DESARROLLO DE UNA APLICACIÓN DE SOFTWARE EN EL LENGUAJE JAVA PARA EL CÁLCULO DE ENGRANES RECTOS Y HELICOIDALES PARALELOS

CRISTIAM GILBERTO COMETTA CONDE

UNIVERSIDAD DE SAN BUENAVENTURA
FACULTAD DE INGENIERÍA
PROGRAMA DE INGENIERIA MECATRÓNICA
BOGOTÁ D.C.
2010

DESARROLLO DE UNA APLICACIÓN DE SOFTWARE EN EL LENGUAJE JAVA PARA EL CÁLCULO DE ENGRANES RECTOS Y HELICOIDALES PARALELOS

CRISTIAM GILBERTO COMETTA CONDE

Proyecto de Grado como requisito para optar al título de Ingeniero Mecatrónico.

ING. BALDOMERO MENDEZ
Director

C.S.P. PATRICIA CARREÑO MORENO
Asesora Metodológica

UNIVERSIDAD DE SAN BUENAVENTURA
FACULTAD DE INGENIERÍA
PROGRAMA DE INGENIERIA MECATRÓNICA
BOGOTÁ D.C.
2010

Nota de aceptación

____

____

____

____

____

Firma del Presidente del Jurado

____

Firma del Jurado

____

Firma del Jurado

____

Firma asesor Metodológico

Bogotá, 10 de mayo de 2010

Este trabajo esta dedicado a todas las personas que me apoyaron para la realización de esta propuesta, y en el transcurso de mi carrera profesional.

A mi madre Silvia Cometta, mis hermanos Silvia Catalina y Carlos Gilberto quienes me apoyaron en todo momento, especialmente a mi madre quien a pesar de todos los obstáculos que hemos tenido, me apoyó de todas las formas posibles para ser la persona que ahora soy.

A mi abuelo que en paz descanse Gilberto Cometta, quien fue una gran persona, a quien en gran medida le debo el honor de poderme llamar “Ingeniero Mecatrónico”.

AGRADECIMIENTOS

Expreso mi más sentido agradecimiento a Dios, por haberme dado esta gran oportunidad tanto profesional como personal.

Quiero agradecer de manera especial al Instituto de Metrología Alemán "Physikalisch-Technische Bundesantalt" y a mi jefe el ingeniero Frank Härtig, quien me dio su apoyo para la presentación de esta propuesta como mi proyecto de grado para alcanzar la meta de graduarme como Ingeniero Mecatrónico, además de ofrecerme una oportunidad laboral para comenzar con pie derecho mi experiencia profesional.

Al ingeniero Baldomero Méndez y al ingeniero Gabriel Jaime Cardona por su apoyo y su gran motivación.

A la profesora Patricia Carreño por su apoyo y colaboración como asesora metodológica.

A todos los profesores de la Universidad de San Buenaventura que me guiaron por el transcurso de mi carrera, gracias por su dedicación, su orientación y su nivel de exigencia, a los que les puedo decir de manera muy cordial y amigable "por favor sigan molestando con trabajos complicados".

CONTENIDO

Pág.

INTRODUCCIÓN 12

1. PLANTEAMIENTO DEL PROBLEMA 13

1.1 ANTECEDENTES 13

1.2 DESCRIPCIÓN Y FORMULACIÓN DEL PROBLEMA 13

1.3 JUSTIFICACIÓN 14

1.4 OBJETIVOS 15

1.4.1 Objetivo General 15

1.4.2 Objetivos Específicos 15

1.5 ALCANCES Y LIMITACIONES 16

1.5.1. Alcances 16

1.5.2 Limitaciones 16

2. MARCO DE REFERENCIA 17

2.1 MARCO TEÓRICO - CONCEPTUAL 17

2.1.1 Engranes Rectos 17

2.1.2 Engranes Helicoidales Paralelos 18

2.1.3 Introducción a la Programación Orientada a Objetos 21

2.1.4 Fundamentos de Java 26

2.1.5 Clases y Objetos 33

2.1.6 Herencia 41

2.1.7 Operaciones Avanzadas en las Clases 44

2.2 MARCO NORMATIVO 52

3. METODOLOGÍA 53

3.1 ENFOQUE DE LA INVESTIGACIÓN 53

3.2 LÍNEA DE CAMPO DE INVESTIGACIÓN DE USB /
SUB – LÍNEA DE FACULTAD / CAMPO TEMÁTICO DEL PROGRAMA 53

3.3 TÉCNICAS DE RECOLECCIÓN DE INFORMACIÓN 54

4. DESARROLLO INGENIERIL 55

4.1 DESARROLLO DE LAS DOS PRIMERAS SÚPER CLASES 55

4.2 DEFINICIÓN DE LOS NOMBRES DE LAS VARIABLES A USAR 55

4.3 DEFINICIÓN DE LOS CONSTRUCTORES Y MÉTODOS
QUE SE USARAN EN LAS SÚPER CLASES 59

4.4 DESARROLLO DE LAS TRES PRIMERAS CLASES
DONDE ESTÁN DEFINIDAS INDIVIDUALMENTE m_n, m_t Y β 62

4.5 DESARROLLO DE LA CLASE PARA PROBAR EL
FUNCIONAMIENTO DEL SOFTWARE 64

4.6 DESCRIPCIÓN DEL FUNCIONAMIENTO HASTA EL MOMENTO 68

4.7 DEFINICIÓN DE LOS PARÁMETROS QUE SE
DESEAN CALCULAR CON SUS RESPECTIVAS ECUACIONES 69

4.8 IMPLEMENTACIÓN DE LAS CLASES
DONDE SE DEFINIRÁN INDIVIDUALMENTE d, z, γ Y m_x 77

4.9 CONTENIDO FINAL DE TODAS LAS CLASES 78

4.10 COMPORTAMIENTO DEL SOFTWARE 80

5. PRESENTACIÓN DE RESULTADOS 83

6. CONCLUSIONES 91

BIBLIOGRAFÍA 92

ANEXOS 93

LISTA DE TABLAS

Pág.

Tabla 1. Palabras reservadas de Java 28

Tabla 2. Operadores de Java 29

Tabla 3. Formatos de comentario de Java 30

Tabla 4. Estado de las banderas una vez inicializado el software 66

Tabla 5. Estado de las banderas al momento de calcular "a" 67

Tabla 6. Ecuaciones para calcular el Módulo normal, m_n. 70

Tabla 7. Ecuaciones para calcular el Módulo transversal, m_t. 71

Tabla 8. Ecuaciones para calcular el Ángulo de hélice, β (Beta). 72

Tabla 9. Ecuaciones para calcular el Diámetro de referencia, d. 73

Tabla 10. Ecuaciones para calcular el Números de dientes, z. 73

Tabla 11. Ecuaciones para calcular el Ángulo de avance del cilindro de referencia, γ. 74

Tabla 12. Ecuaciones para calcular Módulo axial, m_x. 74

Tabla 13. Ecuaciones para calcular el Módulo básico, m_b. 75

Tabla 14. Ecuaciones para calcular el Ángulo de presión normal, α_n. 75

Tabla 15. Ecuaciones para calcular el Ángulo de presión, α_t. 76

Tabla 16. Ecuaciones para calcular el Diámetro base, d_b. 77

LISTA DE GRÁFICOS

Pág.

Gráfico 1. Ejemplo de árbol de herencia 24

Gráfico 2. Ejemplo de un árbol de herencia 41

Gráfico 3. Comportamiento del software 65

Gráfico 4. Diagrama de flujo del funcionamiento del software 80

Gráfico 5. Diagrama de flujo del ingreso de valores. 81

Gráfico 6. Diagrama de flujo del cálculo de valores. 82

GLOSARIO

- CONST.NONSD: bandera establecida en una clase predeterminada ("Const") por el Instituto de Metrología Alemán cuyo valor numérico es igual a 9999999999.9, el nombre de esta bandera significa que almacena una constante sin sentido de tipo doble, que es usada comúnmente para inicializar las variables usadas.

- CONST.NONSI: bandera establecida en una clase predeterminada ("Const") por el Instituto de Metrología Alemán cuyo valor numérico es igual a 999999999, el nombre de esta bandera significa que almacena una constante sin sentido de tipo entero, que es usada comúnmente para inicializar las variables usadas.

- DOUBLE.MAX_VALUE: bandera establecida en una clase predeterminada en java cuyo valor numérico es igual a 1.7976931348623157E308.

INTRODUCCIÓN

Como es bien sabido por la mayoría del mundo, Alemania es uno de los países líderes en tecnología y su industria es una de las más fuertes a nivel mundial; compañías multinacionales como Bosch, BMW, Mercedez Benz, Porsche, Siemens, Henkel, etc., han demostrado que a pesar de los fuertes golpes que sufrió el país germano a causa de las dos guerras mundiales, no han sido un impedimento para su crecimiento científico y tecnológico.

Para el sostenimiento del vanguardismo tecnológico alemán, es necesaria además de una gran inversión en la educación, también una gran inversión en sistemas de reconocimiento de calidad de alta precisión, por esto el Instituto de Metrología Alemán "Physikalisch-Technische Bundesantalt" encargado de expedir certificados de calidad, está compuesto de maquinarias y técnicas de medición de última tecnología; teniendo en cuenta que una de las fortalezas de la industria alemana es la construcción de maquinaria, y que una de las principales partes de cualquier máquina son los engranajes que el Instituto de Metrología Alemán ha desarrollado durante años anteriores, software para el cálculo de las variables que comprenden estos elementos, para ponerlos a disposición del público en general.

Aprovechando que en el mes de junio del presente año se realizará un encuentro de empresas en dicho instituto, el Instituto de Metrología Alemán ha tomado la decisión de realizar otra versión del "Involute Calculator" (nombre que se le ha designado al software para el cálculo de variables de engranajes), para ser mostrado a las compañías que asistirán a dicho evento.

1. PLANTEAMIENTO DEL PROBLEMA

1.1 ANTECEDENTES

En el año 2006 fue desarrollado por el joven investigador polaco Robert Kupiec un software en el lenguaje JAVA, el cual consistía en una interfaz gráfica en la que se podía ingresar valores de variables de los engranes, calculando los valores restantes en el caso de que algún valor no hubiera sido escrito, o de haber sido ingresado todos los datos, se mostraría la diferencia que hay entre el dato ingresado y el dato calculado.

1.2 DESCRIPCIÓN Y FORMULACIÓN DEL PROBLEMA

Durante años anteriores han sido desarrolladas varias aplicaciones de software para el cálculo de parámetros de engranajes por parte del Instituto de Metrología Alemán, la última de estas aplicaciones de software fue “Involute Calculator” la cual fue desarrollada en el año 2006, por lo que es una aplicación obsoleta para ser presentada por el Instituto de Metrología Alemán en el encuentro de compañías del presente año, ante esto, dicho Instituto tomó la decisión de iniciar el desarrollo de una nueva versión de dicha aplicación de software, con lo que consecuentemente se planteó el siguiente problema: ¿Cuáles son las características técnicas y funcionales de una aplicación de software para el cálculo de engranes, una aplicación que calcule todas las variables necesarias para el diseño de un engranaje a partir de unas pocas y reconozca múltiples incongruencias que se puedan encontrar en los parámetros de entrada?

1.3 JUSTIFICACIÓN

La aplicación de software que posee el Instituto de Metrología Alemán para calcular parámetros de engranajes fue desarrollado en el año 2006, por lo que es considerado como obsoleto por dicho instituto, por lo tanto es necesario desarrollar una nueva aplicación de software que cumpliera esta función a partir de un nuevo proceso de cálculo, aprovechando la experiencia que posee este instituto.

El Instituto de Metrología Alemán proyecta esta aplicación como una gran herramienta para el cálculo de parámetros de engranajes, la cual servirá para la identificación y corrección de posibles errores o incongruencias en los valores numéricos de los parámetros de los engranajes y será puesta a disposición a la industria alemana.

Este proyecto significara un gran impulso en la carrera profesional del alumno de la Universidad de San Buenaventura que está en el desarrollo de dicho software para darse a conocer, gracias a un encuentro de empresas que se desarrollara en Instituto de Metrología Alemán, donde uno de los eventos será la presentación de este proyecto por parte de este estudiante.

1.4 OBJETIVOS

1.4.1 Objetivo General

Desarrollar una aplicación de software en el lenguaje de programación Java, para el cálculo de parámetros de engranajes rectos y helicoidales paralelos, de acuerdo a los requerimientos del Instituto de Metrología Alemán.

1.4.2 Objetivos Específicos

- Analizar los requerimientos funcionales y de variables exigidos por el Instituto de Metrología Alemán.
- Diseñar la metodología que regirá la aplicación de software.
- Implementar el algoritmo.
- Probar los posibles casos de entrada.
- Analizar los resultados.

1.5 ALCANCES Y LIMITACIONES

1.5.1 Alcances: El proyecto culmina con la presentación de una parte del software en la que se calcularán siete parámetros de los engranajes rectos y helicoidales paralelos según exigencias del Instituto de Metrología Alemán, con sus respectivas simulaciones.

1.5.2 Limitaciones: Este proyecto está limitado por el tiempo dado que la exigencia del Instituto de Metrología Alemán es que este software sea desarrollado en seis meses y la práctica solamente contempla la estadía por tres meses.

2. MARCO DE REFERENCIA

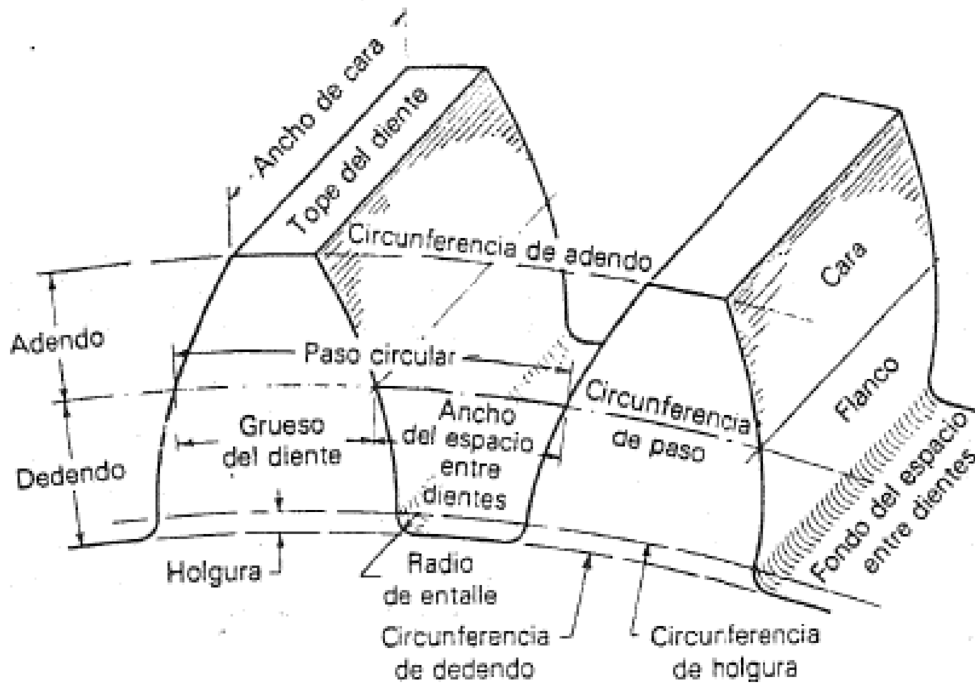
2.1 MARCO TEÓRICO - CONCEPTUAL

2.1.1 Engranes Rectos: En casi todas las máquinas hay transmisión de movimiento de rotación de un eje a otro. Los engranes (o ruedas dentadas) constituyen uno de los mejores medios disponibles para hacerlo.

