Benemérita Universidad Autónoma de Puebla

- FACULTAD DE CIENCIAS DE LA COMPUTACIÓN -

Modelado de un simulador para un sistema dinámico usando UML

TESIS

Para obtener el título de Licenciado en: Ingeniería en Ciencias de la Computación

P R E S E N T A

Israel Mozo Valencia

Matricula: 200117935

Asesor
Dr. Eduardo Ríos Silva

Asesor interno
Dr. Manuel Martín Ortiz

Puebla, Pue. 2008

Modelado de un simulador para un sistema dinámico usando UML

RESUMEN

En el presente trabajo se realizó el estudio del modelo de un sistema, bajo la metodología del lenguaje unificado de modelado, conocido como UML por sus siglas en ingles, *Unified Modeling Language.* El sistema modelado es un sistema dinámico para un robot manipulador de tres grados de libertad con la finalidad de plantear los diagramas necesarios para generar un simulador del mismo.

El lenguaje unificado de modelado, es un conjunto de diagramas que incluyen todos los aspectos para la elaboración de un sistema. Estos diagramas están enfocados en el paradigma de *programación orientado a objetos*. Sin embargo, ésta no es la única razón para emplear éste paradigma, ya que se analizó un aspecto de gran importancia que se incluye en el estudio y es la *complejidad del software*, y gracias a lo cual, se remarca la importancia del uso del enfoque orientado a objetos.

Se muestra el análisis realizado al modelo dinámico para un robot manipulador de 3 grados de libertad bajo el paradigma orientado a objetos y con la metodología UML. El modelo dinámico esta formado por distintos elementos como son: la matriz de inercia, el vector de gravedad, la matriz de fricción, etc. de los cuales surgen las clases, los métodos, y las relaciones que forman parte de cada uno de los correspondientes diagramas UML.

Para la construcción de los diagramas que describen el modelado del sistema se utilizó el software de diseño Visual Paradigm 5.0, que mediante una interfaz gráfica, facilita la elaboración de los diagramas UML.

Finalmente, los resultados presentados son los diagramas UML que contienen las clases, la comunicación entre éstas, el diagrama de actividades, etc. que unidos muestran en forma precisa la manera en que el programador deberá llevar a cabo el desarrollo del software para el caso del simulador del sistema dinámico modelado con UML.

CONTENIDO

JUSTIFICACIÓN

Durante años, los programadores se han dedicado a construir aplicaciones muy parecidas que resolvían una y otra vez los mismos problemas. Para conseguir que los esfuerzos de los programadores pudieran ser utilizados por otras personas se creó la POO, que es una serie de normas de realizar las cosas de manera que otras personas puedan utilizarlas y adelantar su trabajo, de manera que consigamos que el código se pueda reutilizar. A la reutilización, también debemos añadir una característica importante y clave que es la *legibilidad*.

La programación orientada a objetos (POO) es una forma especial de programar, que a diferencia de tipos de programación, es más cercana a como expresaríamos las cosas en la vida real. La POO no es difícil, pero es una manera especial de pensar, a veces subjetiva de quien la programa, de manera que la forma de hacer las cosas puede ser diferente según el programador. Aunque se puedan hacer los programas de formas distintas, no todas ellas son correctas. Lo difícil no es programar orientado a objetos sino programar bien. Programar bien es importante porque así se pueden aprovechar todas las ventajas de la POO.
Con la POO se tiene que aprender a pensar las cosas de una manera distinta, para escribir los programas en términos de objetos, propiedades, métodos y otras.

Estas características, unidas a una metodología para poner en orden las ideas, en cuanto a las especificaciones que se desean en un programa, así como para definir cada uno de los elementos que conformarán el mismo, da como resultado una forma de programar más eficiente. Esta eficiencia es la que se busca al aplicar el UML al realizar el modelo de un problema, ya que se puede tener una perspectiva de software (elementos UML se mapean al SW) y una perspectiva conceptual (descripción del dominio).

Una gran ventaja es que por medio de una notación gráfica, se tiene una manera sencilla de poder representar los modelos, y así, por un lado, una perspectiva sobre el software que será elaborado en base a una notación UML y a la vez una perspectiva conceptual (por ejemplo, la notación del diagrama de clases, define cómo los conceptos e ítems y sus relaciones son representados).

Una respuesta importante a la pregunta de ¿para qué modelar los problemas antes de abordarlos directamente a la programación?, es debido a que si se sigue el último método, el 80% de todos los proyectos de software fallan. Estos proyectos sobrepasan sus presupuestos, no proporcionan las características que los clientes necesitan o desean o, lo que es peor nunca se entregan [4].

Ahora bien, al aplicar el análisis orientado a objetos y describiéndolo mediante los diagramas que UML aporta en un sistema de una complejidad alta, como es el caso del simulador para un sistema dinámico, se tendrá como resultado orden, facilidad en el momento de programar, un código legible, y cuyo mantenimiento será sencillos, gracias a que se cuenta con los diagramas, que actuarían como radiografías del sistema, con lo cual, el sistema será mas entendible, a pesar de que la persona que lo revise no sea quien lo codificó.

ANTECEDENTES

Crear software implica utilizar un proceso de desarrollo. En cualquier proyecto de construcción de software se deben tener en cuenta diferentes objetivos como *la reutilización, la calidad y la productividad [7].*

Otro problema que se encuentra al hacer sistemas, no es, como pudiera parecer, el resolver una problemática con software, sino asegurar que la solución generada sea efectiva y a un costo aceptable.

Además el proceso de desarrollo debe cumplir requisitos. Un requisito representa esencialmente un problema a resolver y el software representa la solución [8]. La mayor parte del software que se produce no cumple con todos los requisitos que cada proceso requiere, apenas resuelve solo algunos aspectos de la problemática de las organizaciones y sus costos se tienen que aceptar, no porque sean buenos, sino porque no existen otras opciones. Siendo estos costos, con mucha frecuencia, sobreestimados, por lo que la relación costo-beneficio se inclina al costo, ya que éste es muy grande y el beneficio es definitivamente pequeño. Para asegurar la solución efectiva del problema, es indispensable contar con herramientas de análisis y diseño, que permitan establecer un acuerdo entre los analistas, desarrolladores y el usuario final, para asegurar el entendimiento del problema y poder proporcionar una solución que satisfaga al cliente.

UML (Unified Modeling Language) es una herramienta que cumple con estas funciones, ya que ayuda a capturar la idea de un sistema y comunicarla posteriormente a quien este involucrado en su proceso de desarrollo; esto se lleva a cabo mediante un conjunto de símbolos y diagramas que tienen fines distintos en el proceso de desarrollo. Los lenguajes de modelado como el Unified Modeling Language (UML) son el puente entre los requisitos y el desarrollo del sistema.

UML es un lenguaje visual de modelado y comunicación para especificar, visualizar, construir y documentar software.

A finas de los 80ôs, 11 compañías crearon el Object Management Group (OMG), que es un consorcio a nivel internacional que integra a los principales representantes de la industria de la tecnología de información OO. El OMG tiene como objetivo central la promoción, fortalecimiento e impulso de la industria OO. El OMG propone y adopta por consenso especificaciones entorno a la tecnología. Mientras que por otro lado se había formado la Rational Software Corporation, cuyo fin era proporcionar herramientas para difundir el uso de practicas de ingeniería de software modernas, en especial la arquitectura modular. El término unificado "unified" se refiere al hecho de que el Object Management Group (OMG) y Rational Software Corporation crearon UML para reunir las mejores prácticas de la industria. Como sucede en otros lenguajes, el UML fue inventado por necesidad. Es más, como en muchos lenguajes, en el UML se usan símbolos para transmitir significado. Sin embargo, a diferencia de otros lenguajes como el inglés y el alemán, que evolucionan con el transcurso del tiempo a partir del uso común y la adaptación, el UML fue inventado por científicos.

INTRODUCCIÓN

UML (*Unified Modeling Language*) es un lenguaje que permite modelar, construir y documentar los elementos que forman un sistema software orientado a objetos. Se ha convertido en el estándar para la industria, debido a que ha sido concebido por los autores de los tres métodos más usados de orientación a objetos: Grady Booch, Ivar Jacobson y James Rumbaugh, apodados ñLos tres amigosò, quienes trabajaban en empresas distintas durante la década de los años ochenta y principios de los noventa y cada uno diseñó su propia metodología para el análisis y diseño orientado a objetos. Sus metodologías dominaron sobre las de sus competidores. A mediados de los años noventa empezaron a intercambiar ideas entre si y decidieron desarrollar su trabajo en conjunto. Los anteproyectos del UML empezaron a circular en la industria del software y las reacciones resultantes trajeron consigo considerables modificaciones. Conforme diversos corporativos vieron que el UML era útil a sus propósitos, se conformó un consorcio del UML. Y poco a poco desde entonces el UML ha llegado a ser el estándar en facto en la industria del software, y su evolución continúa.

Figura 1 - Logo de UML.

Esta notación ha sido ampliamente aceptada debido al prestigio de sus creadores y debido a que incorpora las principales ventajas de cada uno de los métodos particulares en los que se basa: Booch, OMT y OOSE. UML ha puesto fin a las llamadas ñguerras de métodosò que se han mantenido a lo largo de los 90, en las que los principales métodos sacaban nuevas versiones que incorporaban las técnicas de los demás. Con UML se fusiona la notación de estas técnicas para formar una herramienta compartida entre todos los ingenieros software que trabajan en el desarrollo orientado a objetos.

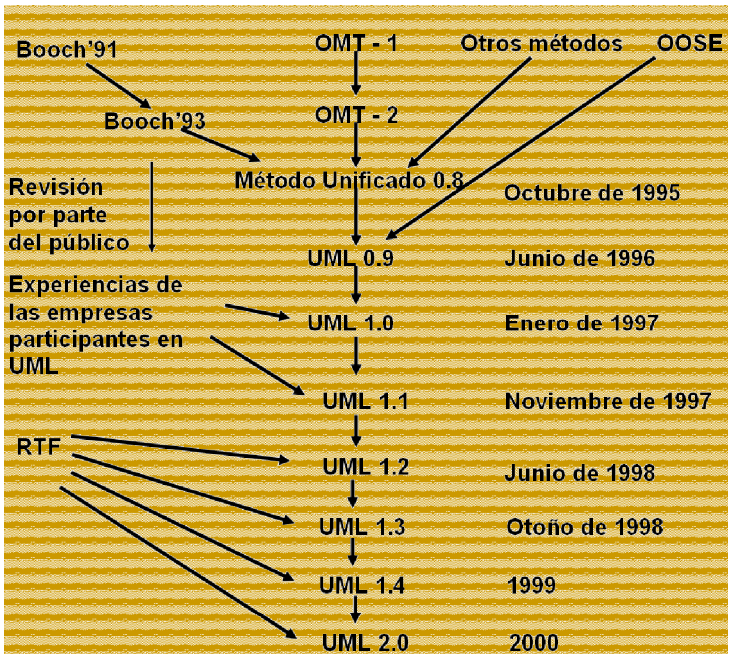

Fig.2: Historia de UML.

Desarrollo Orientado a Objetos con UML

El objetivo principal cuando se empezó a gestar UML era posibilitar el intercambio de modelos entre las distintas herramientas CASE orientadas a objetos del mercado. Para ello era necesario definir una notación y semántica común. En la figura 2 se puede ver cuál ha sido la evolución de UML hasta la creación de UML. Hay que tener en cuenta que el estándar UML no define un proceso de desarrollo específico, tan solo se trata de una notación.

¿Por qué un lenguaje unificado de modelado?

Dado que los desarrolladores necesitan un lenguaje de modelado para ayudarles a discutir los problemas y las soluciones implicados en la construcción del sistema ¿qué debería determinar el lenguaje que utilizan?

El lenguaje elegido debería ser:

1. *Suficientemente expresivo*, de manera que sea posible expresar los aspectos del diseño que será necesario tratar, y que reflejen de forma que tengan

sentido, los cambios en el diseño que se lleven a cabo durante el desarrollo como cambios en el modelo.

