-class* que permite instanciar campos de edición y entrada de datos y *ui-action-button-class*, que permite instanciar botones con la posibilidad de poder activar métodos para realizar acciones secundarias. Estas clases no deben ser modificadas por el usuario, pero se pueden instanciar objetos con herencia especificada a estas clases.

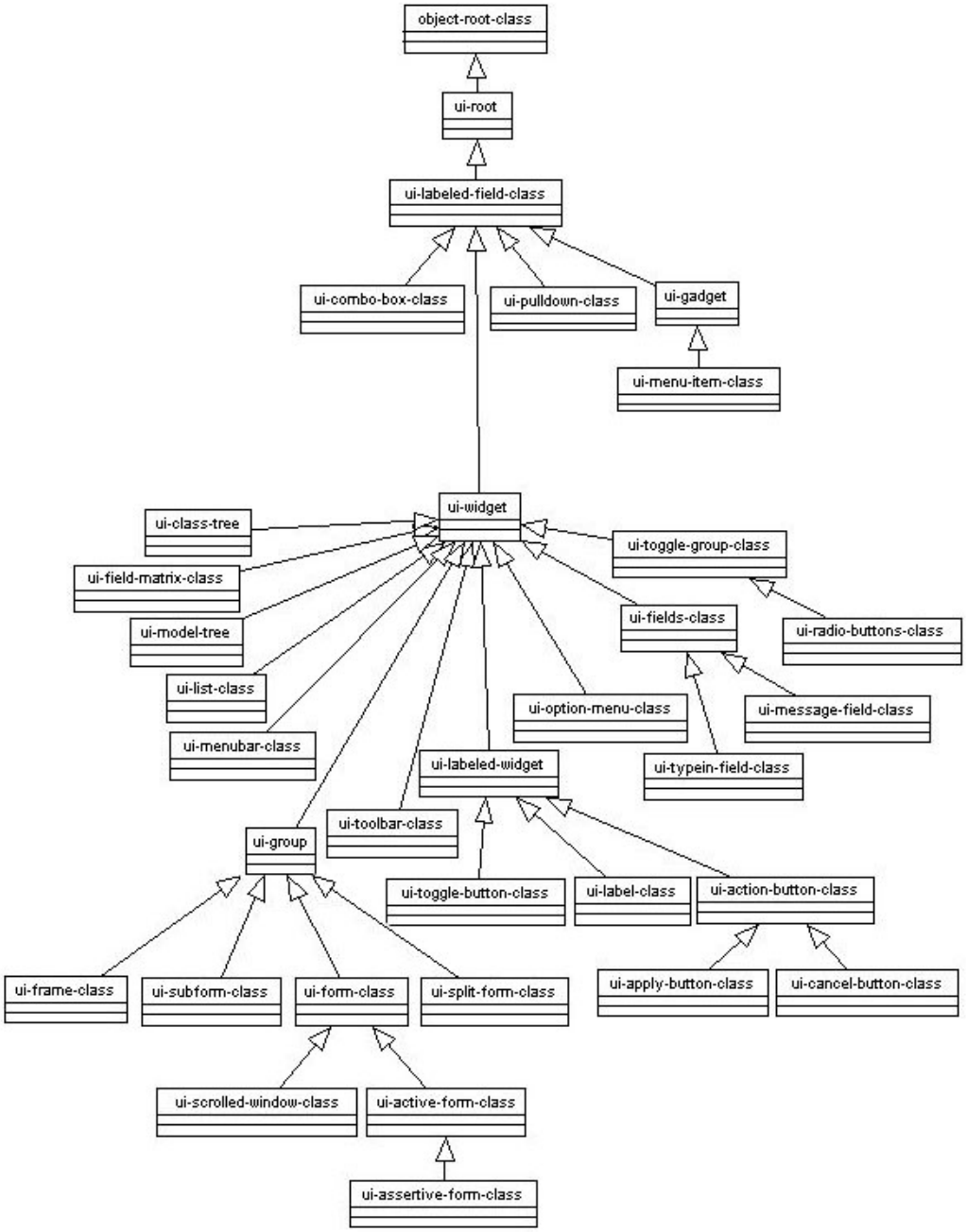


Figura 3. 6.- Diagrama General de clases que definen los elementos de interfaz gráfica GUI.

3.3.3.2 Clases que forman parte del sistema.

El diagrama mostrado en la **figura 3.7.** muestra como estarán relacionadas las clases que son utilizadas para instanciar la interfaz gráfica. Las clases que heredan de la clase *ui-resizable-pane-mixin* instanciaran un panel dividido. En el primer panel, se instanciara como subobjeto el canvas , mientras que en el segundo panel, se instanciara como subobjeto el control de gráficos. Esto se puede ver, debido a la asociación de *pane1-example-class* con *ui-canvas-class* y de *pane2-example-class* con *ui-graphic-control-form-class*, respectivamente. Las características de despliegue serán controladas por las clases split-form-area-class y *split-form-example-class*, por sus herencias con las clases *ui-splitform-class* y *ui-form-class*, respectivamente **(Ver figura 3.7.)**.

El diagrama mostrado en la **figura 3.8.** muestra como estarán relacionadas las clases que son utilizadas para instanciar la barra de herramientas del sistema que se desarrollará en AML. Esta clase instancia una ventana que contiene botones para desencadenar acciones. La clase *ui-form-class* será la clase que proporcione la ventana que contendrá el resto de los subobjetos. La clase *ui-label-class* permitirá instanciar zonas de imágenes para desplegar logotipos. La clase *ui-toolbar-class* instanciará botones de herramientas que sus propiedades permitirán poder usar texto o imágenes para etiquetarlos y cada uno tendrá asociado un método *button1-action ( )* que permitirá ejecutar código adicional. **(Ver figura 3.8.).**

El siguiente diagrama mostrado en la **figura 3.9.** muestra como estarán relacionadas las clases que son utilizadas para instanciar los objetos geométricos *esfera, nuevo-modelo* y *eje coordenado*. Éstos tres objetos geométricos siempre heredaran de la clase *object*. La clase *nuevo-model* será la clase en la cual se agregaran todas las nuevas instancias de objetos geométricos, según sean solicitados. La clase *eje-coordenado* tendrá una herencia con la clase *coordinate-system-class*. La clase *esfera-objeto* tendrá dos asociaciones como subobjetos. La primera asociación será con la clase *sphere-object*, que será la que instanciara la esfera. La segunda asociación, será con *coordinate-system-class*, que instanciara un eje coordenado relativo en el centro de la esfera, pero que gracias a sus propiedades, este eje coordenado no será visible al usuario. Esto es necesario para poder insertar la esfera en el canvas, dado una coordenada. **(Ver figura 3.9.).**

El siguiente diagrama mostrado en la **figura 3.10.** muestra como estarán relacionadas las clases que son utilizadas para instanciar operaciones booleanas de unión, intersección y diferencia de objetos geométricos. La clase union-booleana tendrá una herencia con la clase union-object. La clase diferencia-booleana tendrá una herencia con la clase difference-object. La clase interseccion-booleana tendrá una herencia con la clase intersection-object. Las clases union-object, difference-object, intersection-object heredan de la clase bolean-object. Estas clases no deben ser modificadas por el usuario, pero se puede definir una clase que herede de una clase "object". **(Ver figura 3.10.).**

El diagrama mostrado en la **figura 3.11.** muestra como estarán relacionadas las clases que son utilizadas para instanciar características secundarias a la geometría. Para generar una característica secundaria (ó un acabado final), será necesario crear un polígono con las características necesarias, posicionarlo en el punto de aplicación deseado y girarlo alrededor del eje de simetría. Se utilizará clases que tendrán asociación con la clase *polygon-object* que generarán polígonos cerrados como superficies, también se utilizarán clases que tendrán asociación con la clase *arc-ctr-radius-sta-enda* (para instanciar arcos) y también se clases que tendrán asociación con la clase *line-object* (para generar líneas), en conjunto estas dos ultimas permitirán instanciar polígonos con una configuración diferente.

Con estas clases que instanciaran polígonos, se usaran clases que tendrán asociación con la clase *trimmed-surface-object* que permitirán recortar las superficies utilizando la forma de los polígonos como guías, y utilizando asociaciones con la clase *rotation-sweep-object*, rotaremos esa superficie alrededor del eje de simetría para obtener un sólido. Debido a la naturaleza de la característica secundaria, se decidirá que operación booleana que se aplicara (por lo general, unión ó diferencia), pero eso sólo sucederá en el código del evento que insertara la característica secundaria, y no en la definición de la clase. **(Ver figura 3.11.).**

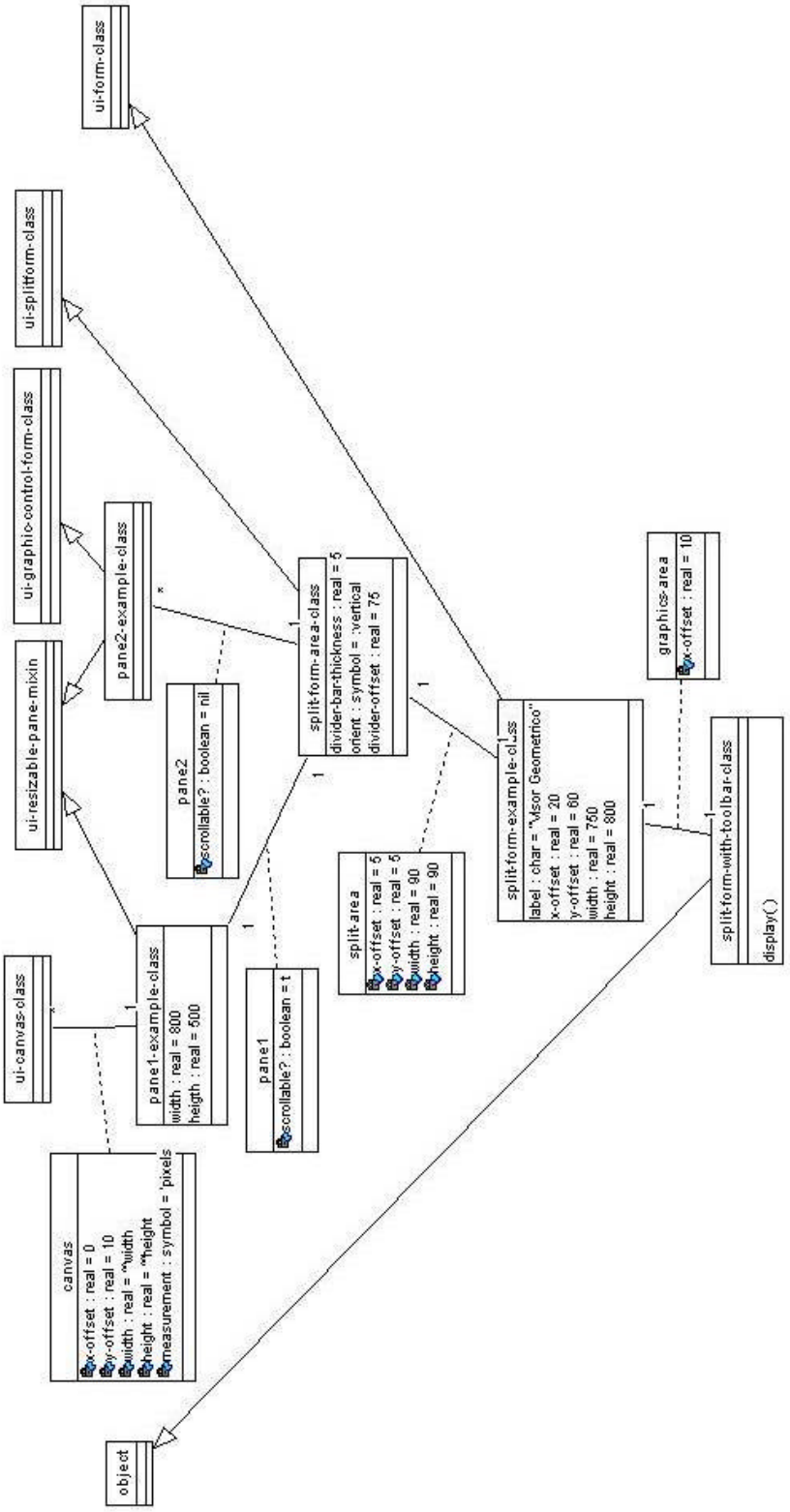


Figura 3. 7.- Diagrama de clases de la zona de dibujo ó Canvas.

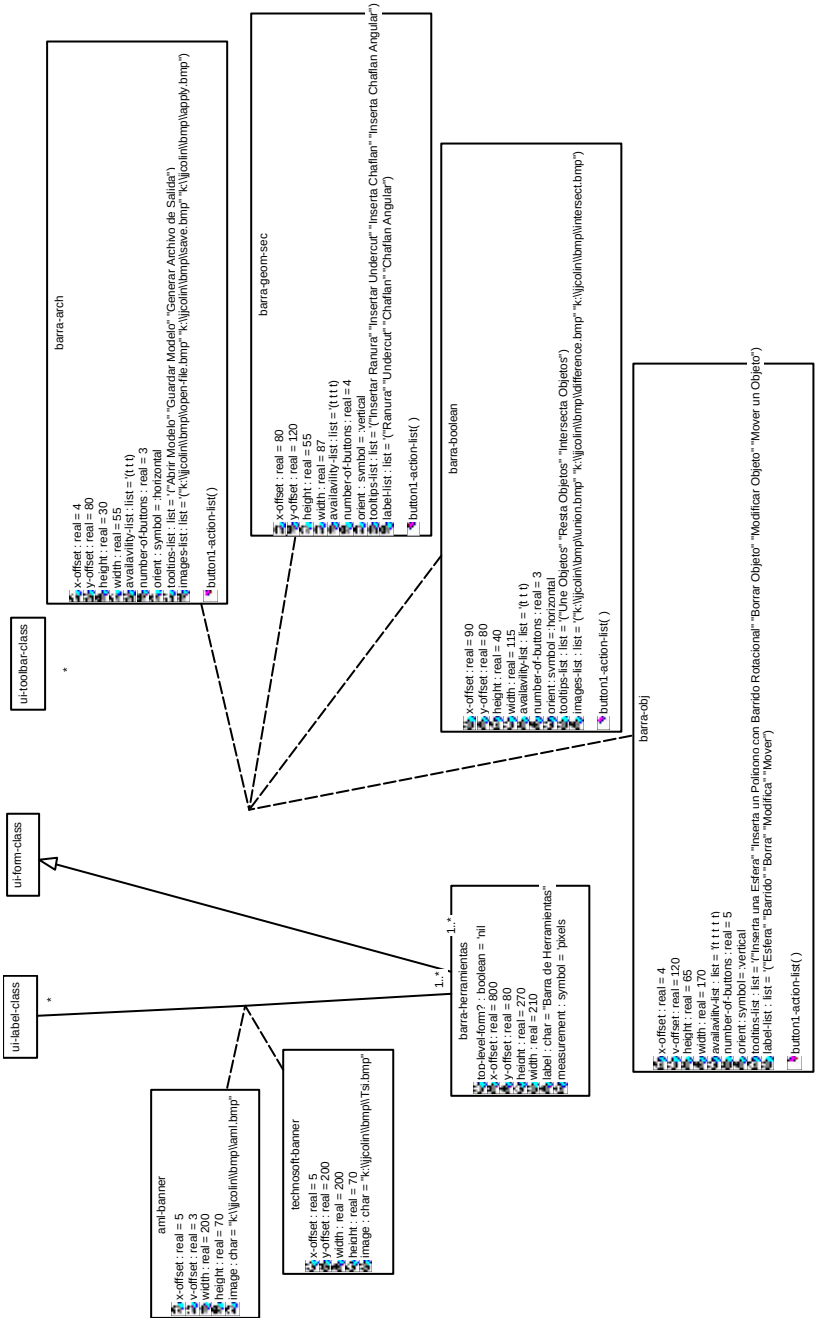


Figura 3. 8.- Diagrama de clases de la barra de herramientas.

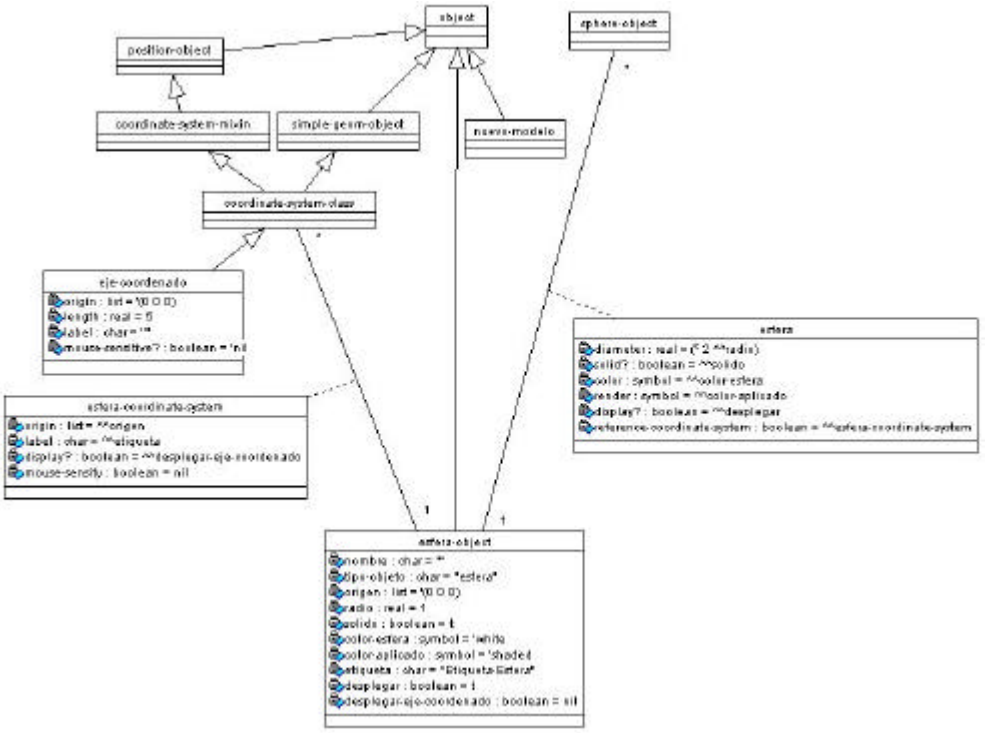


Figura 3. 9.- Diagrama de clases del nuevo modelo, eje coordenado y esfera.

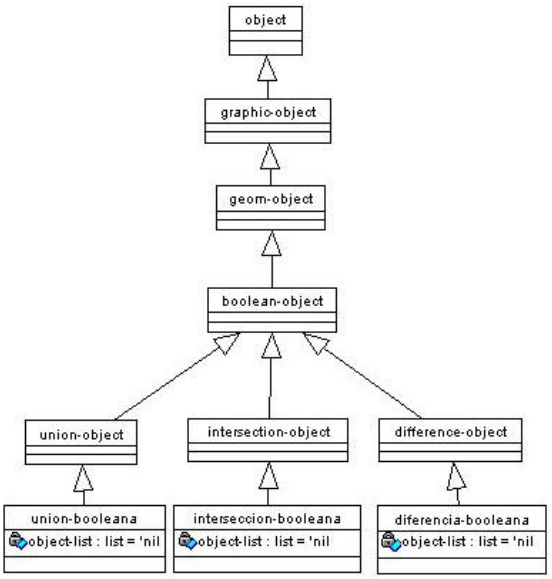


Figura 3. 10.- Diagrama de clases de Operaciones Booleanas.

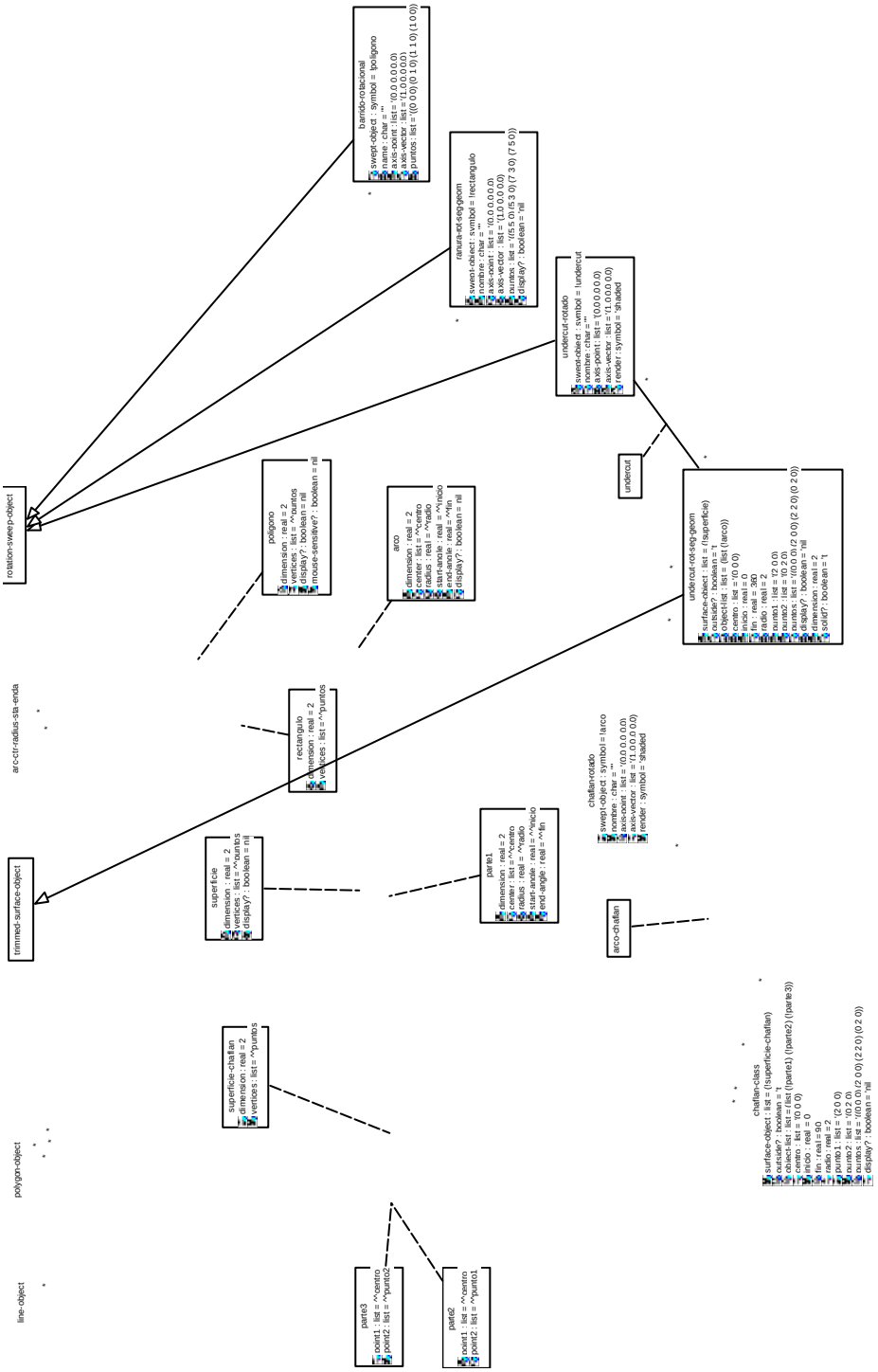


Figura 3. 11.- Diagrama de clases de Geometría y Características Secundarias.

El diagrama mostrado en la **figura 3.12.** muestra como estarán relacionadas las clases que son utilizadas para generar un archivo de salida basado en información geométrica del modelo. Esta clase instancia una ventana que solicitara datos para generar el archivo de salida. La clase *ui-form-class* será la clase que proporcione la ventana que contendrá el resto de los subobjetos. La clase *ui-label-class* permitirá instanciar zonas de texto para mensajes. La clase *ui-typein-field-class* permitirá instanciar campos para la entrada de datos. La clase *ui-action-button-*class permitirá instanciar botones con el método *button1-action( )* que permitirá ejecutar código adicional. La clase *ui-radio-buttons-class* permitirá instanciar una zona de botones de radio, el cual sólo permitirá seleccionar una sola opción de *n* opciones.

El método *button1-action( )* de la clase *busca-arch* abre el cuadro de díalogo para seleccionar el nombre y ruta del archivo a abrir y los guarda en el contenido de la clase *data-arch-text-box.* El método *button1-action( )* de la clase *apply* obtiene los datos de los contenidos de la clase *ui-labeled-field-class* , y si es necesario, abre el archivo seleccionado. Al finalizar, se actualiza la pantalla. El método *button1-action( )* de la clase *close* borra el objeto del cual fue llamado, la clase *gen-arch-sal*. **(Ver figura 3.12.).**

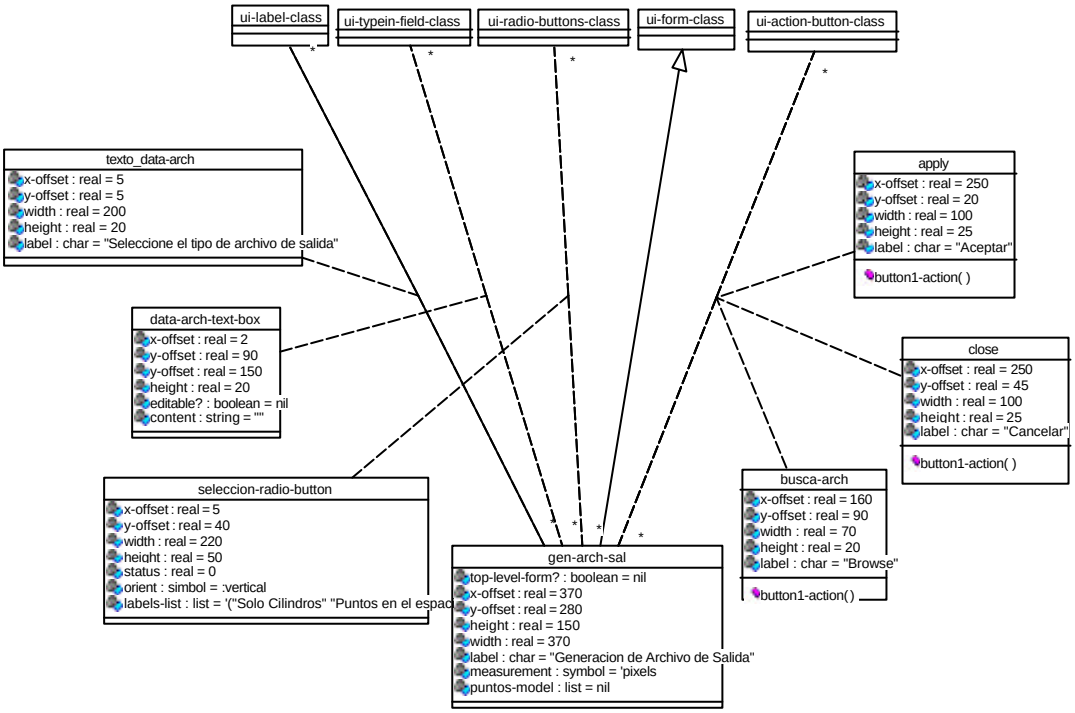


Figura 3. 12.- Diagrama de clases de interfaz "Genera Archivo de Salida".

El diagrama mostrado en la **figura 3.13.** muestra como estarán relacionadas las clases que son utilizadas para instanciar en el nuevo modelo un polígono barrido alrededor del eje de simetría. Esta clase instancia una ventana que solicitara datos para insertar un polígono barrido. La clase *ui-form-class* será la clase que proporcione la ventana que contendrá el resto de los subobjetos. La clase *ui-label-class* permitirá instanciar zonas de texto para mensajes. La clase *ui-typein-field-class* permitirá instanciar campos para la entrada de datos. La clase *ui-labeled-field-class* permitirá instanciar un campo de texto para la entrada de datos y una etiqueta para identificar el campo de texto. La clase *ui-action-button*-class permitirá instanciar botones con el método *button1-action ( )* que permitirá ejecutar código adicional. La clase *ui-radio-buttons-class* permitirá instanciar una zona de botones de radio, el cual sólo permitirá seleccionar una sola opción de *n* opciones.

El método *button1-action( )* de la clase *seleccion-radio-button* verifica que opción esta seleccionada. Si la opción "Editar Puntos" esta seleccionada, sólo los campos necesarios para editar puntos están disponibles. Si la opción "Leer archivo de Puntos" esta seleccionada, sólo los campos necesarios para leer el archivo de puntos esta disponible y permite introducir el nombre y la ruta física del archivo a leer.

El método *button1-action( )* de la clase *busca-arch* abre el cuadro de díalogo para seleccionar el nombre y ruta del archivo a abrir y los guarda en el contenido de la clase *data-arch-text-box.* El método *button1-action( )* de la clase *apply* obtiene los datos de los contenidos de la clase *ui-labeled-field-class* , y si es necesario, abre el archivo de puntos seleccionado y guarda una lista con los puntos, arma el polígono en el espacio y lo inserta en el nuevo modelo, también inserta la clase nueva asociada a la clase *barrido-rotacional* y modifica sus propiedades para rotar el nuevo polígono. Al finalizar, se actualiza la pantalla. El método *button1-action( )* de la clase *close* borra el objeto del cual fue llamado, la clase *inserta-barrido*. **(Ver figura 3.13.).**

El diagrama mostrado en la **figura 3.14.** muestra como estarán relacionadas las clases que son utilizadas para instanciar en el nuevo modelo una esfera. Esta clase instancia una ventana que solicitara datos para insertar una esfera. La clase *ui-form-class* será la clase que proporcione la ventana que contendrá el resto de los subobjetos. La clase *ui-label-class* permitirá instanciar zonas de texto para mensajes. La clase *ui-option-menu-class* permitirá instanciar una cuado combo de opciones desplegables para seleccionar una opción de *n* opciones. La clase *ui-labeled-field-class* permitirá instanciar un campo de texto para la entrada de datos y una etiqueta para identificar el campo de texto. La clase *ui-action-button*-class permitirá instanciar botones con el método *button1-action ( )* que permitirá ejecutar código adicional. La clase *ui-radio-buttons-class* permitirá instanciar una zona de botones de radio, el cual sólo permitirá seleccionar una sola opción de *n* opciones.

El método *button1-action( )* de la clase *apply* obtiene los datos de los contenidos de la clase *ui-labeled-field-class* . Éstos datos modificaran las propiedades de la clase nueva que será insertada en el nuevo modelo, con herencia a la clase *esfera-objeto*. Al finalizar, se actualiza la pantalla. El método *button1-action( )* de la clase *close* borra el objeto del cual fue llamado, la clase *inserta-esfera*. **(Ver figura 3.14.).**

El diagrama mostrado en la **figura 3.15.** muestra como estarán relacionadas las clases que son utilizadas para instanciar en el nuevo mo delo la unión entre objetos. Esta clase instancia una ventana que solicitara el nombre nuevo con el que será referenciada la nueva unión. La clase *ui-form-class* será la clase que proporcione la ventana que contendrá el resto de los subobjetos. La clase *ui-labeled-field-class* permitirá instanciar un campo de texto para la entrada de datos y una etiqueta para identificar el campo de texto. La clase *ui-action-button*-class permitirá instanciar botones con el método *button1-action ( )* que permitirá ejecutar código adicional.