Cuando se piensa en que los engranes del diferencial de un automóvil, por ejemplo, trabajan en un recorrido de cien mil millas o más, antes de que se necesite reemplazarlos o cuando se considera el total de vueltas o revoluciones que han dado, se aprecia el hecho de que el diseño y la fabricación de estos elementos es algo verdaderamente notable. Por lo general no se advierte que complicados han llegado a ser el diseño, análisis y fabricación de engranes; esto se debe a que son elementos de máquinas de uso muy frecuente y extenso.

2.1.1.1 Nomenclatura: Los engranes rectos se emplean para transmitir movimiento de rotación entre ejes paralelos. Su contorno es de forma cilíndrica circular y sus dientes son paralelos al eje de rotación.

Imagen 1. Nomenclatura de los dientes de los engranajes.

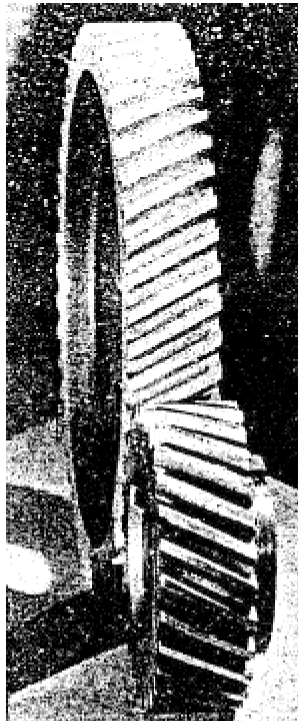


Fuente: SHIGLEY, Joseph Edgard, MITCHELL, Larry D. Diseño en Ingeniería Mecánica. Editorial McGRAWHILL, 1985. Pág. 603.

2.1.1.2 Acción conjugada: Al actuar entre sí para transmitir el movimiento de rotación, los dientes de engranes conectados actúan de modo semejante a las levas. Cuando los perfiles de los dientes se diseñan de modo que produzcan una relación constante de velocidades angulares durante su funcionamiento de contacto, se dice que tienen acción conjugada. En teoría, por lo menos, puede seleccionarse arbitrariamente un perfil para un diente y luego hallar el perfil de dientes en el engrane compañero que producirá acción conjugada. Uno de las soluciones posibles la da el perfil llamado envolvente (o involuta) que, con algunas excepciones, es el de uso universal para dientes de engranes.

2.1.2 Engranes Helicoidales Paralelos: En la siguiente figura se ilustran los engranes helicoidales utilizados para transmitir movimientos entre los ejes paralelos. El ángulo de hélice es el mismo en cada engrane, pero uno debe tener una hélice a la derecha y el otro, una a la izquierda.

Imagen 2. Un par de engranajes helicoidales.



Fuente: Ibid

El contacto inicial de los dientes de engranes rectos es una línea que se extiende a lo largo de toda la cara del diente. El contacto inicial de los dientes de engranes helicoidales es un punto, el cual se convierte en una línea cuando los dientes hacen más contacto. En los engranes la línea de contacto es paralela al eje de rotación; en los helicoidales esta es una diagonal a través de la cara del diente.

Es esta conexión gradual entre los dientes y la transmisión suave de la carga de uno a otro lo que da a los engranes helicoidales la capacidad de transmitir cargas pesadas a altas velocidades. Debido a la naturaleza del contacto entre engranes helicoidales la relación de contacto es de importancia menor y el área de contacto, que es proporcional al ancho de cara del engrane, es la verdaderamente significativa.

Algunas definiciones de los parámetros de los engranajes según ISO 21771:2007:

- Ángulo de hélice, β: es el ángulo entre la tangente a una hélice de referencia y la línea de referencia del cilindro sobre el punto de contacto a través de la tangente.

- Ángulo de presión, α_t : es el ángulo agudo entre la tangente a la involuta en su punto de intersección con el círculo de referencia y el radio a través de este punto de intersección

- Diámetro base, d_b: el círculo de base es la intersección del cilindro de base con un plano de la sección transversal.

- Ángulo de presión normal, α_n: el ángulo de inclinación en el cilindro de referencia es el ángulo de presión normal

- Módulo básico, m_b: el módulo normal de la marcha cilíndrica se encuentra como el módulo de la norma básica perfil de diente de cremallera (series de Módulo ISO 54).

- Ángulo de avance del cilindro de referencia, γ: es el ángulo en el que cruza el plano normal al eje del engranaje. también es el ángulo entre la tangente a una referencia a una hélice de referencia (línea de referencia del flanco) y la sección de referencia a través del punto de contacto tangente.

- Números de dientes, z : Numero de dientes de un engranaje.

- Diámetro de referencia, d : El diámetro de referencia es determinado por:

$$d = |z| m_t = \frac{|z| m_n}{\cos \beta}$$

- Módulo: El Módulo de una cremallera básica se define como la división entre el paso de la cremallera y pi (π).

- Módulo transversal: Para engranajes rectos el Módulo transversal es igual al Módulo normal y para engranajes helicoidales se encuentra de la siguiente manera:

$$m_t = \frac{m_n}{\cos\beta}$$

- Módulo axial: Para engranajes helicoidales se encuentra de la siguiente manera:

$$m_x = \frac{m_n}{sen\beta} = \frac{m_n}{\cos\gamma} = \frac{m_t}{\tan\beta}$$

2.1.3 Introducción A La Programación Orientada A Objetos

2.1.3.1 Programación Orientada A Objetos: La orientación a objetos es un paradigma de programación que facilita la creación de software de calidad por sus factores que potencian el mantenimiento, la extensión y la reutilización del software generado bajo este paradigma.
La programación orientada a objetos trata de amoldarse al modo de pensar del hombre y no al de la máquina. Esto es posible gracias a la forma racional con la que se manejan las abstracciones que representan las entidades del dominio del problema, y a propiedades como la jerarquía o el encapsulamiento.

El elemento básico de este paradigma no es la función (elemento básico de la programación estructurada), sino un ente denominado objeto. Un objeto es la representación de un concepto para un programa, y contiene toda la información necesaria para abstraer dicho concepto: los datos que describen su estado y las operaciones que pueden modificar dicho estado, y determinan las capacidades del objeto.
Java incorpora el uso de la orientación a objetos como uno de los pilares básicos de su lenguaje.

2.1.3.2 Los Objetos: Podemos definir objeto como el "encapsulamiento de un conjunto de operaciones (métodos) que pueden ser invocados externamente, y de un estado que recuerda el efecto de los servicios"[1].

---

[1] PIATTIN, Mario, CALVO-MANZANO, José A., CEVERA, Joaquín y FERNANDEZ, Luis. "Análisis y diseño detallado de Aplicaciones informáticas de gestión". RA-MA.1996.

Un objeto además de un estado interno, presenta una interfaz para poder interactuar con el exterior. Es por esto por lo que se dice que en la programación orientada a objetos "se unen datos y procesos", y no como en su predecesora, la programación estructurada, en la que estaban separados en forma de variables y funciones.
Un objeto consta de:

- Tiempo de vida: La duración de un objeto en un programa siempre está limitada en el tiempo. La mayoría de los objetos sólo existen durante una parte de la ejecución del programa. Los objetos son creados mediante un mecanismo denominado *instanciación*, y cuando dejan de existir se dice que son *destruidos.*

- Estado: Todo objeto posee un estado, definido por sus *atributos*. Con él se definen las propiedades del objeto, y el estado en que se encuentra en un momento determinado de su existencia.

- Comportamiento: Todo objeto ha de presentar una interfaz, definida por sus *métodos*, para que el resto de objetos que componen los programas puedan interactuar con él.

El equivalente de un *objeto* en el paradigma estructurado sería una *variable*. Así mismo la *instanciación de objetos* equivaldría a la *declaración de variables*, y el *tiempo de vida de un objeto* al *ámbito de una variable*.

2.1.3.3 Las Clases: Las clases son abstracciones que representan a un conjunto de objetos con un comportamiento e interfaz común.
Podemos definir una clase como "un conjunto de cosas (físicas o abstractas) que tienen el mismo comportamiento y características... Es la implementación de un tipo de objeto (considerando los objetos como instancias de las clases)"[1].
Una clase no es más que una plantilla para la creación de objetos. Cuando se crea un objeto (*instanciación*) se ha de especificar de qué clase es el objeto instanciado, para que el compilador comprenda las características del objeto.
Las clases presentan el estado de los objetos a los que representan mediante variables denominadas *atributos*. Cuando se instancia un objeto el compilador crea en la memoria dinámica un espacio para tantas variables como atributos tenga la clase a la que pertenece el objeto.

Los *métodos* son las funciones mediante las que las clases representan el comportamiento de los objetos. En dichos métodos se modifican los valores de los atributos del objeto, y representan las capacidades del objeto (en muchos textos se les denomina *servicios*).
Desde el punto de vista de la programación estructurada, una clase se asemejaría a un módulo, los atributos a las variables globales de dicho módulo, y los métodos a las funciones del módulo.

2.1.3.4 Modelo De Objetos: Existen una serie de principios fundamentales para comprender cómo se modeliza la realidad al crear un programa bajo el paradigma de la orientación a objetos. Estos principios son: la abstracción, el encapsulamiento, la modularidad, la jerarquía, el paso de mensajes y el polimorfismo.

- *Principio de Abstracción:* Mediante la abstracción la mente humana modeliza la realidad en forma de objetos. Para ello busca parecidos entre la realidad y la posible implementación de *objetos del programa* que simulen el funcionamiento de los *objetos reales*.
  Los seres humanos no pensamos en las cosas como un conjunto de cosas menores; por ejemplo, no vemos un cuerpo humano como un conjunto de células. Los humanos entendemos la realidad como objetos con comportamientos bien definidos. No necesitamos conocer los detalles de porqué ni cómo funcionan las cosas; simplemente solicitamos determinadas acciones en espera de una respuesta; cuando una persona desea desplazarse, su cuerpo le responde comenzando a caminar.

  Pero la abstracción humana se gestiona de una manera jerárquica, dividiendo sucesivamente sistemas complejos en conjuntos de subsistemas, para así entender más fácilmente la realidad. Esta es la forma de pensar que la orientación a objeto intenta cubrir.

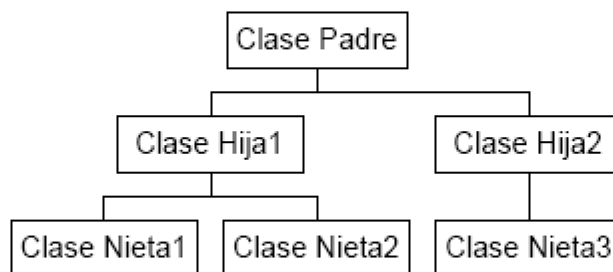
- *Principio de Encapsulamiento:* El encapsulamiento permite a los objetos elegir qué información es publicada y qué información es ocultada al resto de los objetos. Para ello los objetos suelen presentar sus métodos como interfaces públicas y sus atributos como datos privados e inaccesibles desde otros objetos.
  Para permitir que otros objetos consulten o modifiquen los atributos de los objetos, las clases suelen presentar métodos de acceso. De esta manera el

acceso a los datos de los objetos es controlado por el programador, evitando efectos laterales no deseados.
Con el encapsulado de los datos se consigue que las personas que utilicen un objeto sólo tengan que comprender su interfaz, olvidándose de cómo está implementada, y en definitiva, reduciendo la complejidad de utilización.

- *Principio de Modularidad:* Mediante la modularidad, se propone al programador dividir su aplicación en varios módulos diferentes (ya sea en forma de clases, paquetes o bibliotecas), cada uno de ellos con un sentido propio.
  Esta fragmentación disminuye el grado de dificultad del problema al que da respuesta el programa, pues se afronta el problema como un conjunto de problemas de menor dificultad, además de facilitar la comprensión del programa.

- *Principio de Jerarquía:* La mayoría de nosotros ve de manera natural nuestro mundo como objetos que se relacionan entre sí de una manera jerárquica. Por ejemplo, un perro es un mamífero, y los mamíferos son animales, y los animales seres vivos...
  Del mismo modo, las distintas clases de un programa se organizan mediante la *jerarquía*. La representación de dicha organización da lugar a los denominados *arboles de herencia*:

Gráfico 1. Ejemplo de árbol de herencia



Fuente: UNIVERSIDAD DE BURGOS, GUÍA DE INICIACION AL LENGUAJE JAVA. Versión 2.0. 1.999. Pág. 18.

Mediante la *herencia* una *clase hija* puede tomar determinadas propiedades de una *clase padre*. Así se simplifican los diseños y se evita la duplicación de código al no tener que volver a codificar métodos ya implementados.
Al acto de tomar propiedades de una clase padre se denomina *heredar.*

- *Principio del Paso de Mensajes:* Mediante el denominado *paso de mensajes*, un objeto puede solicitar de otro objeto que realice una acción determinada o que modifique su estado. El paso de mensajes se suele implementar como llamadas a los métodos de otros objetos.
  Desde el punto de vista de la programación estructurada, esto correspondería con la llamada a funciones.

- *Principio de Polimorfismo:* Polimorfismo quiere decir "un objeto y muchas formas". Esta propiedad permite que un objeto presente diferentes comportamientos en función del contexto en que se encuentre. Por ejemplo un método puede presentar diferentes implementaciones en función de los argumentos que recibe, recibir diferentes números de parámetros para realizar una misma operación, y realizar diferentes acciones dependiendo del nivel de abstracción en que sea llamado.

2.1.3.5 Relaciones Entre Objetos: Durante la ejecución de un programa, los diversos objetos que lo componen han de interactuar entre sí para lograr una serie de objetivos comunes.
Existen varios tipos de relaciones que pueden unir a los diferentes objetos, pero entre ellas destacan las relaciones de: asociación, todo/parte, y generalización/especialización.

- *Relaciones de Asociación:* Serían relaciones generales, en las que un objeto realiza llamadas a los servicios (métodos) de otro, interactuando de esta forma con él.
  Representan las relaciones con menos riqueza semántica.

- *Relaciones de Todo/Parte:* Muchas veces una determinada entidad existe como conjunción de otras entidades, como un conglomerado de ellas. La orientación al objeto recoge este tipo de relaciones como dos conceptos; la agregación y la composición.
  En este tipo de relaciones un *objeto componente* se integra en un *objeto compuesto*. La diferencia entre agregación y composición es que mientras

que la composición se entiende que dura durante toda la vida del objeto componedor, en la agregación no tiene por qué ser así.
Esto se puede implementar como un objeto (*objeto compuesto*) que cuenta entre sus atributos con otro objeto distinto (*objeto componente*).

- *Relaciones de Generalización/Especialización:* A veces sucede que dos clases tienen muchas de sus partes en común, lo que normalmente se abstrae en la creación de una tercera clase (*padre* de las dos) que reúne todas sus características comunes.
  El ejemplo más extendido de este tipo de relaciones es la herencia, propiedad por la que una clase (*clase hija*) recoge aquellos métodos y atributos que una segunda clase (*clase padre*) ha especificado como "heredables".
  Este tipo de relaciones es característico de la programación orientada a objetos.
  En realidad, la generalización y la especialización son diferentes perspectivas del mismo concepto, la generalización es una perspectiva ascendente (*buttom-up*), mientras que la especialización es una perspectiva descendente (*top-down*).

2.1.4 Fundamentos De Java

2.1.4.1 Introducción: De tal forma que puede ser considerado como un C++ nuevo y modernizado o bien como un C++ al que se le han amputado elementos heredados del lenguaje estructurado C.

2.1.4.2 Tokens: Un *token* es el elemento más pequeño de un programa que es significativo para el compilador. Estos *tokens* definen la estructura de Java.
Cuando se compila un programa Java, el compilador analiza el texto, reconoce y elimina los espacios en blanco y comentarios y extrae *tokens* individuales. Los *tokens* resultantes se compilan, traduciéndolos a código de byte Java, que es independiente del sistema e interpretable dentro de un entorno Java.