2. *Suficientemente fácil de utilizar*, de forma que el lenguaje de modelado ayude a tener un conocimiento claro en vez de proporcionar el camino para tener dicho conocimiento claro.
3. *Inequívoco,* para que el lenguaje de modelado ayude a resolver malos entendidos en vez de presentar más.
4. *Soportado por herramientas adecuadas,* de manera que el esfuerzo de los desarrolladores pueda utilizarse en un trabajo que requiera su habilidad, no en un trabajo rutinario como crear un diagrama con una herramienta de dibujo.
5. *Generalmente utilizado,* por gran variedad de razones. Por supuesto cuanto más general sea la utilización de un lenguaje es más probable que se cumplan los cuatro puntos anteriores. También,
    - cuando se incorpora gente nueva en el proyecto, es una ventaja si ya conocen el lenguaje de modelado en vez de tener que aprenderlo;
    - para tener diseño basado en componentes hay qye ser capaz de leer las descripciones de los componentes, y cuanto más rápido y fácil se pueda hacer es más barato tener en cuanta un componente. Cuanto más general sea la utilización de su lenguaje de modelado, mayor es la posibilidad de que sea el mismo que el escritor del componente decidió utilizar.

¿Por qué es necesario el UML?

En los principios de la computación, los programadores no realizaban análisis muy profundos sobre el problema por resolver. Con frecuencia comenzaban a escribir el programa desde el principio, y el código necesario se escribía conforme se requería. Aunque anteriormente esto agregaba un aura de aventura y atrevimiento al proceso, en la actualidad es inapropiado en los negocios de alto riesgo. Hoy en día, es necesario contar con un plan bien analizado. Un cliente tiene que comprender qué es lo que hará un equipo de

desarrolladores; además tiene que ser capaz de señalar cambios, si no se han captado claramente sus necesidades (o si se cambia de opinión durante el proceso). A su vez el desarrollo es un esfuerzo orientado a equipos, por lo que cada uno de sus miembros tiene que saber qué lugar toma su trabajo en la solución final (así como saber cuál es la solución en general).

Conforme aumenta la complejidad del mundo, los sistemas informáticos también deberán crecer en complejidad. En ellos se encuentran diversas piezas de hardware y software que se comunican a grandes distancias mediante una red, misma que está vinculada a bases de datos que, a su vez, contienen grandes cantidades de información. Si deseamos crear sistemas que nos involucren con este nuevo milenio ¿Cómo manejar tanta complejidad?, pues la clave está en organizar el proceso de diseño de tal forma que los analistas, clientes, desarrolladores y otras personas involucradas en el desarrollo del sistema lo comprendan y convengan con él. El UML proporciona tal organización.

Un arquitecto no podría crear una compleja estructura como lo es un edificio de oficinas sin crear primero un anteproyecto detallado; asimismo no se podrían generar complejos sistemas sin crear un plan de diseño detallado. Tal plan de diseño debe ser el resultado de un cuidadoso análisis de las necesidades del cliente. Otra característica del desarrollo de sistemas contemporáneo es reducir el periodo de desarrollo. Cuando los plazos se encuentran muy cerca uno del otro es absolutamente necesario contar con un diseño sólido.

La necesidad de diseños sólidos ha traído consigo la creación de una notación de diseño que los analistas, desarrolladores y clientes acepten como pauta (tal como la notación en los diagramas esquemáticos sirve como pauta para los trabajadores especializados en electrónica). El UML es esa misma notación. El UML está compuesto por diversos elementos gráficos que se combinan para

conformar diagramas. Debido a que el UML es un lenguaje, cuenta con reglas para combinar tales elementos. La finalidad de los diagramas es presentar diversas perspectivas de un sistema, a las cuales se les conoce como *modelo.* El modelo UML de un sistema es similar a un modelo a escala de un edificio junto con la interpretación del artista del edificio. Es importante destacar que un modelo UML describe lo que supuestamente hará un sistema, pero no indica cómo implementa dicho sistema.

OBJETIVOS

OBJETIVO PRINCIPAL

El objetivo principal es plantear una estrategia para el modelado de un sistema dinámico utilizando la metodología de UML, con el fin de poder incorporar el diseño orientado a objetos para realizar la correspondiente simulación.

OBJETIVOS ESPECÍFICOS

a) Desarrollar y analizar el modelo dinámico para un sistema con tres grados de libertad.

b) Analizar el sistema en su totalidad con el paradigma orientado a objetos.

c) Diseñar el sistema a desarrollar bajo el paradigma orientado a objetos.

d) Utilizar el Lenguaje Unificado de Modelado (UML) para representar el modelo.

I

MODELADO

CAPÍTULO I. MODELADO

Antes de empezar, definamos que es un modelo según [1]:

Un modelo es una simplificación de la realidad.

Un modelo proporciona los planos de un sistema. Los modelos pueden involucrar planos detallados, así como planos más generales que ofrecen una visión global del sistema en consideración. Un buen modelo incluye aquellos elementos que tienen una gran influencia y omite aquellos elementos menores que no son relevantes para el nivel de abstracción. Todo sistema puede ser descrito desde diferentes perspectivas utilizando diferentes modelos, y cada modelos es por tanto una abstracción semánticamente cerrada del sistema. Un modelo puede ser estructural, destacando la organización del sistema, o puede ser de comportamiento, resaltando su dinámica.

A través del modelado, conseguimos cuatro objetivos:

1. Los modelos nos ayudan a visualizar cómo es o queremos que sea un sistema.
2. Los modelos nos permiten especificar la estructura o el comportamiento de un sistema.
3. Los modelos nos proporcionan plantillas que nos guían en la construcción de un sistema.
4. Los modelos documentan las decisiones que hemos adoptado.

El modelado no es sólo para los grandes sistemas. Incluso el equivalente software de una caseta de perro puede beneficiarse de algo de modelado. Sin embargo, es absolutamente cierto que, cuanto más grande y complejo es el sistema, el modelado se hace más importante, por una simple razón:

Construimos modelos de sistemas complejos porque no podemos comprender el *sistema en su totalidad.*

Hay límites a la capacidad humana de comprender la complejidad. A través del modelado, se reduce el problema que se está estudiando, centrándose sólo en un aspecto a la vez. Este es, esencialmente, el enfoque "divide y vencerás" que planteó Edsger Dijkstra años atrás: acometer un problema difícil dividiéndolo en una serie de subproblemas más pequeños que se pueden resolver. Además, a través del modelado, se potencia la mente humana. Un modelo escogido adecuadamente puede permitir al modelador trabajar a mayores niveles de abstracción.

Decir que se debería modelar no significa que necesariamente se esté haciendo. De hecho, una serie de estudios sugieren que la mayoría de las organizaciones software hacen poco, o ningún, modelado formal. Si se representara el uso del modelado frente a la complejidad de un problema, se encontraría que cuanto más sencillo es el proyecto menos probable es que se haya utilizado un modelado formal.

La palabra clave aquí es "formal". En realidad, incluso en el proyecto más simple, los desarrolladores hacen algo de modelado, si bien muy informalmente. Un desarrollador puede bosquejar una idea sobre una un trozo de papel para visualizar una parte de un sistema, o el equipo puede utilizar tarjetas CRC[1] para trabajar a través de un escenario o el diseño de un mecanismo. No hay nada malo con ninguno de estos modelos. Si funcionan, deben utilizarse por todos los medios. Sin embargo, estos modelos informales son muchas veces ad hoc, y no proporcionan un lenguaje común que se pueda compartir fácilmente con otros. Así como existe un lenguaje común de planos para la industria de la construcción, un lenguaje común para la ingeniería eléctrica y un lenguaje común para el modelado

[1] Las tarjetas CRC son una metodología para el diseño de software orientado por objetos creada por Kent Beck y Ward Cunningham.
Como una extensión informal a UML, la técnica de las tarjetas CRC se puede usar para guiar el sistema a través de análisis guiados por la responsabilidad. Las clases se examinan, se filtran y se refinan basándose en sus responsabilidades con respecto al sistema, y las clases con las que necesitan colaborar para completar sus responsabilidades

matemático, también una empresa de software puede beneficiarse utilizando un lenguaje común para el modelado de software.

Cualquier proyecto puede beneficiarse de algo de modelado. Incluso en el dominio del software desechable, donde a veces es más efectivo deshacerse del software inadecuado a causa de la productividad ofrecida por los lenguajes de programación visuales, el modelado puede ayudar al equipo de desarrollo a visualizar mejor el plano de su sistema y a permitirles desarrollar más rápidamente, al ayudarles a construir el producto apropiado. Cuanto más complejo sea un proyecto, más probable es que se fracase o no se construya el producto apropiado si no se hace nada de modelado. Todos los sistemas interesantes y útiles tienen una tendencia natural a hacerse más complejos con el paso del tiempo. Así que, aunque al inicio se pueda pensar que no es necesario modelar, cuando el sistema **evolucione** se lamentará esa decisión y entonces será demasiado tarde.

1.1 PRINCIPIOS DE MODELADO

Grady Booch, Ivar Jacobson y Jim Rumbaugh nos proponen en [2] cuatro principios de modelado:

Primero:

> *La elección de qué modelos crear tiene una profunda influencia sobre cómo se acomete un problema y como se da forma a una solución.*

En otras palabras, hay que elegir bien los modelos. Los modelos adecuados pueden arrojar mucha luz sobre los problemas de desarrollo más horribles, ofreciendo una comprensión que simplemente no podríamos obtener de otra manera; los modelos erróneos desorientarán, haciendo que uno se centre en cuestiones irrelevantes.

En el software, los modelos elegidos pueden afectar mucho a nuestra visión del mundo. Si construimos un sistema con la mirada de un desarrollador de bases de datos, probablemente nos centraremos en los modelos entidad-relación que trasladan el comportamiento a disparadores (triggers) y procedimientos almacenados. Si construimos un sistema con la mirada de un analista estructurado, probablemente se obtendrán modelos centrados en los algoritmos, con los datos fluyendo de proceso en proceso. Si construimos un sistema con la mirada de un desarrollador orientado a objetos, se obtendrá un sistema cuya arquitectura se centra en un mar de clases y los patrones de interacción que gobiernan cómo trabajan juntas esas clases. Cualquiera de estos enfoques podría estar bien para una aplicación y una cultura de desarrollo dadas, aunque la experiencia sugiere que la visión orientada a objetos es superior, al proporcionar arquitecturas flexibles, incluso para sistemas que podrían tener una gran base de datos o una gran componente computacional. No obstante, la cuestión es que cada visión del mundo conduce a un tipo de sistema diferente, con diferentes costes y beneficios.

Segundo:

Todo modelo puede ser expresado a diferentes niveles de precisión.

Si se está construyendo un rascacielos**,** a veces es necesaria una vista de pájaro, por ejemplo, para ayudar a que los inversionistas se imaginen su aspecto externo. Otras veces, hay que descender al nivel de los remaches, por ejemplo, cuando hay un recorrido de tuberías enmarañado o un elemento estructural poco usual.

Eso mismo es cierto con los modelos software. A veces, un pequeño y sencillo modelo ejecutable de la interfaz del usuario es exactamente lo que se necesita; otras veces, hay que bajar y enredarse con los bits, como cuando se están especificando interfaces entre sistemas o luchando con cuellos de botella en redes. En cualquier caso, los mejores tipos de modelos son aquéllos que permiten elegir el grado de detalle, dependiendo de quién está viendo el sistema y por qué necesita

verlo. Un analista o un usuario final se centrará en el qué; un desarrollador se centrará en el cómo. Tanto unos como otros querrán visualizar un sistema a diferentes niveles de detalle en momento diferentes.

Tercero:

Los mejores modelos están ligados a la realidad.

Un modelo físico de un edificio que no responda de la misma forma que los materiales reales tiene sólo un valor limitado; un modelo matemático de una aeronave que asuma sólo condiciones ideales y una fabricación perfecta puede enmascarar algunas características de la aeronave real potencialmente fatales. Es mejor tener modelos que tengan una clara conexión con la realidad, y donde ésta conexión sea débil saber exactamente cómo se apartan esos modelos del mundo real. Todos los modelos simplifican la realidad; el truco está en asegurarse de que las simplificaciones no enmascaran ningún detalle importante.