El método *button1-action( )* de la clase *apply* obtiene los datos de los contenidos de la clase *ui-labeled-field-class* . Éstos datos modificaran las propiedades de la clase nueva que será insertada en el nuevo modelo, con herencia a la clase *union-booleana*. Al finalizar, se actualiza la pantalla. El método *button1-action( )* de la clase *close* borra el objeto del cual fue llamado, la clase *inserta-union*. **(Ver figura 3.15.).**

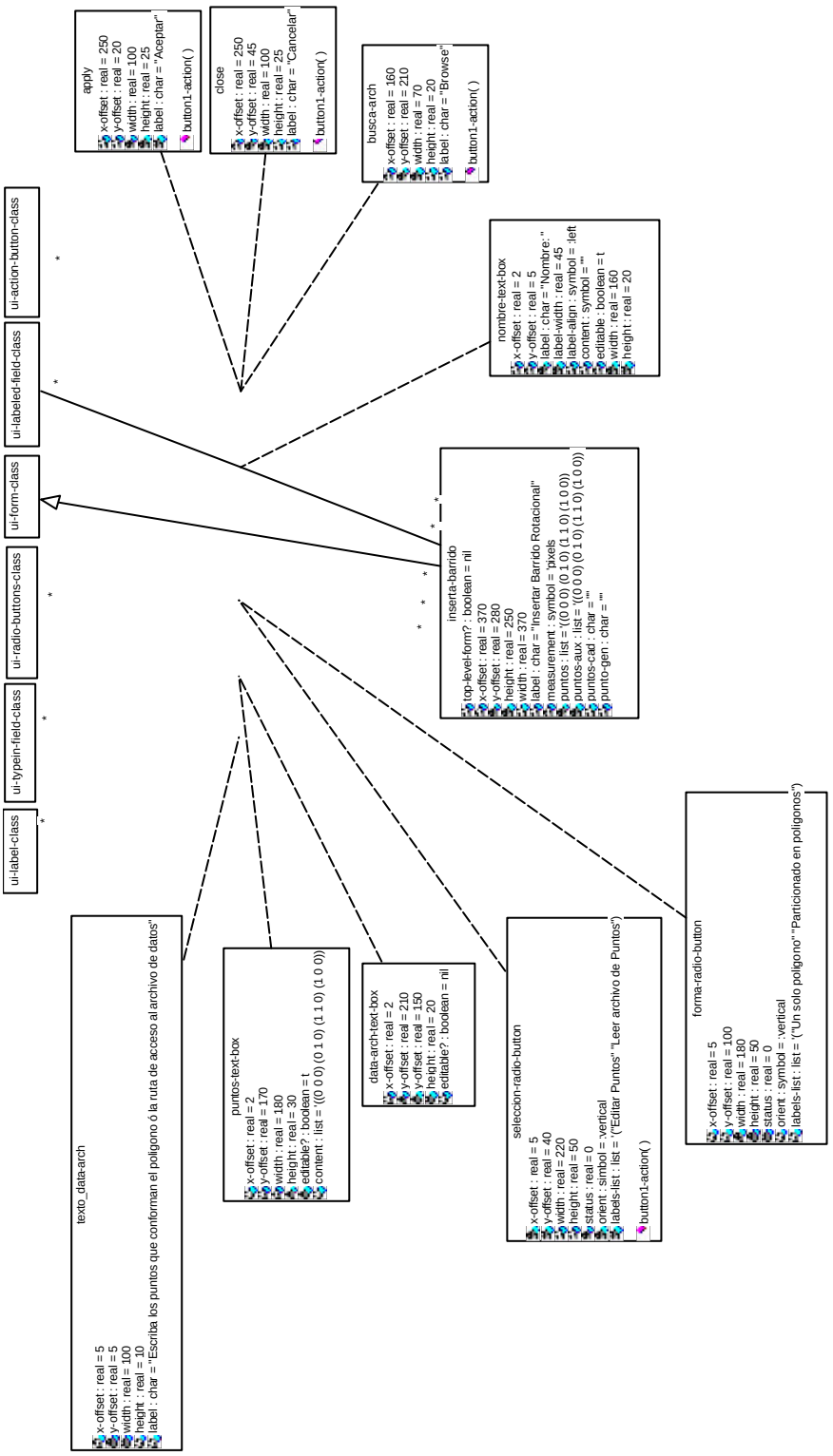


Figura 3. 13.- Diagrama de clases de interfaz “inserta barrido”.

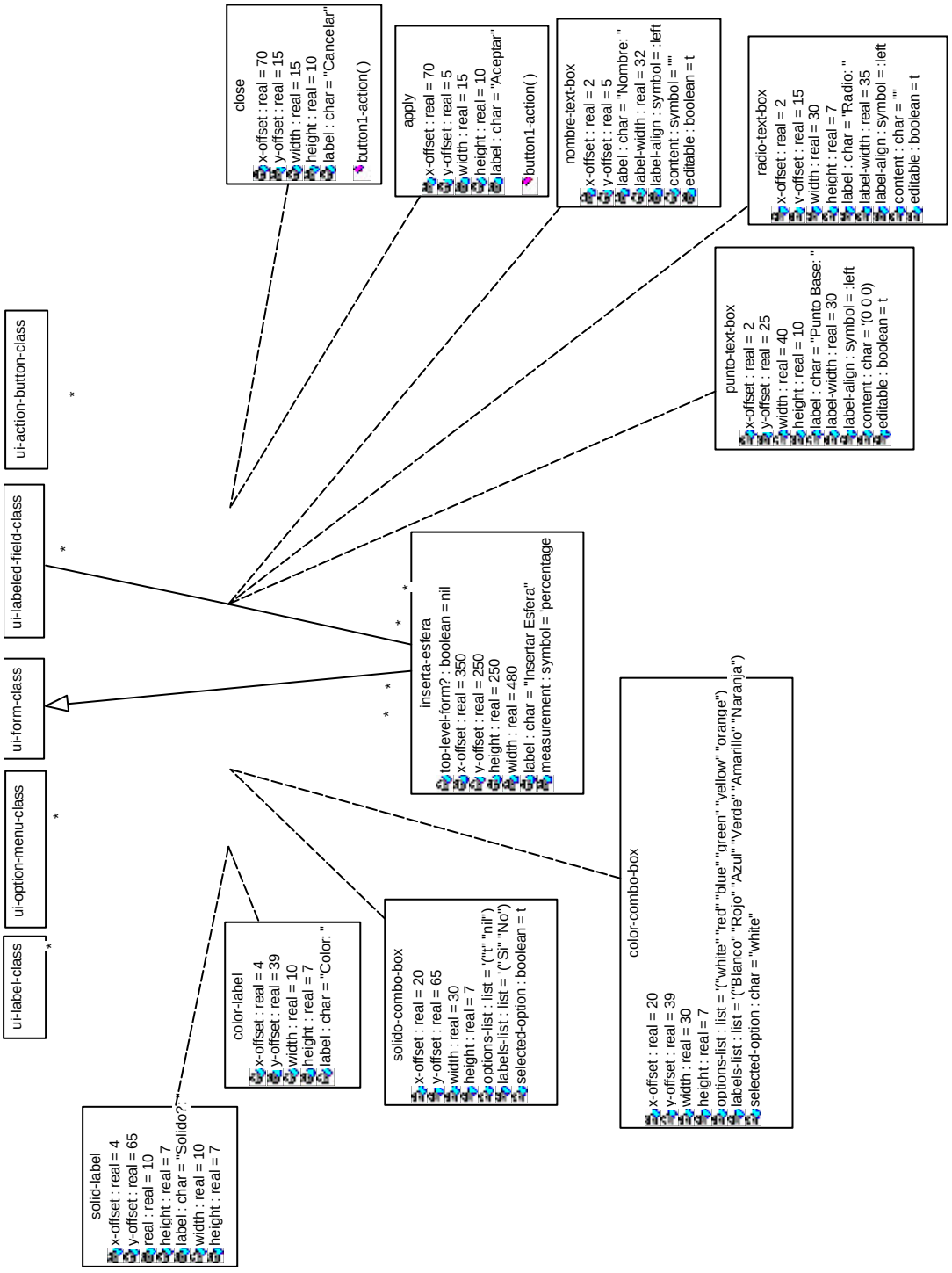


Figura 3. 14.- Diagrama de clases de interfaz “inserta Esfera”.

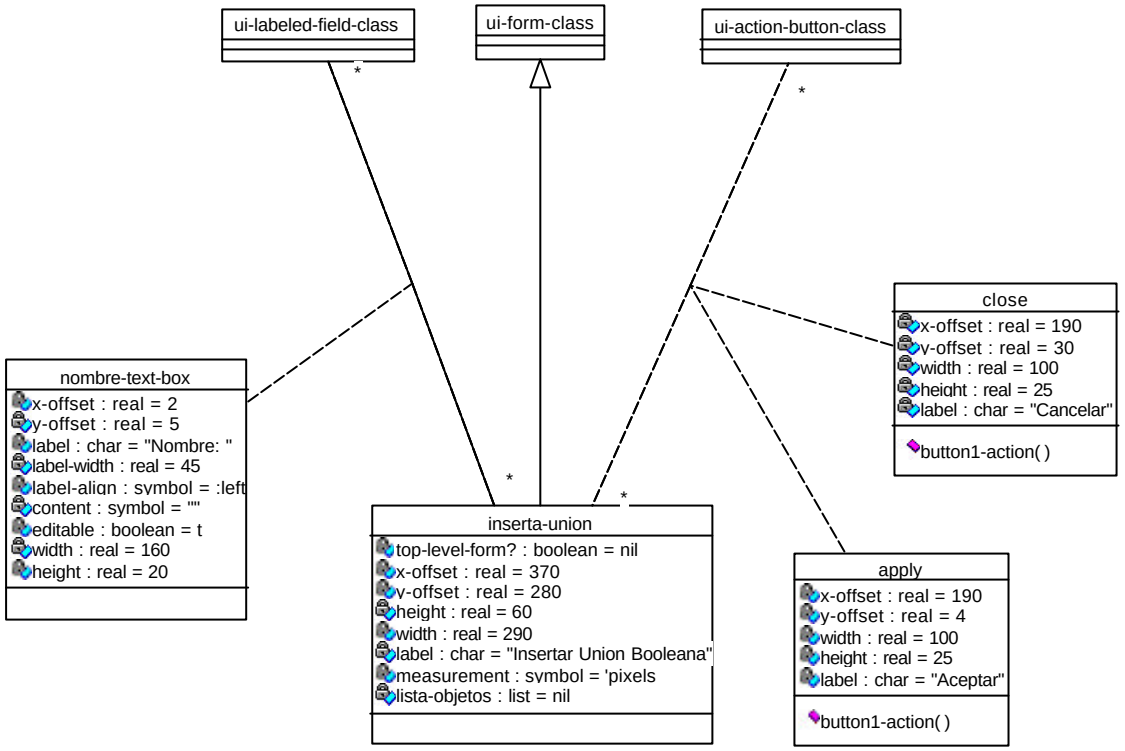


Figura 3. 15.- Diagrama de clases de interfaz "inserta unión".

El diagrama mostrado en la **figura 3.16.** muestra como estarán relacionadas las clases que son utilizadas para instanciar en el nuevo modelo la intersección entre objetos. Esta clase instancia una ventana que solicitara el nombre nuevo con el que será referenciada la nueva intersección. La clase *ui-form-class* será la clase que proporcione la ventana que contendrá el resto de los subobjetos. La clase *ui-labeled-field-class* permitirá instanciar un campo de texto para la entrada de datos y una etiqueta para identificar el campo de texto. La clase *ui-action-button*-class permitirá instanciar botones con el método *button1-action( )* que permitirá ejecutar código adicional.

El método *button1-action( )* de la clase *apply* obtiene los datos de los contenidos de la clase *ui-labeled-field-class* . Éstos datos modificaran las propiedades de la clase nueva que será insertada en el nuevo modelo, con herencia a la clase *interseccion-booleana.* Al finalizar, se actualiza la pantalla. El método *button1-action( )* de la clase *close* borra el objeto del cual fue llamado, la clase *inserta-interseccion*. **(Ver figura 3.16.).**

El diagrama mostrado en la **figura 3.17.** muestra como estarán relacionadas las clases que son utilizadas para instanciar en el nuevo modelo la diferencia entre objetos. Esta clase instancia una ventana que solicitara el nombre nuevo con el que será referenciada la nueva diferencia. La clase *ui-form-class* será la clase que proporcione la ventana que contendrá el resto de los subobjetos. La clase *ui-labeled-field-class* permitirá instanciar un campo de texto para la entrada de datos y una etiqueta para identificar el campo de texto. La clase *ui-action-button*-class permitirá instanciar botones con el método *button1-action( )* que permitirá ejecutar código adicional.

El método *button1-action( )* de la clase *apply* obtiene los datos de los contenidos de la clase *ui-labeled-field-class* . Éstos datos modificaran las propiedades de la clase nueva que será insertada en el nuevo modelo, con herencia a la clase *diferencia-booleana.* Al finalizar, se actualiza la pantalla. El método *button1-action( )* de la clase *close* borra el objeto del cual fue llamado, la clase *inserta-diferencia*. **(Ver figura 3.17.).**

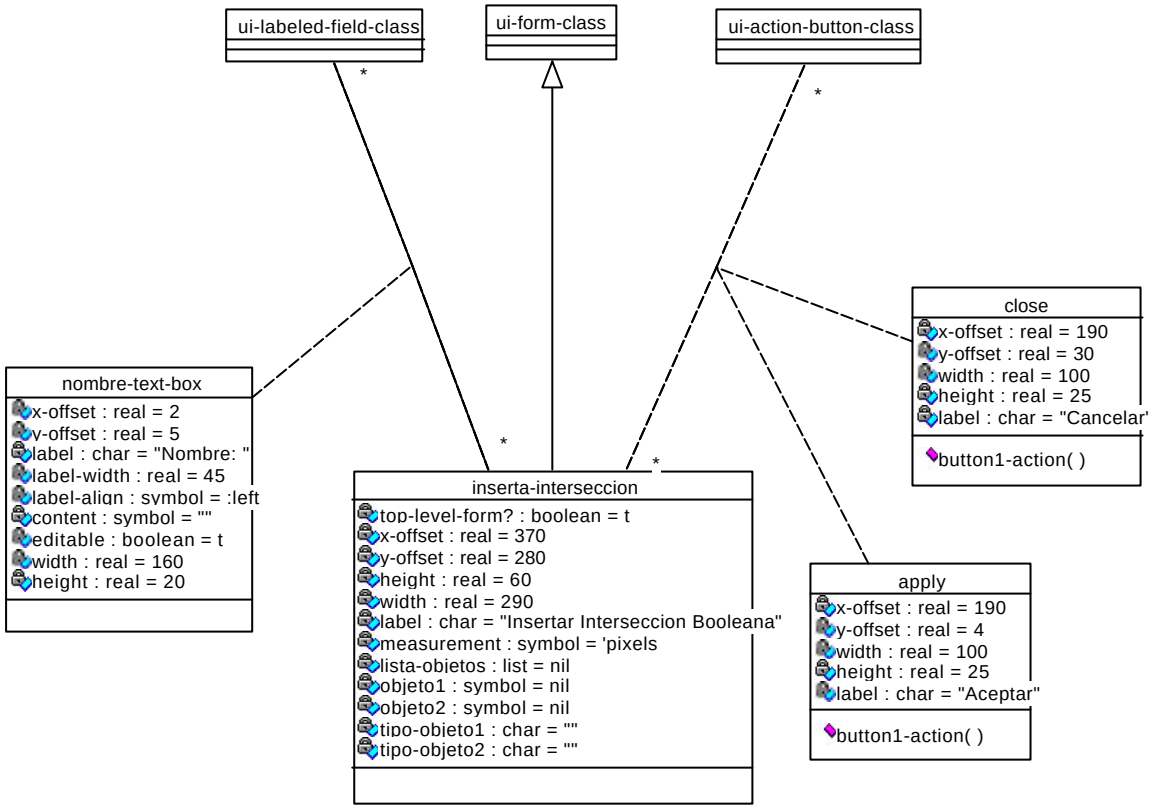


Figura 3.16.- Diagrama de clases de interfaz “inserta intersección”.

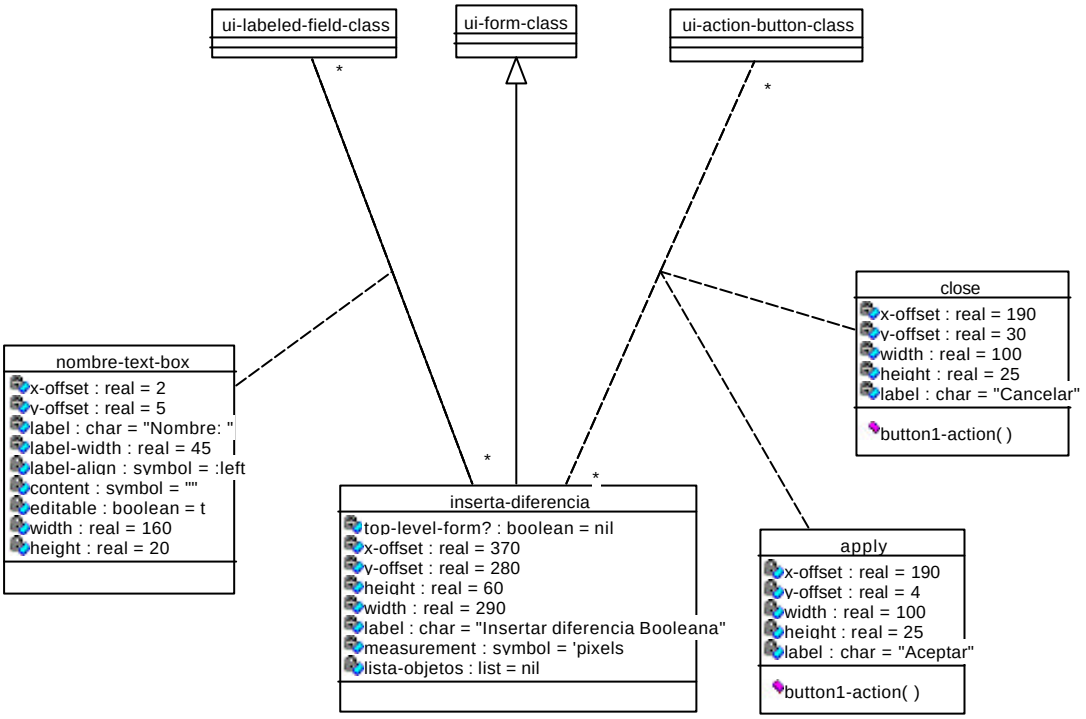


Figura 3. 17.- Diagrama de clases de interfaz "inserta diferencia".

El diagrama mostrado en la **figura 3.18.** muestra como estarán relacionadas las clases que son utilizadas para mover objetos. Esta clase instancia una ventana que solicitara la nueva posición del objeto a mover. La clase *ui-form-class* será la clase que proporcione la ventana que contendrá el resto de los subobjetos. La clase *ui-label-class* permitirá instanciar zonas de texto para mensajes. La clase *ui-radio-buttons-class* permitirá instanciar una zona de botones de radio, el cual sólo permitirá seleccionar una sola opción de *n* opciones. La clase *ui-action-button*-class permitirá instanciar botones con el método *button1-action( )* que permitirá ejecutar código adicional.

El método *button1-action( )* de la clase *apply* obtiene los datos dependiendo del estado de la clase asociada a *ui-radio-buttons-class*. Éstos datos servirán para modificar la posición del objeto a mover. El código adicional moverá los objetos entre el "objeto a mover" y el "objeto junto al cual se moverá", calculando la distancia a mover cada objeto, según el ancho calculado del objeto. Como ultimo paso, trasladara el objeto a mover. Al finalizar, se actualiza la pantalla. El método *button1-action( )* de la clase *close* borra el objeto del cual fue llamado, la clase *mover-objeto-window*. **(Ver figura 3.18.).**

El diagrama mostrado en la **figura 3.19.** muestra como estarán relacionadas las clases que son utilizadas para instanciar en el nuevo modelo la característica secundaria conocida como ranura. Esta clase instancia una ventana que solicitara sobre que objeto insertara, distancias y tamaños de la nueva ranura. La clase *ui-form-class* será la clase que proporcione la ventana que contendrá el resto de los subobjetos. La clase *ui-label-class* permitirá instanciar zonas de texto para mensajes. La clase *ui-labeled-field-class* permitirá instanciar un campo de texto para la entrada de datos y una etiqueta para identificar el campo de texto. La clase *ui-action-button*-class permitirá instanciar botones con el método *button1-action( )* que permitirá ejecutar código adicional.

El método *button1-action( )* de la clase *apply* obtiene los datos de los contenidos de la clase *ui-labeled-field-class*. Éstos datos modificaran las propiedades de la clase nueva que será insertada en el nuevo modelo, con herencia a la clase *ranura-rot-seg-geom.* Al finalizar, se actualiza la pantalla. El método *button1-action( )* de la clase *close* borra el objeto del cual fue llamado, la clase *inserta-ranura*. **(Ver figura 3.19.).**

El diagrama mostrado en la **figura 3.20.** muestra como estarán relacionadas las clases que son utilizadas para instanciar en el nuevo modelo la característica secundaria conocida como chaflán. Esta clase instancia una ventana que solicitara sobre que objeto insertara, distancias y tamaños del nuevo chaflán. La clase *ui-form-class* será la clase que proporcione la ventana que contendrá el resto de los subobjetos. La clase *ui-label-class* permitirá instanciar zonas de texto para mensajes. La clase *ui-labeled-field-class* permitirá instanciar un campo de texto para la entrada de datos y una etiqueta para identificar el campo de texto. La clase *ui-radio-buttons-class* permitirá instanciar una zona de botones de radio, el cual sólo permitirá seleccionar una sola opción de *n* opciones. La clase *ui-action-button*-class permitirá instanciar botones con el método *button1-actio ( )* que permitirá ejecutar código adicional.

El método *button1-action( )* de la clase *apply* obtiene los datos de los contenidos de la clase *ui-labeled-field-class y ui-radio-buttons-class* . Éstos datos modificaran las propiedades de la clase nueva que será insertada en el nuevo modelo, con herencia a la clase *chaflan-rotado*. Al finalizar, se actualiza la pantalla. El método *button1-action( )* de la clase *close* borra el objeto del cual fue llamado, la clase *inserta-chaflan*. **(Ver figura 3.20.).**

El diagrama mostrado en la **figura 3.21.** muestra como estarán relacionadas las clases que son utilizadas para instanciar en el nuevo modelo la característica secundaria conocida como chaflán angular. Esta clase instancia una ventana que solicitara sobre que objeto insertara, distancias y tamaños del nuevo chaflán angular. La clase *ui-form-class* será la clase que proporcione la ventana que contendrá el resto de los subobjetos. La clase *ui-label-class* permitirá instanciar zonas de texto para mensajes. La clase *ui-labeled-field-class* permitirá instanciar un campo de texto para la entrada de datos y una etiqueta para identificar el campo de texto. La clase *ui-radio-buttons-class* permitirá instanciar una zona de botones de radio, el cual sólo permitirá seleccionar una sola opción de *n* opciones. La clase *ui-action-button*-class permitirá instanciar botones con el método *button1-action( )* que permitirá ejecutar código adicional.

El método *button1-action( )* de la clase *apply* obtiene los datos de los contenidos de la clase *ui-labeled-field-class y ui-radio-buttons-class* . Éstos datos modificaran las propiedades de la clase nueva que será insertada en el nuevo modelo, con herencia a la clase *ranura-rot-seg-geom*. Al finalizar, se actualiza la pantalla. El método *button1-action( )* de la clase *close* borra el objeto del cual fue llamado, la clase *inserta-chaflan-angular.* **(Ver figura 3.21.).**

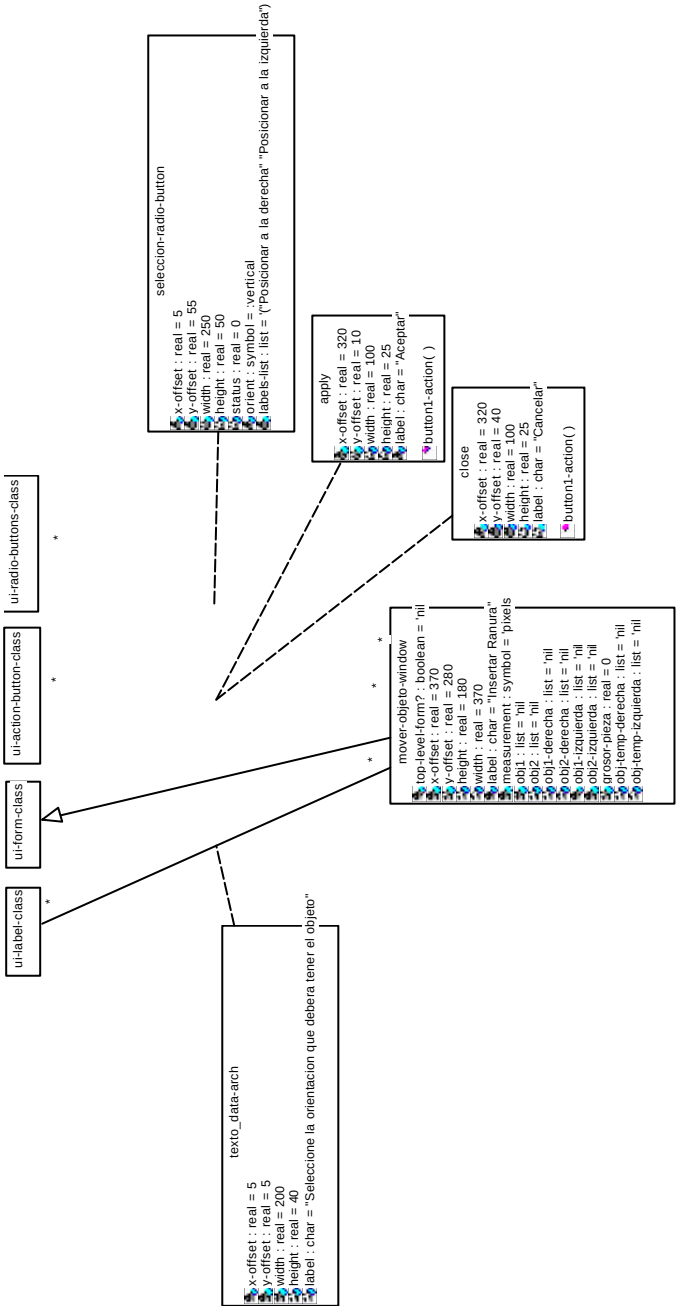


Figura 3. 18.- Diagrama de clases de interfaz “mover-objeto-windows”.

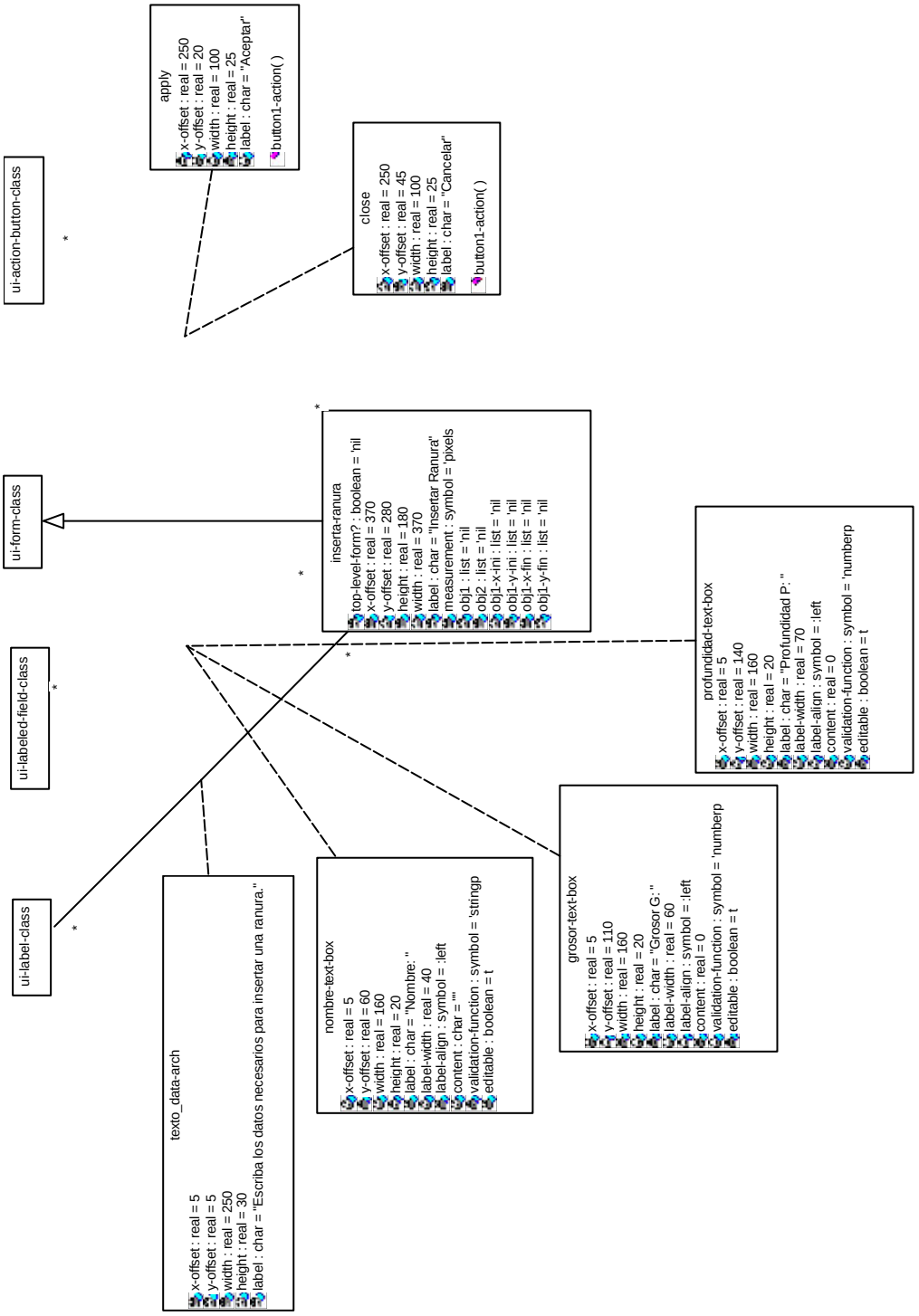


Figura 3. 19.- Diagrama de clases de interfaz “inserta ranura”.