Los códigos de byte se ajustan al sistema de máquina virtual Java, que abstrae las diferencias entre procesadores a un procesador virtual único.
Los *tokens* Java pueden subdividirse en cinco categorías: Identificadores, palabras clave, constantes, operadores y separadores.

- *Identificadores:* Los identificadores son *tokens* que representan nombres asignables a variables, métodos y clases para identificarlos de forma única ante el compilador y darles nombres con sentido para el programador.
  Todos los identificadores de Java diferencian entre mayúsculas y minúsculas (Java es *Case Sensitive o Sensible a mayúscula*) y deben comenzar con una letra, un subrayado (_) o símbolo de dólar ($). Los caracteres posteriores del identificador pueden incluir las cifras del 0 al 9. Como nombres de identificadores no se pueden usar palabras claves de Java.

  Además de las restricciones mencionadas existen propuestas de estilo. Es una práctica estándar de Java denominar:
  - Las clases: *Clase o MiClase*.
  - Las interfaces: *Interfaz o MiInterfaz*.
  - Los métodos: *metodo() o metodoLargo().*
  - Las variables: *altura o alturaMadia.*
  - Las constantes: *CONSTANTE o CONTANTE_LARGA.*
  - Los paquetes: *java.paquete.subpaquete.*

  Sin entrar en más detalle en la siguiente línea de código se puede apreciar la declaración de una variable entera (*int*) con su correspondiente identificador:

  ```
  int alturaMedia;
  ```

- *Palabras claves:* Las palabras claves son aquellos identificadores reservados por Java para un objetivo determinado y se usan sólo de la forma limitada y específica. Java tiene un conjunto de palabras clave más rico que C o que C++, por lo que sí está aprendiendo Java con conocimientos de C o C++, asegúrese de que presta atención a las palabras clave de Java.
  Las siguientes palabras son palabras reservadas de Java:

Tabla 1. Palabras reservadas de Java

abstract	*boolean*	*break*	*byte*	*byvalue*
case	*cast*	*catch*	*char*	*class*
const	*continue*	*default*	*do*	*double*
else	*extends*	*false*	*final*	*finally*
float	*for*	*future*	*generic*	*goto*
if	*implements*	*import*	*inner*	*instanceof*
int	*interface*	*long*	*native*	*new*
null	*operator*	*outer*	*package*	*private*
protected	*public*	*rest*	*return*	*short*
static	*super*	*switch*	*syncroniced*	*this*
throw	*throws*	*transient*	*true*	*try*
var	*void*	*volatile*	*while*	

Fuente: Ibid. Pág.38.

Las palabras subrayadas son palabras reservadas pero no se utilizan. La definición de estas palabras clave no se ha revelado, ni se tiene un calendario respecto a cuándo estará alguna de ellas en la especificación o en alguna de las implementaciones de Java.

- *Literales y constantes:* Los literales son sintaxis para asignar valores a las variables. Cada variable es de un tipo de datos concreto, y dichos tipos de datos tienen sus propios literales.
  Mediante determinados modificadores (*static* y *final*) podremos crear variables *constantes*, que no modifican su valor durante la ejecución de un programa. Las constantes pueden ser numéricas, booleanas, caracteres (Unicode) o cadenas (*String*).
  Las cadenas, que contienen múltiples caracteres, aún se consideran constantes, aunque están implementadas en Java como objetos.
  Veamos un ejemplo de constante declarada por el usuario:

```
final static int ALTURA_MAXIMA = 200;
```

Se puede observar que utilizamos *final static*, para que la variable sea total y absolutamente invariable.

- *Operadores:* Conocidos también como operandos, indican una evaluación o computación para ser realizada en objetos o datos, y en definitiva sobre identificadores o constantes. Los operadores admitidos por Java son:

Tabla 2. Operadores de Java

<table>
<tr><td>+</td><td>^</td><td><=</td><td>++</td><td>%=</td></tr>
<tr><td>>>>=</td><td>-</td><td>~</td><td>>=</td><td>-</td></tr>
<tr><td>&=</td><td>.</td><td>*</td><td>&&</td><td><<</td></tr>
<tr><td>==</td><td><<=</td><td>[</td><td>/</td><td>||</td></tr>
<tr><td>>></td><td>+=</td><td>^=</td><td>]</td><td>%</td></tr>
<tr><td>!</td><td>>>></td><td>=</td><td>!=</td><td>(</td></tr>
<tr><td>&</td><td><</td><td>*=</td><td>)</td><td>|</td></tr>
<tr><td>></td><td>¿!!</td><td>/=</td><td>>></td><td></td></tr>
</table>

Fuente: Ibid.

- *Separadores:* Se usan para informar al compilador de Java de cómo están agrupadas las cosas en el código.
  Los separadores admitidos por Java son: *{ } , : ;*

- *Comentarios y espacios en blanco:* El compilador de Java reconoce y elimina los espacios en blanco, tabuladores, retornos de carro y comentarios durante el análisis del código fuente.
  Los comentarios se pueden presentar en tres formatos distintos:

Tabla 3. Formatos de comentario de Java

Formato	Uso
/*comentario*/	Se ignoran todos los caracteres entre /* */. Proviene del C
//comentario	Se ignoran todos los caracteres detrás de // hasta el fin de línea. Proviene del C++
/**comentario*/	Los mismo que /* */pero se podrán utilizar para documentación automática

Fuente: Ibid. Pág. 39

Por ejemplo la siguiente línea de código presenta un comentario:

```
int alturaMinima = 150; // No menos de 150 centímetros
```

2.1.4.3 Expresiones: Los operadores, variables y las llamadas a métodos pueden ser combinados en secuencias conocidas como expresiones. El comportamiento real de un programa Java se logra a través de expresiones, que se agrupan para crear *sentencias*.
Una expresión es una serie de variables, operadores y llamadas a métodos (construida conforme a la sintaxis del lenguaje) que se evalúa a un único valor.
Entre otras cosas, las expresiones son utilizadas para realizar cálculos, para asignar valores a variables, y para ayudar a controlar la ejecución del flujo del programa. La tarea de una expresión se compone de dos partes: realiza el cálculo indicado por los elementos de la expresión y devuelve el valor obtenido como resultado del cálculo.

Los operadores devuelven un valor, por lo que el uso de un operador es una expresión.
Por ejemplo, la siguiente sentencia es una expresión:

```
int contador=1;

contador++;
```

La expresión *contador++* en este caso particular se evalúa al valor *1*, que era el valor de la variable *contador* antes de que la operación ocurra, pero la variable *contador* adquiere un valor de *2*.
El tipo de datos del valor devuelto por una expresión depende de los elementos utilizados en la expresión. La expresión *contador++* devuelve un entero porque ++ devuelve un valor del mismo tipo que su operando y *contador* es un entero. Otras expresiones devuelven valores booleanos, cadenas...

Una expresión de llamada a un método se evalúa al valor de retorno del método; así el tipo de dato de la expresión de llamada a un método es el mismo que el tipo de dato del valor de retorno de ese método.
Otra sentencia interesante sería:

```
in.read( ) != -1 // in es un flujo de entrada
```

Esta sentencia se compone de dos expresiones:

1. La primera expresión es una llamada al método *in.read()* El método *in.read()* ha sido declarado para devolver un entero, por lo que la expresión *in.read()* se evalúa a un entero.

2. La segunda expresión contenida en la sentencia utiliza el operador *!=* , que comprueba si dos operandos son distintos. En la sentencia en cuestión, los operandos son *in.read()* y -1. El operando *in.read()* es válido para el operador _ porque in.read() es una expresión que se evalúa a un entero, así que la expresión *in.read()!=-1* compara dos enteros. El valor devuelto por esta expresión será verdadero o falso dependiendo del resultado de la lectura del fichero *in*.

Como se puede observar, Java permite construir sentencias (expresiones compuestas) a partir de varias expresiones más pequeñas con tal que los tipos de datos requeridos por una parte de la expresión concuerden con los tipos de datos de la otra.

2.1.4.4 Bloques y Ámbito: En Java el código fuente está dividido en partes separadas por llaves, denominas *bloques*. Cada bloque existe independiente de lo que está fuera de él, agrupando en su interior sentencias (expresiones) relacionadas.

Desde un bloque externo parece que todo lo que está dentro de llaves se ejecuta como una sentencia. Pero, ¿qué es un bloque externo?. Esto tiene explicación si entendemos que existe una *jerarquía de bloques*, y que un bloque puede contener uno o más subbloques anidados.

El concepto de ámbito está estrechamente relacionado con el concepto de bloque y es muy importante cuando se trabaja con variables en Java. El ámbito se refiere a cómo las secciones de un programa (bloques) afectan el tiempo de vida de las variables.
Toda variable tiene un ámbito, en el que es usada, que viene determinado por los bloques. Una variable definida en un bloque interno no es visible por el bloque externo.

Las llaves de separación son importantes no sólo en un sentido lógico, ya que son la forma de que el compilador diferencie dónde acaba una sección de código y dónde comienza otra, sino que tienen una connotación estética que facilita la lectura de los programas al ser humano.

Así mismo, para identificar los diferentes bloques se utilizan sangrías. Las sangrías se utilizan para el programador, no para el compilador. La sangría (también denominada *indentación*) más adecuada para la estética de un programa Java son dos espacios:

```
{
        // Bloque externo
        int x = 1;
        {
        // Bloque interno invisible al exterior
        int y = 2;
        }
        x = y; // Da error: Y fuera de ámbito
}
```

2.1.5 Clases Y Objetos

2.1.5.1 Definición De Una Clase

- *Introducción:* El elemento básico de la programación orientada a objetos en Java es la clase. Una clase define la forma y comportamiento de un objeto. Para crear una clase sólo se necesita un archivo fuente que contenga la palabra clave reservada *class* seguida de un identificador legal y un bloque delimitado por dos llaves para el cuerpo de la clase.

```
class MiPunto {
}
```

  Un archivo de Java debe tener el mismo nombre que la clase que contiene, y se les suele asignar la extensión *".java"* Por ejemplo la clase *MiPunto* se guardaría en un fichero que se llamase *MiPunto.java*. Hay que tener presente que en Java se diferencia entre mayúsculas y minúsculas; el nombre de la clase y el de archivo fuente han de ser exactamente iguales. Aunque la clase *MiPunto* es sintácticamente correcta, es lo que se viene a llamar una *clase vacía*, es decir, una clase que no hace nada. Las clases típicas de Java incluirán variables y métodos de instancia. Los programas en Java completos constarán por lo general de varias clases de Java en distintos archivos fuente.

  Una clase es una plantilla para un objeto. Por lo tanto define la estructura de un objeto y su interfaz funcional, en forma de métodos. Cuando se ejecuta un programa en Java, el sistema utiliza definiciones de clase para crear instancias de las clases, que son los objetos reales. Los términos instancia y objeto se utilizan de manera indistinta. La forma general de una definición de clase es:

```
class Nombre_De_Clase {
        tipo_de_variable nombre_de_atributo1;
        tipo_de_variable nombre_de_atributo2;
        // . . .
        tipo_devuelto nombre_de_método1( lista_de_parámetros ) {
                cuerpo_del_método1;
        }
```

```
    tipo_devuelto nombre_de_método2( lista_de_parámetros ) {
        cuerpo_del_método2;
    }
    // . . .
}
```

Los tipos *tipo_de_variable* y *tipo_devuelto*, han de ser tipos simples Java o nombres de otras clases ya definidas. Tanto *Nombre_De_Clase*, como los *nombre_de_atributo* y *nombre_de_método*, han de ser identificadores Java válidos.

- *Los atributos:* Los datos se encapsulan dentro de una clase declarando variables dentro de las llaves de apertura y cierre de la declaración de la clase, variables que se conocen como atributos. Se declaran igual que las variables locales de un método en concreto.
  Por ejemplo, este es un programa que declara una clase *MiPunto*, con dos atributos enteros llamados *x* e *y.*

```
class MiPunto {
    int x, y;
}
```

  Los atributos se pueden declarar con dos clases de tipos: un tipo simple Java (ya descritos), o el nombre de una clase.

  Cuando se realiza una instancia de una clase (creación de un objeto) se reservará en la memoria un espacio para un conjunto de datos como el que definen los atributos de una clase. A este conjunto de variables se le denomina *variables de instancia*.

- *Los métodos:* Los métodos son subrutinas que definen la interfaz de una clase, sus capacidades y comportamiento.
  Un método ha de tener por nombre cualquier identificador legal distinto de los ya utilizados por los nombres de la clase en que está definido. Los métodos se declaran al mismo nivel que las variables de instancia dentro de una definición de clase.

En la declaración de los métodos se define el tipo de valor que devuelven y a una lista formal de parámetros de entrada, de sintaxis *tipo identificador* separadas por comas. La forma general de una declaración de método es:

```
tipo_devuelto nombre_de_método( lista-formal-de-parámetros ) {
        cuerpo_del_método;
}
```

Por ejemplo el siguiente método devuelve la suma de dos enteros:

```
int metodoSuma( int paramX, int paramY ) {
        return ( paramX + paramY );
};
```

En el caso de que no se desee devolver ningún valor se deberá indicar como tipo la palabra reservada *void*. Así mismo, si no se desean parámetros, la declaración del método debería incluir un par de paréntesis vacíos (sin *void*):

```
void metodoVacio( ) { };
```

Los métodos son llamados indicando una instancia individual de la clase, que tendrá su propio conjunto único de variables de instancia, por lo que los métodos se pueden referir directamente a ellas.
El método *inicia()* para establecer valores a las dos variables de instancia sería el siguiente:

```
void inicia( int paramX, int paramY ) {
        x = paramX;
        y = paramY;
}
```

2.1.5.2 La Instanciación De Las Clases: Los Objetos

- *Referencias a Objetos e Instancias:* Los tipos simples de Java describían el tamaño y los valores de las variables. Cada vez que se crea una clase se añade otro tipo de dato que se puede utilizar igual que uno de los tipos simples. Por ello al declarar una nueva variable, se puede utilizar un

nombre de clase como tipo. A estas variables se las conoce como *referencias a objetos*.

Todas las referencias a objeto son compatibles también con las instancias de subclases de su tipo. Del mismo modo que es correcto asignar un *byte* a una variable declarada como *int*, se puede declarar que una variable es del tipo *MiClase* y guardar una referencia a una instancia de este tipo de clase:

```
MiPunto p;
```

Esta es una declaración de una variable *p* que es una referencia a un objeto de la clase *MiPunto*, de momento con un valor por defecto de *null*. La referencia *null* es una referencia a un objeto de la clase *Object*, y se podrá convertir a una referencia a cualquier otro objeto porque todos los objetos son *hijos* de la clase *Object*.

- *Constructores:* Las clases pueden implementar un método especial llamado *constructor*. Un constructor es un método que inicia un objeto inmediatamente después de su creación. De esta forma nos evitamos el tener que iniciar las variables explícitamente para su iniciación.

  El constructor tiene exactamente el mismo nombre de la clase que lo implementa; no puede haber ningún otro método que comparta su nombre con el de su clase. Una vez definido, se llamará automáticamente al constructor al crear un objeto de esa clase (al utilizar el operador *new*).

  El constructor no devuelve ningún tipo, ni siquiera *void*. Su misión es iniciar todo estado interno de un objeto (sus atributos), haciendo que el objeto sea utilizable inmediatamente; reservando memoria para sus atributos, iniciando sus valores...

  Por ejemplo:

```
MiPunto( ) {
        inicia( -1, -1 );
}
```

  Este constructor denominado *constructor por defecto*, por no tener parámetros, establece el valor *-1* a las variables de instancia *x* e *y* de los objetos que construya.