En el software, el talón de Aquiles de las técnicas de análisis estructurado es el hecho de que hay una desconexión básica entre el modelo de análisis y el modelo de diseño del sistema. No poder salvar este abismo hace que el sistema concebido y el sistema construido diverjan con el paso del tiempo. En los sistemas orientados a objetos, es posible conectar todas las vistas casi independientes de un sistema en un todo semántico.

Cuarto:

Un único modelo no es suficiente. Cualquier sistema no trivial se aborda mejor a través de un pequeño conjunto de modelos casi independientes.

Si se está construyendo un edificio, no hay un único conjunto de planos que revele todos sus detalles. Como mínimo, se necesitarán planos de planta, alzados, planos de electricidad, planos de calefacción y planos de cañerías. La expresión clave aquí es "casi independientes". En este contexto significa tener modelos que

podemos construir y estudiar separadamente, pero aún así están interrelacionados. Como en el caso de un edificio, podemos estudiar los planos eléctricos de forma aislada, pero también podemos ver su correspondencia con los planos de planta y quizás incluso su interacción con los recorridos de las tuberías en el plano de la fontanería.

Lo mismo es cierto para los sistemas software orientados a objetos. Para comprender la arquitectura de tales sistemas, se necesitan varias vistas complementarias y entrelazarlas: una vista de casos de uso (que muestre los requisitos del sistema), una vista de diseño (que capture el vocabulario del espacio del problema y del espacio de la solución), una vista de procesos (que modele la distribución de los procesos e hilos [threads] del sistema), una vista de implementación (que se ocupe de la realización física del sistema) y una vista de despliegue (que se centre en cuestiones de ingeniería del sistema). Cada una de estas vistas puede tener aspectos tanto estructurales como de comportamiento. En conjunto, estas vistas representan los planos del software.

Según la naturaleza del sistema, algunos modelos pueden ser más importantes que otros. Por ejemplo, en sistemas con grandes cantidades de datos, dominarán los modelos centrados en las vistas de diseño estáticas. En sistemas con uso intensivo de interfaces gráficas de usuario (GUI), las vistas de casos de uso estáticas y dinámicas son bastante importantes. En los sistemas de tiempo real muy exigentes, las vistas de procesos dinámicas tienden a ser más importantes. Finalmente, en los sistemas distribuidos, como los encontrados en aplicaciones de uso intensivo de la Web, los modelos de implementación y despliegue son los más importantes.

1.2 MODELADO ORIENTADO A OBJETOS

Los ingenieros civiles construyen muchos tipos de modelos. Lo más frecuente es que usen modelos estructurales que les ayudan a visualizar y especificar partes de los sistemas y la forma en que esas partes se relacionan entre sí. Dependiendo de las

cuestiones más importantes del sistema o de la ingeniería que les preocupen, los ingenieros podrían también construir modelos dinámicos. Por ejemplo, para ayudarles a estudiar el comportamiento de una estructura en presencia de un terremoto. Cada tipo de modelo se organiza de forma diferente, y cada uno tiene su propio enfoque.

En el software hay varias formas de enfocar un modelo. Las dos formas más comunes son la perspectiva orientada a objetos y la perspectiva algorítmica.

La visión tradicional del desarrollo de software toma una perspectiva algorítmica. En este enfoque, el bloque principal de construcción de todo el software es el procedimiento o función. Esta visión conduce a los desarrolladores a centrarse en cuestiones de control y de descomposición de algoritmos grandes en otros más pequeños. No hay nada inherentemente malo en este punto de vista, salvo que tiende a producir sistemas frágiles. Cuando los requisitos **cambian** (tienden a hacerlo) y el sistema **crece** (que generalmente es una característica intrínseca), los sistemas construidos con un enfoque algorítmico se vuelven muy difíciles de mantener.

La visión actual del desarrollo de software toma una perspectiva orientada a objetos. En este enfoque, el principal bloque de construcción de todos los sistemas software es el objeto o clase. Para explicarlo sencillamente, un objeto es una cosa, generalmente extraída del vocabulario del espacio del problema o del espacio de la solución; una clase es una descripción de un conjunto de objetos similares. Todo objeto tiene identidad (puede nombrarse o distinguirse de otra manera de otros objetos), estado (generalmente hay algunos datos asociados a él), y comportamiento (se le pueden hacer cosas al objeto, y él a su vez puede hacer cosas a otros objetos).
Por ejemplo, considérese una arquitectura sencilla de tres capas para un sistema de contabilidad, que involucre una interfaz de usuario, una capa intermedia y una base de datos. En la interfaz de usuario aparecerán objetos concretos tales como botones, menús y cuadros de diálogo. En la base de datos aparecerán objetos

concretos, tales como tablas que representarán entidades del dominio del problema, incluyendo clientes, productos y otras. En la capa intermedia aparecerán objetos tales como transacciones y reglas de negocio[2], así como vistas de más alto nivel de las entidades del problema, tales como clientes, productos y pedidos.

Actualmente, el enfoque orientado a objetos forma parte de la tendencia principal para el desarrollo de software, simplemente porque ha demostrado ser válido en la construcción de sistemas en toda clase de dominios de problemas, abarcando todo el abanico de tamaños y complejidades. Más aún, la mayoría de los lenguajes actuales, sistemas operativos y herramientas son orientados a objetos de alguna manera, lo que ofrece más motivos para ver el mundo en términos de objetos. El desarrollo orientado a objetos proporciona la base fundamental para ensamblar sistemas a partir de componentes utilizando tecnologías como Java Beans o COM+. Varias cuestiones se derivan de la elección de ver el mundo de una forma orientada a objetos: ¿Cuál es la estructura de una buena arquitectura orientada a objetos? ¿Qué artefactos debería crear el proyecto? ¿Quién debería crearlos? ¿Cómo deberían medirse?

Visualizar, especificar, construir y documentar sistemas orientados a objetos es exactamente el propósito del Lenguaje Unificado de Modelado (Unified Modeling Language).

[2] Una regla de negocio es una expresión que define o restringe algún aspecto estructural o dinámico de una organización.

II

COMPLEJIDAD, PROGRAMACIÓN ORIENTADA A OBJETOS, DISEÑO ORIENTADO A OBJETOS Y UML

CAPÍTULO II. COMPLEJIDAD, PROGRAMACIÓN ORIENTADA A OBJETOS, DISEÑO ORIENTADO A OBJETOS Y UML

2.1 LA COMPLEJIDAD DEL SOFTWARE

Debido a los avances tecnológicos, el mundo en general y las diversas áreas de la ciencia y la tecnología se han visto afectados con cambios importantes y significativos, el desarrollo de software, no podía ser la excepción.

Desde la década de los setenta se ha puesto cada vez más atención a la tecnología del desarrollo de software. Conforme los sistemas de cómputo van en aumento, se hacen más complejos y se introducen con mayor profundidad en la sociedad actual, se hace evidente la necesidad de enfoques sistemáticos para el desarrollo de software, así como para su evolución en el tiempo.

La mala planeación o la total carencia de ésta es una de las principales causas de retrasos en el desarrollo de software, lo que ocasiona el incremento de costos, poca calidad y altos costos de mantenimiento[1] en los productos de programación. Para evitar estos problemas, en lo posible, se requiere de una planeación cuidadosa, tanto en el proceso de desarrollo, como en la operación del sistema. Hoy en día, es necesario contar con un plan bien analizado.

Conforme aumenta la complejidad del mundo, los sistemas informáticos también deberán crecer en complejidad, el problema es ¿cómo manejar tanta complejidad?, la clave está en organizar el proceso de diseño de tal forma que los analistas, clientes, desarrolladores y todas las personas involucradas en el desarrollo del sistema lo comprendan. En este capítulo hablaremos acerca de la complejidad inherente al software, así como la descomposición como una herramienta para abordar dicha complejidad.

[1] Los cambios que se producen por la evolución de los sistemas grandes implica lo que comúnmente y de forma errónea se conoce como ***ñmantenimiento de softwareò***. Hablamos de *mantenimiento* cuando se corrigen errores; es *evolución* cuando se responde a requerimientos que cambian; es *conservación* cuando se siguen empleando medios extraordinarios para mantener en operación un elemento de software anticuado.

2.1.1 La complejidad del software de dimensión industrial

Sabemos que algunos sistemas de software no son complejos, dichos sistemas tienden a tener un propósito muy limitado y un ciclo de vida muy corto. Podemos permitirnos desecharlos y reemplazarlos por software completamente nuevo en lugar de intentar reutilizarlos, repararlos o extender su funcionalidad.
La principal característica del software de dimensión industrial, es que resulta sumamente difícil, si no imposible, para el desarrollador individual comprender todos los requerimientos de su diseño, en otras palabras, la complejidad de tales aplicaciones excede la capacidad intelectual humana. Lamentablemente, esta complejidad de la que hablamos parece ser una propiedad de los sistemas de grandes, dicha complejidad desafortunadamente no podemos eliminarla, pero puede controlarse.

2.1.2 Los cuatro elementos de la complejidad del software

La complejidad propia del software, se deriva de cuatro elementos que describiremos a continuación:

- La complejidad del dominio del problema,
- La dificultad de gestionar el proceso de desarrollo,
- La flexibilidad que se puede alcanzar a través del software y
- Los problemas que plantea la caracterización del comportamiento de sistemas discretos.

2.1.3 La complejidad del dominio del problema

Intentar resolver algún problema mediante software lleva consigo cierto grado de complejidad, esto comienza desde la especificación de requisitos, debido a que en muchas ocasiones el usuario suele encontrar grandes dificultades para expresarle sus necesidades a los desarrolladores, en algunas ocasiones ni siquiera tiene una idea clara de lo que un sistema computacional puede ofrecerle. Esto no es atribuible ni al usuario ni al desarrollador; más bien se debe a que

ninguno de los dos grupos es experto en el área del otro. Un problema adicional es que frecuentemente los requisitos de una aplicación cambian durante el desarrollo del sistema.

Además es muy importante señalar que los sistemas de software grandes significan una inversión considerable y como estos evolucionan constantemente, es inadmisible que cada vez que el sistema requiere de una modificación, éste sea desechado y se desarrolle una nueva aplicación.

2.1.4 La dificultad de gestionar el proceso de desarrollo

La tarea fundamental de un equipo de desarrollo de software, es ofrecerle al usuario final del sistema la ilusión de simplicidad. Se hace lo posible por escribir la menor cantidad de líneas de código, pero esto muchas veces no se logra y pasa a segundo término cuando lo que se pretende es brindarle al usuario un sistema simple y funcional.

Actualmente podemos encontrar sistemas cuyo tamaño se mide en cientos de miles, o incluso millones de líneas de código, lo que implica la utilización de un equipo de desarrolladores para la descomposición del mismo y que pueda ser comprensible para su evolución, de forma ideal se debe utilizar un equipo tan pequeño como sea posible, por un lado por los costos y por otro porque un mayor número de miembros implica una comunicación más compleja y por lo tanto una coordinación más difícil. Lo que no se debe perder de vista es mantener la unidad e integridad en el diseño.

2.1.5 La flexibilidad que se puede alcanzar a través del software

El desarrollo de software implica un trabajo sumamente laborioso, porque el software ofrece la flexibilidad máxima para que el desarrollador construya por sí mismo prácticamente todos los módulos fundamentales sobre los que se apoya el desarrollo de la aplicación, algo que no sucede en otro tipo de industria, por ejemplo, la industria de la construcción no construye su propia acería para fabricar

las vigas que utilizará para la construcción de un edificio, en cambio, tiene normativas de construcción y estándares de calidad en los materiales, de los que se carece en la fabricación de software.

2.1.6 Los problemas que plantea la caracterización del comportamiento de sistemas discretos

En un sistema de software de tamaño considerable podemos tener una cantidad importante de variables y flujos de control; el conjunto de todas las variables, sus valores actuales, la dirección de ejecución y la pila en cada uno de los procesos del sistema, conforman el estado actual de la aplicación.