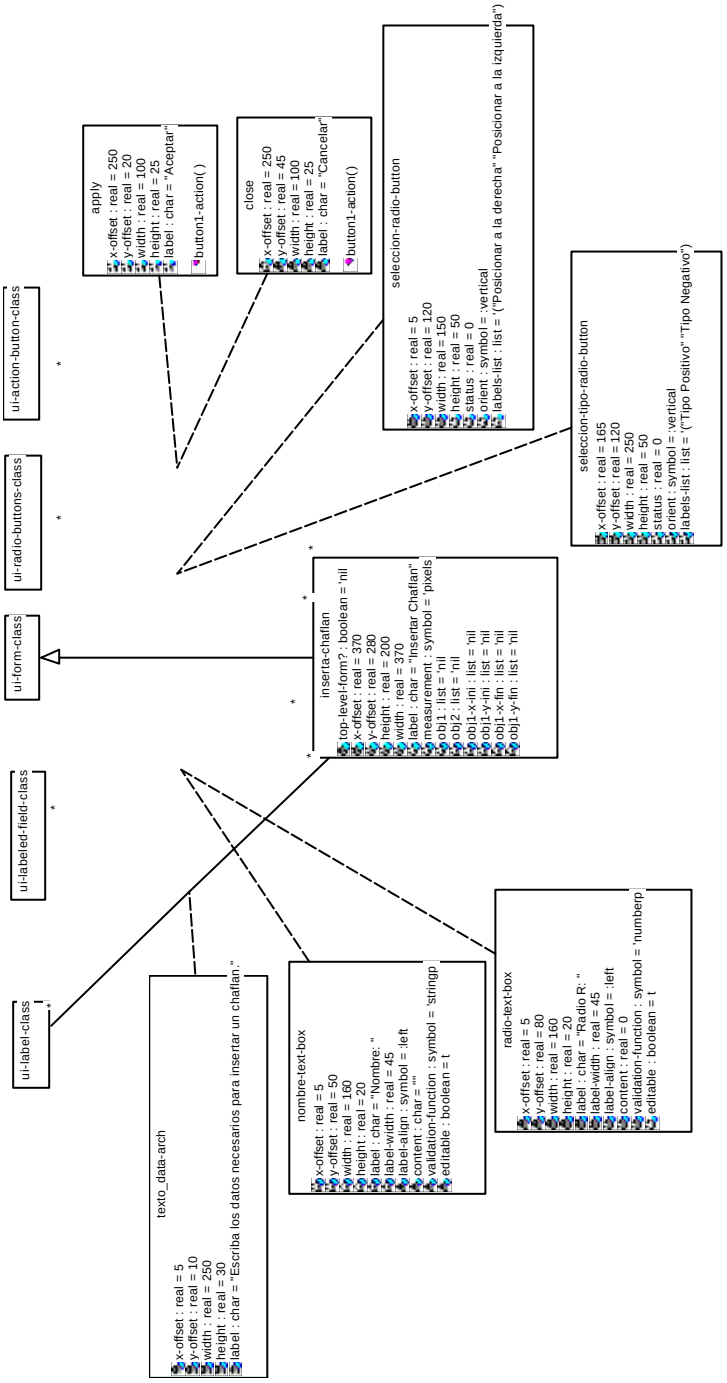


Figura 3. 20.- Diagrama de clases de interfaz “inserta chaflán”.

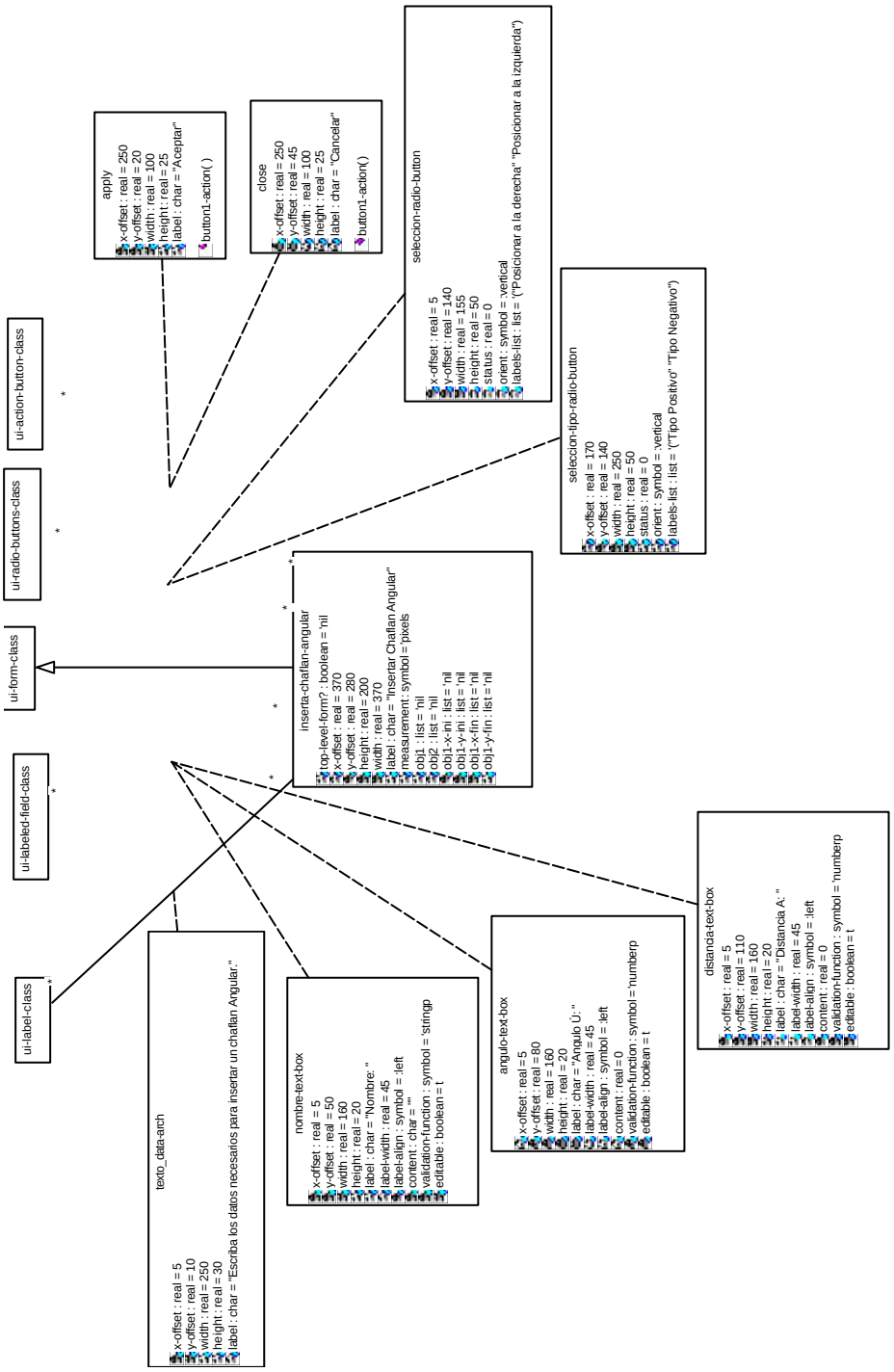


Figura 3. 21.- Diagrama de clases de interfaz “inserta chaflán angular”.

3.3.4 Casos de uso

Los diagramas de casos de uso capturan el comportamiento de un sistema, subsistema ó clase, como es observado por un usuario externo. Dividen la funcionalidad del sistema en transacciones significativas a actores (usuarios idealizados de un sistema). Las piezas de funcionalidad operativa son llamadas casos de uso. Un diagrama de caso de uso describe la interacción con actores como una secuencia de mensajes entre el sistema y uno ó más actores. El termino *actor* incluye personas, así como otros sistemas y procesos de computadora.

Para la descripción del comportamiento del sistema desarrollado en el lenguaje de programación AML, utilizaremos un diagrama de casos de uso general. En este diagrama, mostrado en la **figura 3.22.**, se muestran tres transacciones que resumen los tres tipos de operaciones que realiza el sistema: el *manejo de archivos en AML*, la *inserción de objetos en el modelo* y la *manipulación de objetos en el modelo*.

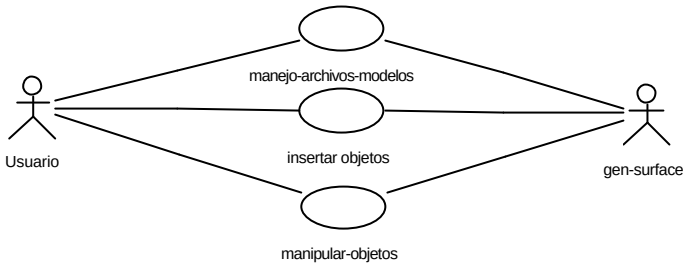


Figura 3. 22.- Diagrama de Casos de Uso General.

El diagrama de caso de uso para el manejo de archivos de modelos en la interfaz desarrollada en AML, se muestra en la **figura 3.23.**, en el cual se muestran las siguientes transacciones: *abrir archivos de modelos*, *guardar archivos de modelos* y *generar archivos de salida basados en información de modelos*.

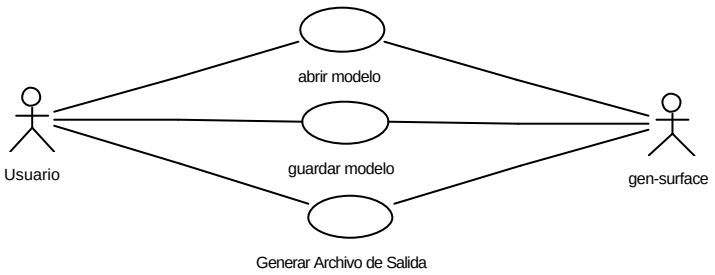


Figura 3. 23.- Diagrama de Casos de Uso para el manejo de archivos de modelos.

El diagrama de caso de uso para la inserción de objetos en los modelos en la interfaz desarrollada en AML, se muestra en la **figura 3.24.**, en el cual se muestran las siguientes transacciones: *insertar esfera*, *insertar polígonos* barridos sobre el eje de simetría, *insertar chaflán*, *insertar chaflán mediante uso de datos angulares*, *insertar ranura* e *insertar sobrecortes*. Todos sobre los objetos que residen en el modelo.

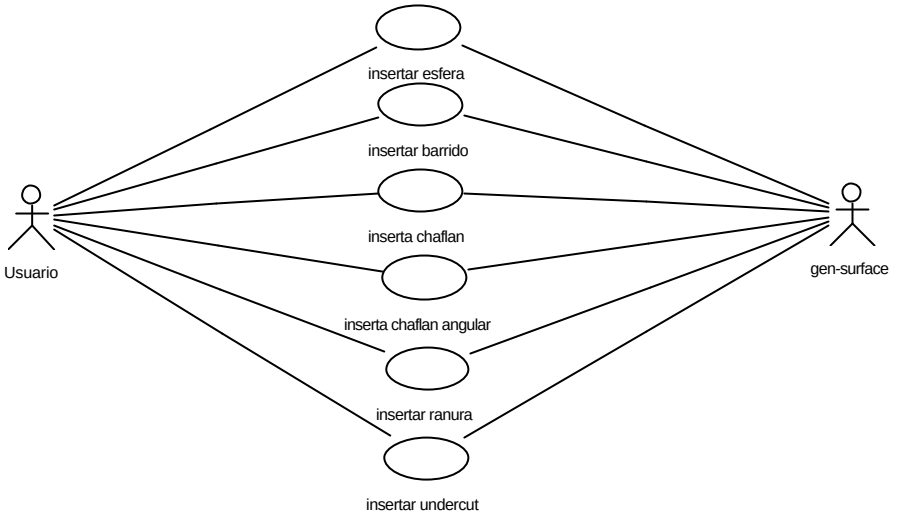


Figura 3. 24.- Diagrama de Casos de Uso para insertar objetos en el modelo.

El diagrama de caso de uso para la manipulación de objetos en los modelos en la interfaz desarrollada en AML, se muestra en la **figura 3.25.**, en el cual se muestran las siguientes transacciones: *mover objetos* a lo largo del eje de simetría, *borrar objetos*, *unir objetos*, *restar objetos* e *intersectar objetos* del modelo.

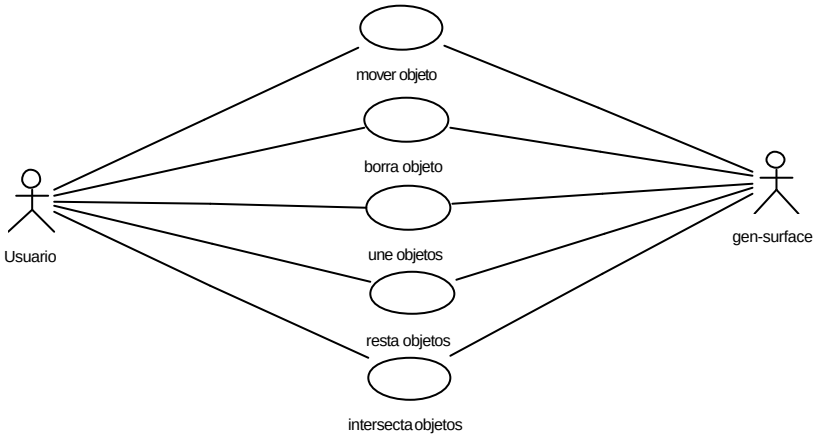


Figura 3. 25.- Diagrama de Casos de Uso para la manipulación de objetos en el modelo.

3.3.5 Diagramas de secuencia

Los objetos interactúan para implementar comportamiento. Esta interacción puede ser representada por una colección de objetos en cooperación. La vista de un modelo que representa el intercambio de mensajes entre objetos para cumplir con un propósito, es conocido como Diagrama de Secuencia. Un diagrama de secuencias muestra una interacción en un diagrama de dos dimensiones. La dimensión vertical representa el tiempo. El tiempo procede hacia debajo de la pagina. La dimensión horizontal representa los papeles clasificadores que representa cada objeto individual en la colaboración. Cada papel clasificador es representado por una columna vertical, que también representa el tiempo de vida del objeto.

El diagrama de secuencia representado en la **figura 3.26.**, muestra el intercambio de mensajes entre el usuario y el modulo para abrir un archivo de modelo en la interfaz desarrollada en AML.

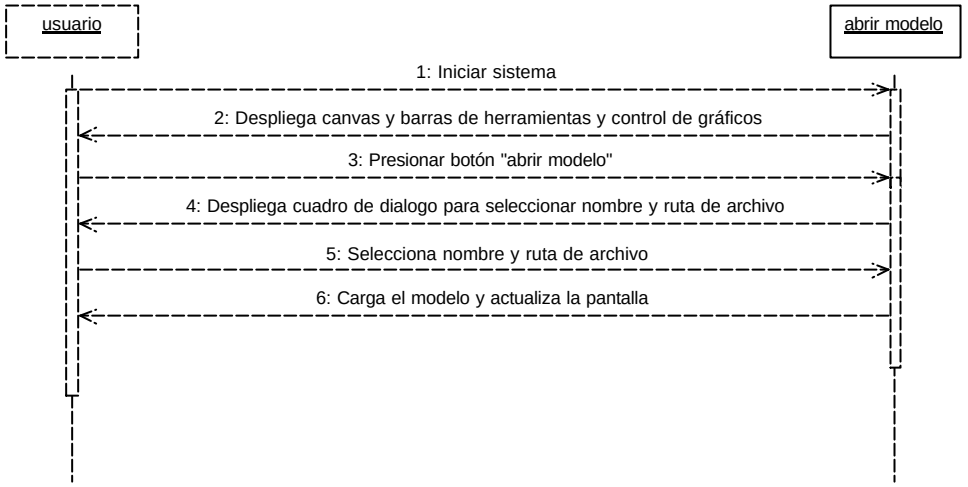


Figura 3. 26.- Diagrama de Secuencia que describe el proceso para abrir un archivo de modelo.

El diagrama de secuencia representado en la **figura 3.27.**, muestra el intercambio de mensajes entre el usuario y el modulo para guardar un archivo de modelo en la interfaz desarrollada en AML.

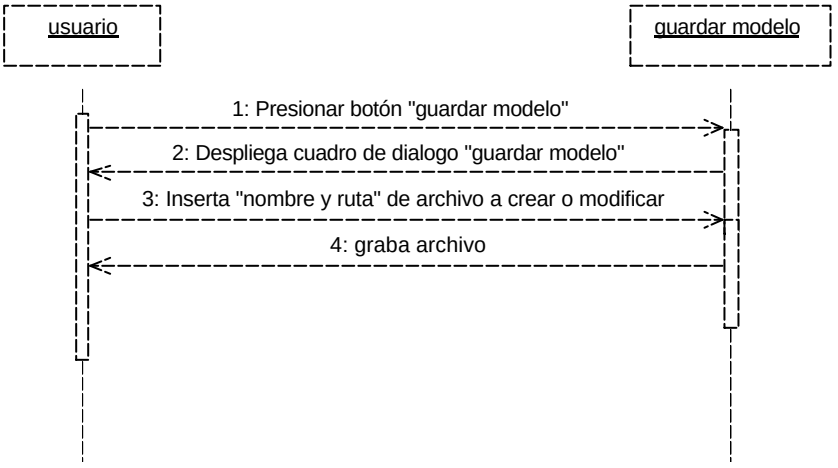


Figura 3. 27.- Diagrama de Secuencia que describe el proceso para guardar un archivo de modelo.

El diagrama de secuencia representado en la **figura 3.28.**, muestra el intercambio de mensajes entre el usuario y el modulo para generar un archivo con información de puntos, diámetros y distancias referentes al modelo realizado en la interfaz desarrollada en AML. Este archivo será usado por el proyecto SADET *(ver sección 3.2.)*.

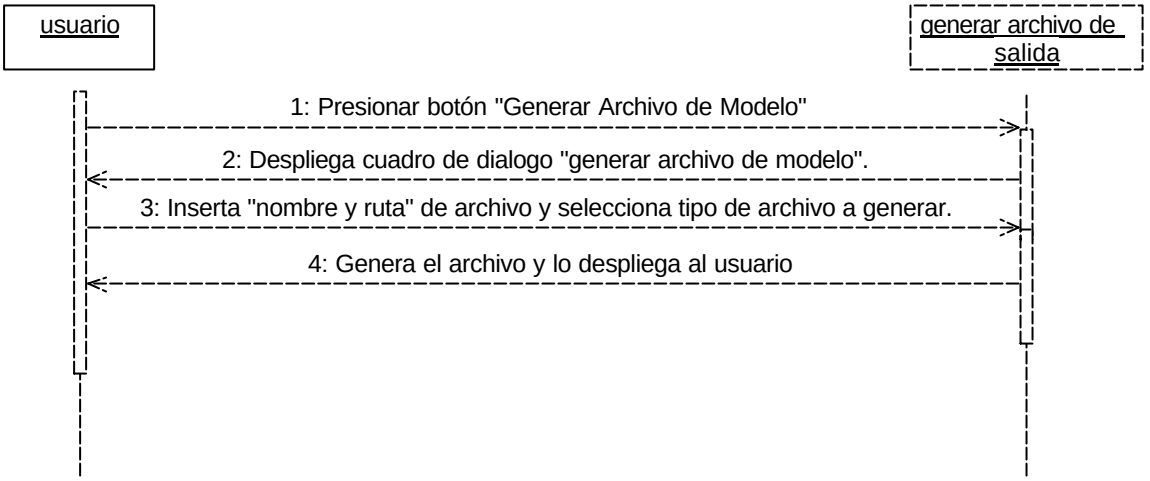


Figura 3. 28.- Diagrama de Secuencia que describe el proceso para generar un archivo de salida referente al modelo, que será usado por SADET.

El diagrama de secuencia representado en la **figura 3.29.**, muestra el intercambio de mensajes entre el usuario y el modulo para insertar una esfera en un modelo realizado en la interfaz desarrollada en AML.

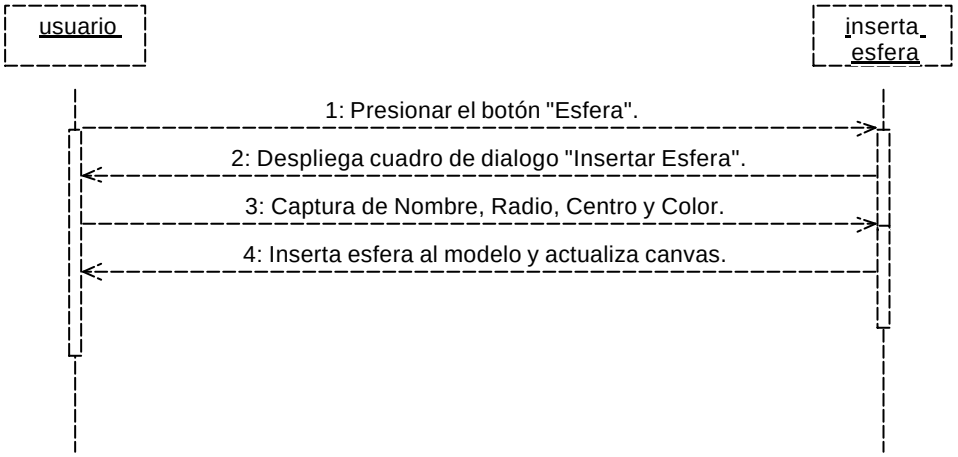


Figura 3. 29.- Diagrama de Secuencia que describe el proceso para insertar una esfera en el modelo.

El diagrama de secuencia representado en la **figura 3.30.**, muestra el intercambio de mensajes entre el usuario y el modulo para insertar un polígono barrido alrededor del eje de simetría en un modelo realizado en la interfaz desarrollada en AML.

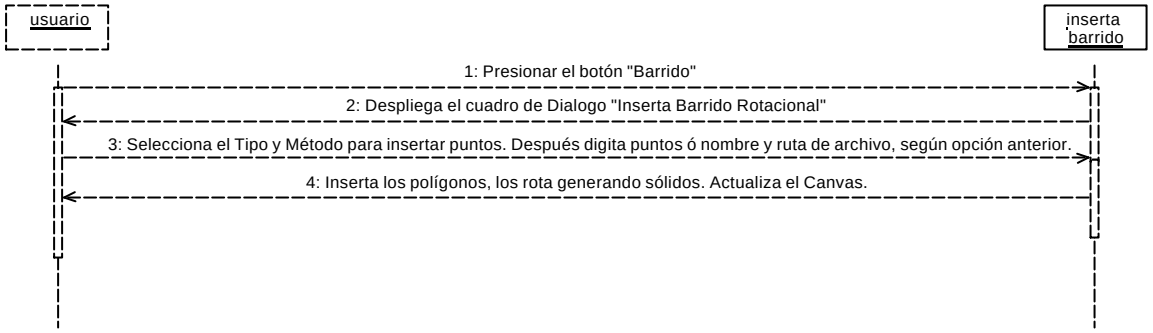


Figura 3. 30.- Diagrama de Secuencia que describe el proceso para inserta polígonos rotados sobre el eje de simetría en un modelo.

El diagrama de secuencia representado en la **figura 3.31.**, muestra el intercambio de mensajes entre el usuario y el modulo para insertar un chaflán en un modelo realizado en la interfaz desarrollada en AML.

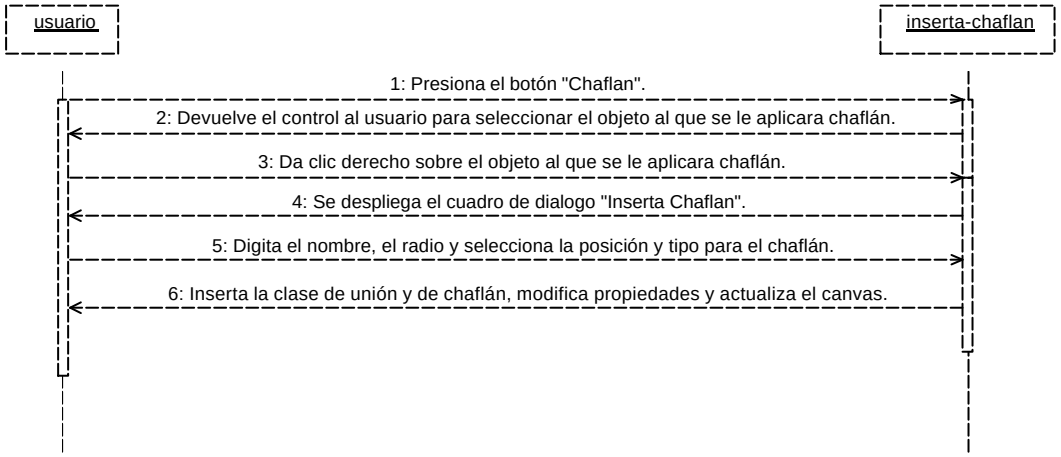


Figura 3. 31.- Diagrama de Secuencia que describe el proceso para insertar un chaflán en un modelo.

El diagrama de secuencia representado en la **figura 3.32.**, muestra el intercambio de mensajes entre el usuario y el modulo para insertar un chaflán usando datos angulares en un modelo realizado en la interfaz desarrollada en AML.

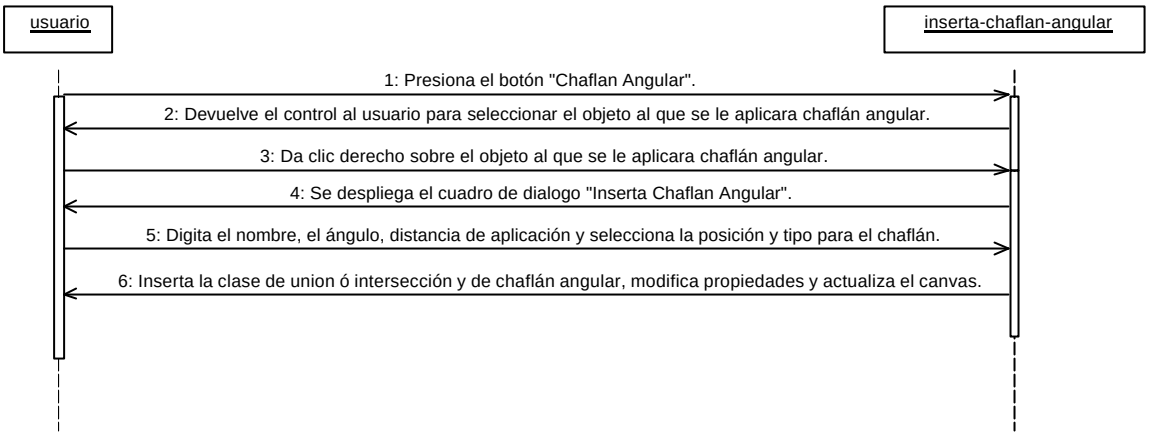


Figura 3. 32.- Diagrama de Secuencia que describe el proceso para insertar un chaflán angular en un modelo.

El diagrama de secuencia representado en la **figura 3.33.**, muestra el intercambio de mensajes entre el usuario y el modulo para insertar una ranura en un modelo realizado en la interfaz desarrollada en AML.

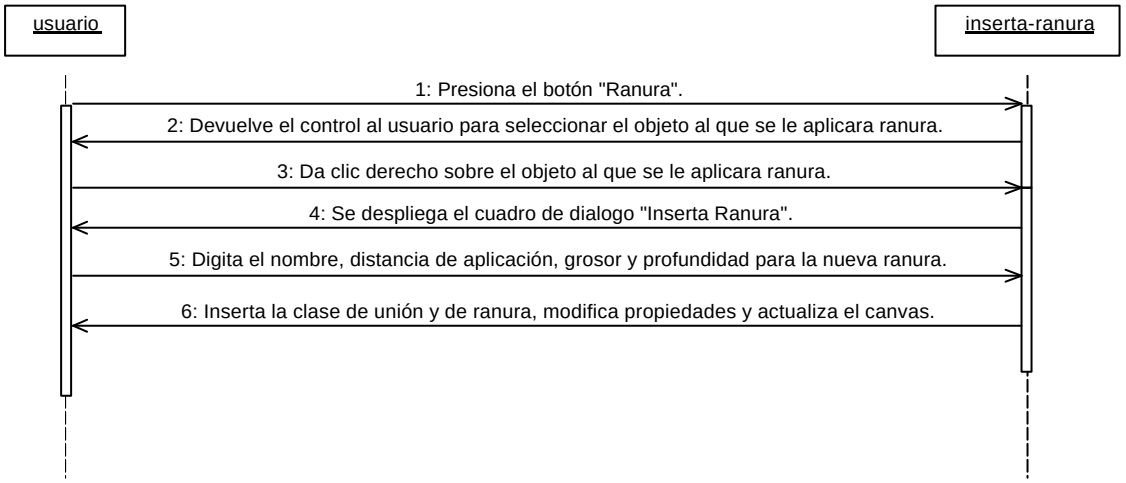


Figura 3. 33.- Diagrama de Secuencia que describe el proceso para insertar una ranura en un modelo.

El diagrama de secuencia representado en la **figura 3.34.**, muestra el intercambio de mensajes entre el usuario y el modulo para insertar un sobrecorte en un modelo realizado en la interfaz desarrollada en AML.

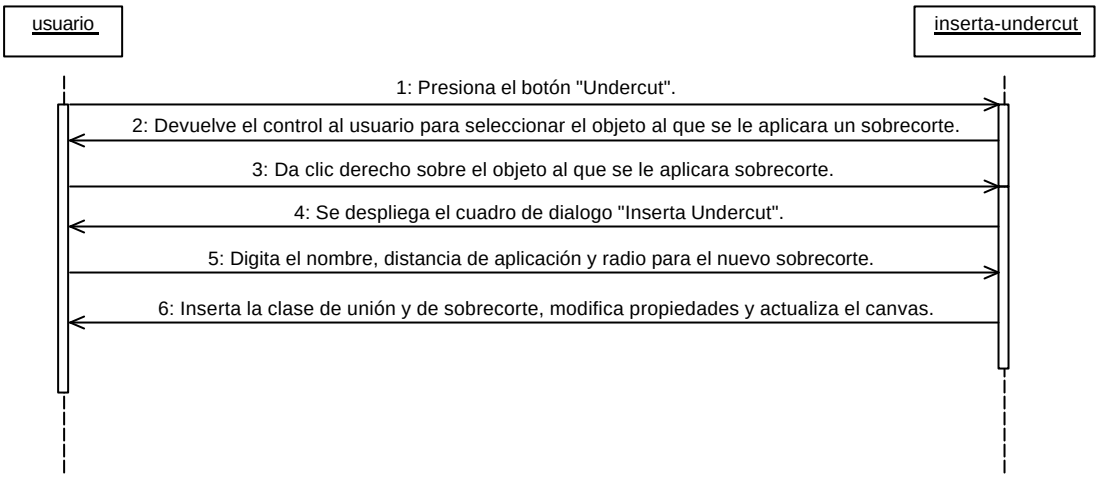


Figura 3. 34.- Diagrama de Secuencia que describe el proceso para insertar un sobrecorte en un modelo.

El diagrama de secuencia representado en la **figura 3.35.**, muestra el intercambio de mensajes entre el usuario y el modulo para mover los elementos generados por los polígonos barridos a lo largo del eje de simetría en un modelo realizado en la interfaz desarrollada en AML.

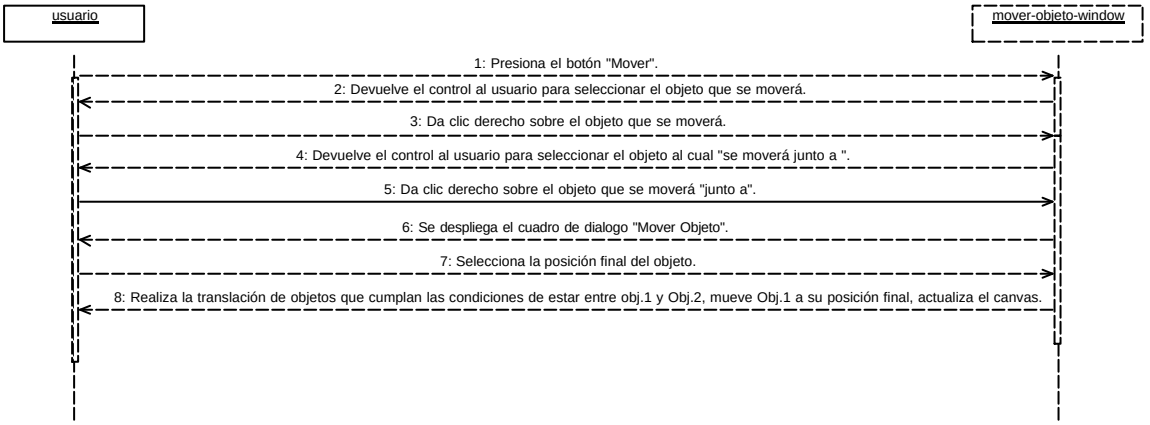


Figura 3. 35.- Diagrama de Secuencia que describe el proceso para mover un objeto de posición en el modelo.