El compilador, por defecto, llamará al constructor de la superclase *Object()* si no se especifican parámetros en el constructor.
Este otro constructor, sin embargo, recibe dos parámetros:

```
MiPunto( int paraX, int paraY ) {
        inicia( paramX, paramY );
}
```

La lista de parámetros especificada después del nombre de una clase en una sentencia *new* se utiliza para pasar parámetros al constructor.

Se llama al método constructor justo después de crear la instancia y antes de que *new* devuelva el control al punto de la llamada.
Así, cuando ejecutamos el siguiente programa:

```
MiPunto p1 = new MiPunto(10, 20);
System.out.println( “p1.- x = “ + p1.x + “ y = “ + p1.y );
```

Se muestra en la pantalla:

```
p1.- x = 10 y = 20
```

Para crear un programa Java que contenga ese código, se debe de crear una clase que contenga un método *main()*. El intérprete *java* se ejecutará el método *main* de la clase que se le indique como parámetro.

- *El operador new:* El operador *new* crea una instancia de una clase (*objetos*) y devuelve una referencia a ese objeto. Por ejemplo:

```
MiPunto p2 = new MiPunto(2,3);
```

Este es un ejemplo de la creación de una instancia de *MiPunto*, que es controlador por la referencia a objeto *p2*.

Hay una distinción crítica entre la forma de manipular los tipos simples y las clases en Java: Las referencias a objetos realmente no contienen a los objetos a los que referencian. De esta forma se pueden crear múltiples referencias al mismo objeto, como por ejemplo:

```
MiPunto p3 =p2;
```

Aunque tan sólo se creó un objeto *MiPunto*, hay dos variables (*p2* y *p3*) que lo referencian. Cualquier cambio realizado en el objeto referenciado por *p2* afectará al objeto referenciado por *p3*. La asignación de *p2* a *p3* no reserva memoria ni modifica el objeto.

De hecho, las asignaciones posteriores de *p2* simplemente desengancharán *p2* del objeto, sin afectarlo:

```
p2 = null; // p3 todavía apunta al objeto creado con new
```

Aunque se haya asignado *null* a *p2*, *p3* todavía apunta al objeto creado por el operador *new*.
Cuando ya no haya ninguna variable que haga referencia a un objeto, Java reclama automáticamente la memoria utilizada por ese objeto, a lo que se denomina *recogida de basura*.

Cuando se realiza una instancia de una clase (mediante *new*) se reserva en la memoria un espacio para un conjunto de datos como el que definen los atributos de la clase que se indica en la instanciación. A este conjunto de variables se le denomina *variables de instancia*.

La potencia de las variables de instancia es que se obtiene un conjunto distinto de ellas cada vez que se crea un objeto nuevo. Es importante el comprender que cada objeto tiene su propia copia de las variables de instancia de su clase, por lo que los cambios sobre las variables de instancia de un objeto no tienen efecto sobre las variables de instancia de otro.

El siguiente programa crea dos objetos *MiPunto* y establece los valores de *x* e *y* de cada uno de ellos de manera independiente para mostrar que están realmente separados.

```
MiPunto p4 = new MiPunto( 10, 20 );
MiPunto p5 = new MiPunto( 42, 99 );
System.out.println(“p4.- x = “ + p4.x + “ y = “ + p4.y);
System.out.println(“p5.- x = “ + p5.x + “ y = “ + p5.y);
```

Este es el aspecto de salida cuando lo ejecutamos.

p4.- x = 10 y = 20
p5.- x = 42 y = 99

2.1.5.3 Acceso Al Objeto

- *El operador punto (.):* El operador punto (.) se utiliza para acceder a las variables de instancia y los métodos contenidos en un objeto, mediante su referencia a objeto:

```
referencia_a_objeto.nombre_de_variable_de_instancia
referencia_a_objeto.nombre_de_método( lista-de-parámetros );
```

Hemos creado un ejemplo completo que combina los operadores *new* y punto para crear un objeto *MiPunto*, almacenar algunos valores en él e imprimir sus valores finales:

```
MiPunto p6 = new MiPunto( 10, 20 );
System.out.println (“p6.- 1. X=“ + p6.x + “ , Y=“ + p6.y);
p6.inicia( 30, 40 );
System.out.println (“p6.- 2. X=“ + p6.x + “ , Y=“ + p6.y);
```

Cuando se ejecuta este programa, se observa la siguiente salida:

p6.- 1. X=10 , Y=20
p6.- 2. X=30 , Y=40

Durante las impresiones (método *println()*) se accede al valor de las variables mediante *p6.x* y *p6.y*, y entre una impresión y otra se llama al método *inicia()*, cambiando los valores de las variables de instancia.
Este es uno de los aspectos más importantes de la diferencia entre la programación orientada a objetos y la programación estructurada. Cuando se llama al método *p5.inicia()*, lo primero que se hace en el método es sustituir los nombres de los atributos de la clase por las correspondientes variables de instancia del objeto con que se ha llamado. Así por ejemplo *x* se convertirá en *p6.x*.

Si otros objetos llaman a *inicia()*, incluso si lo hacen de una manera concurrente, no se producen *efectos laterales*, ya que las variables de instancia sobre las que trabajan son distintas.

- *La referencia this:* Java incluye un valor de referencia especial llamado *this*, que se utiliza dentro de cualquier método para referirse al objeto actual. El valor *this* se refiere al objeto sobre el que ha sido llamado el método actual. Se puede utilizar *this* siempre que se requiera una referencia a un objeto del tipo de una clase actual. Si hay dos objetos que utilicen el mismo código, seleccionados a través de otras instancias, cada uno tiene su propio valor único de *this*.

Un refinamiento habitual es que un constructor llame a otro para construir la instancia correctamente. El siguiente constructor llama al constructor parametrizado *MiPunto(x,y)* para terminar de iniciar la instancia:

```
MiPunto() {
        this( -1, -1 ); // Llama al constructor parametrizado
}
```

En Java se permite declarar variables locales, incluyendo parámetros formales de métodos, que se solapen con los nombres de las variables de instancia.

No se utilizan *x* e *y* como nombres de parámetro para el método *inicia*, porque ocultarían las variables de instancia *x* e *y* reales del ámbito del método. Si lo hubiésemos hecho, entonces *x* se hubiera referido al parámetro formal, ocultando la variable de instancia *x.*

```
void inicia2( int x, int y ) {
        x = x; // Ojo, no modificamos la variable de instancia!!!
        this.y = y; // Modificamos la variable de instancia!!!
}
```

2.1.6 Herencia

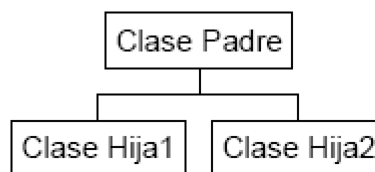
2.1.6.1 Introducción: La verdadera potencia de la programación orientada a objetos radica en su capacidad para reflejar la abstracción que el cerebro humano realiza automáticamente durante el proceso de aprendizaje y el proceso de análisis de información.

Las personas percibimos la realidad como un conjunto de objetos interrelacionados. Dichas interrelaciones, pueden verse como un conjunto de abstracciones y generalizaciones que se han ido asimilando desde la niñez. Así, los defensores de la programación orientada a objetos afirman que esta técnica se adecua mejor al funcionamiento del cerebro humano, al permitir descomponer un problema de cierta magnitud en un conjunto de problemas menores subordinados del primero.

La capacidad de descomponer un problema o concepto en un conjunto de objetos relacionados entre sí, y cuyo comportamiento es fácilmente identificable, puede ser muy útil para el desarrollo de programas informáticos.

2.1.6.2 Jerarquía: La herencia es el mecanismo fundamental de relación entre clases en la orientación a objetos. Relaciona las clases de manera jerárquica; una clase *padre* o *superclase* sobre otras clases *hijas* o *subclases*.

Gráfico 2. Ejemplo de un árbol de herencia.



Fuente: Ibid. Pág. 70

Los descendientes de una clase heredan todas las variables y métodos que sus ascendientes hayan especificado como *heredables*, además de crear los suyos propios.

La característica de herencia, nos permite definir nuevas clases derivadas de otra ya existente, que la especializan de alguna manera. Así logramos definir una jerarquía de clases, que se puede mostrar mediante un árbol de herencia.

En todo lenguaje orientado a objetos existe una jerarquía, mediante la que las clases se relacionan en términos de herencia. En Java, el punto más alto de la jerarquía es la clase *Object* de la cual derivan todas las demás clases.

2.1.6.3 Herencia Múltiple: En la orientación a objetos, se consideran dos tipos de herencia, simple y múltiple. En el caso de la primera, una clase sólo puede derivar de una única superclase. Para el segundo tipo, una clase puede descender de varias superclases.

En Java sólo se dispone de herencia simple, para una mayor sencillez del lenguaje, si bien se compensa de cierta manera la inexistencia de herencia múltiple con un concepto denominado *interface*, que estudiaremos más adelante.

2.1.6.4 Declaración: Para indicar que una clase deriva de otra, heredando sus propiedades (métodos y atributos), se usa el término *extends*, como en el siguiente ejemplo:

```
public class SubClase extends SuperClase {
        // Contenido de la clase_
}
```

Por ejemplo, creamos una clase *MiPunto3D*, hija de la clase ya mostrada *MiPunto*:

```
class MiPunto3D extends MiPunto {
        int z;
        MiPunto3D( ) {
                x = 0; // Heredado de MiPunto
                y = 0; // Heredado de MiPunto
                z = 0; // Nuevo atributo
        }
}
```

La palabra clave *extends* se utiliza para decir que deseamos crear una subclase de la clase que es nombrada a continuación, en nuestro caso *MiPunto3D* es hija de *MiPunto*.

2.1.6.5 Limitaciones En La Herencia: Todos los campos y métodos de una clase son siempre accesibles para el código de la misma clase.
Para controlar el acceso desde otras clases, y para controlar la herencia por las subclases, los miembros (atributos y métodos) de las clases tienen tres modificadores posibles de control de acceso:

- *public*: Los miembros declarados *public* son accesibles en cualquier lugar en que sea accesible la clase, y son heredados por las subclases.

- *private*: Los miembros declarados *private* son accesibles sólo en la propia clase.

- *protected*: Los miembros declarados *protected* son accesibles sólo para sus subclases.

Por ejemplo:

```
class Padre { // Hereda de Object
// Atributos
private int numeroFavorito, nacidoHace, dineroDisponible;
// Métodos
public int getApuesta() {
        return numeroFavorito;
}
protected int getEdad() {
        return nacidoHace;
}
private int getSaldo() {
        return dineroDisponible;
        }
}
class Hija extends Padre {
        // Definición
}
class Visita {
        // Definición
}
```

En este ejemplo, un objeto de la clase *Hija*, hereda los tres atributos (*numeroFavorito*, *nacidoHace* y *dineroDisponible*) y los tres métodos (*getApuesta()*, *getEdad()* y *getSaldo()*) de la clase *Padre*, y podrá invocarlos. Cuando se llame al método *getEdad()* de un objeto de la clase *Hija*, se devolverá el valor de la variable de instancia *nacidoHace* de ese objeto, y no de uno de la clase *Padre*.

Sin embargo, un objeto de la clase *Hija*, no podrá invocar al método *getSaldo()* de un objeto de la clase *Padre*, con lo que se evita que el *Hijo* conozca el estado de la cuenta corriente de un *Padre.*
La clase *Visita*, solo podrá acceder al método *getApuesta()*, para averiguar el número favorito de un *Padre*, pero de ninguna manera podrá conocer ni su saldo, ni su edad

2.1.7 Operaciones Avanzadas En Las Clases

2.1.7.1 Introducción: La programación orientada a objetos en Java va mucho más allá de las clases, los objetos y la herencia. Java presenta una serie de capacidades que enriquecen el modelo de objetos que se puede representar en un programa Java.

2.1.7.2 Operaciones Avanzadas En La Herencia

- *Introducción:* En el capítulo anterior ya se han estudiado los fundamentos de la herencia en Java. Sin embargo, el lenguaje tiene muchas más posibilidades en este aspecto, como estudiaremos a continuación.
  Conviene recordar que estamos utilizando el código de la clase *MiPunto*, cuyo código se puede encontrar en el apartado *"Clases y Objetos"*.

- *Los elementos globales: static:* A veces se desea crear un método o una variable que se utiliza fuera del contexto de cualquier instancia, es decir, de una manera global a un programa. Todo lo que se tiene que hacer es declarar estos elementos como *static.*
  Esta es la manera que tiene Java de implementar funciones y variables globales.
  Por ejemplo:

  ```
  static int a = 3;
  ```

```
static void metodoGlobal() {
	// implementación del método
}
```

No se puede hacer referencia a *this* o a *super* dentro de un método *static*. Mediante atributos estáticos, todas las instancias de una clase además del espacio propio para variables de instancia, comparten un espacio común. Esto es útil para modelizar casos de la vida real.

Otro aspecto en el que es útil *static* es en la creación de métodos a los que se puede llamar directamente diciendo el nombre de la clase en la que están declarados. Se puede llamar a cualquier método *static*, o referirse a cualquier variable *static* utilizando el operador punto con el nombre de la clase, sin necesidad de crear un objeto de ese tipo:

```
class ClaseStatic {
	int atribNoStatic = 42;
static int atribStatic = 99;
	static void metodoStatic() {
		System.out.println("Met. static = " + atribStatic);
	}
	static void metodoNoStatic() {
		System.out.println("Met. no static = " + atribNoStatic);
	}
}
```

El siguiente código es capaz de llamar a *metodoStatic* y *atribStatic* nombrando directamente la clase (sin objeto, sin *new*), por haber sido declarados *static*.

```
System.out.println("At. static = " + ClaseStatic.atribStatic);
ClaseStatic.metodoStatic(); // Sin instancia
new ClaseStatic().metodoNoStatic(); // Hace falta instancia
```

Si ejecutamos este programa obtendríamos:

```
At. static = 99
Met. static = 99
Met. no static = 42
```

Debe tenerse en cuenta que en un método estático tan sólo puede hacerse referencia a variables estáticas.

- *Las clases y métodos abstractos: abstract:* Hay situaciones en las que se necesita definir una clase que represente un concepto abstracto, y por lo tanto no se pueda proporcionar una implementación completa de algunos de sus métodos.

  Se puede declarar que ciertos métodos han de ser sobrescritos en las subclases, utilizando el modificador de tipo *abstract.* A estos métodos también se les llama *responsabilidad de subclases*. Cualquier subclase de una clase *abstract* debe implementar todos los métodos *abstract* de la superclase o bien ser declarada también como *abstract.*

  Cualquier clase que contenga métodos declarados como *abstract* también se tiene que declarar como *abstact*, y no se podrán crear instancias de dicha clase (operador *new*).
  Por último se pueden declarar constructores *abstract* o métodos *abstract static.*
  Veamos un ejemplo de clases abstractas:

```
abstract class claseA {
        abstract void metodoAbstracto();
        void metodoConcreto() {
                System.out.println("En el metodo concreto de claseA");
        }
}
class claseB extends claseA {
        void metodoAbstracto(){
                System.out.println("En el metodo abstracto de claseB");
        }
}
```

  La clase abstracta *claseA* ha implementado el método concreto *metodoConcreto()*, pero el método *metodoAbstracto()* era abstracto y por eso ha tenido que ser redefinido en la clase hija *claseB*.

```
claseA referenciaA = new claseB();
```

```
referenciaA.metodoAbstracto();
referenciaA.metodoConcreto();
```

La salida de la ejecución del programa es:

```
En el metodo abstracto de claseB
En el metodo concreto de claseA
```

2.1.7.3 El Polimorfismo

- *Selección dinámica de método:* Las dos clases implementadas a continuación tienen una relación subclase/superclase simple con un único método que se sobrescribe en la subclase:

```
class claseAA {
        void metodoDinamico() {
                System.out.println("En el metodo dinamico de claseAA");
        }
}
class claseBB extends claseAA {
        void metodoDinamico() {
                System.out.println("En el metodo dinamico de claseBB");
        }
}
```

Por lo tanto si ejecutamos:

```
claseAA referenciaAA = new claseBB();
referenciaAA.metodoDinamico();
```

La salida de este programa es:

```
En el metodo dinamico de claseBB
```

Se declara la variable de tipo *claseA*, y después se almacena una referencia a una instancia de la clase *claseB* en ella. Al llamar al método *metodoDinamico()* de *claseA*, el compilador de Java verifica que *claseA* tiene un método llamado *metodoDinamico()*, pero el intérprete de Java

observa que la referencia es realmente una instancia de *claseB*, por lo que llama al método *metodoDinamico()* de *claseB* en vez de al de *claseA*.
Esta forma de *polimorfismo dinamico en tiempo de ejecución* es uno de los mecanismos más poderosos que ofrece el diseño orientado a objetos para soportar la reutilización del código y la robustez.