Al realizarse la ejecución de un sistema de software en computadoras digitales, se tiene un sistema con estados discretos, en contraste, los sistemas analógicos son sistemas continuos[2]. Por el contrario, en sistemas discretos, se tiene un número finito de estados posibles; en sistemas grandes hay una explosión combinatoria que hace muy grande a este número.

Se pretende diseñar los sistemas de forma que el comportamiento de una parte del sistema tenga mínimo impacto en el comportamiento de otra parte del mismo. Sin embargo, las transiciones de fase entre estados discretos no pueden modelarse en su totalidad con funciones continuas.

Todos los eventos externos a un sistema de software tienen la posibilidad de llevar a este sistema a un nuevo estado, y más aún, la transición de estado a estado no siempre es determinista. En las peores circunstancias, un evento externo puede corromper el estado del sistema, porque los diseñadores olvidaron tener en cuenta ciertas interacciones entre eventos. Esta es la razón principal por la que un sistema debe probarse a conciencia. Ya que no se dispone ni de las herramientas matemáticas ni de la capacidad intelectual necesaria para modelar el

[2] Sistemas continuos: pequeños cambios en las entradas siempre producirán cambios pequeños en las salidas.

comportamiento completo de grandes sistemas discretos, hay que conformarse con un grado de confianza aceptable en lo referente a su corrección.

2.1.7 La estructura de los sistemas complejos

Un sistema complejo es jerárquico por naturaleza, los niveles de esta jerarquía representan diferentes niveles de abstracción[3], cada uno de los cuales se construye sobre el otro, cada uno es comprensible por sí mismo y cada nivel de ésta jerarquía tiene su propia complejidad. Sólo se pueden comprender aquellos sistemas que constan de una estructura jerárquica.

Cuando los sistemas jerárquicos pueden dividirse en partes identificables se les llama *descomponibles*, cuando las partes no son del todo independientes, es decir, no pueden dividirse completamente se les llama *casi descomponibles*.

2.1.8 Las limitaciones de la capacidad humana para enfrentarse a la complejidad

Cuando se analiza por primera vez un sistema complejo se tienen muchos elementos interactuando entre sí de múltiples y diferentes maneras, al trabajar en ello tratando de organizar la información como parte del proceso de diseño se tiene que pensar en muchas cosas al mismo tiempo, sobre todo en el caso de sistemas discretos puesto que tenemos un espacio de estados grande, desafortunadamente es imposible que una persona pueda tener un dominio sobre todos los detalles a la vez.

Grady Booch [1] menciona que: *"Experimentos psicológicos revelan que el máximo número de bloques de información que un individuo puede comprender de forma simultánea es del orden de siete más ó menos dos. Esta capacidad parece*

[3] Por ahora entenderemos la abstracción como la estructuración de un problema confuso en entidades bien definidas por medio de la definición de sus datos y operaciones. Este concepto se definirá formalmente en el Capitulo II. El desarrollo de software orientado a objetos y el Lenguaje Unificado de Modelado (UML).

*estar relacionada con capacidad de memoria a corto plazo. La mente necesita alrededor de cinco segundos para aceptar un nuevo bloque de información***ò**[1].

2.1.9 Métodos de diseño

Debido a la creciente complejidad de los sistemas de software, ha habido una evolución importante en lo que a los métodos de diseño se refiere. A continuación se hará una breve descripción de los métodos más utilizados para el diseño de sistemas, pero antes se darán algunas definiciones importantes para la mejor comprensión de los conceptos.

Un *método* es un proceso disciplinado para la generación de modelos que nos sirven para describir los aspectos del desarrollo de un sistema de software utilizando una notación bien definida.

Una *metodología* es la colección de métodos aplicados a lo largo del ciclo de vida del desarrollo de software.

El *diseño* es la aproximación disciplinada que se emplea para trazar una solución para un problema dado, suministrando de esa forma un conducto, desde los requerimientos hasta la implantación.

2.1.9.1 La descomposición

Como hemos mencionado con anterioridad, para diseñar eficientemente un sistema de software complejo es primordial descomponerlo en partes más pequeñas. La descomposición inteligente ataca directamente la complejidad inherente al software forzando una división del espacio de estados del sistema.

2.1.9.1.1 Descomposición algorítmica

Cada módulo del sistema representa un paso importante del proceso global. La utilización de módulos provoca que los datos sean controlados por él.

2.1.9.1.2 Descomposición orientada a objetos

En vez de descomponer el problema en pasos como en la descomposición algorítmica, se identifican objetos que se derivan automáticamente del vocabulario del dominio del problema. Cada objeto contiene su propio comportamiento y modela un objeto del mundo real.

La descomposición orientada a objetos produce sistemas más pequeños por medio de la reutilización de mecanismos comunes, además son más resistentes al cambio y por lo tanto están mejor preparados para evolucionar en el tiempo. La descomposición orientada a objetos reduce en gran medida el riesgo que representa construir sistemas de software complejos, ya que están diseñados para evolucionar de sistemas más pequeños en los que ya se tiene confianza, es más, esta descomposición resuelve directamente la complejidad innata del software ayudando a tomar decisiones inteligentes respecto a la separación de intereses en un gran espacio de estados.

2.1.9.2 ¿Descomposición algorítmica o descomposición orientada a objetos?

Ambos tipos de descomposición son importantes: La descomposición algorítmica nos proporciona el orden de los eventos y la orientada a objetos resalta los agentes que causan las acciones o los que son sujetos a estas acciones. No obstante no se puede construir un sistema complejo con los dos tipos a la vez, puesto que son dos perspectivas perpendiculares. Inicialmente un sistema complejo deberá descomponerse de alguna de las dos formas y posteriormente utilizar la estructura resultante como marco de referencia para la descomposición elegida.

2.1.10 El desarrollo orientado a objetos y el Lenguaje Unificado de Modelado (UML) como mecanismos para modelar sistemas complejos.

Para poder manejar esta complejidad de la que hablamos, se crean herramientas que contribuyen a la organización del proceso de diseño de tal forma que los analistas, clientes, desarrolladores, y todas las personas involucradas en el desarrollo del sistema lo comprendan, esto hace evidente la necesidad de enfoques sistemáticos para el desarrollo de software, así como para su evolución en el tiempo.

Estos enfoques sistemáticos surgen en la forma de diversas metodologías para el análisis y diseño de sistemas computacionales, metodologías que permiten modelar problemas complejos en sistemas de software de alta calidad y proporcionando una relativa facilidad en el desarrollo de sistemas, así como su evolución en el tiempo.

Una de las herramientas que cobra gran importancia en los últimos años, junto con el desarrollo orientado a objetos, es el Lenguaje Unificado de Modelado (UML por sus siglas en inglés; Unified Modeling Language). Es por ello que se decide trabajar con estos instrumentos.

La complejidad propia del software no puede eliminarse pero puede controlarse por medio de diversas herramientas creadas con este objetivo. Herramientas tales como la programación orientada a objetos y el lenguaje unificado de modelado.

2.2 PROGRAMACIÓN ORIENTADA A OBJETOS

La **Programación Orientada a Objetos** (**POO** u **OOP** según siglas en inglés) es un paradigma de programación[4] que usa objetos y sus interacciones

[4] Un **paradigma de programación** representa un enfoque particular o filosofía para la construcción del software. No es mejor uno que otro sino que cada uno tiene ventajas y desventajas. También hay situaciones donde un paradigma resulta más apropiado que otro.

para diseñar aplicaciones y programas de computadora. Está basado en varias técnicas, incluyendo herencia, modularidad, polimorfismo, y encapsulamiento. Su uso se popularizó recién a principios de la década de 1990. Actualmente varios lenguajes de programación soportan la orientación a objetos.

Los conceptos de la programación orientada a objetos tienen origen en Simula 67, un lenguaje diseñado para hacer simulaciones, creado por Ole-Johan Dahl y Kristen Nygaard del Centro de Cómputo Noruego en Oslo. Según se informa, la historia es que trabajaban en simulaciones de naves, y fueron confundidos por la explosión combinatoria de cómo las diversas cualidades de diversas naves podían afectar unas a las otras. La idea ocurrió para agrupar los diversos tipos de naves en diversas clases de objetos, siendo responsable cada clase de objetos de definir sus *propios* datos y comportamiento. Fueron refinados más tarde en Smalltalk, que fue desarrollado en Simula en Xerox PARC (y cuya primera versión fue escrita sobre Basic) pero diseñado para ser un sistema completamente dinámico en el cual los objetos se podrían crear y modificar "en marcha" en lugar de tener un sistema basado en programas estáticos.

La programación orientada a objetos tomó posición como el estilo de programación dominante a mediados de los años ochenta, en gran parte debido a la influencia de C++, una extensión del lenguaje de programación C. Su dominación fue consolidada gracias al auge de las Interfaces gráficas de usuario, para los cuales la programación orientada a objetos está particularmente bien adaptada.

Las características de orientación a objetos fueron agregadas a muchos lenguajes existentes durante ese tiempo, incluyendo Ada, BASIC, Lisp, Pascal, entre otros. La adición de estas características a los lenguajes que no fueron diseñados inicialmente para ellas condujo a menudo a problemas de compatibilidad y a la capacidad de mantenimiento del código. Los lenguajes

orientados a objetos puros[5], por otra parte, carecían de las características de las cuales muchos programadores habían venido a depender. Para saltar este obstáculo, se hicieron muchas tentativas para crear nuevos lenguajes basados en métodos orientados a objetos, pero permitiendo algunas características imperativas de maneras "seguras". El Eiffel de Bertrand Meyer fue un temprano y moderadamente acertado lenguaje con esos objetivos pero ahora ha sido esencialmente reemplazado por Java, en gran parte debido a la aparición de Internet, y a la implementación de la máquina virtual de Java en la mayoría de navegadores

Una de los elementos básicos en un lenguaje orientado a objetos es el incluir los llamados ñobjetosò, los cuales son entidades que combinan *estado*, *comportamiento* e identidad, lo que los hacen una herramienta poderosa en el ámbito computacional. El estado está compuesto de datos, y el comportamiento por procedimientos o *métodos*. La identidad es una propiedad de un objeto que lo diferencia del resto. La programación orientada a objetos expresa un programa como un conjunto de estos objetos, que colaboran entre ellos para realizar tareas. Esto permite hacer los programas y módulos más fáciles de escribir, mantener y reutilizar.

De esta forma, un objeto contiene toda la información que permite definirlo e identificarlo frente a otros objetos pertenecientes a otras clases e incluso frente a objetos de una misma clase, al poder tener valores bien diferenciados en sus atributos. A su vez, los objetos disponen de mecanismos de interacción llamados métodos que favorecen la comunicación entre ellos. Esta comunicación favorece a su vez el cambio de estado en los propios objetos. Esta característica lleva a tratarlos como unidades indivisibles, en las que no se separan ni deben separarse el estado y el comportamiento.

[5] Esto se refiere a que únicamente utilizan clases y objetos, ya que en un lenguaje de programación no puro, se usan también, tipos de datos primitivos, o bien los Wrappers que son clases que encapsulan tipos de datos primitivos.

Los métodos y atributos están estrechamente relacionados por la propiedad de conjunto. Esta propiedad destaca que una clase requiere de métodos para poder tratar los atributos con los que cuenta. El programador debe pensar indistintamente en ambos conceptos, sin separar ni darle mayor importancia a ninguno de ellos. Hacerlo podría resultar en seguir el hábito erróneo de crear clases contenedoras de información por un lado y clases con métodos que la manejen por el otro.

Esto difiere de la programación estructurada tradicional, en la que los datos y los procedimientos están separados y sin relación, ya que lo único que se busca es el procesamiento de unos datos de entrada para obtener otros de salida. La programación estructurada anima al programador a pensar sobre todo en términos de procedimientos o funciones, y en segundo lugar en las estructuras de datos que esos procedimientos manejan. En la programación estructurada sólo se escriben funciones que procesan datos. Los programadores que emplean éste nuevo paradigma, en cambio, primero definen objetos para luego enviarles mensajes solicitándoles que realicen sus métodos por sí mismos.