El diagrama de secuencia representado en la **figura 3.36.**, muestra el intercambio de mensajes entre el usuario y el modulo para borrar un objeto de un modelo realizado en la interfaz desarrollada en AML.

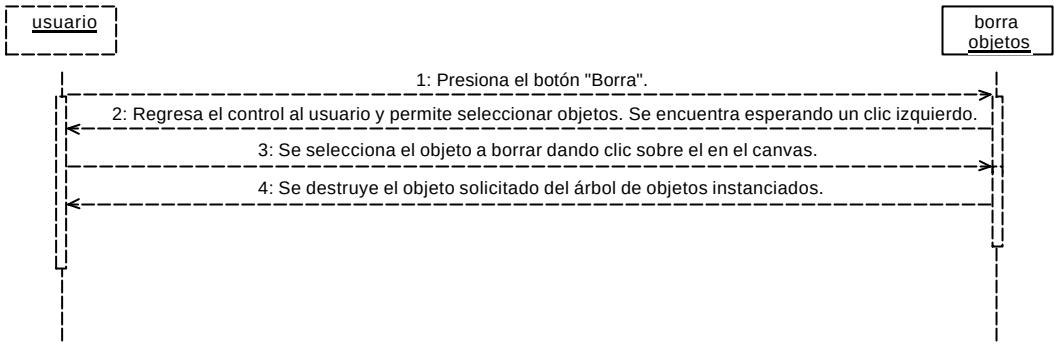


Figura 3. 36.- Diagrama de Secuencia que describe el proceso para borrar un objeto de un modelo.

El diagrama de secuencia representado en la **figura 3.37.**, muestra el intercambio de mensajes entre el usuario y el modulo para unir objetos en un modelo realizado en la interfaz desarrollada en AML.

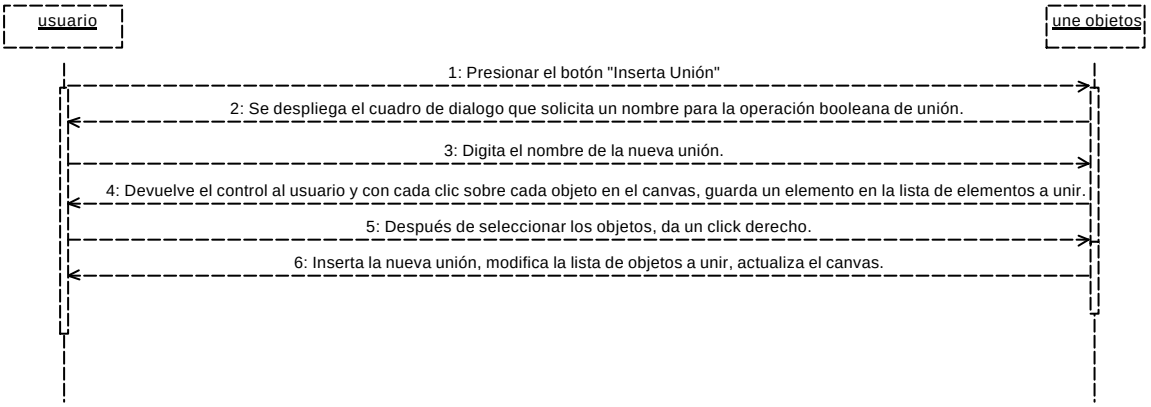


Figura 3. 37.- Diagrama de Secuencia que describe el proceso para unir objetos en un modelo.

El diagrama de secuencia representado en la **figura 3.38.**, muestra el intercambio de mensajes entre el usuario y el modulo para restar objetos en un modelo realizado en la interfaz desarrollada en AML.

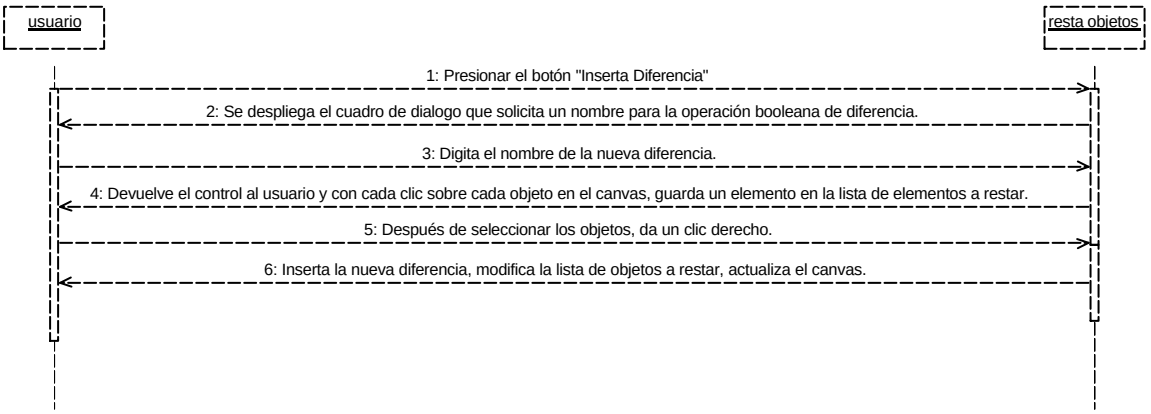


Figura 3. 38.- Diagrama de Secuencia que describe el proceso para restar objetos en un modelo.

El diagrama de secuencia representado en la **figura 3.39.**, muestra el intercambio de mensajes entre el usuario y el modulo para intersectar objetos en un modelo realizado en la interfaz desarrollada en AML.

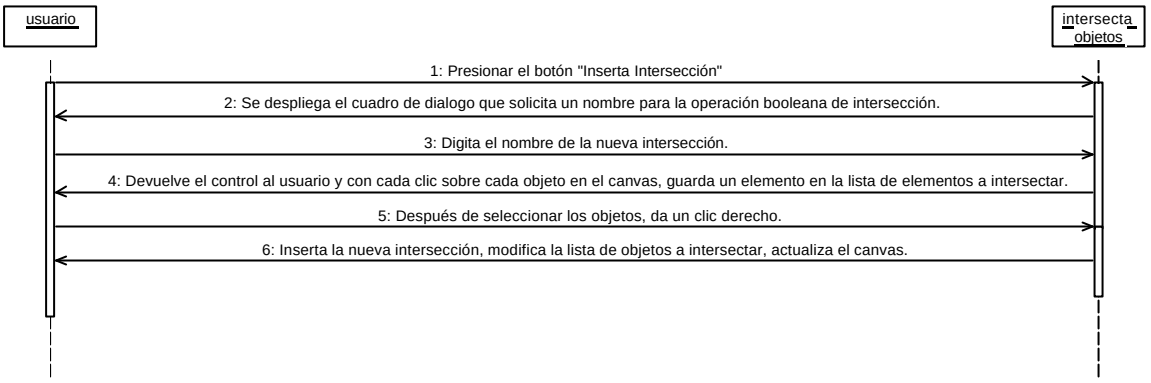


Figura 3. 39.- Diagrama de Secuencia que describe el proceso para intersectar objetos en un modelo.

Capítulo 4.- Descripción del sistema paramétrico bajo AML

4.1 Introducción

En este capítulo, se mostrará el procedimiento que se siguió para la programación e implementación de la interfaz gráfica para el proyecto SADET utilizando el lenguaje de programación AML. (ver sección 3.2.).

Además, se obtendrá una vista general del código que conformara la interfaz gráfica desarrollada en AML. Se analizará código que permite la inserción de objetos, la apertura de archivos de modelos, la destrucción de objetos del árbol de clases instanciadas, del método que permite instanciar el sistema en el entorno gráfico de AML. No se mostrará todo el código que conforma el sistema, pero éstos ejemplos cumplen con las características del diagrama de casos de uso general (ver sección 3.3.3.- Casos de uso).

También se mostrará el manejo y operación de la interfaz gráfica desarrollada en el entorno gráfico de AML. Se reconocerán cada una de las partes que conforman el sistema, el procedimiento para instanciar el sistema en el entorno gráfico de AML y la operación del sistema para abrir y/o guardar archivos de modelos, instanciar modelos, modificar y manipular modelos, así como generar archivos de salida para el proyecto SADET (ver sección 3.2.).

4.2 Implementación en AML

Para realizar la implementación del sistema en AML, se escribirá el código en archivos de texto con extensión ".aml". Éstos archivos se escribirán usando "el bloc de notas" de Windows, o el editor x-emacs de AML, que este último permite tener más control sobre la sintaxis del código. Cuando un código esta listo para ser probado, se utiliza la herramienta "load file", que instancia todas las clases, métodos y funciones contenidos en el archivo seleccionado en el árbol de clases , después se utiliza las herramientas del entorno gráfico AML-GUI para probar el código.

Durante el proceso de programación, las primeras clases que se definirán serán las clases que permitan instanciar objetos geométricos. Estas clases serán revisadas inmediatamente utilizando las herramientas disponibles del entorno GUI de AML, con el fin de verificar las características del objeto geométrico deseado. Las clases que después se definirán serán las que definirán el canvas, la barra de herramientas y el control gráficos. Estas clases, al ser instanciadas, operarán en el entorno GUI. El código de las clases que definen los cuadros de díalogo, será programado en conjunto con la barra de herramientas.

Ya que todas las clases y métodos están instanciados en el entorno de AML, el proceso para iniciar el sistema es el siguiente: se invoca la función "gen-surface" que verifica la existencia del elemento "genera-superficie" en el árbol de clases [the interface forms menubar-header menubar], y en caso de no existir lo crea y define elemento de menú. Cada elemento de menú cuenta con un espacio para declarar un evento. El evento del elemento "Editor de Geometría" crea el modelo 'nuevo-model con herencia a la clase *nuevo-modelo* **(ver sección 3.3.3.2, figura 3.9)**.

Al objeto 'nuevo-model le adiciona:

- El objeto 'eje con herencia a la clase *eje-coordenado* **(ver sección 3.3.3.2, figura 3.9)**
- El objeto 'secundarias con herencia a la clase *nuevo-modelo*;
- El objeto 'op-booleans con herencia a la clase *nuevo-modelo*;
- El objeto 'geometria con herencia a la clase *nuevo-modelo*.

Al objeto [the interface] le adiciona:

- El objeto 'split-form con herencia a la clase *split-form-with-toolbar-class* **(ver sección 3.3.3.2, figura 3.7)**;
- El objeto 'toolbarra con herencia a la clase *barra-herramientas* **(ver sección 3.3.3.2, figura 3.8)**.

Por ultimo, el método del elemento "Editor de Geometría" despliega en pantalla los objetos 'split-form y 'toolbarra que forman el canvas, la barra de herramientas y el control de graficos **(Ver sección 4.3)**.

Para abrir un archivo de modelo (el cual contiene definiciones de geometría para AML), y ver este mismo desplegado en el canvas, el sistema sigue la siguiente secuencia de acciones **(Ver sección 4.3)**:

- Se activa el método *button1-action-list* del objeto [the toolbarra barra-arch] asignado para abrir modelos **(ver sección 3.3.3.2, figura 3.8)**;
- El evento anterior ejecuta la función abrir-objetos();
- La función abrir-objetos() ejecuta el método *select-file-dialog* con el cual obtiene el nombre y la ruta del archivo que contiene el modelo a instanciar, y estos datos se transfieren al método *aml-load* para instanciar el modelo en el objeto 'nuevo-model.

Para insertar un polígono revolucionado alrededor del eje de simetría, el sistema sigue la siguiente secuencia de acciones:

- Se activa el método *button1-action-list* del objeto [the toolbarra barra-arch] asignado para abrir modelos **(ver sección 3.3.3.2, figura 3.8)**;
- Se adiciona el objeto 'menu-barrido, con herencia a la clase *inserta-barrido* **(ver sección 3.3.3.2, figura 3.13)**, y se despliega. Aquí el usuario debe llenar el formulario y presionar <<aceptar>>;
- La acción anterior ejecuta el método *button1-action-list* del objeto [the interface menú-barrido apply];
- El método anterior insertara objetos con herencia a la clase *barrido-rotacional* **(ver sección 3.3.3.2, figura 3.11)** y modificara las propiedades de los polígonos en base a los arreglos de puntos obtenidos;
- Al finalizar, el método anterior actualiza el canvas y destruye el objeto [the interface menú-barrido].

4.3 Código

La importancia del código es grande, ya que este es el que nos permite comunicarnos con el sistema para solicitar que realice las acciones que deseamos. El lenguaje esta definido por una sintaxis en particular, que el compilador de AML interpretara. Es por eso que se presenta parte del código más representativo que conformara la interfaz del sistema.

La función "gen-surface" permitirá instanciar una opción en el menú principal de AML-GUI con el nombre "genera superficie". La opción "Editor de Geometría" permitirá instanciar en el árbol de geometría el eje coordenado y en el árbol de interfaz , instanciara la barra de herramientas, el canvas y el control de gráficos.

```
;;función para Cargar en el entorno de AML el sistema generador de geometria
(defun gen-surface ()
  ;;Las siguientes dos lineas Buscan el menu "Genera-Superficie" en la instancia señalada,
  ;;si no lo encuentran, lo crean en "interface forms menubar-header menubar".
  (when (not (the interface forms menubar-header menubar Genera-Superficie (:error nil)))
    ;;se añade el elemento nuevo de menu.
    (add-header-menu 'Genera-Superficie :menu-items '(
      ;;Se definen Elementos del Menu
      ("Editor de Geometria" '(progn
                                (when (not (the nuevo-modelo (:error nil)))
                                  (create-model 'nuevo-model :class 'nuevo-modelo)
                                (add-object (the nuevo-model) 'eje 'eje-coordenado)
                                  (add-object (the nuevo-model) 'secundarias 'nuevo-modelo)
                                  (add-object (the nuevo-model) 'op-booleans 'nuevo-modelo)
                                  (add-object (the nuevo-model) 'geometria 'nuevo-modelo)
                                )
                              (add-object (the interface) 'split-form 'split-form-with-toolbar-class)
                                (add-object (the interface) 'toolbarra 'barra-herramientas)
                                (display (the interface toolbarra))
                                (display (the interface split-form))
                              (draw (the nuevo-model) :clear-display? t :update-window? t)
                               ))
```

```
            ("Cerrar y Limpiar Datos Cargados" '(progn
                                                  (when (the interface split-form (:error nil))
                                                        (delete-object (the interface split-form)))
                                                  (when (the interface forma-data-arch (:error nil))
                                                    (delete-object (the interface forma-data-arch)))
                                                  (when (the toolbarra (:error nil))
                                                    (delete-object (the toolbarra))))
             )

            )
    :label "Genera Superficie")
  )
)
```

La clase "barra-herramientas" define las características que formaran parte de la barra de herramientas. La clase subobjeto más importante de esta clase es "ui-toolbar-class" que permite instanciar botones con etiquetas, y permite que cada botón tenga asignado código con propiedades heredadas de la clase "ui-action-button-class". A continuación se presenta el código más representativo de la clase:

```
(define-class barra-herramientas
;;Clase que define la ventana que contiene como subobjeto una barra de herramientas.
  :inherit-from (ui-form-class)
  :properties(
              top-level-form? 'nil
              x-offset 800 y-offset 80 height 270 width 210 label "Barra de Herramientas"
              measurement 'pixels
              nombre-objeto ""
              )
  :subobjects(
              .
              .
              .
                (barra-arch :class 'ui-toolbar-class
                   x-offset 4 y-offset 80 height 30 width 55
                   availavility-list '(t t t)
                   number-of-buttons 3
                   orient :horizontal
                   tooltips-list '("Abrir Modelo" "Guardar Modelo" "Generar Archivo de Salida")
                   images-list '("k:\\jjcolin\\bmp\\open-file.bmp" "k:\\jjcolin\\bmp\\save.bmp" "k:\\jjcolin\\bmp\\apply.bmp")
                   button1-action-list '(
                                         ;Codigo Boton 1
                                         (progn
                                            (abrir-objetos))
                                         ;Codigo Boton 2
                                         (progn
                                             (salvar-objetos))
                                         ;Codigo Boton 3
                                          (progn
                                            ((add-object (the interface) 'arch-sal 'gen-arch-sal)
                                             (display (the interface arch-sal)))
                                           )
                                           )
                )
              .
              .
              .
          )
)
```

La función "abrir-objeto" utilizará el método "select-file-dialog" que abre el cuadro de díalogo para seleccionar nombre y ruta de archivo que será utilizado por el método "aml-load" para abrir un archivo de texto con la definición de la geometría de un modelo que será instanciado en el sistema.

```
(defun abrir-objetos()
  ;;Funcion para abrir el modelo diseñado guardado en un archivo (con extension *.mdl como default)
  (aml-load (select-file-dialog :title "Abrir" :x 20 :y 45 :new? nil :filter "*.mld") :addto (the nuevo-model) :as-model? t)
  (draw (the nuevo-model) :clear-display? t :update-window? t))
```

La clase “inserta-barrido” define las características del cuadro de díalogo que solicitara datos para insertar poligonos barridos alrededor del eje de simetría en el árbol de geometría.. Los métodos “get-value” y “change-value” permitirán obtener y cambiar valores de propiedades de la clase. El método “add-object” permitirá instanciar objetos en el árbol de clases indicado, que en este caso, será el árbol de geometría. A continuación se presenta el código más representativo de la clase:

```
(define-class inserta-barrido
  ;;;Esta funcion dibuja la ventana para insertar los datos necesarios para generar un poligono revolucionado utilizando los puntos
  alimentados por un archivo de texto.
  :inherit-from(ui-form-class)
  :properties(
                top-level-form? 'nil
                x-offset 370 y-offset 280 height 250 width  370 label "Insertar Barrido Rotacional"
                measurement 'pixels
               puntos '((0 0 0) (0 1 0) (1 1 0) (1 0 0))
                puntos-aux '((0 0 0) (0 1 0) (1 1 0) (1 0 0))
                puntos-cad ""
                punto-gen ""
                )

  :subobjects(
              .
              .
              .
          (seleccion-radio-button :class 'ui-radio-buttons-class
            ;;Radiobuttonbox para seleccionar entre "Editar Puntos" y "Leer archivo de Puntos" y desactivar elementos de la ventana.
            x-offset 5 y-offset 40 width 220 height 50
            status 0
            orient :vertical
            labels-list '("Editar Puntos" "Leer archivo de Puntos")
            button1-action (
                      (if (equal (the inserta-barrido seleccion-radio-button status) 1)
                          (progn
                           (change-value (the inserta-barrido puntos-text-box editable?) t)
                           (change-value (the inserta-barrido data-arch-text-box editable?) nil)
                           )
                          (progn
                           (change-value (the inserta-barrido puntos-text-box editable?) nil)
                           (change-value (the inserta-barrido data-arch-text-box editable?) t)
                           )
                       )
                      (draw (the inserta-barrido puntos-text-box) :clear-display? t :update-window? t)
                      (draw (the inserta-barrido data-arch-text-box) :clear-display? t :update-window? t)
                      )
           )
          (forma-radio-button :class 'ui-radio-buttons-class
            x-offset 5 y-offset 100 width 180 height 50
            status 0
            orient :vertical
            labels-list '("Un solo poligono" "Particionado en poligonos")
           )
            .
            .
            .
      (apply :class 'ui-action-button-class
            x-offset 250 y-offset 20 width 100 height 25
            label "Aceptar"
            button1-action(
                    (change-value (the inserta-barrido nombre-text-box content) (get-value (the inserta-barrido nombre-text-box)))
                     (if (equal (the inserta-barrido seleccion-radio-button status) 0)
                         (progn
                          ;;Se utilizan los puntos editados
                          (change-value (the inserta-barrido puntos) (get-value (the inserta-barrido puntos-text-box))
                          ))
                         (progn
                          ;;Se utiliza el archivo de puntos
                                      (if (probe-file (first (the inserta-barrido data-arch-text-box content)))
                                          (
```

```
                                        (change-value (the inserta-barrido puntos)
                                        (with-open-file (data-archivo (first(the inserta-barrido data-arch-text-box content))
                                                                      :direction :input)
                                          (loop for linea = (read-line data-archivo nil :eof)
                                                until (equal linea :eof)
                                                for xyz = (read-from-string (format nil "(~a)" linea ))
                                                collect xyz
                                                ))))
                                    (
                                        (message-box "Mensaje del Sistema" "El archivo no ha sido encontrado" :mode :ok
:centered? t :labels '("Aceptar"))
                                    )
                                  )
                                 )
                                )
                (if (equal (the inserta-barrido forma-radio-button status) 0)
                    (progn
                      (eval (read-from-string (concatenate "(add-object (the nuevo-model geometria) '" (the inserta-barrido nombre-
text-box content) " 'barrido-rotacional)")))
                       (eval (read-from-string (concatenate "(change-value (the nuevo-model geometria " (the inserta-barrido nombre-
text-box content) " puntos) (the inserta-barrido puntos))" )))
                                        )
                    (progn
                      (change-value (the inserta-barrido puntos-aux) (the inserta-barrido puntos))
                      (change-value (the inserta-barrido puntos) (list ()))
                      (loop for i from 0 to (length (the inserta-barrido puntos-aux))
                        do (progn
                            (if (equal i 0)
                                (progn
                                    (change-value (the insert a-barrido puntos-cad)(concatenate (the inserta-barrido puntos-cad) (write-
to-string (nth i (the inserta-barrido puntos-aux)))))
                                )
                                (progn
                                    (if (and (equal (nth 1 (nth i (the inserta-barrido puntos-aux))) (nth 1 (nth (- i 1) (the inserta-barrido
puntos-aux)))) (not (equal (nth 0 (nth i (the inserta-barrido puntos-aux))) (nth 0 (nth (- i 1) (the inserta-barrido puntos-aux))))))
                                        (;;Se genera nuevo punto y se inserta nuevo poligono rotado
                                        (progn
                                            (change-value (the inserta-barrido puntos-cad) (concatenate (the inserta-barrido puntos-cad) "
" (write-to-string (nth i (the inserta-barrido puntos-aux)))))
                                            (change-value (the inserta-barrido punto-gen) (concatenate "(" (write-to-string (nth 0 (nth i
(the inserta-barrido puntos-aux)))) " 0 "  (write-to-string (nth 2 (nth i (the inserta-barrido puntos-aux)))) ")"))
                                            (change-value (the inserta-barrido puntos-cad) (concatenate "(" (the inserta-barrido puntos-
cad) " " (the inserta-barrido punto-gen) ")" ))
                                            (change-value (the inserta-barrido puntos)(read-from-string (the inserta-barrido puntos-cad)))
                                            (eval (read-from-string (concatenate "(add-object (the nuevo-model geometria) '" (the inserta-
barrido nombre-text-box content) "-" (write-to-string i) " 'barrido-rotacional)")))
                                            (eval (read-from-string (concatenate "(change-value (the nuevo-model geometria " (the
inserta-barrido nombre-text-box content) "-" (write-to-string i) " puntos) (the inserta-barrido puntos))" )))
                                             (change-value (the inserta-barrido puntos-cad) (the inserta-barrido punto-gen))
                                        ))
                                        (;;Solo se inserta el nuevo punto
                                            (progn
                                                (if (not (equal (write-to-string (nth i (the inserta-barrido puntos-aux))) "nil"))
                                                    (change-value (the inserta-barrido puntos-cad) (concatenate (the inserta-barrido
puntos-cad) " " (write-to-string (nth i (the inserta-barrido puntos-aux))))))
                                            )))
                                )
                            )
                        )
                      )
                    )
                )
                ;;Se actualiza la pantalla y se borra la ventana.
                (draw (the nuevo-model) :clear-display? t :update-window? t)
                (delete-object (the superior superior))
                                )
                )
        .
        …)
```

La función "borra-objetos" destruye los objetos instanciados en el árbol de geometría, mediante el uso del método "delete-object". La selección del objeto se lleva a cabo mediante el método "get-object".

```
(defun borra-objetos ()
  ;;Esta función solicita seleccionar un objeto en el canvas, e inmediatamente lo borra
  ;;del árbol de instancias. Inmediatamente actualiza el canvas.
  (with-error-handler (:error-message "Los ejes coordenados no pueden ser borrados antes que el objeto al cual pertenecen. Borra primero el objeto y despues el eje coordenado.")
      (delete-object (get-object :from (the nuevo-model) :mechanism 'display))
      (draw (the nuevo-model) :clear-display? t :update-window? t)
  ))
```

4.4 Manual de Operación

Al inicializar el sistema, se despliega en pantalla el menú principal de AML-GUI. El sistema genera una opción extra que es visible al final del menú AML-GUI que tiene por nombre ***Genera Superficie*** **(Figura 4.1)**.

Figura 4. 1.- Menú Principal de AML-GUI con una opción extra, Genera Superficie.

El menú ***Genera Superficie*** presenta dos opciones: ***Editor de Geometría*** y ***Cerrar y Limpiar Datos Cargados*** **(Figura 4.2).**

Figura 4. 2.- Opciones del menú Genera Superficie.

La opción ***Editor de Geometría***, inicializa el sistema editor de geometría desarrollado en AML que esta formado por tres elementos en pantalla **(Figura 4.3)**:

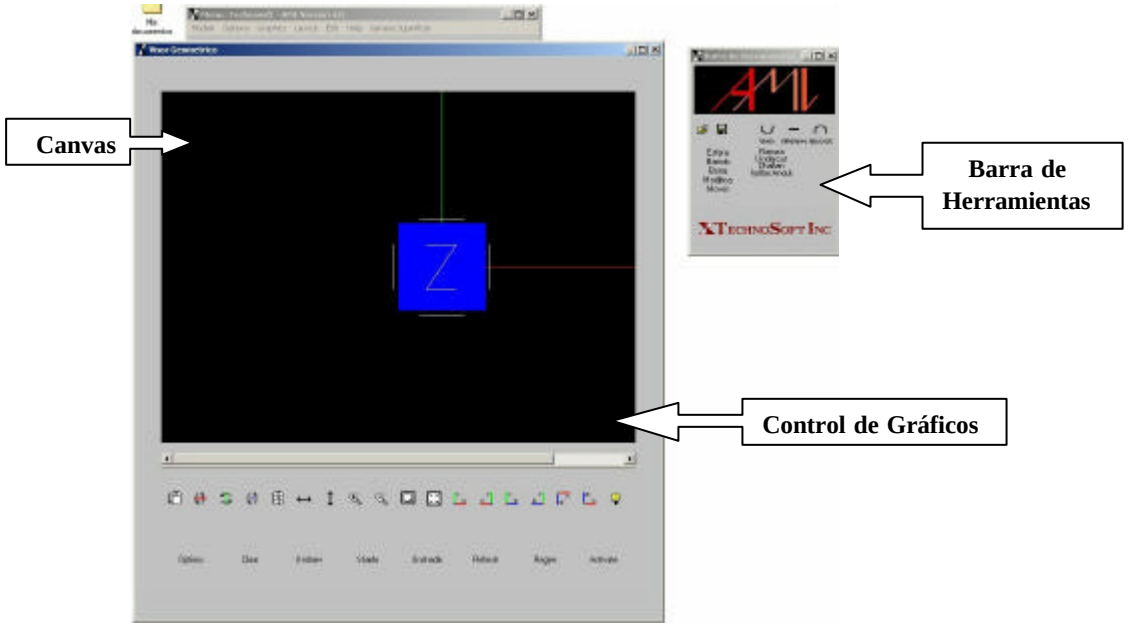


Figura 4. 3.- Estado inicial del editor de objetos.

- **Canvas:** El canvas provee una pantalla de dibujo para los objetos gráficos en AML. Cuando esta ventana esta activada, todas las funciones gráficas llamadas operan sobre esta ventana, hasta que un nuevo canvas sea activado.

- **Control de gráficos:** Es usado para manejar interactivamente los gráficos de la ventana de despliegue activa (canvas). Estas herramientas controlan la forma de ver, delinear, aplicación de luz y otras opciones de los gráficos **(Figura 4.4)**.

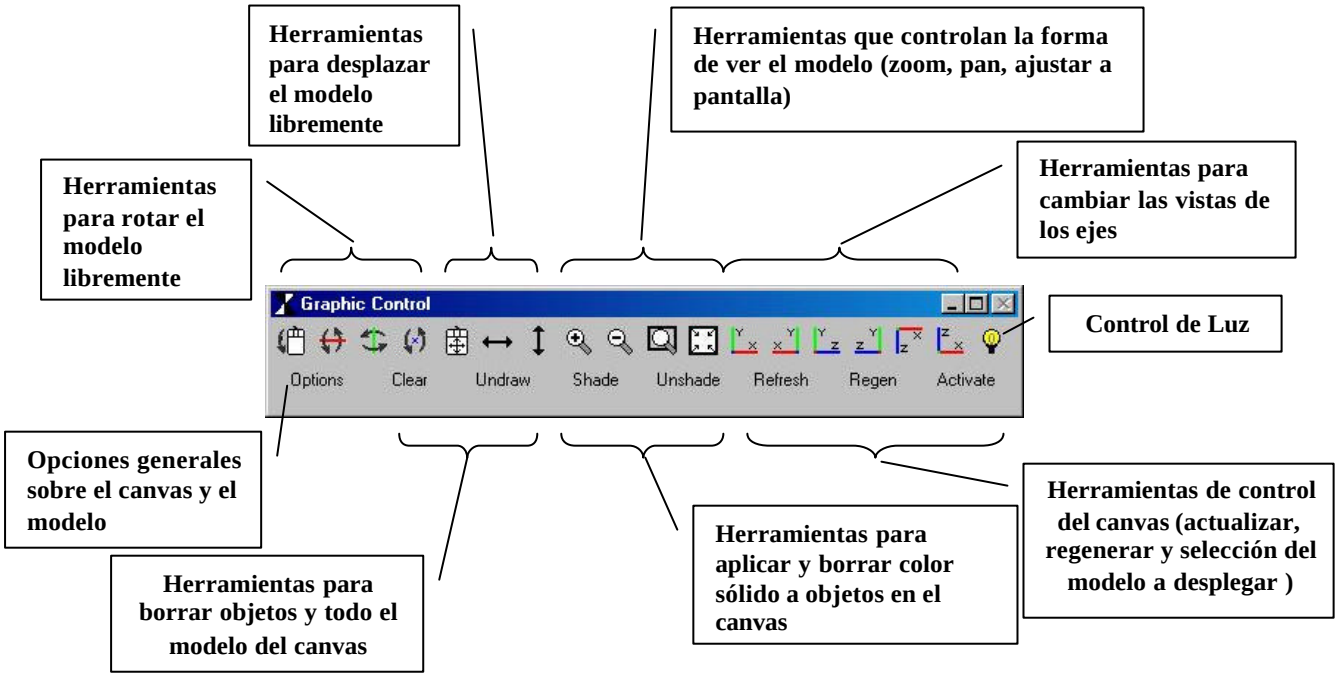


Figura 4. 4.- Descripción de las herramientas contenidas en el Control de gráficos.