- *Sobrescritura de un método:* Durante una jerarquía de herencia puede interesar volver a escribir el cuerpo de un método, para realizar una funcionalidad de diferente manera dependiendo del nivel de abstracción en que nos encontremos. A esta modificación de funcionalidad se le llama sobrescritura de un método.

  Por ejemplo, en una herencia entre una clase *SerVivo* y una clase hija *Persona*; si la clase *SerVivo* tuviese un método *alimentarse()*, debería volver a escribirse en el nivel de *Persona*, puesto que una persona no se alimenta ni como un *Animal*, ni como una *Planta*...

  La mejor manera de observar la diferencia entre sobrescritura y sobrecarga es mediante un ejemplo. A continuación se puede observar la implementación de la sobrecarga de la distancia en 3D y la sobrescritura de la distancia en 2D.

```
class MiPunto3D extends MiPunto {
        int x,y,z;
        double distancia(int pX, int pY) { // Sobrescritura
                int retorno=0;
                retorno += ((x/z)-pX)*((x/z)-pX);
                retorno += ((y/z)-pY)*((y/z)-pY);
                return Math.sqrt( retorno );
        }
}
```

  Se inician los objetos mediante las sentencias:

```
MiPunto p3 = new MiPunto(1,1);
MiPunto p4 = new MiPunto3D(2,2);
```

  Y llamando a los métodos de la siguiente forma:

```
p3.distancia(3,3); //Método MiPunto.distancia(pX,pY)
p4.distancia(4,4); //Método MiPunto3D.distancia(pX,pY)
```

Los métodos se seleccionan en función del tipo de la instancia en tiempo de ejecución, no a la clase en la cual se está ejecutando el método actual. A esto se le llama *selección dinámica de método*.

- *Sobrecarga de método:* Es posible que necesitemos crear más de un método con el mismo nombre, pero con listas de parámetros distintas. A esto se le llama *sobrecarga del método*. La sobrecarga de método se utiliza para proporcionar a Java un comportamiento *polimórfico.*

  Un ejemplo de uso de la sobrecarga es por ejemplo, el crear constructores alternativos en función de las coordenadas, tal y como se hacía en la clase *MiPunto*:

```
MiPunto( ) { //Constructor por defecto
        inicia( -1, -1 );
}
MiPunto( int paramX, int paramY ) { // Parametrizado
        this.x = paramX;
        y = paramY;
}
```

  Se llama a los constructores basándose en el número y tipo de parámetros que se les pase. Al número de parámetros con tipo de una secuencia específica se le llama *signatura de tipo*. Java utiliza estas signaturas de tipo para decidir a qué método llamar.

  Para distinguir entre dos métodos, no se consideran los nombres de los parámetros formales sino sus tipos:

```
MiPunto p1 = new MiPunto(); // Constructor por defecto
MiPunto p2 = new MiPunto( 5, 6 ); // Constructor parametrizado
```

- *Limitación de la sobrescritura: final:* Todos los métodos y las variables de instancia se pueden sobrescribir por defecto. Si se desea declarar que ya no se quiere permitir que las subclases sobrescriban las variables o

métodos, éstos se pueden declarar como final. Esto se utiliza a menudo para crear el equivalente de una constante de C++.
Es un convenio de codificación habitual elegir identificadores en mayúsculas para las variables que sean *final* por ejemplo:

```
final int NUEVO_ARCHIVO = 1;
```

2.1.7.4 Las Referencias Polimorficas: This y Super

- *Acceso a la propia clase: this:* Aunque ya se explicó en el apartado *"Clases y Objetos"* el uso de la referencia *this* como modificador de ámbito, también se la puede nombrar como ejemplo de polimorfismo.
  Además de hacer continua referencia a la clase en la que se invoque, también vale para sustituir a sus constructores, utilizándola como método:

  ```
  this(); // Constructor por defecto
  this( int paramX, int paramY ); // Constructor parametrizado
  ```

- *Acceso a la superclase: super:* Ya hemos visto el funcionamiento de la referencia *this* como referencia de un objeto hacia sí mismo. En Java existe otra referencia llamada *super*, que se refiere directamente a la superclase.
  La referencia *super* usa para acceder a métodos o atributos de la superclase.

  Podíamos haber implementado el constructor de la clase *MiPunto3D* (hija de *MiPunto*) de la siguiente forma:

  ```
  MiPunto3D( int x, int y, int z ) {
          super( x, y ); // Aquí se llama al constructor de MiPunto
          this.z = super.metodoSuma( x, y ); // Método de la superclase
  }
  ```

  Con una sentencia *super.metodoSuma(x,y)* se llamaría al método *metodoSuma()* de la superclase de la instancia *this*. Por el contrario con *super()* llamamos al constructor de la superclase.

2.1.7.5 La Composición: Otro tipo de relación muy habitual en los diseños de los programas es la composición. Los objetos suelen estar compuestos de conjuntos de objetos más pequeños; un coche es un conjunto de motor y carrocería, un motor es un conjunto de piezas, y así sucesivamente. Este concepto es lo que se conoce como *composición*.

La forma de implementar una relación de composición en Java es incluyendo una referencia a objeto de la clase componedora en la clase compuesta.
Por ejemplo, una clase *AreaRectangular*, quedaría definida por dos objetos de la clase *MiPunto*, que representasen dos puntas contrarias de un rectángulo:

```
class AreaRectangular {
        MiPunto extremo1; //extremo inferior izquierdo
        MiPunto extremo2; //extremo superior derecho
        AreaRectangular() {
                extremo1=new MiPunto();
                extremo2=new MiPunto();
        }
        boolean estaEnElArea( MiPunto p ){
                if ( ( p.x>=extremo1.x && p.x<=extremo2.x ) &&
                ( p.y>=extremo1.y && p.y<=extremo2.y ) )
                return true;
                else
                return false;
        }
}
```

Puede observarse que las referencias a objeto (*extreme1* y *extremo2*) son iniciadas, instanciando un objeto para cada una en el constructor. Así esta clase mediante dos puntos, referenciados por *extremo1* y *extremo2*, establece unos límites de su área, que serán utilizados para comprobar si un punto está en su área en el método *estaEnElArea()*.

2.2 MARCO NORMATIVO.

Para el sustento y la realización del proyecto, se sujetara a los estándares internacionales ISO y DIN, en cuanto a las ecuaciones que definen los parámetros de los engranajes, las normas que se utilizarán son:

- ISO 21771:2007 specifies the geometric concepts and parameters for cylindrical gears with involute helicoid tooth flanks. Flank modifications are included. It also covers the concepts and parameters for cylindrical gear pairs with parallel axes and a constant gear ratio, which consist of cylindrical gears according to it. Gear and mating gear in these gear pairs have the same basic rack tooth profile[2].
- DIN 3960, Definitions, parameters and equations for involute cylindrical gears and gear pairs[3].

[2] http://www.iso.org/iso/catalogue_detail.htm?csnumber=35989
[3] http://www.nam.din.de/cmd?artid=1340388&contextid=nam&bcrumblevel=1&subcommitteeid=66502451&level=tpl-art-detailansicht&committeeid=54738979&languageid=en

3 METODOLOGÍA

3.1 ENFOQUE DE LA INVESTIGACIÓN

El enfoque de este proyecto de grado es empírico-analítico, dado que esta aplicación de software será puesta a prueba mediante el análisis de los resultados que arroja esta aplicación respecto a los posibles casos de entrada, detectando así posibles falencias o incongruencias respecto a los requerimientos del Instituto de Metrologia Alemán, dichos requerimientos se basan en la experiencia tanto teórica como practica en el tema de engranajes que posee este Instituto.

3.2 LÍNEA DE INVESTIGACIÓN DE USB/ SUB-LÍNEA DE FACULTAD/ CAMPO TEMÁTICO DEL PROGRAMA

Línea de investigación:

Tecnologías actuales y sociedad.

Sub-línea de facultad:

Sistemas de información y comunicación.

Campo temático del programa:

Software de usuario (aplicaciones).

3.3 TÉCNICAS DE RECOLECCIÓN DE INFORMACIÓN

La recolección de información del proyecto se realizará realizando pruebas, ingresando diferentes posibilidades de datos y con las posibles combinaciones de tal forma que cumpla con los requerimientos del Instituto de Metrología Alemán.

4 DESARROLLO INGENIERIL

En la presente sección se describirán el camino que se tomó y los parámetros que fueron definidos y seguidos para el desarrollo del Involute Calculator, de acuerdo a los requerimientos del Instituto de Metrología Alemán.

4.1 DESARROLLO DE LAS DOS PRIMERAS SÚPER CLASES.

El primer paso en el desarrollo de esta aplicación de software fue la creación de la súper clase principal llamada "VariableGeneralSuper" la cual contendrá las banderas que se utilizarán en el programa en general, con sus respectivas explicaciones acerca de su utilidad; de esta clase se extiende las súper clases: "VariableRealSuper" y "VariableIntSuper", pero para este primer paso solo se implemento la súper clase "VariableRealSuper". Estas dos súper clases tienen una estructura muy similar por no decir igual, su diferencia radica en que de VariableRealSuper se extenderá las clases en las que están definidas aquellos parámetros de los engranajes que pueden tener como valores números de tipo doble (m_n, m_t, β,….) y de la súper clase VariableIntSuper se extenderá las clases en las que están definidas aquellas variables de los engranajes que pueden ser números enteros (z).

En las súper clases VariableRealSuper y VariableIntSuper se definirán las banderas que almacenarán tales valores como: el valor de entrada, el valor calculado, el valor máximo y mínimo de los valores calculados, etc., de cada una de los parámetros de los engranajes; también se definirán aquellos métodos que puedan ser usados por todas las clases que se derivan de ella.

4.2 DEFINICIÓN DE LOS NOMBRES DE LAS VARIABLES A USAR.

Una de las principales cualidades con la que debe cumplir esta aplicación de software a cabalidad es ser sumamente fácil de entender, tanto para comprender la función de una bandera solo con leer su nombre, como en las explicaciones, con esto se quiere decir que ya que las explicaciones y los nombres de las

banderas esta en ingles, se pensó en utilizar palabras y expresiones que para entenderlas solo es necesario tener conocimientos básicos del idioma ingles, porque se desea a tiempo futuro mejorar y extender esta aplicacion de tal forma que cualquier persona que sea contratada para esta labor le sea sumamente fácil entender lo que se hizo y el camino que se tomó para su desarrollo, sin olvidar que cuando el proyecto esté finalizado se redactarán las memorias y las reglas que sigue el programa.

Banderas definidas en la súper clase “VariableGeneralSuper” (todas son de tipo booleano):

De tipo constante:

- *CALCULATION_ONCEMORE (true):* Constante que indica que es necesario realizar una vez más todas las ecuaciones.

- *CALCULATION_ONCEMORE_NOT (false):* Constante que indica que no es necesario realizar una vez más todas las ecuaciones.

- *CALCULATION_RUN (true):* Constante que indica que un cálculo fue ejecutado.

- *CALCULATION_RUN_NOT (false):* Constante que indica que un cálculo no fue ejecutado.

- *CALCULATION_SUCCEED (true):* Constante que indica que un cálculo fue realizado exitosamente.

- *CALCULATION_SUCCEED_NOT (false):* Constante que indica que un cálculo no fue realizado exitosamente.

- *INPUT_FIXED (true):* Constante que indica que el valor de salida tiene que ser el mismo de entrada.

- *INPUT_FIXED_NOT (false):* Constante que indica que el valor de salida puede ser diferente al de entrada.

- *VALUE_KNOWN (true):* Constante que indica que fue dado un valor valido en la entrada y/o un valor valido en el cálculo.

- *VALUE_KNOWN_NOT (false):* Constante que indica que no fue dado un valor valido en la entrada y/o un valor no valido en el cálculo.

- *VALUE_FIRST (true):* Constante que indica que fue encontrado el primer valor calculado.

- *VALUE_FIRST_NOT (false):* Constante que indica que no fue encontrado el primer valor calculado.

De tipo variable:

- *CalculationOnceMore:* Bandera que indica si es necesario realizar una vez más todos los cálculos o no.
  Si es necesario, el estado de la bandera será: CALCULATION_ONCEMORE.
  Si no es necesario, el estado de la bandera será: CALCULATION_ONCEMORE_NOT.
  El estado por defecto de esta bandera es: CALCULATION_ONCEMORE.

- *calculationSucceed:* Bandera que indica si un cálculo fue realizado exitosamente o si por el contrario fallo.
  Si el cálculo fue realizado exitosamente, el estado de la bandera será: CALCULATION_SUCCEED.
  Si el cálculo fallo, el estado de la bandera será: CALCULATION_SUCCEED_NOT.
  El estado por defecto de esta bandera es: CALCULATION_SUCCEED.

- *calculation_01:* Bandera que indica si la primera ecuación fue ejecutada o no.
  Si esta fue ejecutada, el estado de la bandera será CALCULATION_RUN.
  Si esta no fue ejecutada, el estado de la bandera será CALCULATION_RUN_NOT.
  El estado por defecto de esta bandera es: CALCULATION_RUN_NOT.

- *calculation_02:* Bandera que indica si la segunda ecuación fue ejecutada o no.
  Si esta fue ejecutada, el estado de la bandera será CALCULATION_RUN.
  Si esta no fue ejecutada, el estado de la bandera será CALCULATION_RUN_NOT.
  El estado por defecto de esta bandera es: CALCULATION_RUN_NOT.

- *calculation_03:* Bandera que indica si la tercera ecuación fue ejecutada o no.
  Si esta fue ejecutada, el estado de la bandera será CALCULATION_RUN.
  Si esta no fue ejecutada, el estado de la bandera será CALCULATION_RUN_NOT.
  El estado por defecto de esta bandera es: CALCULATION_RUN_NOT.

- *inputFixed:* Bandera que indica si el valor de entrada está bloqueado (el valor de salida tiene que ser el mismo del de entrada) o no.
  Si el valor está bloqueado, el estado de la bandera será: INPUT_FIXED.
  Si el valor no está bloqueado, el estado de la bandera será: INPUT_FIXED_NOT.
  El estado por defecto de esta bandera es: INPUT_FIXED_NOT.

- *valueFirst:* Bandera que indica si el primer valor calculado fue hallado o no.
  Si este fue hallado, el estado de la bandera es: VALUE_FIRST.
  Si este no fue hallado, el estado de la bandera es: VALUE_FIRST_NOT.
  El estado por defecto de esta bandera es: VALUE_FIRST.