Las principales diferencias entre la programación estructurada y la orientada a objetos son:

- La programación orientada a objetos es más moderna, es una evolución de la programación estructurada que plasma en el diseño de una familia de lenguajes conceptos que existían previamente con algunos nuevos.
- La programación orientada a objetos se basa en lenguajes que soportan sintáctica y semánticamente la unión entre los tipos abstractos de datos y sus operaciones (a esta unión se la suele llamar clase).
- La programación orientada a objetos incorpora en su entorno de ejecución mecanismos tales como el polimorfismo y el envío de mensajes entre objetos.

Erróneamente se le adjudica a la programación estructurada clásica ciertos problemas como si fueran inherentes a la misma. Esos problemas fueron haciéndose cada vez más graves y antes de la programación orientada a objetos diversos autores (de los que podemos destacar a Yourdon) encontraron soluciones basadas en aplicar estrictas metodologías de trabajo. De esa época son los conceptos de cohesión y acoplamiento. De esos problemas se destacan los siguientes:

- Modelo mental anómalo. Nuestra imagen del mundo se apoya en los seres, a los que asignamos nombres sustantivos, mientras la programación clásica se basa en el comportamiento, representado usualmente por verbos.
- Es difícil modificar y extender los programas, pues suele haber datos compartidos por varios subprogramas, que introducen interacciones ocultas entre ellos.
- Es difícil mantener los programas. Casi todos los sistemas informáticos grandes tienen errores ocultos, que no surgen a la luz hasta después de muchas horas de funcionamiento.
- Es difícil reutilizar los programas. Es prácticamente imposible aprovechar en una aplicación nueva las subrutinas que se diseñaron para otra.
- Es compleja la coordinación y organización entre programadores para la creación de aplicaciones de media y gran envergadura.

2.2.1 Conceptos fundamentales de la POO

La programación orientada a objetos es una nueva forma de programar que trata de encontrar una solución a estos problemas. Introduce nuevos conceptos, que superan y amplían conceptos antiguos ya conocidos. Entre ellos destacan los siguientes:

- **Objeto**: entidad provista de un conjunto de propiedades o atributos (datos) y de comportamiento o funcionalidad (métodos). Corresponden a los objetos reales del mundo que nos rodea, o a objetos internos del sistema (del programa).

- **Clase**: definiciones de las propiedades y comportamiento de un tipo de objeto concreto. La instanciación es la lectura de estas definiciones y la creación de un objeto a partir de ellas.
- **Método**: algoritmo asociado a un objeto (o a una clase de objetos), cuya ejecución se desencadena tras la recepción de un "mensaje". Desde el punto de vista del comportamiento, es lo que el objeto puede hacer. Un método puede producir un cambio en las propiedades del objeto, o la generación de un "evento" con un nuevo mensaje para otro objeto del sistema.
- **Evento**: un suceso en el sistema (tal como una interacción del usuario con la máquina, o un mensaje enviado por un objeto). El sistema maneja el evento enviando el mensaje adecuado al objeto pertinente. También se puede definir como evento, a la reacción que puede desencadenar un objeto, es decir la acción que genera.
- **Mensaje**: una comunicación dirigida a un objeto, que le ordena que ejecute uno de sus métodos con ciertos parámetros asociados al evento que lo generó.
- **Propiedad o atributo**: contenedor de un tipo de datos asociados a un objeto (o a una clase de objetos), que hace los datos visibles desde fuera del objeto y esto se define como sus características predeterminadas, y cuyo valor puede ser alterado por la ejecución de algún método.
- **Estado interno**: es una propiedad invisible de los objetos, que puede ser únicamente accedida y alterada por un método del objeto, y que se utiliza para indicar distintas situaciones posibles para el objeto (o clase de objetos).
- **Componentes de un objeto**: atributos, identidad, relaciones y métodos.
- **Representación de un objeto**: un objeto se representa por medio de una tabla o entidad que esté compuesta por sus atributos y funciones correspondientes.
- **Tipo**: conjunto de firma de métodos con un nombre que lo identifica. Un tipo puede ser definido a través de una Clase o una Interface.

En comparación con un lenguaje imperativo, una "variable", no es más que un contenedor interno del atributo del objeto o de un estado interno, así como la "función" es un procedimiento interno del método del objeto.

2.2.2 Características de la POO

Hay un cierto desacuerdo sobre exactamente qué características de un método de programación o lenguaje le definen como "orientado a objetos", pero hay un consenso general en que las características siguientes son las más importantes (para más información, seguir los enlaces respectivos):

- **Abstracción**: Cada objeto en el sistema sirve como modelo de un "agente" abstracto que puede realizar trabajo, informar y cambiar su estado, y "comunicarse" con otros objetos en el sistema sin revelar *cómo* se implementan estas características. Los procesos, las funciones o los métodos pueden también ser abstraídos y cuando lo están, una variedad de técnicas son requeridas para ampliar una abstracción.
- **Encapsulamiento**: Significa reunir a todos los elementos que pueden considerarse pertenecientes a una misma entidad, al mismo nivel de abstracción. Esto permite aumentar la cohesión de los componentes del sistema. Algunos autores confunden este concepto con el principio de ocultación, principalmente porque se suelen emplear conjuntamente.
- **Principio de ocultación**: Cada objeto está aislado del exterior, es un módulo natural, y cada tipo de objeto expone una *interfaz* a otros objetos que específica cómo pueden interactuar con los objetos de la clase. El aislamiento protege a las propiedades de un objeto contra su modificación por quien no tenga derecho a acceder a ellas, solamente los propios métodos internos del objeto pueden acceder a su estado. Esto asegura que otros objetos no pueden cambiar el estado interno de un objeto de maneras inesperadas, eliminando efectos secundarios e interacciones inesperadas. Algunos lenguajes relajan esto, permitiendo un acceso directo a los datos internos del objeto de una manera controlada y limitando el grado de

abstracción. La aplicación entera se reduce a un agregado o rompecabezas de objetos.

- **Polimorfismo**: comportamientos diferentes, asociados a objetos distintos, pueden compartir el mismo nombre, al llamarlos por ese nombre se utilizará el comportamiento correspondiente al objeto que se esté usando. O dicho de otro modo, las referencias y las colecciones de objetos pueden contener objetos de diferentes tipos, y la invocación de un comportamiento en una referencia producirá el comportamiento correcto para el tipo real del objeto referenciado. Cuando esto ocurre en "tiempo de ejecución", esta última característica se llama *asignación tardía* o *asignación dinámica*. Algunos lenguajes proporcionan medios más estáticos (en "tiempo de compilación") de polimorfismo, tales como las plantillas y la sobrecarga de operadores de C++.
- **Herencia**: las clases no están aisladas, sino que se relacionan entre sí, formando una jerarquía de clasificación. Los objetos heredan las propiedades y el comportamiento de todas las clases a las que pertenecen. La herencia organiza y facilita el polimorfismo y el encapsulamiento permitiendo a los objetos ser definidos y creados como tipos especializados de objetos preexistentes. Estos pueden compartir (y extender) su comportamiento sin tener que reimplementar su comportamiento. Esto suele hacerse habitualmente agrupando los objetos en *clases* y estas en *árboles* o *enrejados* que reflejan un comportamiento común. Cuando un objeto hereda de más de una clase se dice que hay *herencia múltiple*; esta característica no está soportada por algunos lenguajes (como Java).

2.3 DISEÑO ORIENTADO A OBJETOS

El diseño orientado a objetos es una fase de la metodología orientada a objetos para el desarrollo de Software. Su uso induce a los programadores a pensar en términos de objetos, en vez de procedimientos, cuando planifican su código. Un objeto agrupa datos encapsulados y procedimientos para representar

una entidad. La 'interfaz del objeto', esto es las formas de interactuar con el objeto, también es definida en esta etapa. Un programa orientado a objetos es descrito por la interacción de esos objetos. El diseño orientado a objetos es la disciplina que define los objetos y sus interacciones para resolver un problema de negocio que fue identificado y documentado durante el análisis orientado a objetos.

El énfasis en los métodos de programación está puesto principalmente en el uso correcto y efectivo de mecanismos particulares del lenguaje que se utiliza. Por contraste, los métodos de diseño enfatizan la estructuración correcta y efectiva de un sistema complejo. ¿Qué es entonces el diseño orientado a objetos? [1] sugiere que:

> *El diseño orientado a objetos es un método de diseño que abarca el proceso de descomposición orientada a objetos y una notación para describir los modelos lógico y físico, así como los modelos estático y dinámico del sistema que se diseña.*

Hay dos partes importantes en esta definición: el diseño orientado a objetos (1) da lugar a una descomposición orientada a objetos y (2) utiliza diversas notaciones para expresar diferentes modelos del diseño lógico (estructura de clases y objetos) y físico (arquitectura de módulos y procesos) de un sistema, además de los aspectos estáticos y dinámicos del sistema.

El soporte para la descomposición orientada a objetos es lo que hace el diseño orientado a objetos bastante diferente del diseño estructurado: el primero utiliza abstracciones de clases y objetos para estructurar lógicamente los sistemas, y el segundo utiliza abstracciones algorítmicas. Se utilizará el término *diseño orientado a objetos* para referirse a cualquier método que encamine a una descomposición orientada a objetos.

2.3.1 Análisis orientado a objetos

El modelo de objetos ha influido incluso en las fases iniciales del ciclo de vida del desarrollo del software. Las técnicas de análisis estructurados tradicionales se centran en el flujo de datos dentro de un sistema. El análisis orientado a objetos (o AOO, como se le llama en ocasiones) enfatiza la construcción de modelos del mundo real, utilizando una visión del mundo orientada a objetos:

> *El análisis orientado a objetos es un método de análisis que examina los requisitos desde la perspectiva de las clases y objetos que se encuentran en el vocabulario del dominio del problema [1].*

2.3.2 ¿Cómo se relacionan AOO, DOO y POO?

Básicamente, los productos del análisis orientado a objetos sirven como modelos de los que se puede partir para un diseño orientado a objetos; los productos del diseño orientado a objetos pueden utilizarse entonces como anteproyectos para la implementación completa de un sistema utilizando métodos de programación orientada a objetos.

2.4 Lenguaje unificado de modelado (UML)

La notación UML, se ha convertido desde finales de los 90 en un estándar para modelar con tecnología orientada a objetos todos aquellos elementos que configuran la arquitectura de un sistema de información y, por extensión, de los procesos de negocio de una organización. De la misma manera que los planos de un arquitecto disponen el esquema director a partir del cual levantamos un edificio, los diagramas UML suministran un modelo de referencia para formalizar los procesos, reglas de negocio, objetos y componentes de una organización. La interacción de todos estos elementos es una representación de nuestra realidad.

Nuestros límites para entender esta realidad están en nuestro lenguaje. El mundo es la suma total de lo que podemos definir, organizar y visualizar. Cabe preguntarse ¿de qué manera un modelo en UML representa nuestra experiencia? Enseñar a utilizar un lenguaje formal siempre es problemático. Es necesario *mostrar* como este lenguaje puede ser aplicado a la realidad tal como la conocemos y tal como la compartimos con los demás. Con esta guía visual mostramos de manera precisa las técnicas básicas para usar UML en diferentes contextos. La clave está en discriminar cuales son aquellos procedimientos esenciales que nos permiten evitar costosas confusiones conceptuales.

No es mediante el descubrimiento de nuevos objetos y de sus múltiples conexiones que avanzamos en las respuestas a nuestros interrogantes cuando modelamos un dominio, sino mediante la disolución de las contradicciones que existen entre los términos (conceptos) ya conocidos y, en último caso, mediante la reducción de su número despejando aquellos conceptos que no añaden valor a la comprensión de dicho dominio.