- **Barra de Herramientas:** Esta barra de herramientas, contiene botones que controlan la creación de modelos. Las opciones permiten guardar modelos, abrir modelos preexistentes, insertar esferas, planos barridos a lo largo del eje de simetria, mover objetos, crear operaciones booleanas entre objetos y aplicar características secundarias como ranuras, chaflanes y cortes **(Figura 4.5)**.

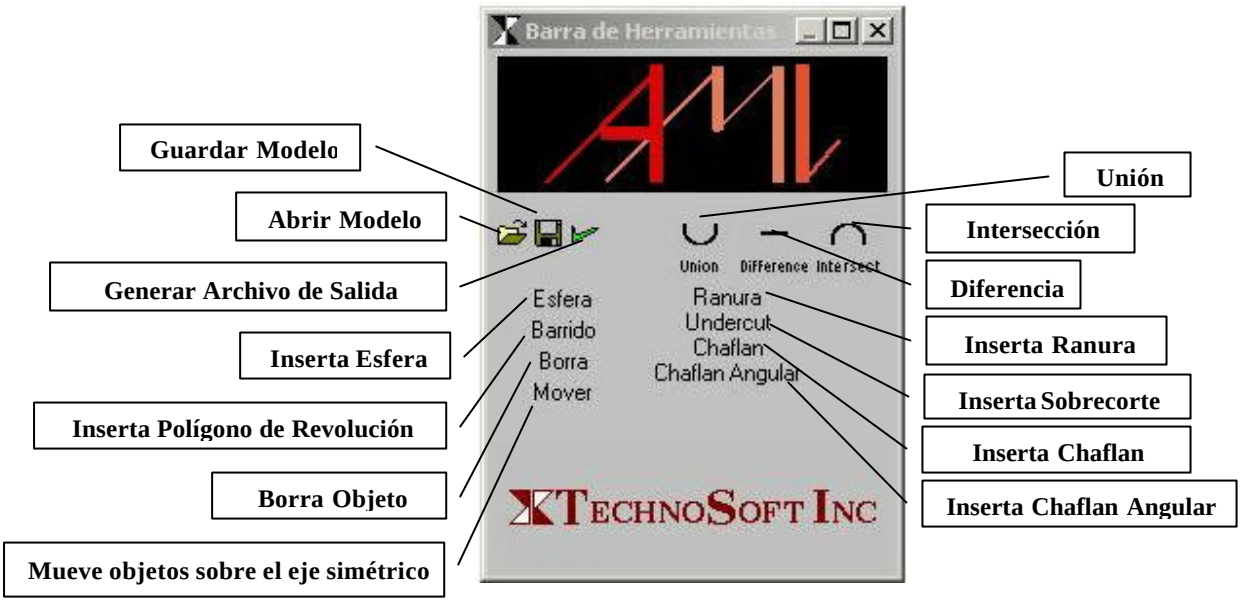


Figura 4. 5.- Descripción de las herramientas contenidas en la Barra de Herramientas.

Abrir un Modelo.

Para cargar un modelo preexistente, en la barra de herramientas, presionar el botón de "abrir modelo"  , el cual abre el cuadro de díalogo para seleccionar el modelo a cargar. Como valor predeterminado, se buscan los archivos con extensión ".mdl", pero se pueden cargar cualquier modelo con cualquier extensión que haya sido generado en AML **(Figura 4.6)**.

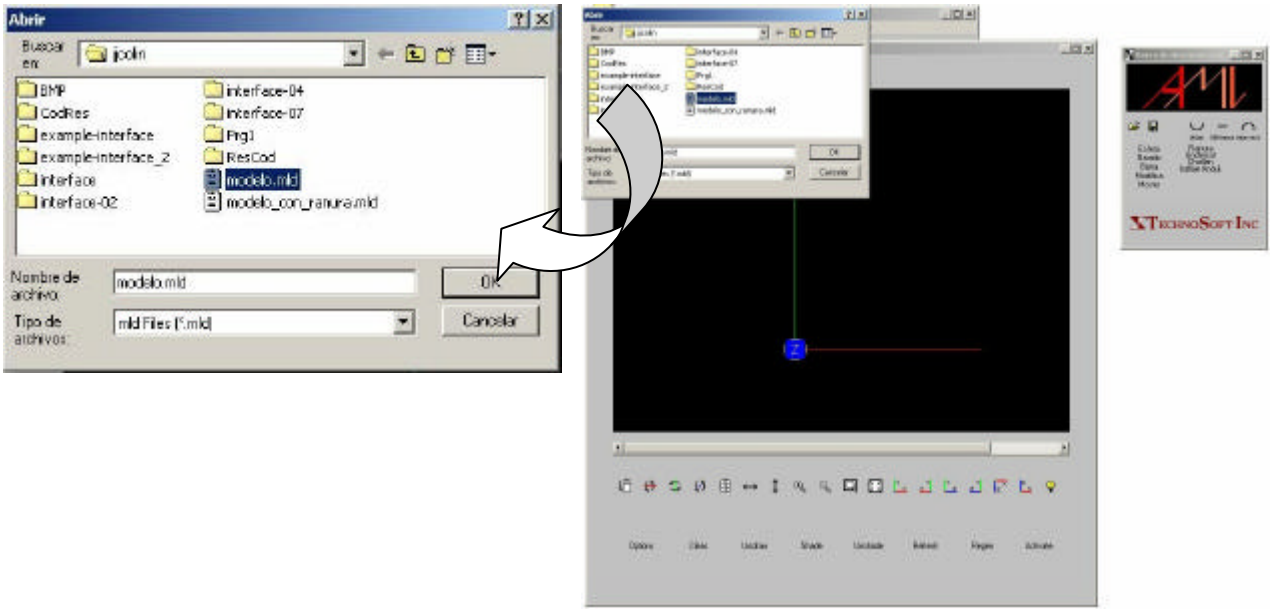


Figura 4. 6.- Ejemplo de cuadro de díalogo para abrir un modelo.

Guardar un Modelo.

Para guardar un modelo, en la barra de herramientas, presionar el botón "guardar modelo"  , el cual despliega el cuadro de díalogo para asignar un nombre y una extensión al archivo del modelo a guardar. Como valor predeterminado, se sugieren guardar los archivos con extensión ".mdl", pero se pueden guardar cualquier modelo con cualquier extensión **(Figura 4.7)**.

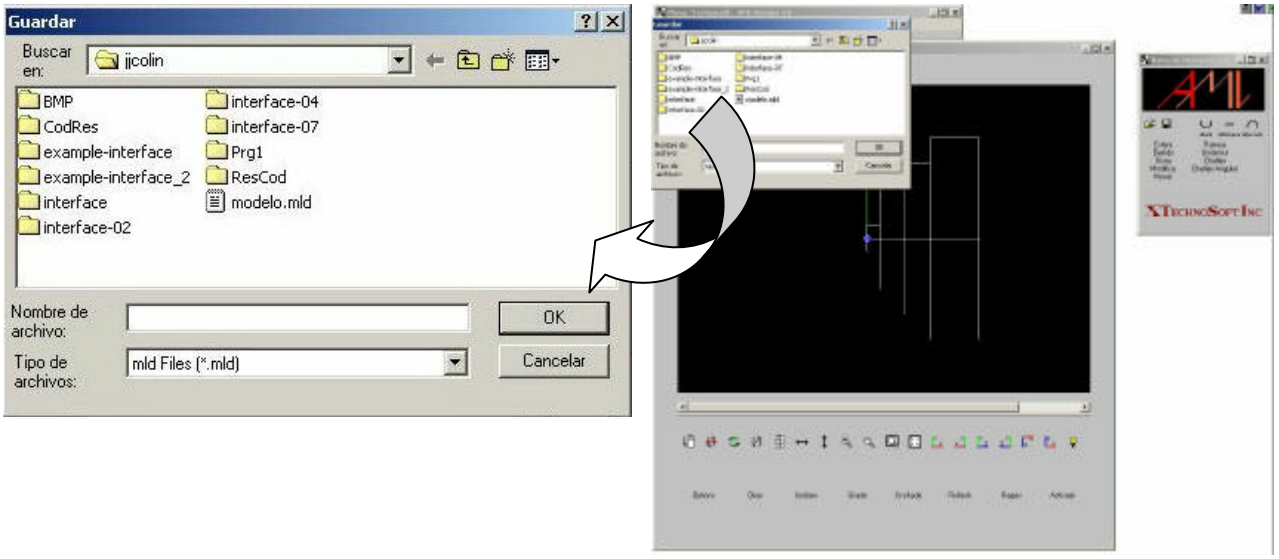


Figura 4. 7.- Ejemplo de cuadro de díalogo para guardar un modelo.

Generar Archivo de Salida.

Para generar un archivo de salida, presionar el botón "generar archivo de salida"  , el cual despliega un cuadro de díalogo para asignar un nombre y una extensión al archivo de salida a generar. Como valor predeterminado, se sugieren guardar los archivos con extensión ".sal", pero se puede guardar cualquier archivo generado con cualquier extensión. También hay que seleccionar el tipo de archivo que se generará, ya que se puede generar con la información basada en cilindros ó solamente puntos en el espacio. El archivo de salida contiene información de puntos, diámetros y distancias referentes al modelo realizado, que será usado por el proyecto SADET *(ver sección 3.2.)* **(ver Figura 4.8)**.

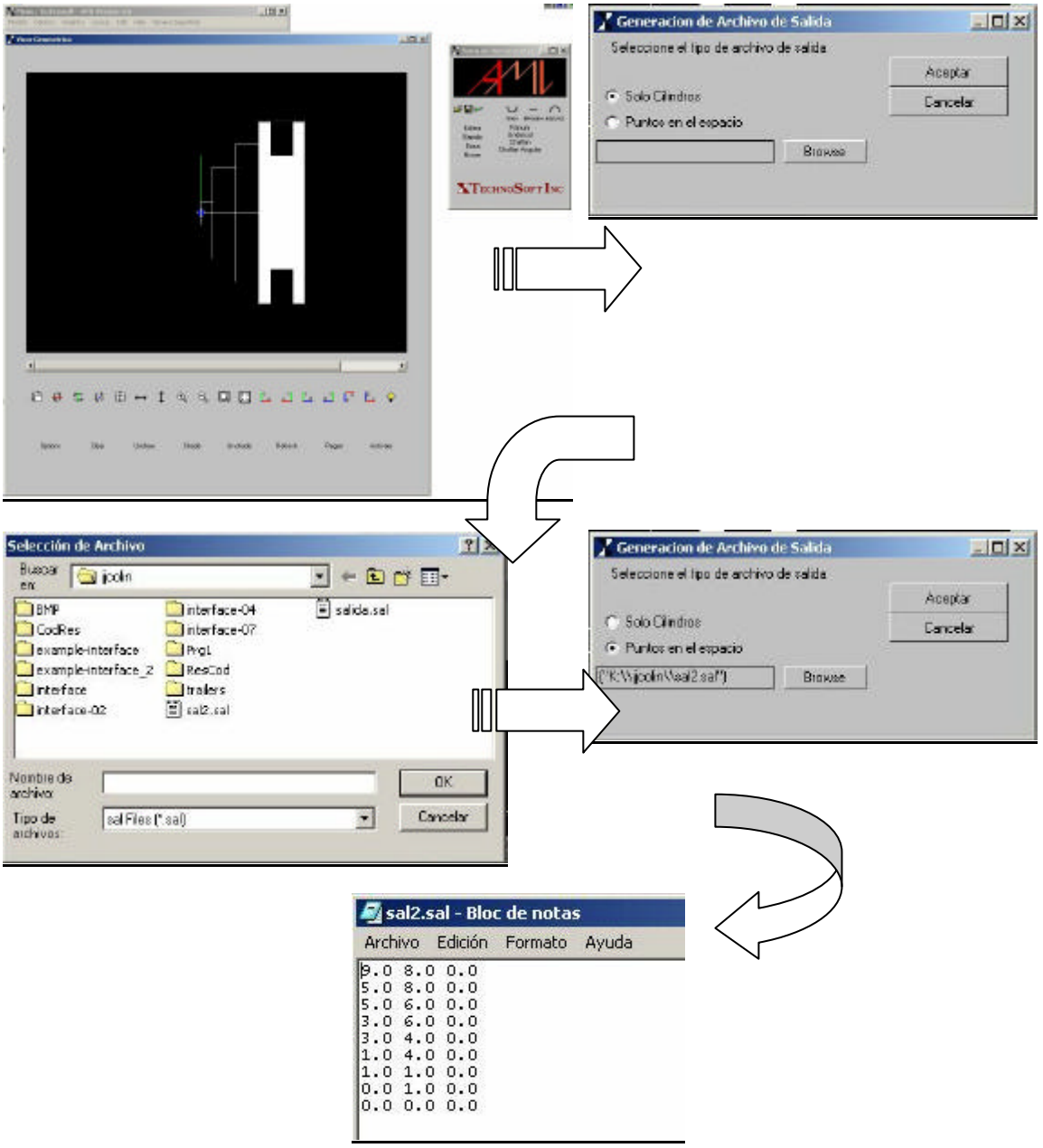


Figura 4. 8.- Ejemplo de cuadro de díalogo para generar un archivo de salida con información acerca del modelo.

Insertar Objetos.

Las siguientes herramientas permiten insertar objetos básicos en el espacio del canvas. Para realizar esta acción, basta presionar el botón correspondiente localizado en la barra de herramientas que despliega el formulario y proporcionar datos necesarios para realizar la petición.

La primera herramienta es ***inserta esfera***, que al momento de presionar el botón Esfera, se despliega el formulario que solicita *nombre de la esfera, radio de la esfera, coordenadas del centro de la esfera, color de la esfera*, y si es definida como un *sólido*. Sólo hay que proporcionar los datos solicitados y presionar "aceptar". Los cambios se ven reflejados en el canvas **(Figura 4.9)**.

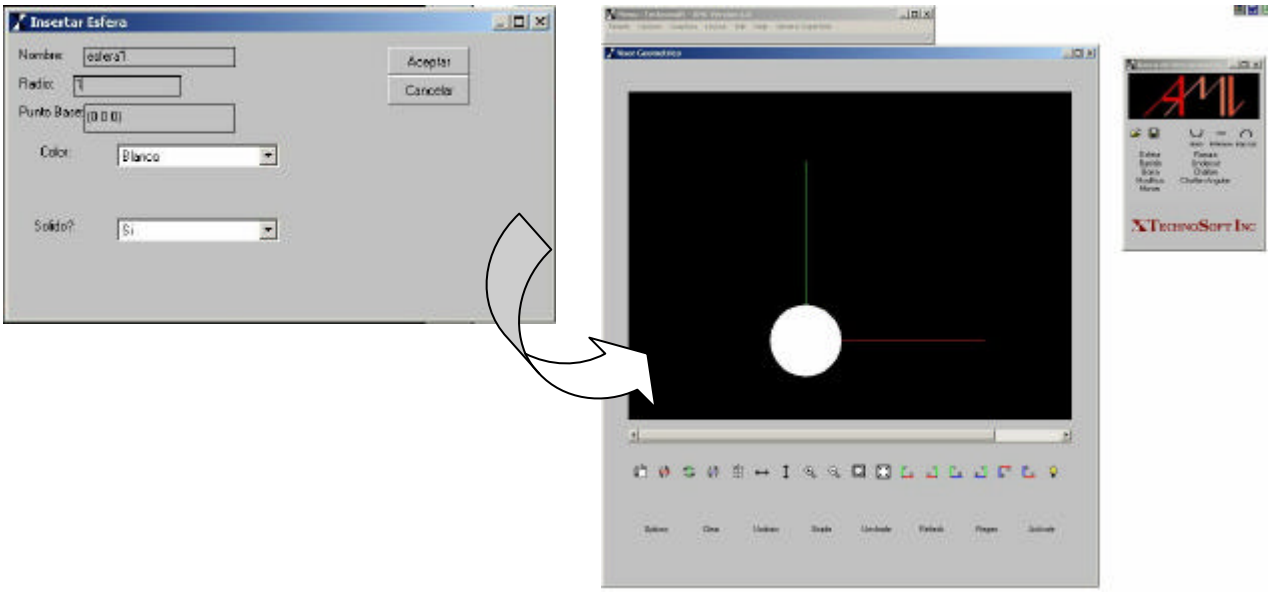


Figura 4.9.- Ejemplo de cuadro de díalogo para insertar una esfera.

La segunda herramienta es la que ***inserta un polígono barrido*** a largo del eje simétrico, que al momento de presionar el botón Barrido, se despliega el formulario que solicita el *nombre del polígono*, la opción de *editar los puntos de los vértices* ó de *cargar desde un archivo de texto*, si se forma un *sólo polígono* ó se parte en *múltiples polígonos*, la *lista de puntos* ó la *ruta del archivo de texto con los puntos*, según sea el caso. Sólo hay que proporcionar los datos solicitados y presionar "aceptar". Los cambios se ven reflejados en el canvas **(Figura 4.10).**

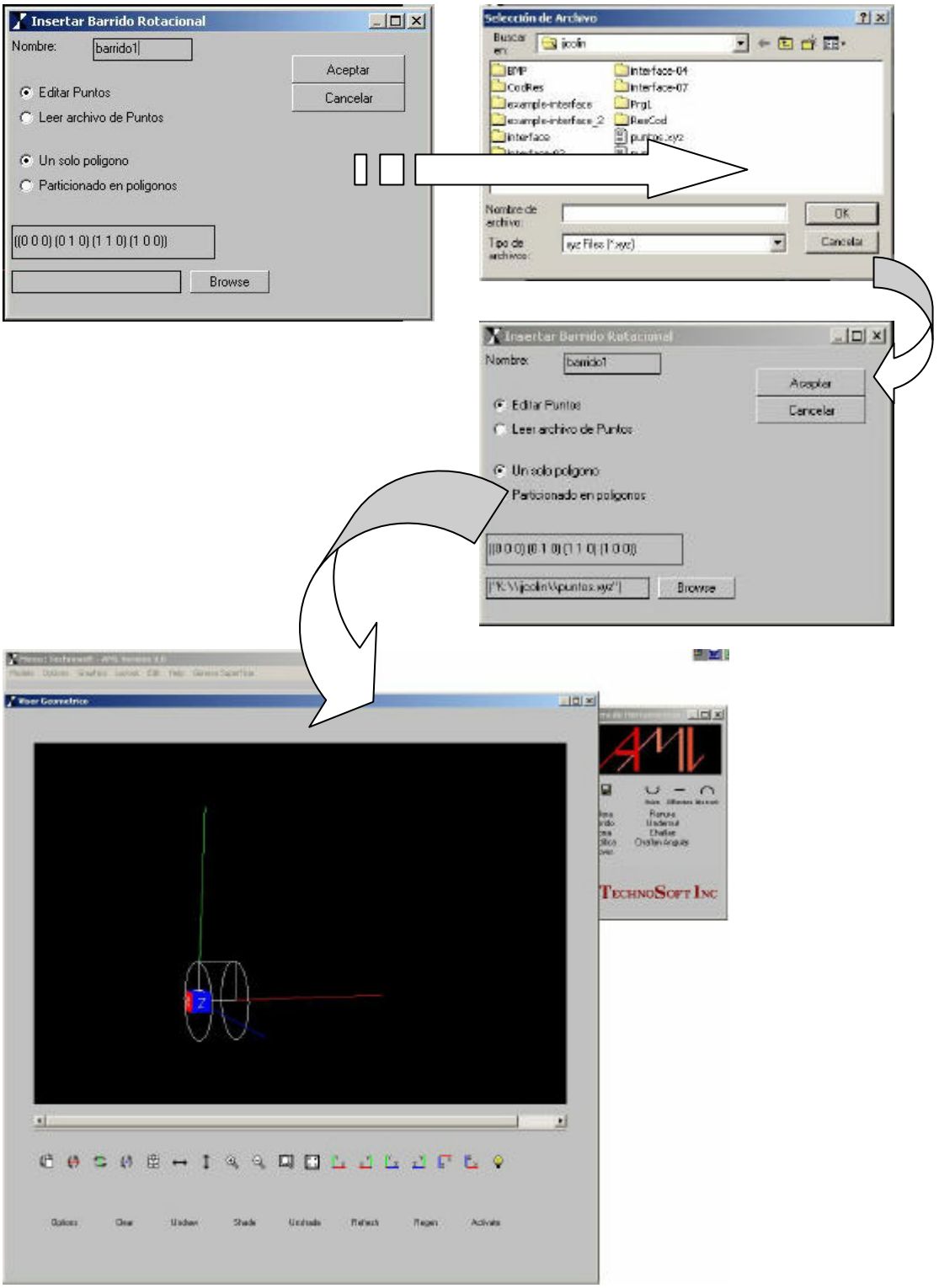


Figura 4. 10.- Ejemplo de cuadro de díalogo para insertar un polígono barrido alrededor de un eje.

Borrar Objetos.

Esta herramienta permite borrar un objeto del canvas. Para realizar esta acción, basta presionar el botón Borra, seleccionar el objeto, mediante el uso del ratón y dando clic izquierdo. Los cambios se ven reflejados en el canvas **(Figura 4.11).**

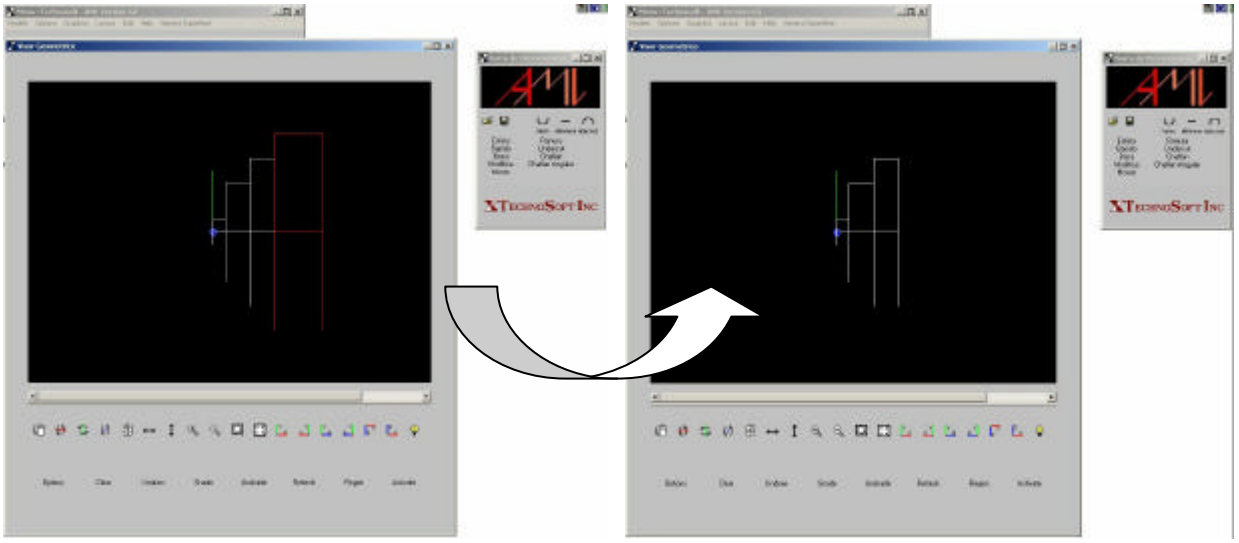
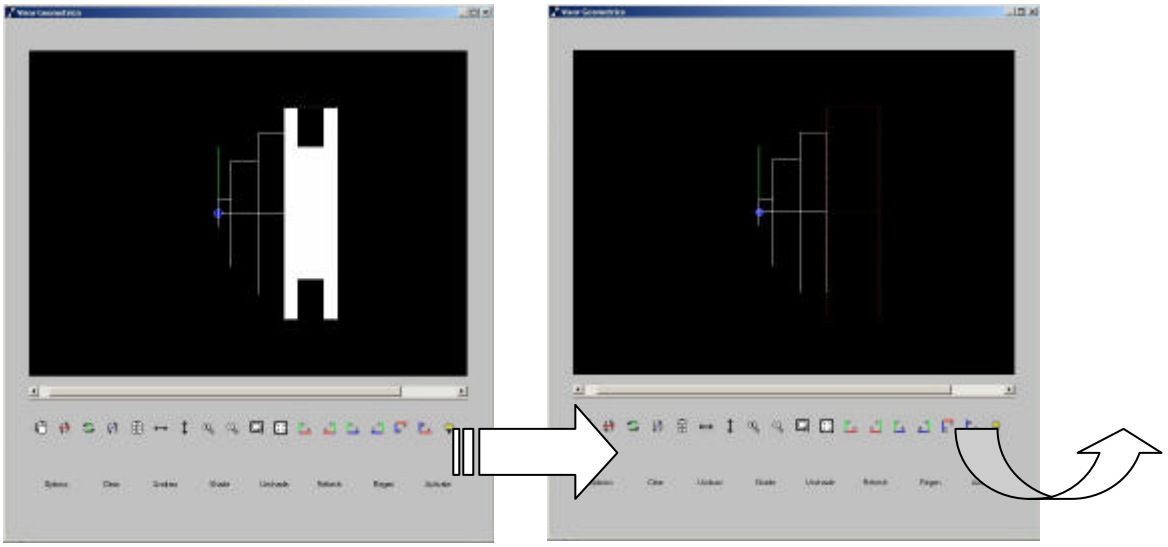


Figura 4. 11.- Ejemplo de procedimiento para borrar objetos.

Mover Objetos.

Esta herramienta permite mover un objeto del canvas de una posición a otra, pero sólo a lo largo del eje simétrico. para realizar esta acción, basta presionar el botón Mover, seleccionar el objeto que se desea mover y el objeto junto al cual se desea mover el primer objeto, mediante el uso del raton y dando clic izquierdo para seleccionar. Los cambios se veran reflejados en el canvas **(Figura 3.3.12).**



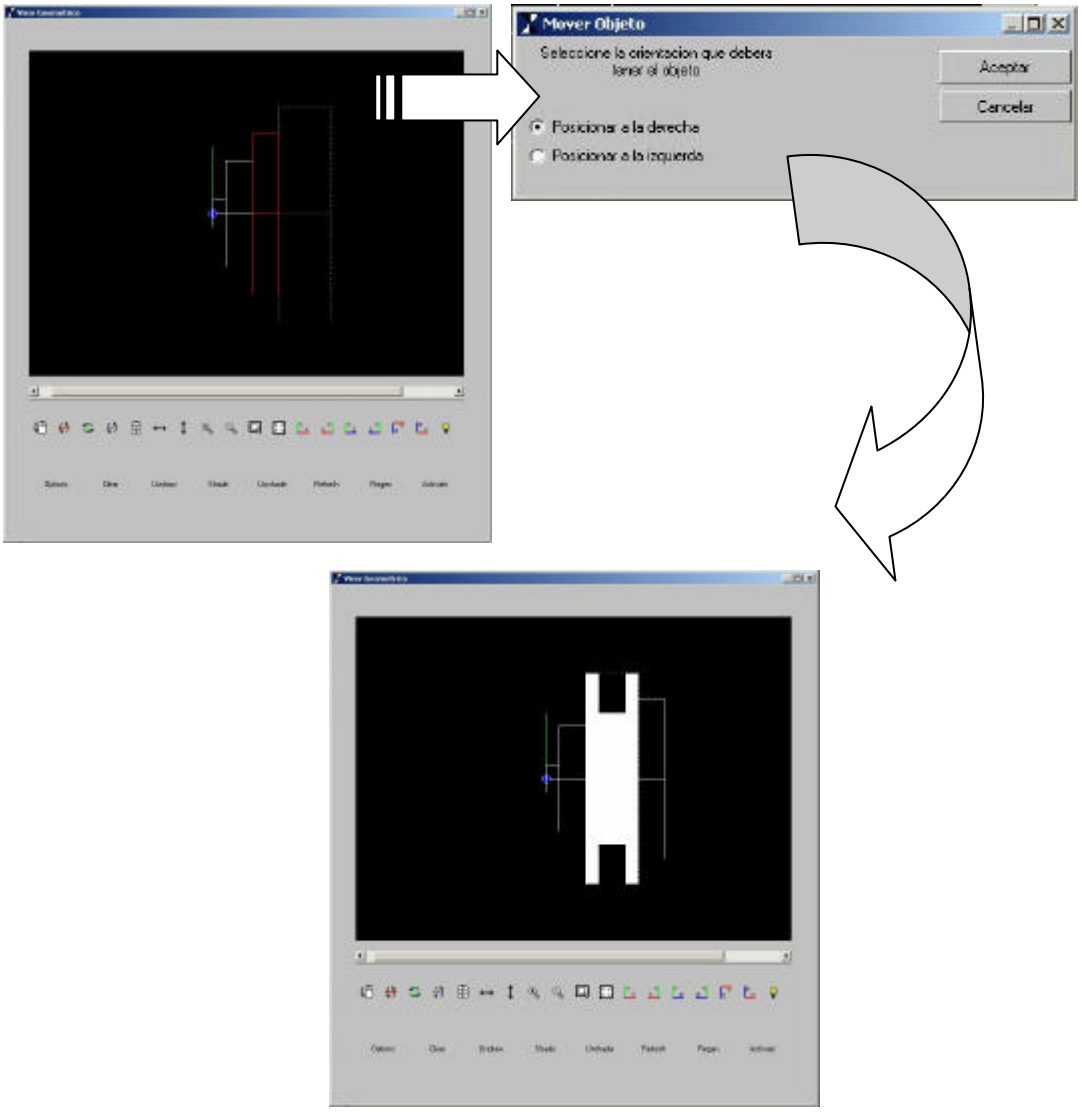


Figura 4. 12.- Ejemplo de procedimiento para mover objetos.

Insertar Operaciones Booleanas.

En la barra de herramientas hay tres botones **(Figura 4.13)** que permiten insertar tres tipos de operaciones booleanas: la unión **(Figura 4.14)**, la intersección **(Figura 4.15)** y la sustracción **(Figura 4.16)**. Para hacer uso de estas herramientas, hay que presionar el botón de la operación que deseamos realizar:

Figura 4. 13.- Botones de la barra de herramientas para insertar las operaciones booleanas de Unión, Intersección y Diferencia.

Después hay que seleccionar, mediante el uso del ratón y un clic izquierdo por cada uno, los objetos que participaran en la operación booleana y presionar un clic derecho para cerrar la selección. El resultado de las operaciones se ve reflejado inmediatamente en el canvas.