- *valueKnown:* Bandera que indica si un parámetro tiene almacenado un numero valido o no.
  Si este es válido, el estado de la bandera es: *VALUE_KNOWN.*
  Si este no es válido, el estado de la bandera es: *VALUE_KNOWN_NOT.*
  El estado por defecto de esta bandera es: *VALUE_KNOWN_NOT.*

Banderas definidas en las súper clases “VariableRealSuper” y “VariableIntSuper” son (teniendo en cuenta que las siguientes banderas están definidas en las dos súper clases, variando únicamente el tipo de valor que almacenan, en la súper clase VariableRealSuper las banderas almacenan valores reales, y en VariableIntSuper enteros):

- *value:* Bandera que almacena el valor calculado.
  Para la súper clase VariableRealSuper, El valor por defecto es Const.NonsD.
  Para la súper clase VariableIntSuper, El valor por defecto es Const.NonsI.

- *valueInput:* Bandera que almacena el valor de entrada.
  Para la súper clase VariableRealSuper, El valor por defecto es Const.NonsD.
  Para la súper clase VariableIntSuper, El valor por defecto es Const.NonsI.

- *valueLimitMax:* Bandera que almacena el valor del límite máximo permitido.
  Para la súper clase VariableRealSuper, El valor por defecto es Const.NonsD.
  Para la súper clase VariableIntSuper, El valor por defecto es Const.NonsI.

- *valueLimitMin:* Bandera que almacena el valor del límite mínimo permitido.
  Para la súper clase VariableRealSuper, El valor por defecto es Const.NonsD.
  Para la súper clase VariableIntSuper, El valor por defecto es Const.NonsI.

- *valueMin:* Bandera que almacena el mínimo valor calculado.
  Para la súper clase VariableRealSuper, El valor por defecto es Const.NonsD.
  Para la súper clase VariableIntSuper, El valor por defecto es Const.NonsI.

- *valueMax:* Bandera que almacena el valor calculado.
  Para la súper clase VariableRealSuper, El valor por defecto es - Const.NonsD.
  Para la súper clase VariableIntSuper, El valor por defecto es - Const.NonsI.

- *firstValue:* Bandera que almacena el valor del primer cálculo.
  Para la súper clase VariableRealSuper, El valor por defecto es Const.NonsD.
  Para la súper clase VariableIntSuper, El valor por defecto es Const.NonsI.

4.3 DEFINICIÓN DE LOS CONSTRUCTORES Y MÉTODOS QUE SE USARAN EN LAS SÚPER CLASES.

De acuerdo a los requerimientos del Instituto de Metrología Alemán en la súper clase “VariableGeneralSuper”, no se definirán ningún constructor, ni método; en la súper clase “VariableRealSuper” se definieron dos constructores, uno para inicializar ciertas variables cuando el valor de entrada no es conocido, y el otro constructor para cuando el valor de entrada es conocido.

El primer constructor (cuando el valor de entrada no es conocido) seguirá las siguientes instrucciones:

Importar el valor mínimo (min) y máximo (max) que puedo tener la variable.

- *valueLimitMin* = min.
- *valueLimitMax* = max.
- *valueKnown* = VALUE_KNOWN_NOT.
- *inputFixed* = INPUT_FIXED_NOT.
- *valueInput* = Const.NonsD.
- *value* = Const.NonsD.

El segundo constructor (cuando el valor de entrada es conocido) seguirá las siguientes instrucciones:

- Importar *valueInput, inputFixed,* min y max.
- *valueLimitMin* = min.
- *valueLimitMax* = max.
- Exportar la variable *valueInput* e ir al método setValueInput.

Los métodos que fueron definidos en la súper clase “VariableRealSuper” son los siguientes:

- getValueMin: el cual retorna el valor inmediato de *valueMin.*

- getValueMax: el cual retorna el valor inmediato de *valueMax.*

- calMinMax: en el cual se establece el máximo y mínimo valor calculado en las banderas *valueMax* y *valueMin* respectivamente.

- checkLimits: método donde se comprueba que tanto los valores de entrada como los calculado se encuentran dentro de los límites establecidos de ser así se seguirá la siguiente instrucción:
  valueKnown = VALUE_KNOWN.
  De no estar el valor dentro de los límites:
  valueKnown = VALUE_KNOWN_NOT
  calculationSucceed = CALCULATION_SUCCEED_NOT.
  Este método retorna el valor de la bandera *valueKnown.*

- setValueInput: método en el que se establece el valor de entrada en la bandera *valueInput*, lo envía al método setValue y realiza la siguiente instrucción:
  valueMin = Double.MAX_VALUE
  valueMax = -100

- setValue: método en el que se establece el valor calculado (valueIn) en la bandera *value*, primero comprueba que dicho valor este dentro de los límites establecidos pasando el valor calculado por el método checkLimits, de estar dentro de los limites se sigue la siguiente instrucción:
  value = valueIn
  calMinMax (va a este método)
  fixed (va a este método).
  Si valueIn no está dentro de los límites:
  Value = Const.NonsD
  valueMin = *value*
  valueMax = *value*

- fixed: método en el que se comprueba el estado de la bandera *inputFixed*, primero comprueba la siguiente instrucción:
  si *inputFixed* = INPUT_FIXED entonces:
  value = *valueInput*
  calMinMax (va a este método),
  Según requerimientos del instituto de metrología por ahora el valor final que será mostrado será el primer valor calculado; en este método se comprobara que fue hallado el primer valor calculado y almacenado de la siguiente forma:

Si *valueFirst* = VALUE_FIRST entonces:
value = *firstValue.*
Y son la siguiente instrucción se estará constantemente refrescando la variable *value* con el primer valor calculado:
Si *valueFirst* = VALUE_FIRST_NOT entonces:
firstValue = *value*

- getValue: el cual retorna el valor inmediato de *value.*

- getValueInput: el cual retorna el valor de *valueInput.*

- check: el cual establece los valores de salida como el mismo de entrada, para el caso en el que el valor de entrada se haya dado, sea un numero que se encuentra dentro de los límites establecidos y no se haya podido calcular dicho parámetro.

4.4 DESARROLLO DE LAS TRES PRIMERAS CLASES DONDE ESTÁN DEFINIDAS INDIVIDUALMENTE m_n, m_t Y β.

Después de haber sido creadas las dos primeras súper clases se procedió a crear tres clases las cuales se extenderán de la súper clase “VariableRealSuper”, dichas clases serán llamadas “Beta0”, “ModuleNormal” y “ModuleTransverse”, donde serán definidos los métodos para el cálculo de las variables beta, Módulo normal y Módulo transversal respectivamente.

Estas tres primeras clases servirán para desarrollar una clase de formato para el posterior desarrollo de las clases de los parámetros de los engranajes que faltan; la principal función de estas clases es contener las ecuaciones que definan su parámetro.

Para el cálculo de cada parámetro se diseño el Gráfico 3, en el cual se expone el resultado que se obtendrá de acuerdo a las posibilidades, donde “A” simboliza la variable que se desea calcular y “B” las variables necesarias para calcular “A”.

El primer paso que se seguirá para el cálculo de cualquier variable será la evaluación de la bandera *calcSucceed*, ya que de haber sido entrado un valor que

se encuentra fuera de los límites establecidos hará que no se ejecute ningún cálculo; este procedimiento se pensó entorno a que si el software ejecutaba los respectivos cálculos y daba estos resultados, es posible que el usuario entienda que los valores que él entro eran consentidos por el software.

Después de evaluar la bandera *calcSucceed* se procederá a reconocer el estado de la bandera *valueKnown* para la entrada de “A”, seguidamente se evaluara el estado de esta bandera para la entrada de “B”, en caso de que en estas dos su estado sea verdadero, se evaluara el estado de la bandera *inputFixed* para “A”.

Posteriormente al desarrollo del Gráfico 3 se desarrollaron las tablas 4 y 5 donde se presentan los posibles casos de acuerdo a las diferentes posibilidades de entradas, donde:

- Como en el esquema 1 “a” simboliza la variable que se desea calcular.
- Al igual que en el esquema 1 “b…” simboliza las variables que se usaran para calcular “a”.
- “3” simboliza el valor numérico de entrada de “a”.
- “10” simboliza los valores numéricos para calcular “a”.
- “6” simboliza el valor calculado de “a”.
- “-” simboliza que no fue dado el valor numérico.
- “NonSens” simboliza que es un valor que se encuentra fuera de los limites numéricos establecidos en cada clase según la variable.

Cada clase está constituida por:

- Definición de los valores de los límites numéricos de cada variable.
- Dos constructores para los casos en los que el valor de entrada sea dado y no sea dado, estos constructores se remiten a los constructores de la súper clase “VariableRealSuper”.
- Un método llamado “Calculate”, en el cual se evalúa si se puede calcular la variable deseada con respecto a las banderas (ver tabla 4 y 5).

- Un método cuyo nombre es el mismo del de su clase seguido por el numero “01”, en el cual se encuentra la ecuación para calcular la variable.

4.5 DESARROLLO DE UNA CLASE PARA PROBAR EL FUNCIONAMIENTO DEL SOFTWARE.

Inmediatamente creadas las clases: Beta0, ModuleNormal y ModuleTransverse, fue desarrollada la clase “InvCalculatorMain”, la cual contendrá las siguientes instrucciones para probar que el software estaba funcionando de acuerdo a las condiciones ya anteriormente dichas:

- Se crean los objetos beta0, mn, mt.
- Se dan los valores de entrada de los parámetros.
- Se definió una estructura de tipo “while” donde se encuentras las instrucciones para que sean calculados los parámetros, en la que se corrobora que no hubo ningún problema al momento de realizar los cálculos.
- Están las instrucciones para mostrar los valores de salida.

Gráfico 3. Comportamiento del software

CalcSucceed A=true &
CalcSucceed B=true

- TRUE
  - InputKnown A
    - TRUE
      - InputKnown B
        - TRUE
          - InputFixed A
            - TRUE
              - Calcula A
              - Value = ValueIn
              - ValueMin = Changed
              - ValueMax = Changed
            - FALSE
              - Calcula A
              - Value = Value A
              - ValueMin = Changed
              - ValueMax = Changed
              - InputKnown = True
              - CalcSucceed A = True
        - FALSE
          - Value = ValueIn
          - ValueMin = ValueIn
          - ValueMax = ValueIn
    - FALSE
      - InputKnown B
        - TRUE
          - Calcula A
          - Value = ValueCalc
          - ValueMin = Changed
          - ValueMax = Changed
          - InputKnown = True
          - CalcSucceed A = True
        - FALSE
          - Value = NonSensD
          - ValueMin = NonSensD
          - ValueMax = NonSensD
- FALSE
  - Value = NonSensD
  - ValueMin = NonSensD
  - ValueMax = NonSensD
  - InputKnown = False
  - CalcSucceed A = False

Tabla 4. Estado de las banderas una vez inicializado el software.

			BANDERAS					
Caso	**Variables**	**Valores**	**InputFixed a**	**CalcSucceed a**	**ValueKnown a**	**Value**	**ValueMin**	**ValueMax**
1	a	3	True	True	True	3	3	3
	b....	-						
2	a	3	False	True	True	3	3	3
	b....	-						
3	a	3	True/False	True	True	3	3	3
	b....	NonSens						
4	a	3	True	True	True	3	3	3
	b....	10						
5	a	3	False	True	True	3	3	3
	b....	10						
6	a	-	True/False	True	False	NonSens	NonSens	NonSens
	b....	10						
7	a	-	True/False	True	False	NonSens	NonSens	NonSens
	b....	NonSens						
8	a	NonSens	True/False	False	False	NonSens	NonSens	NonSens
	b....	-						
9	a	NonSens	True/False	False	False	NonSens	NonSens	NonSens
	b....	NonSens						

Tabla 5. Estado de las banderas al momento de calcular "a".

			BANDERAS							
Caso	**Variables**	**Valores**	**InputFixed a**	**ValueKnown a**	**ValueKnown b**	**CalcSucceed a**	**CalcSucceed b**	**Value**	**ValueMin**	**ValueMax**
1	a	3	True	True	False	True	True	3	3	3
	b....	-								
2	a	3	False	True	False	True	True	3	3	3
	b....	-								
3	a	3	True/False	False	False	False	False	NonSens	NonSens	NonSens
	b....	NonSens								
4	a	3	True	True	True	True	True	3	3	6
	b....	10								
5	a	3	False	True	True	True	True	6	6	6
	b....	10								
6	a	-	False	True	True	True	True	6	6	6
	b....	10								
7	a	-	True/False	False	False	False	False	NonSens	NonSens	NonSens
	b....	NonSens								
8	a	NonSens	True/False	False	False	False	True	NonSens	NonSens	NonSens
	b....	-								
9	a	NonSens	True/False	False	False	False	False	NonSens	NonSens	NonSens
	b....	NonSens								

4.6 DESCRIPCIÓN DEL FUNCIONAMIENTO HASTA EL MOMENTO.

- Cada parámetro tiene un cierto número de ecuaciones en su clase, y por lo tanto este es capaz de hallar diferentes valores para el mismo parámetro.
- Cada parámetro tiene sus propios límites para el valor de entrada (límite máximo y mínimo), si el valor de entrada no se encuentra dentro de estos límites, la salida (*value*), *valueMin* y *valueMax* serán establecidos con el valor Const.NonSensD o Const.NonSensI.
- El orden de cálculo de los parámetros son:

  1. Módulo Normal.
  2. Módulo Transversal.
  3. Beta.

  Y el grado de importancia es:

  1. Beta.
  2. Módulo Transversal.
  3. Módulo Normal.

  Resumiendo, el grado de importancia es inversamente proporcional al orden de los cálculos.
- Si el estado de un parámetro esta como bloqueado (FIXED), la salida de este parámetro es el mismo valor de entrada, y también este valor de entrada es usado para ser establecido en las banderas *valueMin* y *valueMax.*
- Si el estado de un parámetro esta como no bloqueado (FIXED_NOT), y el cálculo de este parámetro es posible (hay los parámetros necesarios para este fin) y están dentro de los límites establecidos, en las banderas *value, valueMin* y *valueMax* serán almacenados el valor o los valores resultantes de los cálculos, de otro modo se almacenara el mismo valor del de la entrada.

- Si un cálculo falla o el valor de entrada no es válido (esta fuera de los limites), en las banderas *value, valueMin* y *valueMax* se almacenarán el valor de Const.NonsD o Const.NonsI, y los parámetros que tengan este parámetro dentro de alguna de sus ecuaciones, tendrán el mismo valor (Const.NonsD o Const.NonsI) en sus banderas *value, valueMin* y *valueMax.*
- La salida de un parámetro es igual al resultado de la primera ecuación que se haya realizado, y este valor es también usado en todas las ecuaciones que contengan este parámetro como una de sus variables.
- *valueMin* y *valueMax* almacenarán el resultado numérico más pequeño y el más grande de las ecuaciones respectivamente.

4.7 DEFINICIÓN DE LOS PARÁMETROS QUE SE DESEAN CALCULAR CON SUS RESPECTIVAS ECUACIONES.

Según requerimientos del Instituto de Metrologuia Alemán el software calculara los siguientes parámetros:

- Números de dientes, z.
- Diámetro de referencia, d.
- Ángulo de hélice, β.
- Ángulo de presión, α_t.
- Diámetro base, d_b.
- Ángulo de presión normal, α_n.
- Módulo básico, m_b.
- Módulo normal, m_n.
- Módulo transversal, m_t.
- Módulo axial, m_x.
- Ángulo de avance del cilindro de referencia, γ.

Con los estándares ISO 21771 y DIN 3960 se procedió a realizar diferentes tablas en las que se catalogaron las ecuaciones con las que se pueden calcular cada uno

de los parámetros que se desea que aparezcan en la actual aplicación de software, de manera que sea de fácil desarrollo las clases en las que se definirán los parámetros faltantes hasta este momento, partiendo de los clases ya desarrolladas. Estas tablas significan una gran ayuda tanto para la respectiva documentación de este proyecto como para la verificación en el código del software de los conjuntos de ecuaciones y sus requerimientos para ser calculadas (como se explico en el numeral correspondiente a las limitaciones de este proyecto, este software solo calculará siete parámetros, por lo que no serán usadas todas las ecuaciones que posteriormente se mostrarán, se especifica que las ecuaciones que no serán usadas en esta propuesta servirán para la continuación y finalización de este proyecto en el Instituto de Metrologia Alemán).