Reconsiderar lo obvio, desenmascarar presunciones, desambiguar conceptos conocidos, todo en busca de la *simplicidad* y la *usabilidad*.
La tecnología orientada a objetos persigue el antiguo principio del divide y vencerás. Su objetivo es descomponer la *complejidad* en partes más manejables y comprensibles. No parece que esto sea algo novedoso con respecto a la tradicional descomposición funcional de los métodos estructurados. Sin embargo, la gran diferencia reside en aplicar la dualidad *estructura-función* en pequeñas unidades capaces de comunicarse y reaccionar en base a la aparición de una serie de eventos. El esquema dominante de la separación de estructuras de datos y funciones (bases de datos y programas) está amenazado pero aún se resiste a desaparecer.

Mucha gente cree que la principal utilidad de la orientación a objetos es la *reutilización del código* para conseguir un desarrollo más rápido de las

aplicaciones (Rapid Application Development). Si hay algo que caracteriza un entorno de desarrollo actual es la constante del *cambio*. Todo proyecto que sobrepase los tres meses está amenazado por la aparición de nuevas plataformas más exigentes, la extinción de herramientas sin previo aviso y, de manera sistemática, por la rotación del personal crítico encargado del proyecto. También está sometido, como no, a los cambios de requerimientos del cliente que a su vez están plenamente justificados por la readaptación de sus procesos de negocio a un mercado inestable.

Ante este cuadro de incertidumbre, el mayor desafío de una metodología de desarrollo es su *adaptación* para el cambio. Esto significa crear modelos que faciliten la *comunicación* entre todos los *agentes involucrados* en el sistema en construcción; que hagan visible la *trazabilidad* entre la concepción de los componentes, su especificación, implementación e instalación; significa el diseño de arquitecturas que faciliten la gestión de las dependencias entre estos componentes, que sean en fin, fácilmente *reemplazables* por otros más optimizados o bien por componentes que aporten una mayor funcionalidad y/o facilidad de uso.

La dinámica de cambio no se desarrolla en la ingeniería del software con la misma velocidad vertiginosa con que nos tiene acostumbrados la tecnología del hardware. La clave reside en que a diferencia de la electrónica, en los dominios del desarrollo de software no existe un *vocabulario unificado*. Desde la fase de concepción de un sistema a la instalación de sus componentes hay que *mapear* entre sí una gran diversidad de lenguajes orientados al análisis, diseño, código ejecutable, esquemas de bases de datos, componentes de páginas web, entre otros. Esta distancia entre la concepción y la usabilidad de un producto o de un proceso de negocio, exige cada vez más la capacidad de cooperación y comunicación de un equipo interdisciplinar muy especializado. Esta guía visual de UML está pensada para facilitar este proceso cooperativo y para ayudar a establecer una buena práctica fundamentada en un lenguaje común.

2.4.1 Elementos en UML

El UML tiene cuatro tipos de elementos: elementos estructurales, de comportamiento, de agrupación y de anotación [2].

2.4.1.1 Elementos estructurales

Los elementos estructurales son en su mayoría las partes estáticas de un modelo, representan cosas conceptuales o materiales. En total hay siete tipos de elementos estructurales: Clases, Interfaces, Colaboraciones, Casos de uso, Clases activas, Componentes y Nodos.

Clases. Son un conjunto de objetos que comparten los mismos atributos, operaciones relaciones y semántica. Gráficamente se representa como un rectángulo que contiene su nombre, atributos y operaciones. Ver figura 2.4.1.

Figura 2.4.1: Representación de una clase en UML.

Interfaz. Son un conjunto de operaciones que especifican un servicio de una clase o componente. Ver figura 2.4.2.

Figura 2.4.2: Representación de una interfaz en UML.

Colaboración. Define una interacción y es una sociedad de roles y otros elementos que colaboran para producir un comportamiento cooperativo mayor. Una clase puede participar en varias colaboraciones. Ver figura 2.4.3.

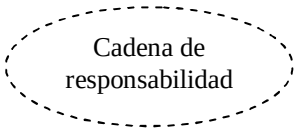

Figura 2.4.3: Representación gráfica de colaboraciones en UML.

Casos de uso. Descripción de un conjunto de secuencias de acciones que un sistema ejecuta y produce un resultado observable de interés para un actor en particular. Ver figura 2.4.4.

Figura 2.4.4: Representación gráfica de casos de uso.

Los elementos restantes: clases activas, componentes y nodos son similares a las clases, en el sentido de que también describen un conjunto de objetos que comparten los mismos atributos, operaciones, relaciones y semántica. Sin embargo se justifica su tratamiento especial por las diferencias necesarias para modelar ciertos aspectos de un sistema orientado a objetos.

Un **actor** es una entidad externa al sistema que realiza algún tipo de interacción con el mismo. Se representa mediante una figura humana. Esta representación sirve tanto para actores que son personas como para otro tipo de actores (otros sistemas, sensores, etc.).

Clases activas. Es una clase cuyos objetos tienen uno o más procesos de ejecución y por lo tanto pueden dar origen a actividades de control, es decir, una clase activa es igual que una clase, excepto que sus objetos representan elementos cuyo comportamiento es concurrente con otros elementos. Gráficamente se representa como la figura 2.4.5.

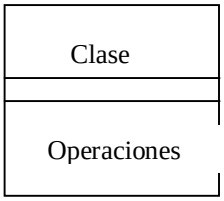

Figura 2.4.5: Una clase activa.

Componente. Parte física y reemplazable de un sistema que conforma un conjunto de interfaces y proporciona la implementación de dicho conjunto. Un componente representa típicamente el empaquetamiento físico de diferentes elementos lógicos, como clases, interfaces y colaboraciones. Ver la figura 2.4.6.

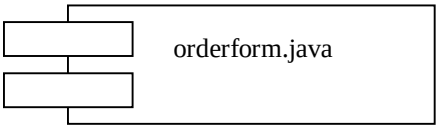

Figura 2.4.6: Componentes.

Nodo. Elemento físico que existe en tiempo de ejecución y representa un recurso computacional, que por lo general dispone de memoria y con frecuencia capacidad de procesamiento. Un conjunto de componentes puede residir en un nodo y puede también migrar de un nodo a otro. Ver figura 2.4.7.

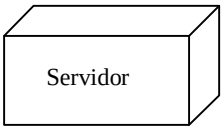

Figura 2.4.7: Nodos.

Existen variaciones de estos siete elementos, tales como actores, señales, utilidades (tipos de clases), procesos e hilos (tipos de clases activas) y aplicaciones, documentos, archivos, bibliotecas, páginas, tablas (tipos de componentes).

2.4.1.2 Elementos de comportamiento

Los elementos de comportamiento son las partes dinámicas de los modelos UML. Estos son los verbos de un modelo y representan comportamiento en el tiempo y el espacio. Hay dos tipos principales de elementos de comportamiento: Interacción y Máquina de estados [2].

Interacción. Es un comportamiento que comprende un conjunto de mensajes intercambiados entre un conjunto de objetos, dentro de un contexto particular para alcanzar un propósito específico. Una interacción involucra muchos otros elementos incluyendo mensajes, secuencias de acción (comportamiento invocado por un mensaje) y enlaces (conexiones entre objetos). Gráficamente se representa por una línea dirigida con el nombre de la operación ver figura 2.4.8.

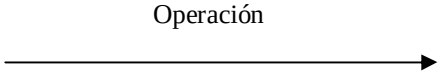

Figura 2.4.8: Interacción (Mensajes).

Máquina de estados. Es un comportamiento que especifica las secuencias de estados por las que pasa un objeto o una interacción durante su vida en respuesta a eventos, junto con sus reacciones a los mismos. Una Máquina de estados involucra estados, transiciones (flujo de un estado a otro), eventos (que disparan una transición) y actividades (la respuesta a una transición). (Ver fig. 2.4.9).

Figura 2.4.9: Estados.

2.4.1.3 Elementos de agrupación

Los elementos de agrupación son las partes encargadas de la organización de los modelos UML. Hay un solo elemento de agrupación principal, los paquetes.

Paquete. Es un mecanismo de propósito general para organizar elementos en grupos. En un paquete pueden incluirse elementos estructurales, elementos de comportamiento, e incluso otros elementos de agrupación. Un paquete es exclusivamente conceptual, es decir, sólo existe en tiempo de desarrollo y no en tiempo de ejecución. El gráfico que lo representa se muestra en la figura 2.4.10.

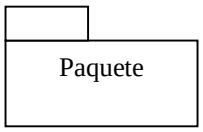

Figura 2.4.10: Paquetes.

2.4.1.4 Elementos de anotación

Los elementos de anotación son las partes explicativas de los modelos UML. Son comentarios que se pueden aplicar para describir, clarificar y hacer observaciones sobre cualquier elemento de un modelo. Hay un tipo de anotación llamado nota.

Nota. Es un símbolo para mostrar restricciones y comentarios junto a un elemento o una colección de elementos. El gráfico que lo representa se muestra en la figura 2.4.11.

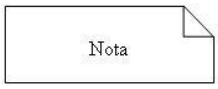

Figura 2.4.11: Notas.

2.4.2 Relaciones

Hay cuatro tipos de relaciones en UML: Dependencia, Asociación, Generalización y Realización.

2.4.2.1 Dependencia.

Es una relación semántica entre dos elementos, en la cual un cambio a un elemento (elemento independiente) puede afectar a la semántica del otro elemento (elemento dependiente). Ver la figura 2.4.12.

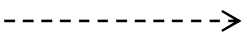

Figura 2.4.12: Dependencia.

2.4.2.2 Asociación.

Es una relación estructural que describe un conjunto de enlaces las cuales son conexiones entre objetos. Ver la figura 2.4.13.

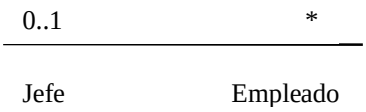

Figura 2.4.13: Asociaciones.

2.4.2.3 Generalización.

Relación de especialización/generalización en la cual los objetos del elemento especializado (hijo) pueden sustituir a los elementos del objeto general (padre). De esta forma el hijo comparte la estructura y el comportamiento del padre. Ver la figura 2.4.14.

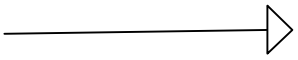

Figura 2.4.14: Generalizaciones.

2.4.2.4 Realización.

Relación semántica entre clasificadores, en donde un clasificador especifica un contrato que otro clasificador garantiza que cumplirá. Las relaciones de realización pueden encontrarse entre las interfaces y las clases y componentes que las realizan, y entre los casos de uso y las colaboraciones que los realizan. Ver la figura 2.4.15.

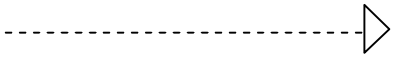

Figura 2.4.15: Realización.

También existen variaciones de estos cuatro elementos relacionales.

2.4.3 Diagramas en UML

Un diagrama es la representación gráfica de un conjunto de elementos, visualizado la mayoría de veces como un grafo conexo de nodos y arcos (elementos y relaciones). Los diagramas se dibujan para visualizar un sistema desde diferentes perspectivas, de forma que un diagrama es una proyección de un sistema. En teoría un diagrama puede contener cualquier combinación de elementos y relaciones. En la práctica, sin embargo, sólo surge un pequeño número de combinaciones [2].

UML incluye nueve tipos de diagramas, esta no es una lista cerrada, sin embargo son los que con mayor frecuencia aparecerán en la práctica:

2.4.3.1 Diagrama de clases.

Muestra un conjunto de clases, interfaces y colaboraciones, así como sus relaciones. Son el tipo de diagramas más comunes en el modelado de sistemas orientados a objetos. Este tipo de diagramas son estáticos.

2.4.3.2 Diagrama de objetos.

Muestra un conjunto de objetos y sus relaciones, representan instantáneas de instancias de los elementos encontrados en los diagramas de clases. Es un tipo de diagrama estático.

2.4.3.3 Diagrama de casos de uso.

Muestra un conjunto de casos de uso y actores (tipo especial de clases) y sus relaciones. Es un diagrama estático. Estos diagramas son especialmente importantes en el modelado y organización del comportamiento de un sistema.