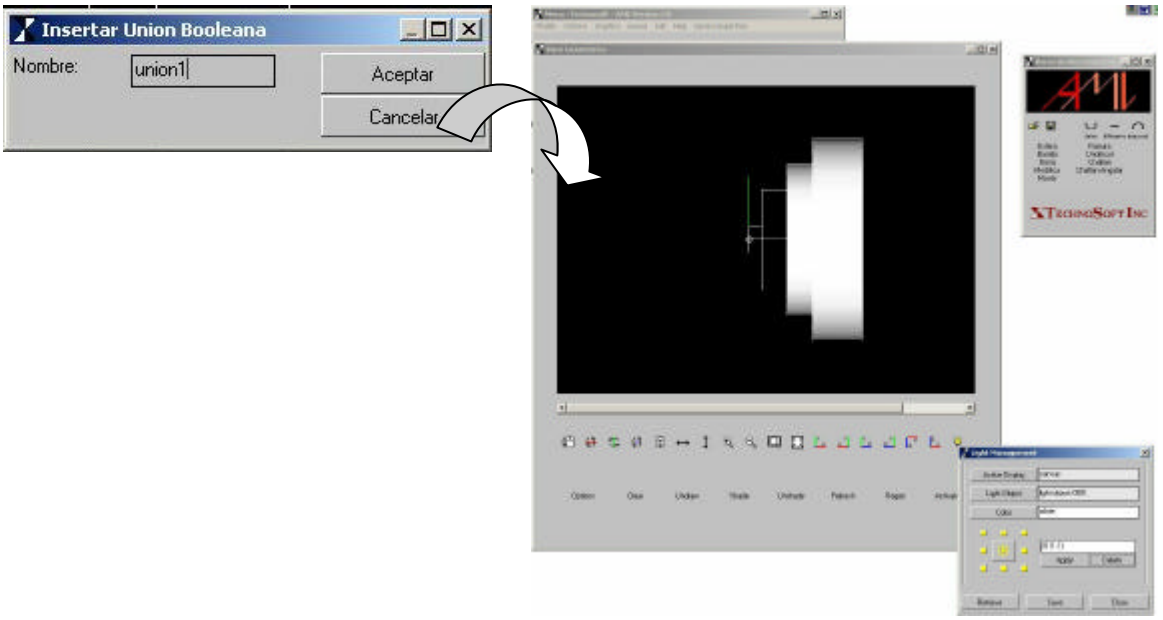


Figura 4. 14.- Ejemplo de cuadro de díalogo para insertar una unión entre objetos.

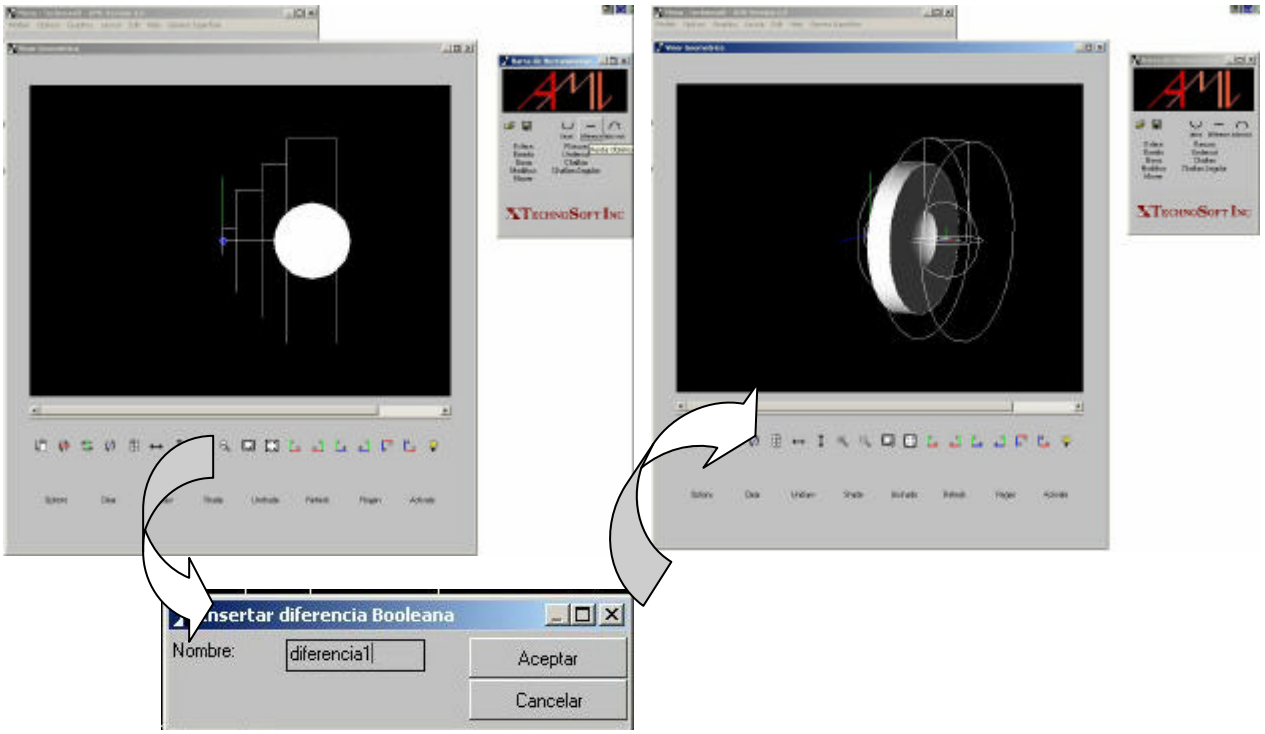


Figura 4. 15.- Ejemplo de cuadro de díalogo para insertar una Diferencia entre objetos.

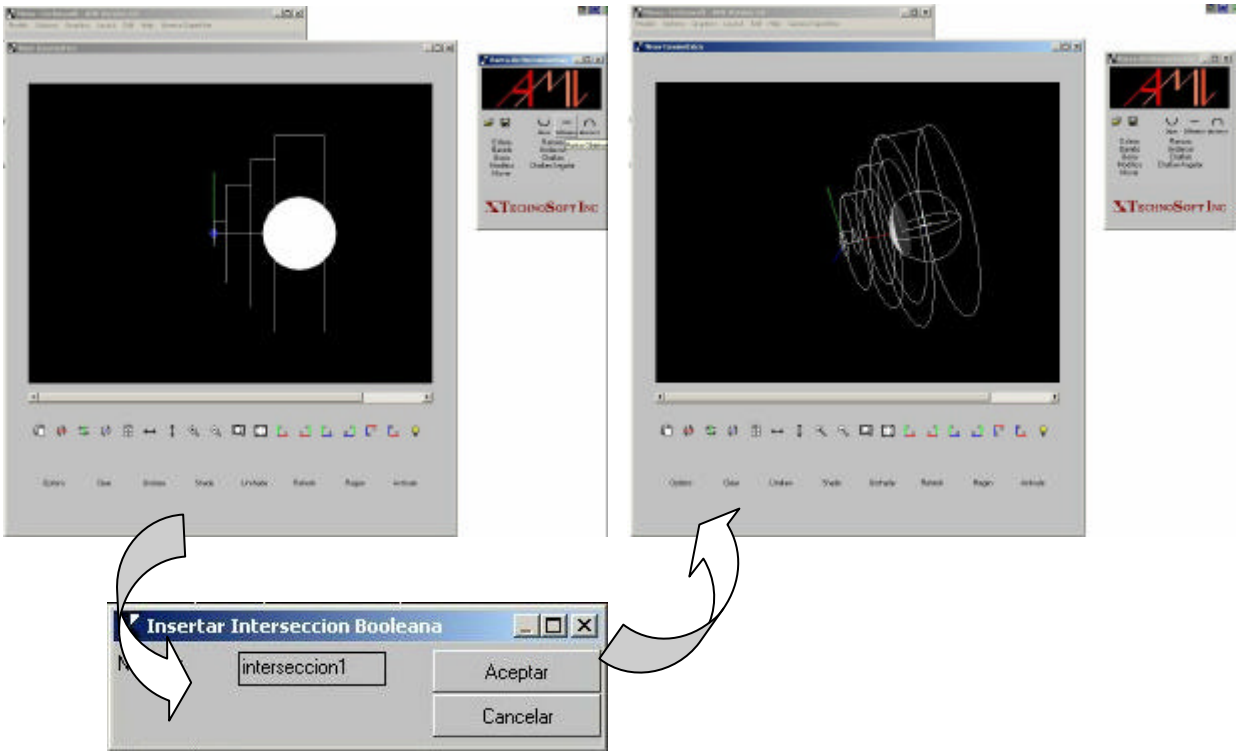


Figura 4. 16.- Ejemplo de cuadro de díalogo para insertar una intersección entre objetos.

Insertar Características Secundarias.

En la barra de herramientas hay cuatro botones que permiten insertar varios tipos de características secundarias: ***Ranura, Sobrecorte, Chaflán*** y ***Chaflán Angular.*** Estas características secundarias se aplican sólo sobre elementos primarios como los polígonos barridos sobre un sólo eje. Para poder hacer uso de estas herramientas, hay que presionar el botón de la operación que deseamos realizar, seleccionar, mediante el uso del ratón y un clic izquierdo, el objeto al cual se le aplicara una característica secundaria.

La herramienta ***Ranura*** inserta una ranura formada por un polígono de cuatro puntos. Después de presionar el botón Ranura, se despliega el formulario este solicita el *nombre de la ranura*, el *punto de aplicación de la ranura* sobre el objeto seleccionado, el *grosor de la ranura*, la *profundidad de la ranura*. Sólo basta con llenar los campos con los datos solicitados y presionar el botón "aceptar" del formulario. El resultado de las operaciones se ve reflejado inmediatamente en el canvas. **(Figura 4.17)**.

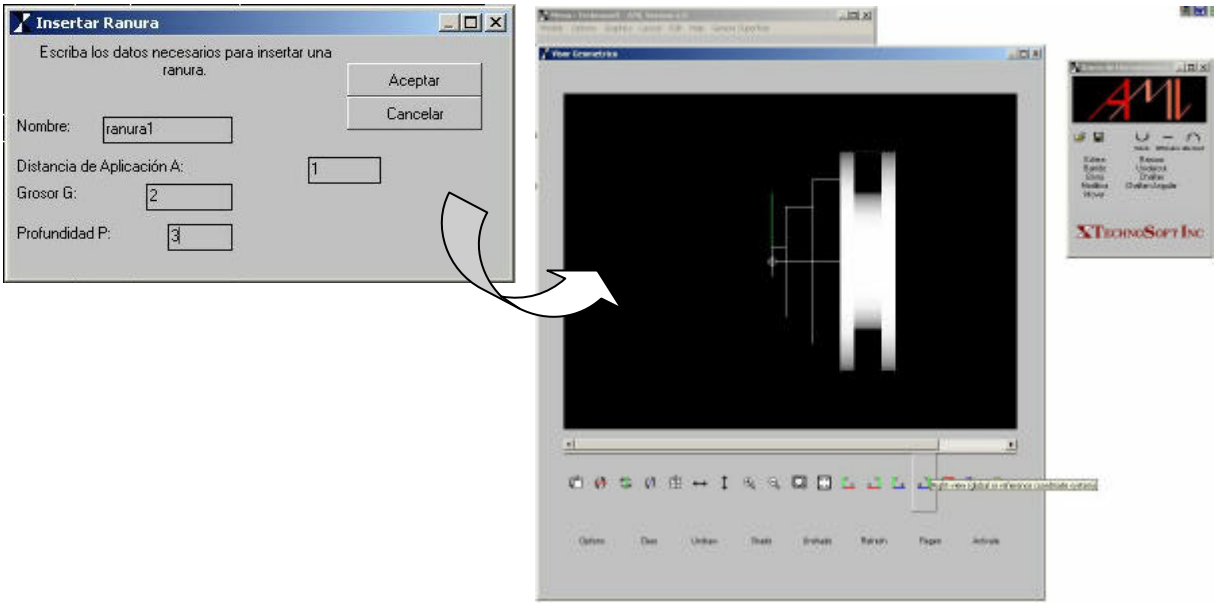


Figura 4. 17.- Ejemplo de cuadro de díalogo para insertar una Ranura en un objeto.

La herramienta ***Sobrecorte*** inserta una ranura formada por una curva de 180°. Después de presionar el botón Undercut, se despliega el formulario este solicita *el nombre del sobrecorte*, la *distancia de aplicación del sobrecorte* y el *radio del sobrecorte*. Sólo basta con llenar los campos con los datos solicitados y presionar el botón "aceptar" del formulario. El resultado de las operaciones se ve reflejado inmediatamente en el canvas. **(Figura 4.18)**.

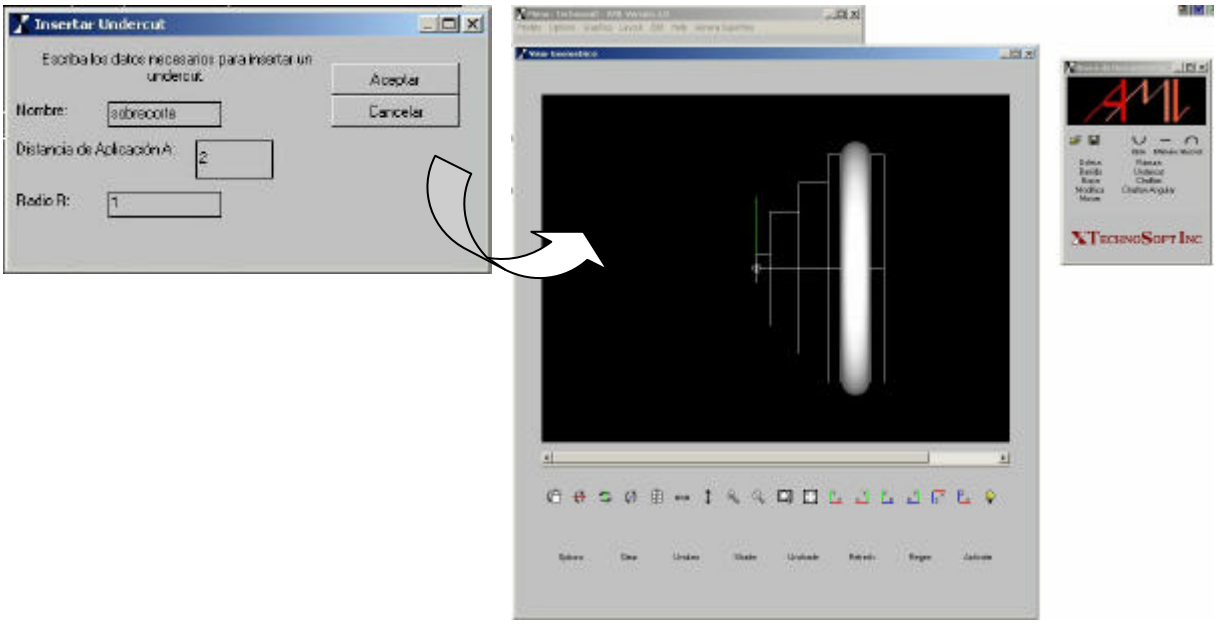


Figura 4. 18.- Ejemplo de cuadro de díalogo para insertar un Sobrecorte en un objeto.

La herramienta ***Chaflán*** inserta un chaflán formado por un arco cerrado rotado sobre el eje simétrico. Después de presionar el botón Chaflan, se selecciona, mediante el uso del ratón, el objeto al cual se le aplicara un chaflán, y se despliega el formulario este solicita el *nombre del chaflán*, el *radio del chaflán*, la *posición del chaflán* sobre el objeto ("derecha" o "izquierda") y si es de *tipo positivo* (se adiciona la geometría generada al objeto) ó de *tipo negativo* (se sustrae la geometría generada al objeto). Sólo basta con llenar los campos con los datos solicitados y presionar el botón "aceptar" del formulario. El resultado de las operaciones se ve reflejado inmediatamente en el canvas **(Figura 4.19 y Figura 4.20)**.

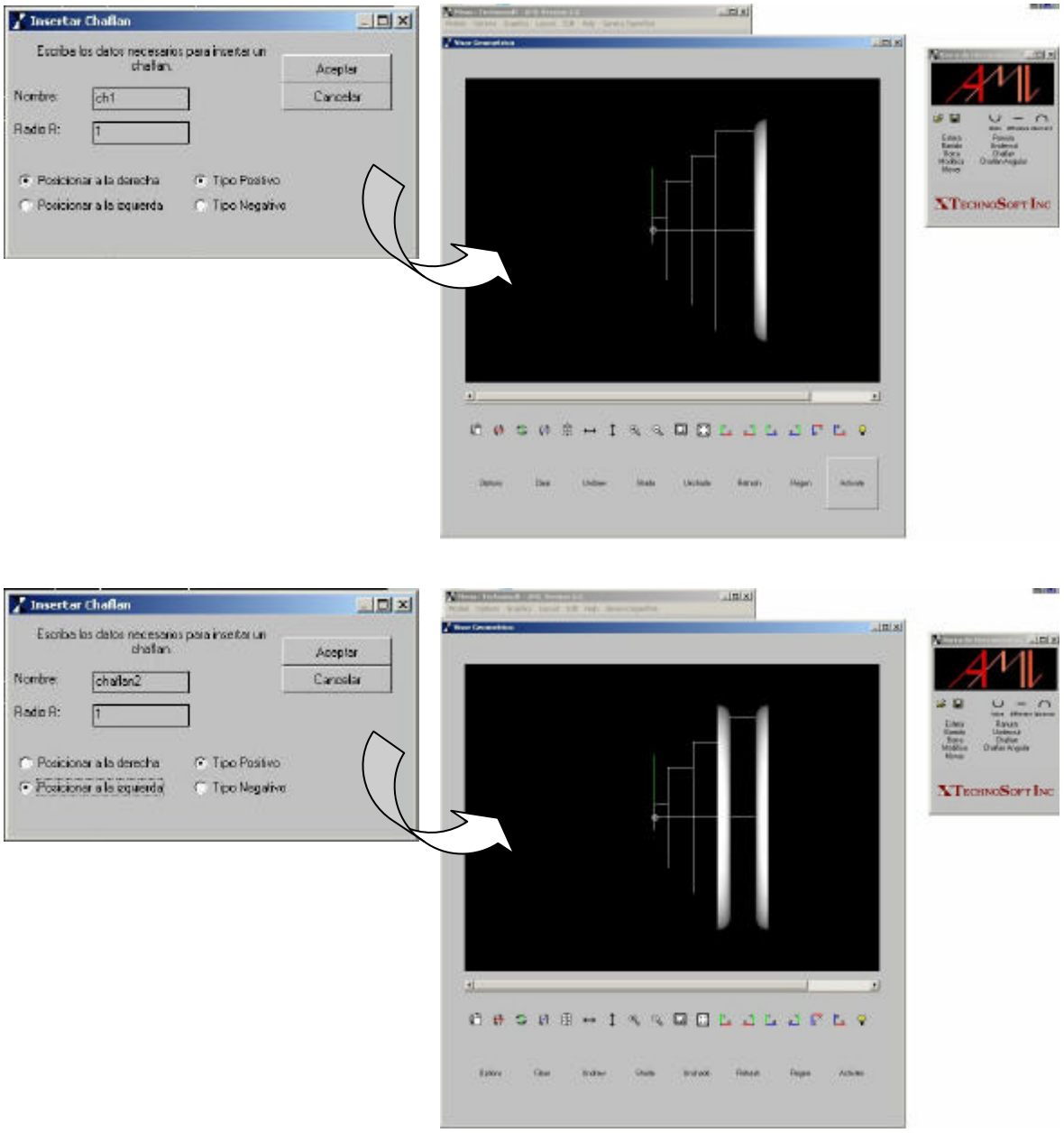


Figura 4. 19.- Ejemplo de cuadro de díalogo para insertar un Chaflán Positivo en un objeto.

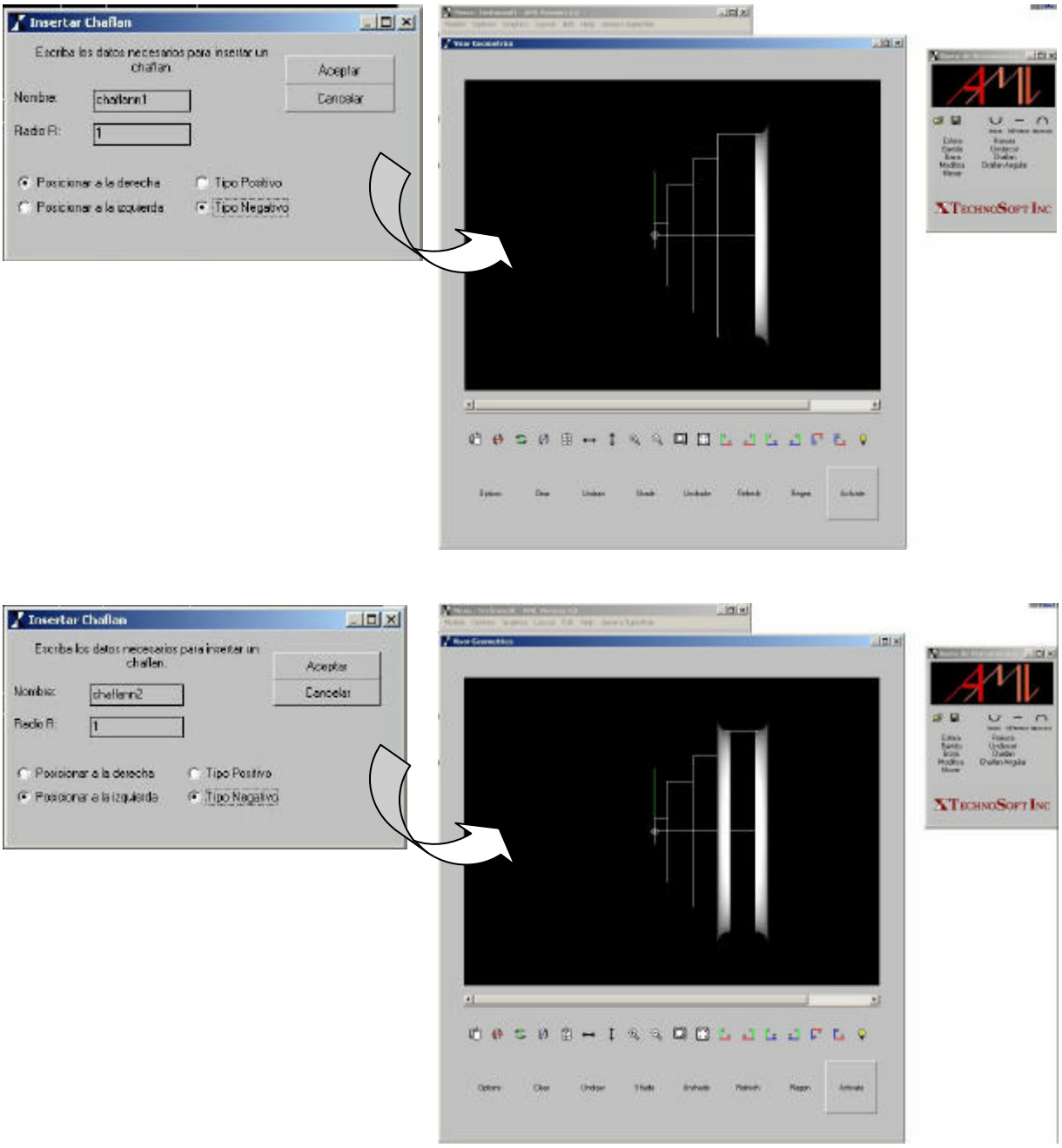


Figura 4. 20.- Ejemplo de cuadro de díalogo para insertar un Chaflán Negativo en un objeto.

La herramienta ***Chaflán Angular*** inserta un chaflán formado por un arco cerrado rotado sobre el eje simétrico. Después de presionar el botón Chaflan Angular, se despliega el formulario este solicita el *nombre del chaflán angular*, el *ángulo del chaflán* angular, la *distancia de aplicación del chaflán angular*, la *posición del chaflán angular* sobre el objeto ("derecha" o "izquierda") y si es de *tipo positivo* (se adiciona la geometría generada al objeto) ó de *tipo negativo* (se sustrae la geometría generada al objeto). Sólo basta con llenar los campos con los datos solicitados y presionar el botón "aceptar" del formulario. El resultado de las operaciones se ve reflejado inmediatamente en el canvas **(Figura 4.21 y Figura 4.22)**

Figura 4. 21.- Ejemplo de cuadro de díalogo para insertar un Chaflán Angular Positivo sobre un objeto.

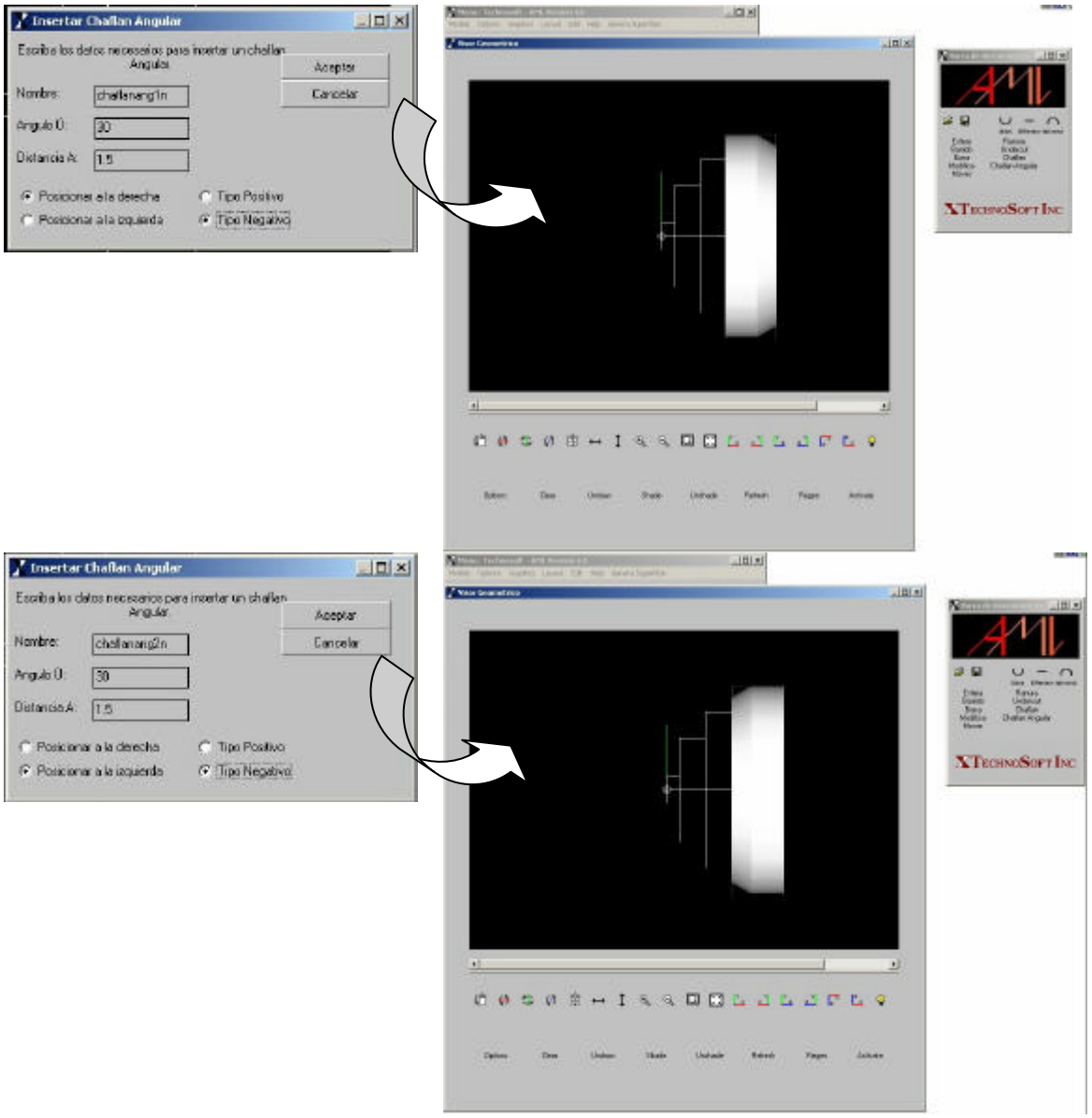


Figura 4. 22.- Ejemplo de cuadro de díalogo para insertar un Chaflán Angular Negativo en un objeto.

Capítulo 5.- Caso de estudio

5.1 Introducción.

En este capítulo se definirá las características del caso de estudio para demostrar que la aplicación de la teoría, junto con la programación de la interfaz gráfica desarrollada en AML, son suficientes para resolver una de las necesidades que presenta el proyecto SADET (ver sección 3.2.).

También en este capítulo se mostrará la operación de la interfaz desarrollada en AML, se instanciaran las características primarias de una flecha propuesta, usando como base, datos obtenidos del sistema SADET. Esta pieza será instanciada en el sistema desarrollado en AML, se le aplicarán características secundarias mediante el uso del método de generación de polígonos barridos a lo largo del eje simétrico, como ranuras y chaflanes. Así también se modelará la figura en la interfaz gráfica de AML. Se generará un archivo de salida con información sobre los puntos en el espacio proyectados sobre el plano XY, este archivo servirá para actualizar la base de datos del sistema SADET.

También, definiremos los procedimientos de mantenimiento y liberación para la interfaz gráfica desarrollada en AML.

5.2 Definición del caso de estudio.

Para realizar las pruebas de sistema para la interfaz desarrollada en el entorno gráfico de AML, se propone usar los datos obtenidos, del sistema SADET (ver sección 3.2.), de una pieza mecánica conocida como "flecha", que esta formada por una serie de cilindros en línea, compartiendo el mismo centro, de diferentes largos y radios **(ver figura 5.1.)**.

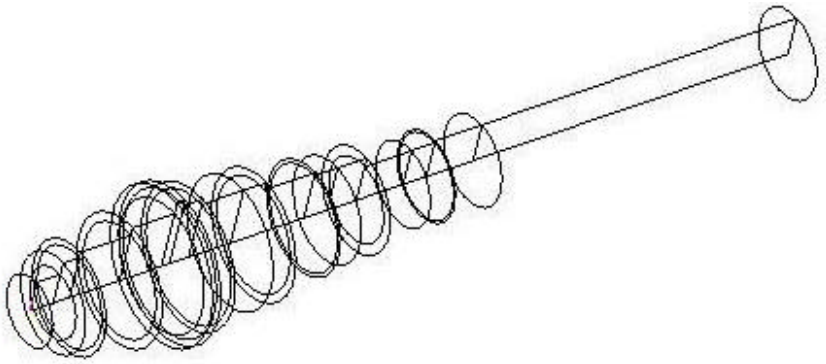


Figura 5. 1.- Pieza Mecánica "Flecha", que será usada en el caso de estudio.

Esta pieza mecánica será instanciada en la interfaz gráfica desarrollada en AML, pero sólo se instanciara las piezas primarias. Con las herramientas desarrolladas, se le aplicara a la flecha instanciada, características secundarias como ranuras, chaflanes y sobrecortes. Como acto final, se generará el archivo de salida, con el cual, será actualizada la base de datos del sistema SADET.

5.3 Pruebas del sistema.

Para realizar las pruebas del sistema necesitaremos definir el archivo de entrada, que contará con los datos relacionados a los vértices del polígono proyectado sobre el plano XY de la pieza mecánica "flecha" **(ver figura 5.2)**.

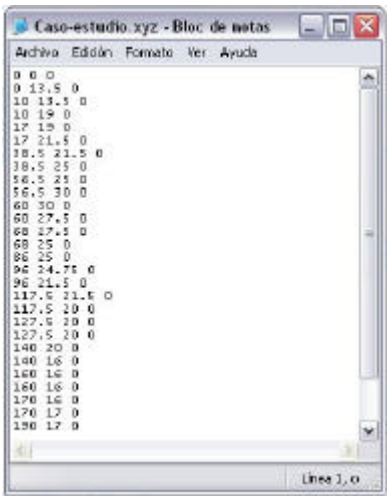
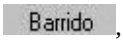


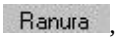
Figura 5. 2.- Archivo de Entrada ".xyz" con información de la flecha a instanciar.