Tabla 6. Ecuaciones para calcular el Módulo normal, m_n.

z	d	β	α_t	d_b	α_n	m_b	m_x	γ	m_t	Fórmula
X	X	X								$\frac{d * \cos\beta}{z}$
		X							X	$m_t * \cos\beta$
X		X	X	X						$\frac{d_b * \cos\beta}{z * \cos\alpha_t}$
X		X		X	X					$\frac{d_b\sqrt{\tan^2\alpha_n + \cos^2\beta}}{z}$
		X			X	X				$m_b * \sqrt{\tan^2\alpha_n + \cos^2\beta}$
							X	X		$m_x * \cos\gamma$
		X						X		$m_x * sen\beta$
X		X	X	X						$\frac{d_b * \cos\beta\sqrt{\tan^2\alpha_t + 1}}{z}$
		X	X			X				$\frac{m_b * \cos\beta}{\cos\alpha_t}$
		X						X	X	$\frac{\cos\gamma\tan\beta}{m_t}$

Fuente: ISO 21771, DIN 3960

Tabla 7. Ecuaciones para calcular el Módulo transversal, m_t.

z	d	m_n	β	α_t	d_b	α_n	m_b	m_x	γ	Fórmula
X	X									$\frac{d}{z}$
		X	X							$\frac{m_n}{\cos\beta}$
X				X	X					$\frac{d_b}{z*\cos\beta}$
		X	X	X		X				$\frac{m_n}{\cos\alpha_t\sqrt{\tan^2\alpha_n+\cos^2\beta}}$
				X			X			$\frac{m_b}{\cos\alpha_t}$
			X					X		$m_x*\tan\beta$
		X	X						X	$\frac{m_n*\tan\beta}{\cos\gamma}$
X				X	X					$\frac{d_b\sqrt{\tan^2\alpha_t+1}}{z}$

Fuente: Ibid

Tabla 8. Ecuaciones para calcular el Ángulo de hélice, β (Beta).

z	d	m_n	α_t	d_b	α_n	m_b	m_x	γ	m_t	Fórmula
		X							X	$\arccos\left(\frac{m_n}{m_t}\right)$
			X		X					$\arccos\left(\frac{\tan\alpha_n}{\tan\alpha_t}\right)$
X	X	X								$\arccos\left(\frac{z * m_n}{d}\right)$
								X		$90° - \gamma$
		X			X	X				$\arccos\left(\frac{m_n^2}{m_b^2} - \tan^2 \alpha_n\right)$
		X	X			X				$\arccos\left(\frac{m_n * \cos\alpha_t}{m_b}\right)$
		X						X	X	$\arctan\left(\frac{m_t * \cos\gamma}{m_n}\right)$

Fuente: ISO 21771

Tabla 9. Ecuaciones para calcular el Diámetro de referencia, d .

z	m_n	β	α_t	d_b	α_n	m_b	m_x	γ	m_t	Fórmula
X										$d * m_t$
X	X	X								$\frac{z * m_n}{\cos\beta}$
			X	X						$\frac{d_b}{\cos\alpha_t}$
X	X	X	X		X					$\frac{z * m_n}{\cos\alpha_t \sqrt{\tan^2\alpha_n + \cos^2\beta}}$
X			X			X				$\frac{m_b * z}{\cos\alpha_t}$

Fuente: ISO 21771, DIN 3960

Tabla 10. Ecuaciones para calcular el Números de dientes, z .

d	m_n	β	α_t	d_b	α_n	m_b	m_x	γ	m_t	Fórmula
X									X	$\frac{d}{m_t}$
X	X	X								$\frac{d * \cos\beta}{m_n}$
			X	X					X	$\frac{d_b}{m_t * \cos\alpha_t}$
	X	X		X	X					$\frac{d_b \sqrt{\tan^2\alpha_n + \cos^2\beta}}{m_n}$
			X	X					X	$\frac{d_b \sqrt{\tan^2\alpha_t + 1}}{m_t}$
				X		X				$\frac{d_b}{m_b}$

Fuente: ISO 21771

Tabla 11. Ecuaciones para calcular el Ángulo de avance del cilindro de referencia, γ .

z	d	m_n	β	α_t	d_b	α_n	m_b	m_x	m_t	Fórmula
			X							$90° - \beta$
		X						X		$ar\cos\left(\frac{m_n}{m_x}\right)$
		X	X						X	$ar\cos\left(\frac{m_n \tan\beta}{m_t}\right)$

Fuente: ISO 21771, DIN 3960

Tabla 12. Ecuaciones para calcular Módulo axial, m_x .

z	d	m_n	β	α_t	d_b	α_n	m_b	γ	m_t	Fórmula
		X	X							$\frac{m_n}{sen\beta}$
		X						X		$\frac{m_n}{\cos\gamma}$
			X						X	$\frac{m_t}{\tan\beta}$

Fuente: DIN 3960

Tabla 13. Ecuaciones para calcular el Módulo básico, m_b.

z	d	m_n	β	α_t	d_b	α_n	m_x	γ	m_t	Fórmula
		X	X			X				$\frac{m_n}{\sqrt{\tan^2 \alpha_n + \cos^2 \beta}}$
X					X					$\frac{d_b}{z}$
		X	X	X						$\frac{m_n * \cos \alpha_t}{\cos \beta}$
				X					X	$m_t * \cos \alpha_t$
X	X			X						$\frac{d * \cos \alpha_t}{z}$

Fuente: ISO 21771, DIN 3960

Tabla 14. Ecuaciones para calcular el Ángulo de presión normal, α_n.

z	d	m_n	β	α_t	d_b	m_b	m_x	γ	m_t	Fórmula
			X	X						$\arctan(\cos \beta \tan \alpha_t)$
X		X	X		X					$\arctan\left(\sqrt{\left(\frac{z * m_n}{d_b}\right)^2 - \cos^2 \beta}\right)$
		X	X			X				$\arctan\left(\sqrt{\frac{m_n^2}{m_b^2} - \cos^2 \beta}\right)$
		X		X					X	$\arctan\left(\frac{m_n}{m_t} \tan \alpha_t\right)$

Fuente: ISO 21771

Tabla 15. Ecuaciones para calcular el Ángulo de presión, α_t .

z	d	m_n	β	d_b	α_n	m_b	m_x	γ	m_t	Fórmula
			X		X					$arcsen\left(\frac{\tan\alpha_n}{\sqrt{\tan^2\alpha_n+\cos^2\beta}}\right)$
			X		X					$ar\cos\left(\frac{\cos\beta}{\sqrt{\tan^2\alpha_n+\cos^2\beta}}\right)$
			X		X					$\arctan\left(\frac{\tan\alpha_n}{\cos\beta}\right)$
	X			X						$ar\cos\left(\frac{d_b}{d}\right)$
X				X					X	$ar\cos\left(\frac{d_b}{z*m_t}\right)$
X		X	X	X						$ar\cos\left(\frac{d_b*\cos\beta}{z*m_n}\right)$
		X	X			X				$ar\cos\left(\frac{m_b*\cos\beta}{m_n}\right)$
						X			X	$ar\cos\left(\frac{m_b}{m_t}\right)$
X	X					X				$ar\cos\left(\frac{m_b*z}{d}\right)$

Fuente: Ibid

Tabla 16. Ecuaciones para calcular el Diámetro base, d_b.

z	d	m_n	β	α_t	α_n	m_b	m_x	γ	m_t	Fórmula
	X			X						$d * \cos \alpha_t$
X				X					X	$z * m_t * \cos \alpha_t$
X		X	X	X						$\frac{z * m_n * \cos \alpha_t}{\cos \beta}$
X				X					X	$\frac{m_t * z}{\sqrt{\tan^2 \alpha_t + 1}}$
X						X				$z * m_b$

Fuente: Ibid

4.8 IMPLEMENTACIÓN DE LAS CLASES DONDE SE DEFINIRÁN INDIVIDUALMENTE d, z, γ Y m_x.

Estas clases fueron llamadas “DiameterReference”, “TeethNumber”, “gamma” y “ModuleAxial” respectivamente, las cuales fueron basadas en las clases: Beta0, ModuleNomal y ModuleTransverse.

La clase que define el número de dientes (TeethNumber) es la única clase en la que se definieron todas sus variables de tipo entero, y por esto fue desarrollada la súper clase llamada “VariableIntSuper”, la cual es sumamente parecida a “VaribleRealSuper”, su única diferencia es que en la primera todas las variables están definidas de tipo entero, como se explicó anteriormente; la clase “TeethNumber” es la única que se extiende de “VariableIntSuper”.

4.9 CONTENIDO FINAL DE TODAS LAS CLASES.

a) InvCalculatorMain: Un solo método donde:

- Se crean los objetos beta0, mn, mt, dr, z, gamma y mx.
- Se dan los valores de entrada de los parámetros.
- Se definió una estructura de tipo “while” donde se encuentras las instrucciones para que sean calculados los parámetros, en la que se corrobora que no hubo ningún problema al momento de realizar los cálculos.
- Están las instrucciones para mostrar los valores de salida.

b) VariableGeneralSuper: Esta clase tiene los elementos que se explicaron en numerales anteriores.

c) VariableRealSuper: Esta clase tiene los elementos que se explicaron en numerales anteriores, y se le agrego un método llamado “round” el cual servirá para tomar los valores de salida y aproximarlos de la siguiente manera: solo se mostrara dos decimales, donde el segundo decimal se aproximara de acuerdo al valor del tercer decimal.

d) VariableIntSuper: Esta clase tiene los mismos elementos que la super clase VariableRealSuper, la única diferencia como se explicó en numerales anteriores es que en esta clase las variables están definidas como variables numéricas de tipo entero.

e) Beta0, ModuleNormal, ModuleTransverse, DiameterReference, TeethNumber, gamma, ModuleAxial: Estas clases tienen los elementos que se explicaron en numerales anteriores y de los cuales se dará una breve explicación.

- Dos constructores los que se remitirán a dos constructores de la súper clase de la que se extienden. La función de estos constructores es inicializar los valores dependiendo de los valores que se tienen en la entrada.

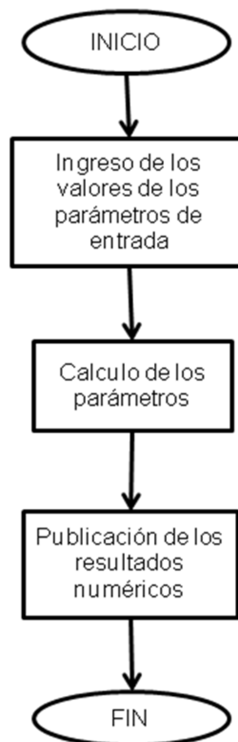
- Un método llamado "calculate" como se explico en el numeral 4.4.

- Varios métodos los cuales tendrán como nombre, el mismo nombre de la clase seguido por un numero el cual obedecerá a conteo ascendente; en estos métodos están las ecuaciones para poder calcular el parámetro que es definido en la clase.

4.10 COMPORTAMIENTO DEL SOFTWARE.

El comportamiento que sigue el software está definido en el siguiente diagrama de flujo:

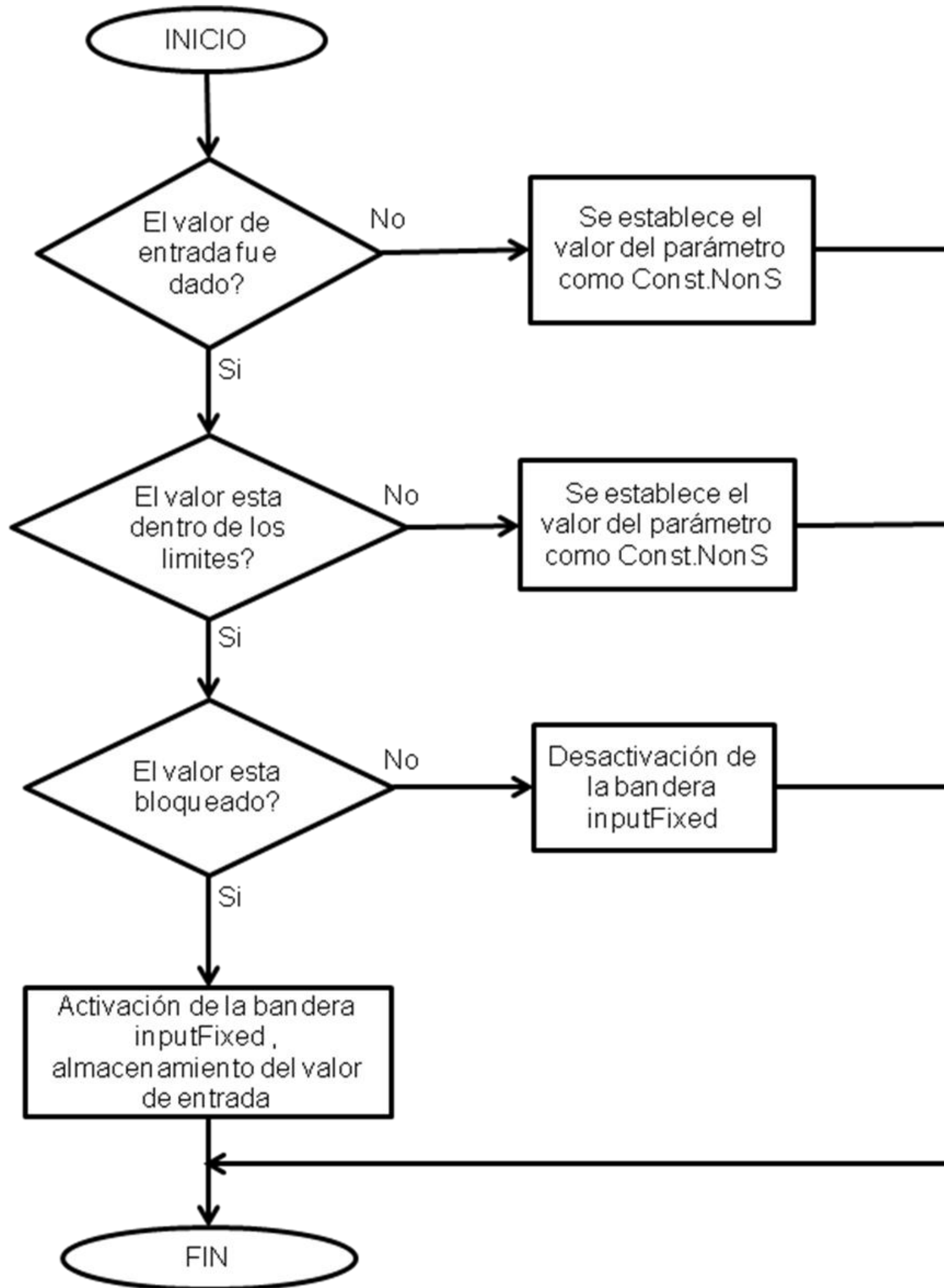
Gráfico 4. Diagrama de flujo del funcionamiento del software.



Por la complejidad del software se definirán posteriormente diagramas de flujo donde se podrá detallar el procedimiento que sigue el software.