2.4.3.4 Diagrama de secuencia.

Es un diagrama de interacción, que resalta la ordenación temporal de los mensajes. Un diagrama de interacción, como su nombre lo indica, muestra una interacción que consta de un conjunto de objetos y sus relaciones, incluyendo los mensajes que pueden ser enviados entre ellos. Los diagramas de interacción son dinámicos.

2.4.3.5 Diagrama de colaboración.

Al igual que el diagrama de secuencia, el diagrama de colaboración es un diagrama de interacción, que resalta la organización estructural de los objetos que envían y reciben mensajes. Los diagramas de secuencia y de colaboración son isomorfos, es decir, se puede tomar uno y transformarlo en otro.

2.4.3.6 Diagrama de estados (statechart).

Muestra una máquina de estados, que consta de estados, transiciones, eventos y actividades. Son de tipo dinámico, resaltan el comportamiento dirigido por eventos de un objeto.

2.4.3.7 Diagrama de actividades.

Es un tipo especial de diagrama de estados que muestra el flujo de actividades dentro de un sistema. Son de tipo dinámico y resaltan el flujo de control entre objetos.

2.4.3.8 Diagrama de componentes.

Muestra la organización y las dependencias entre un conjunto de componentes. Son de tipo estático, se relacionan con los diagramas de clases, en que un componente se corresponde por lo común, con una o más clases, interfaces o colaboraciones.

2.4.3.9 Diagrama de despliegue.

Muestra la configuración de nodos de procesamiento en tiempo de ejecución y los componentes que residen en ellos. Son de tipo estático, se relacionan con los diagramas de componentes en que un nodo incluye uno o más componentes.

III

MODELO DIN MICO

CAPÍTULO III. MODELO DINÁMICO

3.1 Robots manipuladores

Existen varias definiciones de robots manipuladores industriales. De acuerdo a la definición adoptada por la Federación Internacional de Robótica bajo la norma ISO/TR 8373, un robot manipulador se define de la siguiente manera:

"Un robot manipulador industrial es una máquina manipuladora con varios grados de libertad controlada automáticamente, reprogramable y de múltiples usos, pudiendo estar en un lugar fijo o móvil para su empleo en aplicaciones industriales" [6].

Los robots manipuladores son sistemas mecánicos articulados formados por eslabones conectados entre sí a través de uniones o articulaciones, formando un "brazo" o una "mano" para tomar objetos y herramientas, siendo capaces de realizar una gran cantidad de operaciones físicas (Ver figura 3.1). Dichas articulaciones son básicamente de dos tipos: **rotacionales** y **traslacionales**.

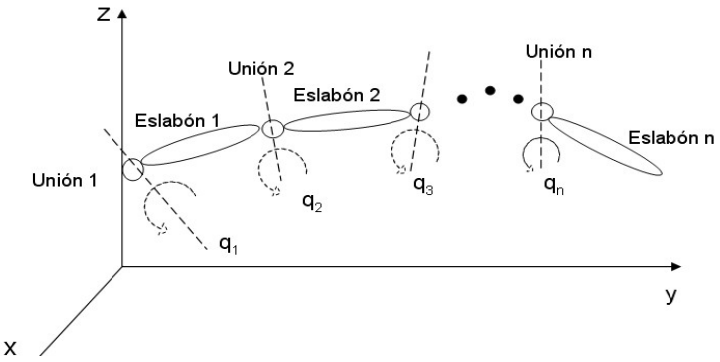

Figura 3.1: Diagrama abstracto de un robot manipulador de n grados de libertad (g.d.l.).

Habitualmente, se coloca un marco de referencia cartesiano de tres dimensiones en cualquier lugar de la base del robot, denotado por las coordenadas $[x \quad y \quad z]^T$. Los eslabones se numeran consecutivamente desde la base (eslabón 1) hasta el final (eslabón *n*).

Las uniones son los puntos de contacto entre los eslabones y se numeran de tal forma que la unión *i* conecta los eslabones *i* e *i-1*. Cada unión se controla independientemente a través de un accionador que se coloca generalmente en dicha unión, y el movimiento de las uniones produce el movimiento relativo de los eslabones.

Denotaremos por z_i al eje de movimiento de la unión *i*. La coordenada articular generalizada denotada por q_i es el desplazamiento angular alrededor de z_i si la unión *i* es traslacional. En el caso típico donde los accionadores se localizan en las uniones entre los eslabones, las coordenadas articulares generalizadas reciben el nombre de ***posiciones articulares***.

Las posiciones articulares correspondientes a cada articulación del robot, se agrupan en el vector de posiciones articulares ***q***. Para un robot de *n*-articulaciones, es decir de *n* - g.d.l., el vector de posiciones articulares ***q***, tendrá *n*-elementos:

$$q = [q_1, q_2, ..., q_n]^T \tag{1}$$

Por otro lado, la determinación de la posición y orientación del dispositivo terminal del robot es de gran interés puesto que este dispositivo realiza la tarea encomendada al robot. Dicha posición y orientación se expresa en términos del marco de referencia coordenado cartesiano *(x,y,z)* , así como en ángulos de Euler.

Dichas coordenadas son agrupadas en el vector *x* de posiciones cartesianas:

$$x = [x_1, x_2, ..., x_m]^T \quad \text{donde } m \leq n. \tag{2}$$

3.2 Modelo dinámico

El modelo dinámico de un sistema es aquel que representa los aspectos del sistema que tienen que ver con el tiempo y los cambios. Para los efectos prácticos

del presente trabajo, nos basaremos en el modelo dinámico de un robot manipulador de tres grados de libertad, el cual se muestra en la siguiente figura:

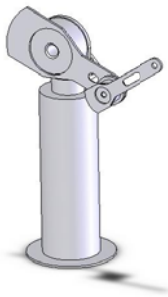

Figura 3.2: Robot manipulador de tres grados de libertad.

Las partes principales de este tipo de robots, son la base, el hombro y el codo, que a su vez son los que definen sus grados de libertad. El modelo dinámico de un sistema es la regla matemática que vincula las variables de entrada y salida del sistema, éste puede obtenerse tradicionalmente por una de las dos técnicas siguientes [6]:

- *Analítica*. Este procedimiento se basa en las ecuaciones de la física que rigen el comportamiento del sistema. Esta metodología puede proporcionar un modelo matemático preciso a condición de dominar las leyes de la física que están involucradas en el sistema.
- *Experimental*. Este procedimiento requiere una serie de datos experimentales del sistema. Frecuentemente se trata de examinar el comportamiento del sistema ante entradas específicas. El modelo obtenido a partir de este procedimiento es, en general, más impreciso que el conseguido a partir del método analítico. No obstante, su principal ventaja radica en la facilidad y el corto espacio de tiempo requerido para disponer del modelo.

En algunas ocasiones se procede a una simplificación del modelo del sistema que desea controlarse con miras a obtener posteriormente un sistema de

control relativamente sencillo. Esta etapa puede, no obstante, tener la desventaja de dar como resultado un sistema de control que funcione inadecuadamente, fenómeno conocido como *falta de robustez*[1].

El modelado dinámico de robots manipuladores se realiza tradicionalmente de forma analítica, esto es, a partir de las leyes de la física. Debido a la naturaleza mecánica de los robots manipuladores, las leyes de la física involucrada son simplemente las leyes de la mecánica. Desde el punto de vista de los sistemas dinámicos, un robot manipulador de *n* grados de libertad, puede ser considerado como un sistema no lineal multivariable, teniendo n entradas (los pares y las fuerzas que son aplicados en las articulaciones por medio de accionadores electromecánicos, hidráulicos o neumáticos) y *2n* variables de estado, normalmente asociadas con las *n* posiciones ***q*** y *n* velocidades de las articulaciones. Mostrándolo en un diagrama a bloques, y suponiendo que las variables de estado son a su vez las variables de salida, esto seria:

Figura 3.3: Diagrama a bloques del sistema

Los modelos dinámicos de los robots manipuladores son generalmente caracterizados por ecuaciones diferenciales ordinarias no lineales y no autónomas. Este hecho tiene como consecuencia, que las técnicas de diseño tradicionales para el control de sistemas lineales tengan una aplicación limitada en la síntesis de controladores con alto desempeño para robots manipuladores.

Debido a esto, y a los requerimientos actuales de alta precisión y rapidez en los movimientos de los robots, se ha hecho necesario el uso de técnicas más elaboradas de control para el diseño de controladores con mayores prestaciones.

[1] Entendamos la robustez como el grado de capacidad que presenta un sistema o un componente para funcionar correctamente frente a entradas de información erróneas, o carga de trabajo elevada.

Esta clase de sistemas de control pueden o no incluir, por ejemplo, controles no lineales y controles adaptables.

El modelo dinámico del robot de 3 grados de libertad que se esta tratando, fue realizado en base a la metodología Euler-Lagrange (Apédice A), para la cual, es necesario calcular en primer lugar las expresiones para la energía cinética y potencial correspondientes al sistema bajo estudio. Posteriormente se emplean los resultados obtenidos para calcular el lagrangiano, y finalmente se manipula mediante las ecuaciones de movimiento de Euler-Lagrange para obtener el modelo dinámico buscado.

La energía cinética de un cuerpo se debe a su movimiento relativo con respecto a un sistema de referencia espacial no inercial, y está dada por la mitad del producto de la masa de dicho cuerpo y su velocidad al cuadrado. La energía potencial, a su vez, está definida como aquella que un cuerpo posee en virtud de su posición en el espacio (nuevamente, respecto de un sistema ortogonal no inercial), y se calcula como el producto resultante de tres factores, que son la masa del cuerpo, su altura con respecto al nivel de referencia y la constante de aceleración de la gravedad.

Considérese el robot antropomórfico presentado en la figura 3.2, constituido por una cadena cinemática abierta de tres grados de libertad que se pueden comparar con el cuerpo humano (de ahí el calificativo antropomórfico).

El primer grado de libertad es aquel que hace girar a la base o cintura (uno de cuyos extremos se encuentra anclado a un punto fijo en el espacio, y el otro unido al segundo eslabón); luego se tiene el segundo grado de libertad (que conecta a la base con el segundo eslabón) denominado también como hombro y finalmente está el tercer grado de libertad (asimismo llamado codo), que une al segundo eslabón con el tercero, uno de cuyos extremos se encuentra posicionado en algún punto del espacio de trabajo del manipulador.

Los ejes de referencia pertenecientes a cada articulación han sido asignados (empleando el procedimiento de Denavit-Hartenberg) siendo la posición que el robot mantiene en la figura aquella denominada como casa, home o posición inicial, según la literatura conocida.

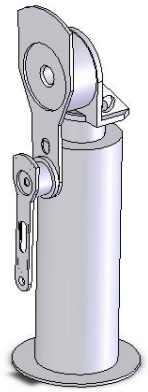

Figura 3.4: Posición de casa, *home* o inicial del robot antropomórfico.

Cabe mencionar que una característica de los robots antropomórficos es que el eje de giro de la base es ortogonal con los ejes de giro correspondientes al hombro y codo, que a su vez son paralelos entre sí.

La obtención de las expresiones que determinan la posición de cada elemento de masa se realiza mediante el análisis de la cinemática directa. Ésta consiste en las expresiones matemáticas (que en este caso son obtenidas mediante relaciones trigonométricas) que relacionan las coordenadas en el espacio de trabajo con los valores de las coordenadas articulares del sistema; o sea:

(3)

Dichas posiciones sirven de manera casi inmediata al cálculo de la energía potencial de cada elemento de masa componente del sistema. Para el cálculo de la energía cinética, dado que se requiere conocer la velocidad de cada elemento de masa, es necesario derivar la cinemática directa con respecto al tiempo, lo que se puede expresar como:

(4)

Una vez que se obtienen las expresiones para las funciones y que denotan la energía cinética y potencial, se puede calcular el lagrangiano del robot, definido como: la energía cinética del robot menos su energía potencial:

(5)

Y finalmente se aplican las ecuaciones de movimiento de Lagrange [9] dadas por:

(6)

Donde se refiere a un vector de pares de fricción pertenecientes a cada articulación, así como las fuerzas no conservativas. Esta ecuación puede escribirse en forma vectorial, lo que permite expresar convenientemente el modelo dinámico de una forma compacta como:

(7)

Con lo que obtenemos los elementos que describen el modelo dinámico del robot. A continuación se presentan cada uno de los elementos que forman parte del modelo dinámico de un robot antropomórfico de 3 grados de libertad.