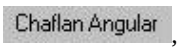
También, necesitaremos tener la interfaz gráfica desarrollada cargada en el entorno GUI de AML.

1) Abrir el editor gráfico "x-emacs" de AML.
2) Ejecutar AML mediante el uso de la herramienta "Run AML" 

.
3) Mediante el uso de la herramienta "load file" 

, cargar el sistema de interfaz (archivo *.aml), en el entorno gráfico de AML **(Ver Figura 5.3)**.
4) En la ventana de "linea de comando", escribir en nombre de la funcion principal entre parestesis "(gen-surface)", y presionar <<Enter>>.
5) Después de haber inicializado AML-GUI, se despliega la barra de menú de AML. Se ejecuta la opción "Editor de Geometria" del menú "Genera Modelo", que desplegara la barra de herramientas, el canvas y la barra de control de gráficos.
6) Utilizando la herramienta "inserta barrido" 

, se desplegara el cuadro de díalogo que solicita la información necesaria para abrir el archivo de entrada de datos, con extensión ".xyz". Con los puntos definidos en este archivo, se instanciaran los poligonos barridos alrededor del eje simétrico, que generaran los sólidos que conformarán la pieza mecanica "flecha" **(Ver Figura 5.4)**.
7) Utilizando la herramienta "inserta ranura" 

, se despliega el cuadro de díalogo que solicita los datos necesarios para insertar ranuras sobre objetos en el modelo. Se insertan las ranuras necesarias para obtener el modelo deseado **(Ver Figura 5.5)**.
8) Utilizando la herramienta "inserta chaflan angular" 

, se despliega el cuadro de díalogo que solicita los datos necesarios para insertar chaflanes en base a datos angulares sobre objetos en el modelo. Se insertan los chaflanes necesarios para obtener el modelo deseado **(Ver Figura 5.6)**.

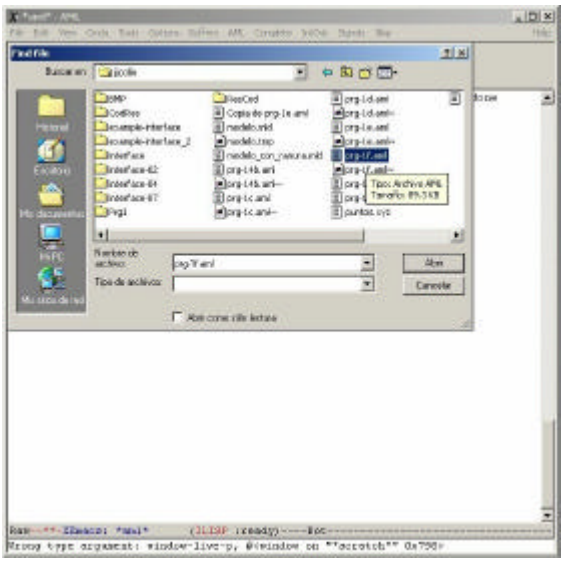


Figura 5. 3.- Selección de archivo ".aml" con Codigo de interfaz gráfica.

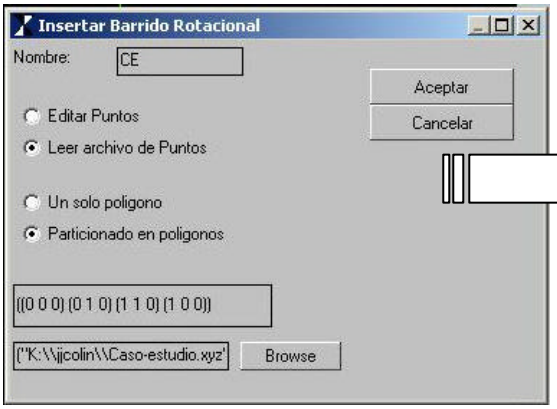


Figura 5. 4.- Cuadro de díalogo para insertar Poligonos Barridos alrededor del eje simétrico.

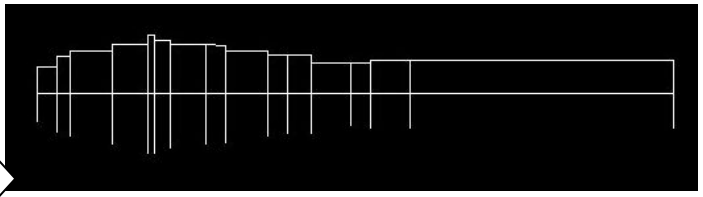


Figura 5. 5.- Poligonos barridos generados con el archivo de entrada.

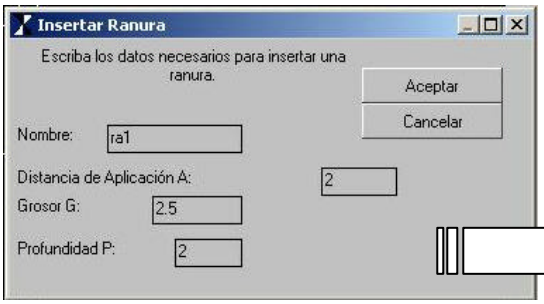


Figura 5. 6.- Cuadro de díalogo para insertar ranuras.

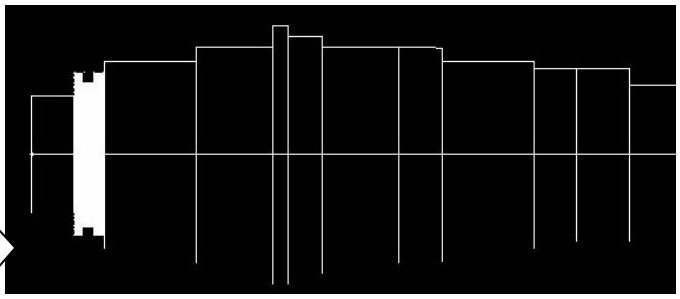


Figura 5. 7.- Ranura insertada sobre el modelo de la "flecha".

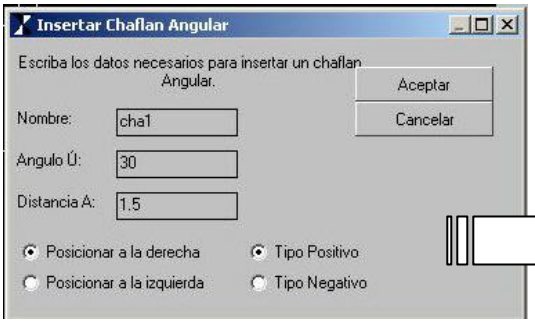


Figura 5. 8.- Cuadro de dialogo para insertar chaflanes.

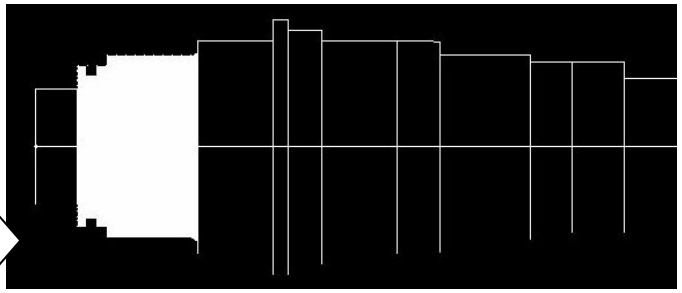


Figura 5. 9.- Chaflan insertado sobre el modelo de la "flecha".

Cada vez que se usa una herramienta, se inserta una característica secundaria. Después de un proceso más extenso utilizando estas herramientas, se obtiene el modelo final de la pieza mecánica "flecha" **(ver figura 5.7)**.

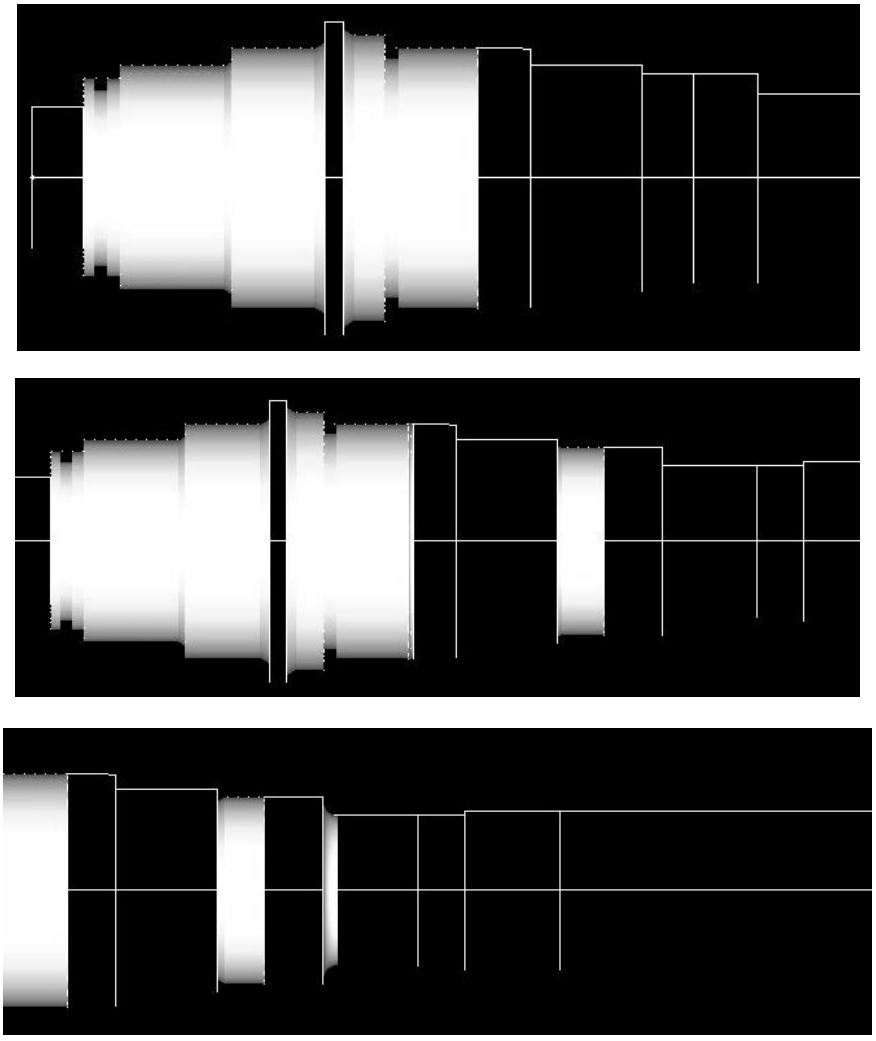


Figura 5. 10.- Avance del modelado de la pieza mecanica "flecha".

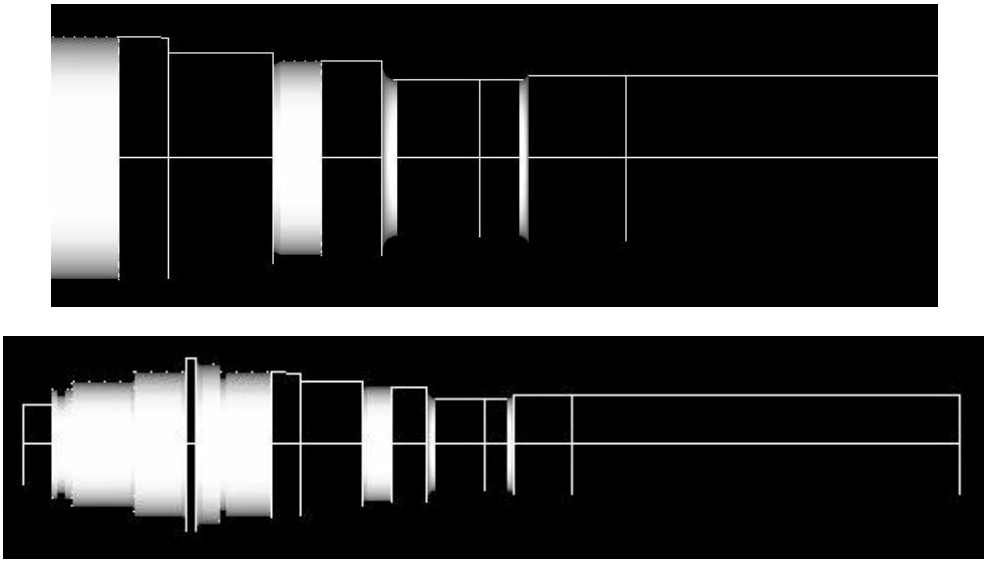


Figura 5. 11.- Avance del modelado de la pieza mecanica "flecha".

Después de modelar la pieza mecánica, se procede a generar el archivo de salida, con los puntos relacionados con la geometría del modelo de la pieza mecánica "flecha". Utilizando la herramienta "genera archivo de salida", se despliega el cuadro de díalogo que define el tipo de archivo de salida, así como el nombre y ruta del archivo nuevo a crear. Después de presionar "Aceptar", el archivo es generado **(ver figura 5.8)**.

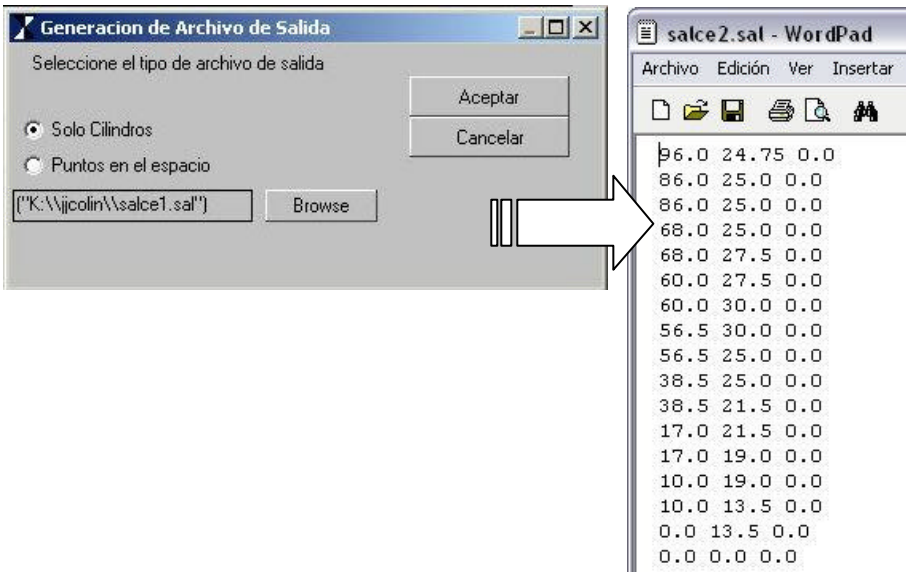


Figura 5. 12.- Generación del archivo de salida del modelo "flecha".

5.4 Mantenimiento.

Las acciones a realizar para el mantenimiento de la interfaz gráfica desarrollada en AML, son las siguientes:

1) Programar clases que definan más objetos geométricos como cubos, cilindros y otros.
2) Programar herramientas que permitan insertar en el modelo más objetos geométricos.
3) Programar clases que definan objetos geométricos de propósito general, como partes de ejes de transmision predefinidos y paramétrizados.
4) Programar la herramienta que permita conectar la interfaz gráfica desarrollada en AML, con la base de datos de SADET, a travez de herramientas ODBC, y poder desplegar modelos de piezas directamente.

Como se puede observar, las acciones de mantenimiento estan enfocadas en convertir la interfaz desarrollada en AML, en una aplicación de proposito particular. Dependiendo de nuevas necesidades, se pueden hacer crecer las capacidades del sistema.

5.5 Liberación.

Al momento de finalizar este trabajo de tesis, se puede decir que el sistema tiene una funcionalidad de 85 a 90 por ciento. Esto es debido a que hacen falta pruebas de caja negra donde intervengan usuarios finales, a fin de corregir errores. También faltan las actividades descritas en la **sección 5.4.**

Conclusiones y Tr abajos Futuros

Conclusiones

El objetivo de este trabajo se ha cumplido en un 100%, debido a que se ha programado la interfaz gráfica utilizando el lenguaje de programación AML. El desarrollo de esta interfaz incluye:

- El canvas, control de gráficos y b arra de herramientas;
- herramientas que realizan operaciones booleanas de unión, intersección y diferencia;
- herramientas que insertan objetos primarios en el modelo (polígonos barridos alrededor de un eje de simetría);
- herramientas que insertan objetos secundarios en el modelo (chaflanes, sobrecortes y ranuras);
- métodos para generar modelos a partir de un archivo de entrada y generar archivos de salida con información de la geometría del modelo en pantalla.

Haber estudiado la carrera de Ingeniería en Computación me ayudo a desarrollar este sistema, ya que esta me aporto el conocimiento de computadoras y programación, así como Ingeniería de Sistemas y Software, para desarrollar el sistema, y al mismo tiempo, poder desarrollar una técnica y disciplina, para escribir el código que forma parte del proyecto. También obtuve nuevo conocimiento acerca del lenguaje de modelado unificado UML, del proceso de especificación de requerimientos, del desarrollo de sistemas, así como de las pruebas de caja blanca y caja negra que son realizadas para poder asegurar la calidad del proyecto.

También me aporto el conocimiento científico para poder realizar la investigación necesaria que permitiría obtener los conocimientos necesarios para poder desarrollar este proyecto, ya que muchos de los antecedentes obtenidos pertenecen a la carrera de Ingeniería Mecánica, entre ellos los pertenecientes a la teoría de sistemas CAD / CAM y entidades geométricas.

Los conocimientos aportados por la carrera de Ingeniería en Computación han sido enormes y de gran utilidad para la realización de este proyecto. Solo recomiendo, que para futuras revisiones del plan de estudios de la carrera de Ingeniería en Computación, se busque darle un mayor espacio a la programación orientada a objetos, a la conexión entre bases de datos alojadas en servidores y en sistemas que se encuentran residentes en el dominio de una red local ó en la Internet. Entre las actividades que me hubiera gustado haber realizado durante la carrera es haber realizado algún proyecto para alguna asignatura que implicara programar en un lenguaje orientado a objetos, como JAVA. Afortunadamente, la carrera de Ingeniería en Computación me preparo lo suficiente para poder obtener conocimientos sobre la programación orientada a objetos, así como poder comprender la estructura del lenguaje de programación AML.

Entre los beneficios obtenidos por haber realizado este trabajo se cuentan la adquisición de nuevo conocimiento, principalmente sobre sistemas CAD / CAM, entidades geométricas, AML y programación orientada a objetos. Además, realizar este trabajo de tesis me acerco a una de las facetas de la carrera de ingeniería mecánica, el diseño mecánico asistido por computadora y me permitió ejercitar esa habilidad del ingeniero en computación, en la cual debe en algunas ocasiones trabajar con profesionales de otras áreas de conocimiento.

Entre los problemas a resolver que se presentaron durante la elaboración de este trabajo y el desarrollo de este proyecto, el mas representativo, fue que para poder utilizar el lenguaje de programación AML, es necesario configurar la licencia académica en el servidor disponible en el laboratorio de LIMAC, ubicado en el CDM de la Facultad de Ingeniería. Utilizando el servidor de licencias FlexLM, se configura la licencia que la empresa Technosoft provee. La configuración de la licencia es un proceso difícil, ya que esta presenta problemas diversos como que el periodo de uso de la licencia es corto, es necesario renovar la licencia y verificar el contenido de esta, así como, el servidor ha sido atacado por virus y hackers, lo que hace que además se tuviera que reparar la configuración del mis mo servidor en varias ocasiones.

Trabajos Futuros.

Existen trabajos futuros para este proyecto, los cuales servirán para mejorar el desempeño de la interfaz realizada en el lenguaje de programación AML:

- Programar las clases y métodos necesarios para que la interfaz se conecte directamente con la base de datos del proyecto SADET *(ver sección 3.2.)*;.
- Programar más clases y herramientas para insertar más objetos geométricos como cubos, cilindros, etc.;
- Programar los métodos y clases necesarios para obtener datos calculados acerca del modelo, como centros de masa, centros de inercia, áreas, volúmenes, cantidad de material necesario para manufacturarlo, etc.;
- Programar clases y métodos que permitan insertar partes de ejes de transmisión ya predefinidos y con parámetros ajustables, según lo necesite el usuario.

Apéndice A: Constructores en AML.

En la siguiente sección se detallara la sintaxis de los constructores en AML para definir clases y/o objetos.

Notas:

- AML no es sensible a las mayúsculas.
- Los tipos de datos no son verificados en el momento de compilación en AML, pero algunos tipos de datos son esperados, como string, symbol, list, number o object.
- "(in-package:aml)" es requerido al principio de todo archivo de AML.
- AML es un lenguaje interpretado y compilado.

OBJECT [Clase].

Object es el nivel más alto de clase de la cual las clases definidas por el usuario deben heredar. Mientras se desarrolla código fuente en AML, las clases no deben heredar de ninguna clase que sea una súper-clase de un *Object*. *Object* es principalmente usado para definir clases que no tendrán ninguna geometría asociada ó representación gráfica. Frecuentemente, esta clase es usada para definir clases que serán mezcladas con otras clases cuando se definen objetos. Muchas de las clases predefinidas en AML tienen *Object* como una súper-clase.

DEFINE-CLASS [Constructor].

Define-class, un constructor fundamental en AML, es usado para describir las estructuras de las nuevas clases. Todas las clases en AML predefinidas y definidas por el usuario son definidas usando este constructor, incluyendo aquellas en la interfaz de usuario.

En cualquier *Define-class*, lo siguiente debe ser especificado:

1) La clase ó clases de las cuales la nueva clase debe heredar (súper-clases). Esto es requerido para todas las nuevas definiciones de clases.
2) Las propiedades de la nueva clase y sus formulas (atributos).
3) Los subobjetos de la nueva clase (hijos).

Las propiedades y los subobjetos dados en la definición de clase se adicionan a aquellos que existen en las definiciones de clases súper-clases. Propiedades y subobjetos de una superclase serán remplazados (sobrescritos) si tienen el mismo nombre que cualquiera dado en la nueva definición de clase. Una vez definida usando el constructor *Define-class*, instancias de las nuevas clases pueden ser creadas. Crear una instancia es hecho usando los comandos *create-model* y *add-object*. *Create-model* creara una nueva instancia de la clase y hará esa instancia el principio de un nuevo modelo (objeto raíz del modelo). *Add-object* es usado para crear una instancia de una clase como un subobjeto de un objeto que ya existe.

Formato:

```
(DEFINE-CLASS nombre-de-la-clase
: inherit-from ()
[:properties (especificación-de-propiedades)]
[:subobjetos (lista-de-especificaciones-de-subobjetos)]
)
```

Argumentos:

nombre-de-la-clase Cualquier símbolo puede ser usado como nombre de la clase a ser definida.

:inherit-from	Una lista de clases predefinidas de donde hereda esta clase.
:properties	Una lista de especificaciones de propiedades. Cada especificación puede ser un simple par propiedad/formula o una lista de especificación de objetos que contiene el nombre, clase y el par propiedad/formula para la propiedad del objeto.
:subobjetos	Una lista de listas de especificación de subobjetos conteniendo el nombre, clase, y par propiedad/formula para cada subobjeto.

2.5.4.1 Especificando Herencia.

La herencia es un mecanismo para el reuso de clases. A través de la herencia, una clase tendrá las mismas propiedades, subobjetos y métodos que las clases de las cuales esta heredando de (sus superclases). La sección *: inherit-from* del constructor *Define-class* acepta una lista de clases a ser usada como superclases. Si una propiedad, subobjeto o método es presentado en más de una de sus superclases, el orden de precedencia es de izquierda a derecha. Esto significa que si un *: inherit-from* contiene la lista (box-object cylinder-object), la geometría *box-object* será creada. Si el orden de la lista invertido a (cylinder-object box-object), la geometría de *Cylinder-object* será creada. Si una clase se quiere usar para mejorar otras clases, esta debe heredar de la clase *object*.

2.5.4.2 Especificando Propiedades.

La sección opcional *:properties* del constructor *Define-class* es usada para asignar propiedades y sus formulas asociadas a las clases. Las propiedades pueden ser especificadas como una simple propiedad/formula o como la especificación de propiedad objeto. Las propiedades pueden ser los nombres de nuevas propiedades a ser adicionadas a las clases o nombres de propiedades que están en las superclases. Si los nombres son los mismos que aquellos en las superclases, son considerados propiedades nulificadotas. Las especificaciones de propiedades son usadas como los valores predeterminados durante la creación de instancias. La formula de una propiedad puede ser un llamado a función, cualquier cálculo o un valor simple que es evaluada.

2.5.4.3 Especificando Subobjetos.

La sección opcional *:subobjetos* del constructor *Define-class* puede ser una lista de especificaciones de subobjetos. Los subobjetos son creados como instancias que son hijos de la clase a ser definida. La forma de los subobjetos requiere del siguiente formato:

(nombre-instanciado-de- subobjecto: class clase-a-ser-instanciada especificación-de-propiedades)

El nombre-instanciado-de- subobjecto es cualquier símbolo que pueda convertirse en el nombre de la instancia que es creada desde la especificación. La clase-a-ser-instanciada puede ser cualquier expresión que evalúa el nombre de la clase. La especificación de propiedades en el nivel de los subobjetos son exactamente las mismas como en el nivel de *:properties* para *Define-class*.

LIST [función]

Una lista es una colección de elementos. Ejemplos son (1 2 3), (a b c), (“Bob” “Jim” “Steve”).

Formato:

(list arg1 arg2 … argn)
ó
‘(arg1 arg2 … argn)

2.5.4.4 Creando una lista:

Ambos ‘(1 2 3) y (list 1 2 3) crean una lista que contiene elementos enteros de 1, 2 y 3. Sin embargo, las dos maneras de especificar la lista no son idénticas. Usando (list …) provoca que cada elemento de la lista sea

evaluado. Cuando se usa ‘(…), AML no evalúa los elementos dentro del paréntesis. El ejemplo (list 1 2 3) regresa el mismo resultado que ‘(1 2 3) porque cada uno de los elementos se evalúan a ellos mismos. El ejemplo (list (* 1 2) (+ 3 4)) regresa (2 -1 7) porque AML evalúa cada elemento dentro de la lista. Algunas funciones para ext racción de listas son: **first, rest, last, nth.**

(first list)
(rest list)
(last list)
(nth index list)

Nota: nth regresa el elemento de la lista en la posición especificada por *index*. El índice del primer elemento es 0 (cero). Si el índice va más allá del largo de la lista, el resultado *nil* es regresado. La función *first* regresa el primer elemento de la lista, es el equivalente de *nth* con un índice 0 (cero). La función *rest* y *last* ambas regresan listas. *Rest* regresa todo excepto el primer elemento de la lis ta. *Last* regresa una lista conteniendo sólo el ultimo elemento de la lista.

COORDINATE-SYSTEM-CLASS [Clase].

La clase *coordinate-system-class* provee un marco de referencia ortogonal que puede ser usado para posicionar otros objetos en un modelo. Añade un indicador visible de la posición y orientación del marco coordenado y es usado como un marco de referencia para otros objetos. Un objeto *coordinate-system-class* es dibujado como un juego de ejes en las direcciones locales x, y y z; una caja desplegando x, y y z; y el nombre del objeto *coordinate-system-class*. Éstos componentes pueden ser desactivados individualmente.

Propiedades:

origen	La propiedad origen especifica la posición del origen del objeto *coordinate-system-class* con respecto con su sistema coordenado de referencia.
Vector-I	La dirección del eje I del objeto *coordinate-system-class* con respecto con su sistema coordenado de referencia. Valores predeterminados ‘(1 0 0).
Vector-J	La dirección del eje J del objeto *coordinate-system-class* con respecto con su sistema coordenado de referencia. Valores predeterminados ‘(0 1 0).

Ejemplo:

```
(define-class EJEMPLO-DE-CLASE-DE-SISTEMA-COORDENADO
        :inherit-from (coordinate-system-class)
        :properties (
                origin ‘(0 0 0)
                vector-i ‘(1 0 1)
                vector-j ‘(1 -1 0)
        )
)
```

CREATE-MODEL [función].

Create-model permite al usuario crear un nuevo modelo (instanciar una clase) basada en la clase dado su nombre. Esto crea un modelo (una instancia, también conocida como “objeto”) de una clase en particular. Esta instancia se convierte en el modelo actual, también conocido como la “raíz” del árbol. En AML, todos los modelos son ubicados en una jerarquía tal que todos los modelos definidos por el usuario son hijos del *manejador de modelos*. El *manejador de modelos* tiene un subobjeto predefinido llamado *interfaz* que es el objeto donde todos los objetos de interfaz de usuario de AML son almacenados. El *model-manager* es la raíz

absoluta de todos los objetos de AML. Para seleccionar los varios modelos definidos por el usuarios usar la función *select-model*.

Format:

(CREATE-MODEL nombre)

Argumentos:

nombre	El nombre para el modelo a ser creado en forma simbólica; el nombre debe ser un nombre valido para una clase.

Ejemplo:

```
(define-class CLASE-MATERIAL
        :inherit-from (object)
:properties (
        material  'madera
        densidad  0.0
        )
)
```

AML> (create-model 'clase-material)
#<clase-material @ #x2224280a>

2.5.4.5 Expresiones

La aritmética en AML usa notación polaca en forma de lista. Eso significa que el operador aparece primero seguido por sus argumentos. Esto es porque toda la aritmética en AML usa funciones. La evaluación de los argumentos ocurre antes de la operación aritmética. Esto permite incrustación de cálculos dentro de otros cálculos.

(+ (-1 2) (* 3 4)) ? (+ -1 (* 3 4)) ? (+ -1 12) ? 11

La total sintaxis es una expresión que regresa el dato valor de 11, y los argumentos (- 1 2) y (*3 4) son expresiones que regresan -1 y 12, respectivamente. Los argumentos -1 y 12 son también considerados expresiones que se regresan a si mismos.

GRAPHIC-OBJECT [Clase].

Todos los objetos que tienen geometría y gráficos asociados con ellos deben heredar de *graphic-object*. Toda la geometría es creada con el centro en el origen global '(0 0 0), que es valor predeterminado.

Propiedades:

color	El valor puede ser un símbolo o una cadena de caracteres. Valor predeterminado "white"
display?	Cuando es verdadero (t), una instancia de esta clase será capaz de ser dibujada. Cuando es falso (nil), la instancia no puede ser desplegada. Valor predeterminado t
render	*'boundary* para gráficos de alambre, *'shaded* para representaciones con color sólido y *'facet* para conectar las facetas de las superficies con líneas.
line-width	El ancho de las líneas usadas para dibujar los objetos.
line-type	El estilo de las líneas usadas para dibujar los objetos.

POSITION-OBJECT [Clase].

La clase *position-object* provee la habilidad para los objetos de ser orientados en el espacio. Todos los objetos que aparecen desplegados en el espacio heredan de la clase *graphics-object* que hereda de *position-object*.

Propiedades:

orientation La lista de comandos de orientación usada para posicionar el objeto

SOLID-OBJECT [Clase]

Solid-object provee a tres objetos primitivos dimensionales la propiedad de determinar si la geometría es sólida o una coraza hueca.

Propiedades:

solid? Cuando es verdadero, la geometría será un sólido. Cuando sea falso, la geometría será creada como una coraza hueca. Cambiar la propiedad causara que se actualice la geometría. Valor predeterminado t.

SPHERE-OBJECT [Clase]

Sphere-object puede ser referido como una coraza geométrica o un sólido. Esta clase creara una esfera.