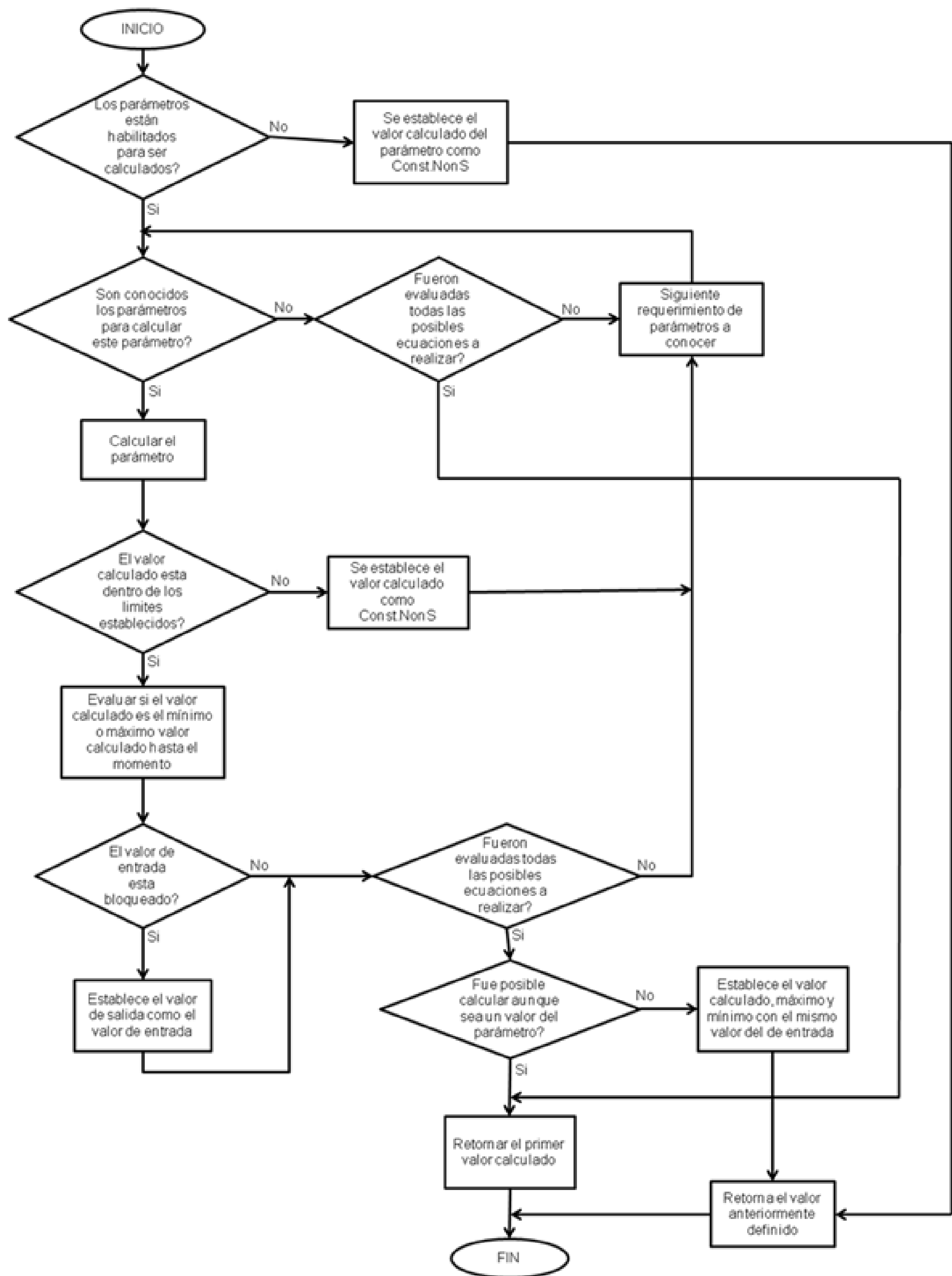
En el siguiente diagrama de flujo se encuentra definido el proceso que sigue el software en el momento de ingresar valores de entrada.

Gráfico 5. Diagrama de flujo del ingreso de valores.



En el siguiente diagrama de flujo se encuentra definido el proceso que sigue el software en al momento de realizar los cálculos de parámetros.

Gráfico 6. Diagrama de flujo del cálculo de valores.



5 PRESENTACIÓN DE RESULTADOS

a) Cuando los valores de entrada están dentro de los límites establecidos y no hay ningún valor bloqueado, el software procederá a calcular todos los valores posibles para cada uno de los parámetros, almacenando el valor máximo y mínimo, y comportándose de acuerdo a las reglas establecidas.

Valores de entrada:

```
mn = new ModuleNormal(1, VariableGeneralSuper.INPUT_FIXED_NOT);
mt = new ModuleTransverse(2, VariableGeneralSuper.INPUT_FIXED_NOT);
beta0 = new Beta0(Math.toRadians(10),
VariableGeneralSuper.INPUT_FIXED_NOT);
dr = new DiameterReference(30,VariableGeneralSuper.INPUT_FIXED_NOT);
z = new TeethNumber(9,VariableGeneralSuper.INPUT_FIXED_NOT);
gamma = new
gamma(Math.toRadians(70),VariableGeneralSuper.INPUT_FIXED_NOT);
mx = new ModuleAxial(5,VariableGeneralSuper.INPUT_FIXED_NOT);
```

Valores de salida:

```
mnIn:  1.0
mn:    1.97
mnMin: 0.87
mnMax: 3.88


mtIn:  2.0
mt:    2.0
mtMin: 0.88
mtMax: 3.33


beta0In: 10.0
beta0: 10.0
beta0 Min: 10.0
beta0 Max: 53.78


drIn:  30.0
dr:    18.0
drMin: 18.0
drMax: 30.0


zIn:  9
z:    9
zMin: 9
zMax: 9
```

```
gammaIn:  70.0
gamma:    80.0
gammaMin: 66.8
gammaMax: 80.0


mxIn:  5.0
mx:    11.34
mxMin: 5.0
mxMax: 11.34
```

b) Con los mismos valores del punto anterior, cambiando que el primer valor está bloqueado, observándose que el valor de salida del primer valor es el mismo del de entrada y modificando el segundo valor.

Valores de entrada:

```
mn = new ModuleNormal(1, VariableGeneralSuper. INPUT_FIXED);
mt = new ModuleTransverse(2, VariableGeneralSuper.INPUT_FIXED_NOT);
beta0 = new Beta0(Math.toRadians(10),
VariableGeneralSuper.INPUT_FIXED_NOT);
dr = new DiameterReference(30,VariableGeneralSuper.INPUT_FIXED_NOT);
z = new TeethNumber(9,VariableGeneralSuper.INPUT_FIXED_NOT);
gamma = new
gamma(Math.toRadians(70),VariableGeneralSuper.INPUT_FIXED_NOT);
mx = new ModuleAxial(5,VariableGeneralSuper.INPUT_FIXED_NOT);
```

Valores de salida:

```
mnIn:  1.0
mn:    1.0
mnMin: 0.87
mnMax: 3.88


mtIn:  2.0
mt:    1.01
mtMin: 0.51
mtMax: 3.33


beta0In: 10.0
beta0: 10.0
beta0 Min: 10.0
beta0 Max: 72.54


drIn:  30.0
dr:    9.14
```

```
drMin: 9.14
drMax: 30.0


zIn:  9
z:    9
zMin: 9
zMax: 9


gammaIn:  70.0
gamma:    80.0
gammaMin: 70.0
gammaMax: 80.0


mxIn:  5.0
mx:    5.76
mxMin: 5.0
mxMax: 5.76
```

c) Cuando uno de los valores de entrada esta fuera de los límites establecidos, de tal forma que el software no realizará ningún cálculo, con el fin de que el usuario comprenda que ha ingresado un valor que se encuentra fuera de los límites establecidos.

Valores de entrada

```
mn = new ModuleNormal(1000000, VariableGeneralSuper.INPUT_FIXED_NOT);
mt = new ModuleTransverse(2, VariableGeneralSuper.INPUT_FIXED_NOT);
beta0 = new Beta0(Math.toRadians(10),
VariableGeneralSuper.INPUT_FIXED_NOT);
dr = new DiameterReference(30,VariableGeneralSuper.INPUT_FIXED_NOT);
z = new TeethNumber(9,VariableGeneralSuper.INPUT_FIXED_NOT);
gamma = new
gamma(Math.toRadians(70),VariableGeneralSuper.INPUT_FIXED_NOT);
mx = new ModuleAxial(5,VariableGeneralSuper.INPUT_FIXED_NOT);
```

Valores de salida

```
mnIn:  1000000.0
mn:    9.9999999999E9
mnMin: 9.9999999999E9
mnMax: 9.9999999999E9


mtIn:  2.0
mt:    9.9999999999E9
mtMin: 9.9999999999E9
mtMax: 9.9999999999E9
```

```
beta0In: 10.0
beta0: 9.9999999999E9
beta0 Min: 9.9999999999E9
beta0 Max: 9.9999999999E9


drIn:  30.0
dr:    9.9999999999E9
drMin: 9.9999999999E9
drMax: 9.9999999999E9


zIn:  9
z:    999999999
zMin: 999999999
zMax: 999999999


gammaIn:  70.0
gamma:    9.9999999999E9
gammaMin: 9.9999999999E9
gammaMax: 9.9999999999E9


mxIn:  5.0
mx:    9.9999999999E9
mxMin: 9.9999999999E9
mxMax: 9.9999999999E9
```

d) Cuando el valor que se encuentra fuera de los límites está bloqueado, con lo que se puede observar que sin importa si el valor que se encuentra fuera de los límites establecidos esta bloqueado o no, el software no deja que sea calculada ninguna ecuación.

Valores de entrada

```
mn = new ModuleNormal(1000000, VariableGeneralSuper.INPUT_FIXED);
mt = new ModuleTransverse(2, VariableGeneralSuper.INPUT_FIXED_NOT);
beta0 = new Beta0(Math.toRadians(10),
VariableGeneralSuper.INPUT_FIXED_NOT);
dr = new DiameterReference(30,VariableGeneralSuper.INPUT_FIXED_NOT);
z = new TeethNumber(9,VariableGeneralSuper.INPUT_FIXED_NOT);
gamma = new
gamma(Math.toRadians(70),VariableGeneralSuper.INPUT_FIXED_NOT);
mx = new ModuleAxial(5,VariableGeneralSuper.INPUT_FIXED_NOT);
```

Valores de salida

```
mnIn:  1000000.0
mn:    9.9999999999E9
mnMin: 9.9999999999E9
mnMax: 9.9999999999E9


mtIn:  2.0
mt:    9.9999999999E9
mtMin: 9.9999999999E9
mtMax: 9.9999999999E9


beta0In: 10.0
beta0: 9.9999999999E9
beta0 Min: 9.9999999999E9
beta0 Max: 9.9999999999E9


drIn:  30.0
dr:    9.9999999999E9
drMin: 9.9999999999E9
drMax: 9.9999999999E9


zIn:  9
z:    999999999
zMin: 999999999
zMax: 999999999


gammaIn:  70.0
gamma:    9.9999999999E9
gammaMin: 9.9999999999E9
gammaMax: 9.9999999999E9


mxIn:  5.0
mx:    9.9999999999E9
mxMin: 9.9999999999E9
mxMax: 9.9999999999E9
```

e) Se rectificarán que los resultados dados en el literal a, están correctos, con lo que se puede corroborar ya que tanto el valor mínimo y máximo calculado y el valor calculado son iguales al de entrada.

Valores de entrada

```
mn = new ModuleNormal(1.97, VariableGeneralSuper.INPUT_FIXED_NOT);
mt = new ModuleTransverse(2, VariableGeneralSuper.INPUT_FIXED_NOT);
beta0 = new Beta0(Math.toRadians(10),
VariableGeneralSuper.INPUT_FIXED_NOT);
dr = new DiameterReference(18,VariableGeneralSuper.INPUT_FIXED_NOT);
z = new TeethNumber(9,VariableGeneralSuper.INPUT_FIXED_NOT);
gamma = new
gamma(Math.toRadians(80),VariableGeneralSuper.INPUT_FIXED_NOT);
mx = new ModuleAxial(11.34,VariableGeneralSuper.INPUT_FIXED_NOT);
```

Valores de salida

```
mnIn:  1.97
mn:    1.97
mnMin: 1.97
mnMax: 1.97


mtIn:  2.0
mt:    2.0
mtMin: 2.0
mtMax: 2.0


beta0In: 10.0
beta0: 10.0
beta0 Min: 10.0
beta0 Max: 10.0


drIn:  18.0
dr:    18.0
drMin: 18.0
drMax: 18.0


zIn:  9
z:    9
zMin: 9
zMax: 9


gammaIn:  80.0
gamma:    80.0
gammaMin: 80.0
gammaMax: 80.0
```

```
mxIn:  11.34
mx:    11.34
mxMin: 11.34
mxMax: 11.34
```

f) Cuando el valor de entrada de Beta es “0”, lo que significa que se desean calcular los parámetros para un engranaje recto, y por lo tanto no se calculara el Módulo axial.

Valores de entrada

```
mn = new ModuleNormal(1, VariableGeneralSuper.INPUT_FIXED_NOT);
mt = new ModuleTransverse(2, VariableGeneralSuper.INPUT_FIXED_NOT);
beta0 = new Beta0(Math.toRadians(0),
VariableGeneralSuper.INPUT_FIXED_NOT);
dr = new DiameterReference(30,VariableGeneralSuper.INPUT_FIXED_NOT);
z = new TeethNumber(9,VariableGeneralSuper.INPUT_FIXED_NOT);
gamma = new
gamma(Math.toRadians(90),VariableGeneralSuper.INPUT_FIXED_NOT);
mx = new ModuleAxial(5,VariableGeneralSuper.INPUT_FIXED_NOT);
```

Valores de salida

```
mnIn:  1.0
mn:    2.0
mnMin: 1.0
mnMax: 3.33


mtIn:  2.0
mt:    2.0
mtMin: 2.0
mtMax: 3.33


beta0In: 0.0
beta0: 0.0
beta0 Min: 0.0
beta0 Max: 53.13


drIn:  30.0
dr:    18.0
drMin: 18.0
drMax: 30.0


zIn:  9
z:    9
```

```
zMin: 9
zMax: 9


gammaIn:  90.0
gamma:    90.0
gammaMin: 90.0
gammaMax: 90.0


mxIn:  5.0
mx:    9.9999999999E9
mxMin: 9.9999999999E9
mxMax: 9.9999999999E9
```

6 CONCLUSIONES

- El software de programación en lenguaje Java "Eclipse", es un software que facilita la elaboración de instrucciones en dicho lenguaje, identificando errores y peligros, dando posibles soluciones, para que el programador seleccione la solución que él desea de acuerdo a su diseño.
- Una de las grandes ventajas de Java fue, el almacenamiento de varios valores en una misma variable que es definida de manera global, gracias a la declaración "*static*".
- El almacenamiento de métodos y variables que serían usados por las clases, en las super clases, significa una gran reducción en el tamaño del software.
- No hubo necesidad de recargar ninguna clase con demasiadas líneas de código, gracias a que el lenguaje de programación Java, tiene la ventaja de dar la libertad al programador de usar cuantas clases quiera para el desarrollo de un software.
- Al analizar las reglas que se plantearon para el funcionamiento del software mediante diagramas y tablas, y posteriormente realizar la simulación, se pudo observar donde se iban presentando las falencias de acuerdo a las reglas ya establecidas.

BIBLIOGRAFÍA

- DIN 3960, Definitions, parameters and equations for involute cylindrical gears and gear pairs.
- ISO 21771:2007 specifies the geometric concepts and parameters for cylindrical gears with involute helicoid tooth flanks. Flank modifications are included. It also covers the concepts and parameters for cylindrical gear pairs with parallel axes and a constant gear ratio, which consist of cylindrical gears according to it. Gear and mating gear in these gear pairs have the same basic rack tooth profile.
- SHIGLEY, Joseph Edward, MITCHELL, Larry D. Diseño en Ingeniería Mecánica. Editorial McGRAWHILL, 1985. 914p. ISBN 968-51-607-X.
- UNIVERSIDAD DE BURGOS, GUIA DE INICIACION AL LENGUAJE JAVA. Versión 2.0. 1.999.

ANEXOS

ANEXO A