Propiedades:

diameter El diámetro de la esfera. Valor predeterminado 1.0

Ejemplo:

```
(define-class CLASE-PELOTA-PLAYA
:inherit-from (sphere-object)
:properties (
        diameter 2.5
        color 'blue
        render 'shaded
        solid? Nil
        )
)
```

OPEN-CYLINDER-OBJECT [Clase].

Open-cylinder-object crea un cilindro abierto hueco.

Propiedades:

diameter El diámetro del cilindro. Valor predeterminado 1.0
height La altura es definida paralela al eje z. Valor predeterminado 2.0

Ejemplo:

```
(define-class CLASE-COLUMNA-ABIERTA
:inherit-from (open-cylinder-object)
:properties (
        diameter  40.0
        height  150.0
        color  'white
        )
)
```

OPEN-TRUNCATED-CONE-OBJECT [Clase].

Open-truncated-cone-object es definido como un cono abierto truncado hueco.

Propiedades:

start-diameter	El diámetro de la cara inicial del cono truncado. Valor predeterminado 1.0
end-diameter	El diámetro de la cara final del cono truncado. Valor predeterminado 1.0
height	La altura del cono truncado. Valor predeterminado 2.0

2.5.4.6 Funciones de orientación.

Todos los objetos que heredan de *position-object* pueden ser orientados en el espacio del modelo. La orientación de un objeto puede consistir de cualquier combinación de translaciones y rotaciones. Esta orientación puede ser "construida para" el objeto a través de su definición de clase, o aplicada al objeto antes de la instanciación (creación) a través de la forma de orientación en la interfaz de usuario gráfica de AML. El formato de la propiedad de orientación es el siguiente:

```
orientation     (list    (operación-1  args)
                         (operación-2  args)
                         …
                )
```

La formula de orientación no puede ser definida como una lista referida (') porque no es evaluada en la misma manera que otras propiedades.

Considere el siguiente ejemplo de un objeto cuadrado *box-object* trasladado y rotado de su posición original:

```
(define-class ejemplo-objeto-cuadrado
:inherit-from (box-object)
:properties (
        orientation     (list    (translate  '(5 0 0))
                                 (rotate  45  '(0 0 1))
                        )
        )
)
```

La propiedad de orientación especifica que la caja debe ser primero trasladada una distancia de 5 unidades a lo largo del eje x (del marco del sistema de coordenadas global). Después de eso, es rotado 45 grados en el eje z, o el vector (0 0 1) del sistema de coordenadas global. Las operaciones de orientación son construidas en el objeto. Cualquier instancia de este objeto será inmediatamente transformada a una nueva orientación en creación. Hay que mencionar que el orden de las operaciones de orientación es importante. Invertir el orden de las operaciones resultara en una orientación final diferente. Las funciones de orientación pueden contener expresiones que serán evaluadas cuando el objeto es creado. Por lo que, para mover el *box-object* a lo largo

del eje x una distancia igual al doble de su largo y luego rotarlo, la siguiente orientación puede ser usada usando la referencia *the.*

```
orientation (list   (translate   (list   (* 2 ^width)   0  0)
                                 (rotate  45  '(0 0 1))
                    )
```

CREATE-MODEL [función].

Como se demostró anteriormente, *create-model* permite al usuario crear un nuevo modelo basado en la clase especificada.

Formato:

(CREATE-MODEL nombre [:class nombre-clase])

Argumentos:

nombre	Un símbolo que describe el nombre definido por el usuario para el modelo a ser creado. Si el argumento de clase no es proporcionado, el nombre debe ser un nombre de clase valido.
:class	Un símbolo que describe el nombre de la clase. El valor predeterminado para clase es el nombre. Un nombre de clase valido es requerido

Ejemplo:

AML> (create-model 'modelo-1 :class 'object)
#<OBJECT @ #x2224280a>

2.5.4.7 Referencia THE

The es un constructor en AML que interroga a un objeto por sus propiedades y subobjetos. Un desarrollador puede usar la referencia *the* para obtener propiedades o objetos de otros lugares de la jerarquía de la instancia y describir donde están localizados en la jerarquía de la instancia. Por ejemplo, considere la instancia jerárquica de un aeroplano mostrada en un diagrama de instancias, donde las mayúsculas representan objetos y minúsculas representan propiedades de esos objetos.

```
AEROPLANO                                   [nivel 1]
    Máxima-velocidad                        [nivel 2]
    Largo-alas                              [nivel 2]
    Número-de-motores                       [nivel 2]
    ALAS                                    [nivel 2]
        ALAS-0001                           [nivel 3]
            Largo                           [nivel 4]
        ALAS-002                            [nivel 3]
            Largo                           [nivel 4]
            COSTILLAS                       [nivel 4]
                COSTILLAS-0001              [nivel 5]
                    Largo                   [nivel 6]
                    Ancho                   [nivel 6]
    FUSELAJE                                [nivel 2]
        Largo                               [nivel 3]
        Radio                               [nivel 3]
```

```
SECCION-COLA                    [nivel 2]
        ELEVADOR                [nivel 3]
                Angulo de deflexión     [nivel 4]
```

Asumiendo esta jerarquía, el desarrollador puede navegar a través de las instancias y propiedades a través de la referencia *the*. Conceptualmente, la referencia *the* tiene un punto inicial y un punto final. El punto final es el nombre del objeto o propiedad objetivo. El punto de inicio varia dependiendo de donde la referencia *the* sea codificada. Cuando se escribe código en archivos, la referencia *the* empieza desde el nivel actual en el árbol.

SUPERIOR

La propiedad *superior* hace a la referencia *the* más eficiente y ayuda a producir código más legible. Esencialmente, cada *superior* toma a la referencia *the* arriba un nivel. Note que esto sigue siendo la referencia *the* y continuara buscando arriba del árbol si no encuentra el objetivo un nivel arriba de si mismo.

Ejemplo:

Asuma las siguientes definiciones de clases:

```
(define-class EJEMPLO-SUBOBJETO-CLASE
        :inherit-from (object)
        :properties (
                id "OU812"
                número-parte 2.0
                código-distrito nil
        )
)

(define-class EJEMPLO-MODELO-CLASE
:inherit-from (object)
:properties (
        nombre-modelo "Modelo para AML"
        código 45242
        )
:subobjects (
        (sub-1 :class 'ejemplo-subobjeto-clase
                id (the superior superior sub-2 id)
                número-parte 3.0
                código-distrito (the superior superior código)
        )
        (sub-2 :class 'ejemplo-subobjeto-clase
                id "ZZ184
                código-distrito (the superior superior código)
        )
)
)
```

El código de distrito de *sub-1* y *sub-2* obtienen sus valores de su padre de padres; *ejemplo-modelo-clase*. Esto solamente ilustra la automatización del flujo de datos en MAL con el uso de *superior superior*. El *id* de *sub-2* obtiene su valor de la redefinición de la instanciación como subobjeto de *ejemplo-modelo-clase*. El *id* de *sub-1* obtiene su valor al emplear la referencia *the* hacia el *id* de *sub-2*. Existe una abreviación para superior en AML especificada con una ^ (que se le como "the superior). Esto se define como:

^xyz = (the superior xyz)

^^xyz = (the superior superior xyz) y así sucesivamente…

POSITION-OBJECT [Clase].

Position-object provee la habilidad para los objetos de ser orientados en el espacio. Todos los objetos que aparecen en el despliegue gráfico heredan de la clase *graphic-object* que hereda de *position-object*.

Propiedades:

orientation	La lista de comandos de orientación usada para posicionar un objeto.
Reference-coordinate-system	Especifica el objeto *coordinate-system-class* que será el marco de referencia para este objeto. La propiedad *referente-coordinate-system* debe apuntar a un objeto de clase *coordinate-system-class*. Un valor de nil indica que el marco global de referencia debe ser usado. Si *referente-coordinate-system* se refiere a un objeto que no hereda de *coordinate-system-class*, el marco global será usado. Por valor predeterminado, el marco de referencia es global. (Valor predeterminado nil)

Ejemplo:

```
(define-class clase-ejemplo-sistema-de-referencia-coordenado
        :inherit-from (object)
        :properties (
                )
        :subobjects (
                        (sistema-coordenadas-absoluto :class 'coordinate-system-class
                                )
                        (sistema-de coordenadas -caja :class 'coordinate-system-class
                                origin (list 2 0 0)
                                vector-i (list 1 1 0)
                                vector-j (list -1 1 0)
                                )
                        (caja-sin-orientación :class box-object
                                reference-coordinate-system ^^ sistema -de coordenadas-caja
                                )
                        (caja-con-orientación :class box-object
                                reference-coordinate-system ^^ sistema -de coordenadas-caja
                                orientation (list
                                                (translate (list 1 2 0))
                                                )
                                )
                )
)
```

NAME-GENERATOR [Class].

Esta clase provee la habilidad de generar nombres únicos al pegar número al final de los nombres. Esto puede ser usado cuando el usuario agrega varios objetos al modelo sin específicamente nombrar cada objeto manualmente (o escribir un algoritmo para hacer esto). Especialmente para depurar, un usuario puede crear un modelo de tipo *name-generator* y dinámicamente añadir objetos a este modelo sin tener que especificar nombres para todos los objetos. La clase *name-generator* tiene métodos que automatizan el rendimiento y la administración de nombres como *generate-name*.

Propiedades:

Previous-name-list	Esta propiedad es usada para almacenar los nombres y números actuales de nombres que han sido generador. Valor predeterminado nil.
Auto-naming?	Un valor de t causara que el método de generación de nombres automáticamente genere el siguiente nombre para el prefijo especificado. Un valor de nil causara que el método generador de nombres solicite al usuario que digite un nombre. Valor predeterminado t.

ADD-OBJECT [método].

Add-object instancia una clase, adiciona el objeto a la instancia de objetos especificada con el nombre dado por el argumento *name*.

Format:

(ADD-OBJECT padre nombre clase)

Argumentos:

padre	La instancia de objeto donde el nuevo objeto será agregado.
nombre	El nombre del objeto de la instancia de objeto a ser instanciado.
clase	La clase de la instancia de objeto a ser agregado.

Ejemplos:

AML> (create-model ‘generador-nombre)
#<GENERADOR-NOMBRE @ #x117792a>
AML>(add-object (the) ‘esfera-001 ‘sphere-object)
#<SPHERE-OBJECT @ #x522567a>
AML>(add-object (the) ‘cilindro-001 ‘open-cylinder-object)
#<OPEN-CYLINDER -OBJECT @ #x392887a>
AML> **(create-model ‘generador-nombre)**
#<GENERADOR-NOMBRE @ #x182579f>

AML>(add-object (the) ‘esfera-0064 ‘sphere-object)
#<SPHERE-OBJECT @ #x21f3f7a2>
AML>(add-object (the) ‘cilindro-0185 ‘open-cylinder-object)
#<OPEN-CYLINDER-OBJECT @ #x21f5d5b2>
Éstos comandos crearan la siguiente estructura de árbol:

MODEL-MANAGER
GENERADOR-NOMBRE
ESFERA-0001
CILINDRO-0001
GENERADOR-NOMBRE
ESFERA-0064
CILINDRO-0185

CASE [Función].

La declaración *case* compara un valor de prueba con un número de valores y evalúa la expresión que este incluida con el primer valor que sea igual.

Formato:

(CASE valor-prueba (valor1 expresiones)[(valor2 expresiones)]...[(valorN expresiones)]

IF [función].

La declaración *if* es el condicional más simple. Si la *expresión-prueba* no es nil, la cláusula *expresión-verdadera* es evaluada y si la *expresión-prueba* es nil, ,a cláusula *expresión-falsa* es evaluada. La cláusula *expresión-falsa* es opcional y regresara nil si no es incluida. Porque solamente permite una simple expresión para las cláusulas *expresión-verdadera* y *expresión-falsa,* es a veces necesario usar la declaración *progn,* que trata muchas expresiones como una simple función, para una de estas cláusulas.

Formato:

(IF expresión-prueba expresión-verdadera [expresión-falsa])

Argumentos:

expresión-prueba	Cualquier expresión que pueda ser usada para determinar si evalúa la *expresión-verdadera* o la *expresión-falsa.*
expresión-verdadera	Cualquier expresión que será evaluada cuando *expresión-prueba* no sea nil.
expresión-falsa	Cualquier expresión opcional que será evaluada cuando *expresión-prueba* sea nil.

DEFAULT [función].

Cuando se especifica como la formula de una propiedad, *default* buscara en el árbol por un objeto con la propiedad con el mismo nombre. Si se encuentra una, el valor de esa propiedad es regresado. De lo contrario, el valor predeterminado especificado en la formula es usado.

Formato:

(DEFAULT [formula -default nil])

Argumentos:

formula-default	Si la búsqueda en el árbol por una propiedad con el mismo nombre es exitosa, la formula especificada aquí será usada.

Ejemplo:

```
(define-class CLASE-PRUEBA-DEFAULT
:inherit-from (object)
:properties (
        height 15
        width 10
        depth 6
        material 'acero
:subobjects (
        (caja1 :class 'box-object
```

```
                height (default 13)
                width 9
                density 6
                )
        (caja2 :class 'box-object
                height 11
                depth (default 5)
                material (default 'wood)
                )
        )
)
```

2.5.4.8 Comentarios

Los comentarios en AML están especificados con un punto y coma ";". Cualquier palabra, número, expresión o carácter después del punto y coma serán ignorados por el compilador hasta el inicio de la siguiente línea.

OPEN-CONE-OBJECT [Clase].

Un *open-cone-object* es definido como un cono abierto hueco.

Propiedades:

Diameter	El diámetro del circulo final. Valor predeterminado 0.5
height	La altura es definida paralela al eje z. Valor predeterminado 2.0

DIFFERENCE-OBJECT [Clase].

El *difference-object* toma una lista de objetos en el *object-list*. Empezando con el primer objeto en el *object-list*, remueve las partes de su geometría que se interceptan con los subsecuentes objetos en la lista.

Propiedades:

Object-list	La lista de objetos que van a ser diferenciados. Notar que el orden de los objetos determina la geometría final.

Ejemplo:

```
(define-class CLASE-TUBO
:inherit-from (diferrence-object)
:properties (
        id 0.5
        od 1.0
        height 1.0
        render 'shaded

        object-list (list ^estaca ^hoyo)

        (estaca  :class 'cylinder-object
                height ^^height
                diameter ^^od
                solid? t
                )
        (hoyo  :class 'cylinder-object
```

```
                    height ^^height
                    diameter ^^id
                    solid? t
                    )
            )
    )
```

CYLINDER-OBJECT [Clase].

Un *cylinder-object* puede ser un cilindro sólido o hueco. Es hueco si la propiedad *solid?* evalúa en nil, de otra forma, es un sólido.

Propiedades:

diameter	El diámetro del cilindro. Valor predeterminado 1.0.
height	La altura es definida paralela al eje z. valor predeterminado 2.0.

LET* [Constructor].

La forma *let** es el mecanismo más común para crear variables locales. Let* une los valores a las variables en secuencia, lo que significa que la variable puede usar una variable previamente definida en la misma asignación let*.

Formato:

(LET* (cláusulas-de-asignación) cuerpo)

Argumentos:

cláusulas-de-asignación	Las cláusulas let son una lista de listas de formulas variables o de variables.
cuerpo	Cualquier número de expresiones a ser evaluadas dentro del contexto de las variables asignadas.

Ejemplo:

```
(LET*   ((a  3.0)
         (b  (*  2  a))
         )
(+  1.0  b)
)
```

POLYGON-OBJECT [Clase].

El *polygon-object* es un polígono compuesto de segmentos de líneas. Si los vértices no especifican un polígono cerrado, el sistema automáticamente conecta los puntos iniciales con los puntos finales.

Propiedades:

vertices	La lista de puntos que definen el segmento de la poli-línea-objeto. Valor predeterminado nil.
dimension	La dimensión determina si el objeto es una entidad 1d (una línea) o una entidad 2d (una superficie). El valor predeterminado es 1 para 1d.

Ejemplo:

```
(define-class EJEMPLO-POLIGONO-CLASE
:inherit-from (polygon-object)
:properties (
        vértices  '((0  0  0 )
                    (1    0  0)
                    (1    0  1)
                    (1    1  1))
        dimension 2
                )
)
```

EXTRUSION-OBJECT [Clase].

El *extrusion-object* barre un objeto a lo largo de un vector. La geometría resultante puede ser un sólido o superficie y puede estar tapada o no tapada. Cuando la geometría es sólida debe estar tapada.

Propiedades:

Swept-object	El objeto a ser extruido.
vector	La dirección a extruir.
distance	El largo de la extrusión.

Ejemplo:

```
(define-class EJEMPLO-POLIGONO-CLASE
:inherit-from (polygon-object)
:properties (
        vértices  '((0  0  0 )
                    (1 0  0)
                    (1  1  0)
                    (0  1  0))
        dimension 2
                )
)
(define-class EJEMPLO-EXTRUSION-BARRA-CLASE
:inherit-from (extrusion-object)
:properties (
                swept-object            ^profile
                vector                  '(0 0 1)
                distance        10
                (profile :class 'ejemplo-poligono-clase)
                )
        )
```

CAPPED-SURFACE-OBJECT [Clase].

Un *capped-surface-object* toma una superficie abierta como un objeto fuente, y crea, ya sea, una superficie tapada (solid? = nil) o un sólido delimitado por una superficie tapada (solid?= t).

Propiedades:

Source-object	Una instancia de la superficie abierta a tapar.
Solid?	Valor predeterminado t para crear un sólido tapado. Cuando nil, la superficie es solamente tapada y ninguna geometría sólida es creada.

CIRCULAR-ARRAY-OBJECT [Class].

El arreglo es creado al hacer copias de la geometría del source-object y posicionándolas en un circulo alrededor del centro a un diámetro dado. El circulo es normal al eje de rotación. La primera copia es trasladada a lo largo del eje de translación y luego rotado en el eje de rotación en contra de las manecillas del reloj por el ángulo de inicio. Copias subsecuentes son cada una colocadas a una posición más alejada en dirección contraria de las manecillas del reloj incrementado por un Angulo de repetición. El *circular-array-object* crea un sólo objeto ensamble geométrico.

Propiedades:

Source-object	El objeto a ser copiado.
diameter	El tamaño del circulo del cual las copias serán colocadas (valor predeterminado 1.0.
Start-angle	El ángulo en el cual se coloca el primer arreglo de elementos (con referencia al eje de translación). Valor predeterminado 0.0.
Repeat- angle	El espacio angular entre dos arreglos de elementos consecutivos.
Translate-axis	El vector de referencia para posicionar la primera copia . Valor predeterminado es el eje x).
Rotate-axis	Define el eje perpendicular al plano del arreglo circular. Valor predeterminado es el eje y.
Rotate-clones?	Valor predeterminado t, decide si o no las copias serán rotadas individualmente como son colocadas en el patrón circular.
center	El centro del patrón circular, su valor predeterminado es el centro del objeto fuente
quantity	El número de copias a crear. Valor predeterminado t.
Assembly?	T para crear un ensamble y nil para crear una unión. Valor predeterminado t.
Ref-point	El punto local del objeto fuente (y cada copia) que resta en el circulo descrito. Tiene valor predeterminado en el centro del objeto fuente.

2.5.4.9 Herencia Múltiple.

La herencia es un mecanismo para el reuso de clases. A través de la herencia, una clase tendrá todas las mismas propiedades, subobjetos y métodos de las clases que hereda de (sus superclases). La palabra *inherit-from* acepta una lista de clases a ser usada como superclases. . si una propiedad, subobjeto o método esta presente en más de una de sus superclases, el orden de precedencia es de izquierda a derecha. Esto significa que si *:inherit-from* contiene la lista *(box-object cylinder-object)*, la geometría de *box-object* será creada. Si el orden de la lista es revertido *(cylinder-object box-object)*, la geometría de *cylinder-object* será creada.

CHILDREN [Método].

Regresa el subobjeto inmediato de una instancia.

Format:

(CHILDREN instancia [:class t])

Argumentos:

instancia	El objeto instancia de los cuales subobjetos desea encontrar.
:class	Si una clase es especificada, sólo los hijos que heredan de esa clase serán regresados. Valor predeterminado t para todas las clases.

VOLUME-OF-OBJECT objeto gráfico [Método].

El valor regresado por este método depende de la dimensión del objeto. Si la dimension = 1, entonces el largo es regresado. Si la dimension = 2, entonces el área de superficie es regresa. Si la dimension = 3, entonces el volumen es regresado.

Format:

(VOLUME-OF-OBJECT instancia)

Argumentos:

instancia	Una instancia de un objeto gráfico.

LOOP

La facultad de ciclos puede ser usada virtualmente para todas las iteraciones requeridas. Debido a la versatilidad de la sentencia *loop*, existen muchos parámetros de control que son usados para realizar las tareas requeridas. Éstos parámetros de control son discutidos de acuerdo a su función.

2.5.4.10 Control de iteración.

Los siguientes controles de iteración deben proceder otros argumentos de ciclos excepto para los argumentos *with, initially* y *finally.* Puede haber cualquier número de controles de iteración en una sola declaración de ciclo. Las iteraciones ocurren simultáneamente y el ciclo terminara cuando cualquiera de las iteraciones sea completada.

FOR

El argumento *For* es un control de incremento (decremento) general. Existe un número de parámetros que pueden ser usados para controlar como las iteraciones deben proceder. Los parámetros son usados con los parámetros de los ciclos para el control de completar muchos diferentes tipos de iteraciones.

Ejemplo:

```
AML> (loop for i from 0 to 3
        Do (print i))
0
1
2
3
NIL
```

Acumulación de Valor: El valor acumulado resultante será regresado por la declaración de ciclo. Cuando más de una acumulación es necesaria en una declaración de ciclo, el parámetro *into* es necesario para crear una variable local para retener los resultados de las acumulaciones. Si éstos valores acumulados necesitan ser regresados, usar las declaraciones *final* y *return*.

SUM

El argumento *sum* acumula el total de su parámetro.

Ejemplo:

```
AML> (loop for i from 1 to 10
        Sum i)
55
```

2.5.4.11 Ejecución condicional.

When.

El argumento *when* es usado para realizar algunas operaciones cuando una condición es verdadera.

Ejemplo:

```
AML> (loop for i from 0 to 10 by 3
        When (evenp i)
        Do (print i)
        Collect i)
0
6
(0 3 6 9)
```

2.5.4.12 La referencia THE con la palabra *:from*.

El constructor referencia *the* tiene varias palabras reservadas para aumentar su funcionalidad. Un apalabra de ellas es *:from*, que colócale punto de inicio de la instancia de una referencia *the* a un objeto especifico o propiedad dada después de la palabra *:from*. Este argumento debe evaluarse a un objeto o propiedad. Note que todas las palabras de la referencia *the* son dadas dentro de un grupo de paréntesis para distinguir el final de la ruta transversal de una instancia y el empiezo de palabras reservadas.

Format:

(the nombre1 nombre2 … nombreN (:from (the nombre1 nombre2 … nombreM)))

Ejemplo:

Asuma la siguiente estructura de árbol:

```
AEROPLANO                                   [nivel 1]
    Maxima-velocidad                        [nivel 2]
    Cruce-alas                              [nivel 2]
    Número-de-motores                       [nivel 2]
    ALAS                                    [nivel 2]
        ALAS-0001                           [nivel 3]
            Cruce                           [nivel 4]
        ALAS-0002                           [nivel 3]
            Cruce                           [nivel 4]
            CUERPO                          [nivel 4]
                CUERPO-0001                 [nivel 5]
                    Largo                   [nivel 6]
                    Ancho                   [nivel 6]
```

AML> (the aeroplano alas alas-0001 cruce)
40.1
AML>(the cruce (:from (the aeroplano alas alas-0001)))
40.1

2.5.5 Diseño en la interfaz gráfica para usuarios de AML.

Usando AML, el desarrollador de aplicaciones puede construir interfaz gráficas de usuario usando AML, clases, funciones y herramientas. Construir una interfaz de usuario en AML puede ser hecho en diferentes niveles. Si el desarrollador esta buscando un diseño para máxima flexibilidad y control, el desarrollo de procedimientos en GUI puede ser el nivel básico, en otras palabras, los componentes GUI están construidos uno por uno ligados al modelo de aplicación a través de métodos y funciones definidos por el usuario. Este proceso requiere de mucho tiempo de desarrollo. Estas clases y funciones basadas en GUI están disponibles en *ui-base-classes* en el sistema AML. Por otro lado, el desarrollo en GUI puede ser de un nivel mucho más alto si el diseñador decide usar el juego de clases estándar de AML, que son capaces de automatizar la interfaz de un modelo en AML, proporcionando más del control estándar que el usuario requiere para una aplicación. Esto reduce el tiempo de desarrollo dramáticamente. Estas clases están disponibles en *ui-advanced-classes* en el sistema AML.

Model-interfaz-property-class [Clase].

Esta es la superclase de cada interfaz de modelo. Esta clase no es típicamente instanciada por el usuario.

Hereda de: property-object.

Propiedades:

Available? Una bandera (*t* o *nil*) que especifica si la propiedad esta disponible / usable, usualmente basada en los valores actuales de otras propiedades de *data-model*. Una propiedad no disponible es marcada por GUI.
label Una cadena de caracteres que especifican la etiqueta de la propiedad. La etiqueta es típicamente usada por GUI la formula predeterminada es *(write-to-string (object-name (the superior)).*

Computed-data-property-class [Clase].

Esta clase es usada para propiedades de propósito general de los cuales sus valores no se suponen editables por el usuario. Estas propiedades son típicamente propiedades de salida de datos.

Hereda de: model-interfaz-property-class.

Editable-data-property-class [Clase].

Esta clase es usada para propiedades de propósito general de los cuales sus valores se suponen editables por el usuario. Estas propiedades son típicamente propiedades de entrada de datos.

Hereda de: model-interfaz-property-class.

Flag-property-class [Clase].

Esta clase es usada para propiedades "bandera" que esperan un valor de t o nil. Estas propiedades son típicamente representadas en el GUI con un botón.

Hereda de: model-interfaz-property-class.

Option-property-class [Clase].

Esta clase es usada para propiedades en la cuales el valor es igual a un elemento de una lista de opciones disponibles. Estas propiedades pueden ser representadas con un menú de opciones, un grupo de botones de radio, o una caja combo. Una caja combo es típicamente usada cuando el valor de *option-property-class* esta permitido a ser diferente de cualquiera de las opciones disponibles.

Hereda de: model-interfaz-property-class.

Propiedades:

Labels-list	Lista de cadena de caracteres que especifican la etiqueta para cada opción. *Label-list* es ignorada cuando *mode* es *'combo* porque a el usuario le es permitido editar una opción al hacer opciones y etiquetas una entidad.
mode	Modo GUI especificando la representación GUI de la propiedad. Permite valores como *'radio, 'menu* y *'combo*. Formula predeterminada es *'radio*.
Options-list	Lista de opciones disponibles.

Data-model-node-mixin [Clase].

Esta clase esta diseñada para representar un nodo del modelo de datos. Contiene conocimiento especifico de la aplicación acerca del nodo que representa. También administra las propiedades de ese nodo que son parte del modelo de datos por lo que debe ser parte de la aplicación GUI. *Data-model-node-mixin* es típicamente heredado hacia clases definidas por el usuario. Formas de aplicación estándar hacen uso de los nodos del modelo datos para la generación automática de modelos GUI. las formas de aplicación que contienen modelos de datos en su nombre de clase son típicamente diseñadas para hacer interfaz con *Data-model-node-mixin*.

Hereda de: Object.

Propiedades:

Available?	Formula predeterminada es t. Cuando es nil, no permite acceso GUI a la *property-object-list* de un nodo desde el árbol de modelo de datos.
Label:	La etiqueta del nodo es desplegada en el árbol de modelo de datos GUI
Property-object-list	Lista de instancias de *model-interfaz-property* que definen las propiedades *data-model* de este nodo. Propiedades incluidas en esta lista estarán disponibles en las formas generadas automáticamente que usen *Data-model-node-mixin*. La lista sigue el mismo formato que *property-object-list* de *ui-multiple-property-subform-class.*

Índice de Figuras

Indice de Tablas.

Glosario de Terminos.

- **AML:** "Lenguaje de Modelado Adaptativo" (Adaptative Modeling Language).
- **CAD:** "Diseño Asistido por Computadora" (Computer-Aided Design).
- **CAM:** "Manufactura Asistida por Computadora" (Computer-Aided Manufacturing).
- **CAE:** "Ingeniería Asistida por Computadora" (Computer-Aided Engineering).
- **FEA:** "Analisis de Elemento Finito" (Finite Element Analisis).
- **OO:** "Orientado a Objetos" (Object Oriented).
- **OODB:**"Base de Datos Orientada a Objetos" (Object Oriented Data Base).
- **SADET:** Sistema Auxiliar para el Diseño de Ejes de Transmisión.
- **UML:** "Lenguaje de Modelado Unificado" (Unified Modelling Language).

Bibliografía

"Adaptative Modeling Language, Basic Training Manual Versión 2.03"
Technosoft Inc.
4434 Carver Woods Dr. Cincinnati, OH 45242
2000
http://www.technosoft.com

"The Adaptative Modeling Language. A Technical Perspective".
Technosoft Inc.
4434 Carver Woods Dr. Cincinnati, OH 45242
2000
http://www.technosoft.com

"AML vs CAD/CAE/CAM systems"
Technosoft Inc.
4434 Carver Woods Dr. Cincinnati, OH 45242
2003
http://www.technosoft.com

A. Ayala, V. Borja, S. Martínez, L. Villarruel, G. Avila
"Sistema para la verificación de factibilidad de manufactura de productos".
Laboratorio de Ingeniería Mecánica Asistida por Computadora,
Circuito Exterior, Talleres del Anexo
Facultad de Ingeniería, Cd. Universitaria, D.F., México 04510
alvaroar@dimei.fi-b.unam.mx

Luis Joyanes Aguilar.
"Programación orientada a objetos"
Segunda Edición.
Osborne McGraw-Hill, 1998.

Chris McMahon and Jimmie Browne.
"CAD / CAM Principles, Practice and Manufacturing Management".
Second Printing.
Addison-Wesley. 1998.

Roger S. Pressman.
"Ingeniería del Software. Un enfoque práctico".
Quinta Edición.
Adaptación de Darle Ince.
McGraw-Hill. 2002.

James Rumbaugh, Ivar Jacobson, Grady Booch
"The Unified Modeling Language, Reference Manual"
First Printing, December 1998.
Addison Wesley Longman, Inc.

Ibrahim Zeid.
"CAD / CAM Theory and Practice".
First Printing.
McGraw-Hill. 1991.

Referencias

Technosoft Inc.
"AML vs CAD/CAE/CAM systems"
4434 Carver Woods Dr. Cincinnati, OH 45242
2003
http://www.technosoft.com

A. Ayala, V. Borja, S. Martínez, L. Villarruel, G. Avila
"Sistema para la verificación de factibilidad de manufactura de productos".
Laboratorio de Ingeniería Mecánica Asistida por Computadora,
Circuito Exterior, Talleres del Anexo
Facultad de Ingeniería, Cd. Universitaria, D.F., México 04510
alvaroar@dimei.fi-b.unam.mx

Chris McMahon and Jimmie Browne.
"CAD / CAM Principles, Practice and Manufacturing Management".
Second Printing.
Addison-Wesley. 1998.

Roger S. Pressman.
"Ingeniería del Software. Un enfoque práctico".
Quinta Edición.
Adaptación de Darle Ince.
McGraw-Hill. 2002.