3.2.1 Matriz de Inercias

Puede considerarse que la masa de un cuerpo representa de un modo cuantitativo la propiedad de la materia que se describe cualitativamente con la palabra inercia. [12] Cuando es necesario una gran fuerza para aumentar o disminuir la velocidad de un cuerpo, o bien para desviarlo lateralmente si esta moviéndose, la masa del cuerpo es grande. En el lenguaje ordinario diríamos que

el cuerpo tiene una gran inercia. Si solo es necesaria una fuerza pequeña por unidad de aceleración, la masa es pequeña y la inercia es pequeña.

La inercia es la propiedad de los cuerpos que hace que éstos tiendan a conservar su estado de reposo o de movimiento. La inercia es una propiedad mensurable. [11]

Para el caso del modelo dinámico de un robot manipulador se debe tomar en cuenta la inercia, y está representada mediante la llamada matriz de inercia, que es una matriz simétrica definida positiva de *n x n* cuyos elementos son funciones solamente de ***q.*** la matriz de inercia juega un papel importante en el modelado dinámico ya que la matriz de inercia se encuentra íntimamente relacionada con energía cinética.

Los elementos de la matriz *M*(*q*), que en este caso es una matriz de 3x3, se anotan a continuación:

Que son los elementos de la matriz de inercia, *M(q):*

(8)

3.2.2 Matriz de Fuerzas Centrífugas y de Coriolis

La matriz de fuerzas centrífugas y de Coriolis es una matriz *n x n* cuyos elementos son funciones de y . Además tiene algunas propiedades entre las cuales están:

1. La matriz puede no ser única, pero el vector es único.
2. para todo vector .

La matriz de fuerzas centrífugas y de Coriolis se puede calcular a partir de la matriz de inercias mediante los llamados símbolos de Christoffel, mediante:

(9)

Donde denota el ij-ésimo elemento de la matriz de inercia . El kj-ésimo elemento de la matriz puede obtenerse de la siguiente manera:

(10)

Por ejemplo, para obtener el elemento tenemos que:

(11)

Sustituyendo:

(12)

Derivando parcialmente y reduciendo:

(13)

Con lo que se obtiene:

(14)

Y haciendo todos los cálculos necesarios obtenemos que:

3.2.3 Vector de pares gravitacionales g

El vector de pares gravitacionales está presente en robots que no han sido diseñados desde el punto de vista mecánico, con compensación de pares de gravedad, por ejemplo, contrapesos o resortes, o para robots destinados a desplazarse fuera del plano horizontal.

El vector de pares gravitacionales de *n x 1* depende sólo de las posiciones articulares ***q***. El vector está acotado si ***q*** lo está también.

Para el caso del modelo dinámico del sistema de tres grados de libertad tenemos:

(15)

Como se observa en el vector de pares gravitacionales, el valor para el primer eslabón, que corresponde a la base, es cero.

3.2.4 Vector de fricciones

Los efectos de fricción en sistemas mecánicos son fenómenos complicados que dependen de múltiples factores como la naturaleza de los materiales en contacto, lubricación entre ellos, temperatura, etc. Una característica importante de las fuerzas de fricción es que éstas son disipativas.

Un modelo estático "clásico" de fricción es aquel que combina los denominados fenómenos de fricción viscosa, y de fricción Coulomb. Este modelo establece que el vector esta dado por:

(16)

Donde y son matrices de *n x n* diagonales definidas positivas. Los elementos de la diagonal corresponden a los parámetros de fricción viscosa mientras que los de a los de fricción de Coulomb.

3.2.4.1 Matriz de Fricción Viscosa B

La matriz de fricción viscosa está definida como sigue:

(17)

3.2.4.2 Matriz de Fricción F_c

La matriz de fricción de Coulomb está definida como sigue:

(18)

Finalmente el vector de fricciones con que trabajaremos en el modelo esta dado como sigue:

(19)

3.3 Variables de Estado

Podemos representar el sistema de otra forma, la llamada representación en variables de estados, que nos da la forma final del sistema y es como se muestra a continuación:

—

(20)

En este punto tenemos todos los elementos que conforman el modelo dinámico de un robot de 3 grados de libertad, ya sea que lo representemos en variables de estados o de acuerdo a la ecuación (7). El cálculo del modelo dinámico bajo la metodología Euler-Lagrange, a pesar de que su grado de dificultad, en cuanto a los cálculos se refiere, va aumentando conforme aumentan los grados de libertad del sistema, todo se resume a manipulación matemática.

I

DIAGRAMAS UML DEL SISTEMA DIN MICO

CAPÍTULO IV. DIAGRAMAS UML DEL SISTEMA DINÁMICO

Recordemos que el UML (de Unified Modeling Language) es un lenguaje gráfico que permite modelar, visualizar, especificar, construir y documentar los elementos que forman un sistema de software orientado a objetos. Para fines del presente trabajo se ha propuesto el modelado de un sistema dinámico con tres grados de libertad, perteneciente a un robot antropomórfico. Los diagramas que se presentan a continuación fueron elaborados en la herramienta visual para diagramas UML Visual Paradigm For UML 5.0 mostrado en la siguiente figura.

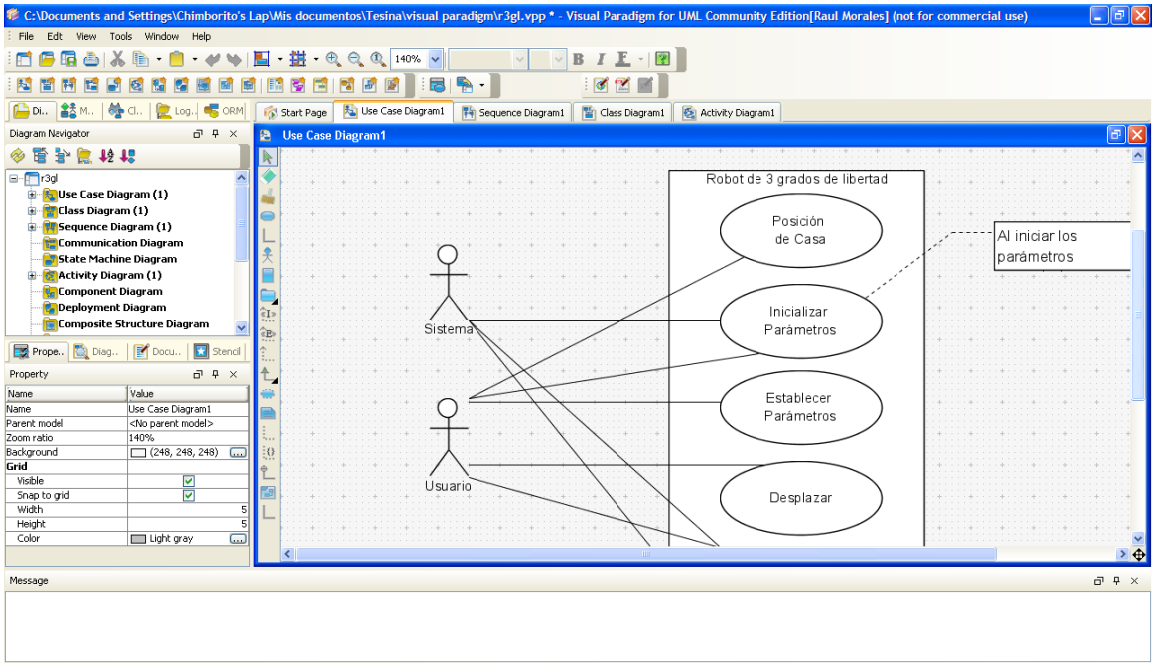

Fig.: Visual Paradigm 5.0

4.1 Diagrama de casos de uso

El primer diagrama que se muestra es el diagrama de casos de uso, un diagrama de casos de uso muestra la relación entre los actores y los casos de uso del sistema. Representa la funcionalidad que ofrece el sistema en lo que se refiere a su interacción externa. En el diagrama de casos de uso se representa también el sistema como una caja rectangular con el nombre en su interior. Los casos de uso están en el interior de la caja del sistema, y los actores fuera, y cada actor está

unido a los casos de uso en los que participa mediante una línea. En la Figura 4.1 se muestra el de diagrama de casos de uso para el sistema tratado.

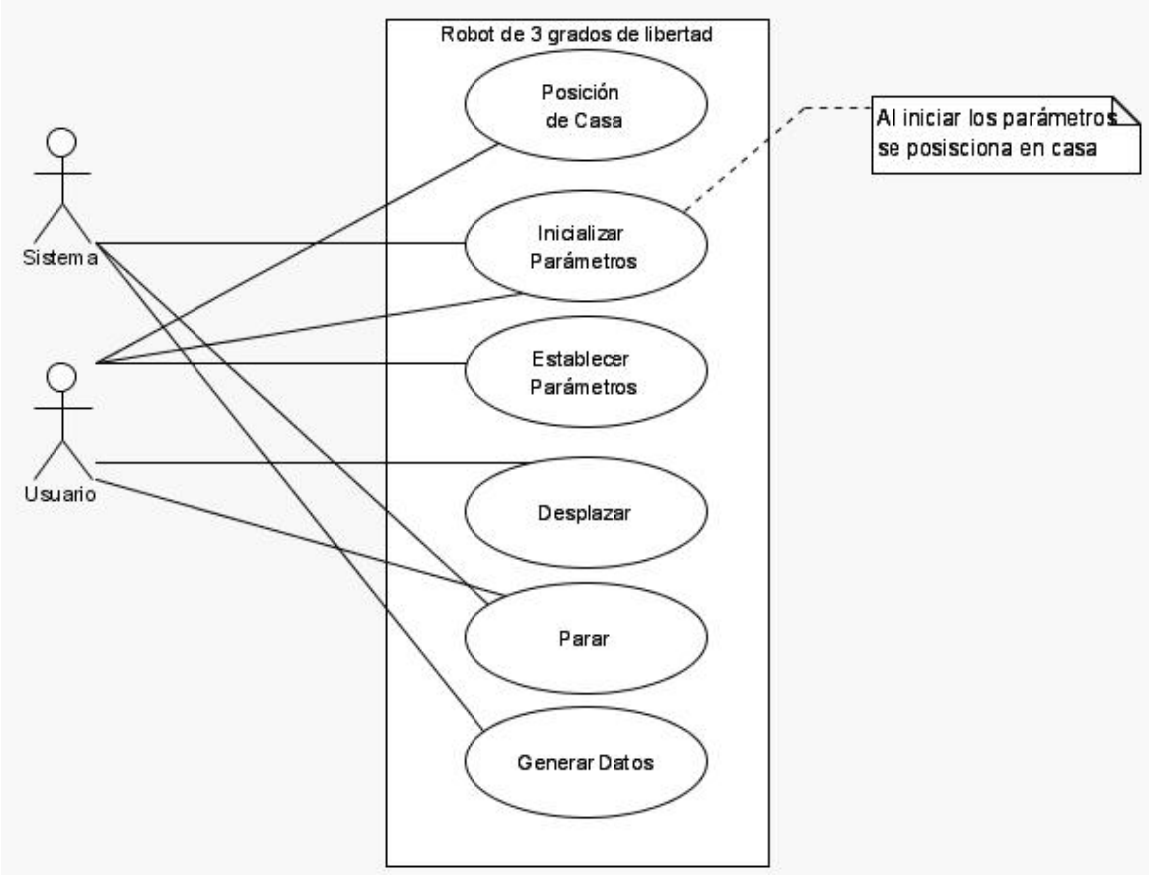

Figura 4.1: Diagrama de Casos de Uso

4.2 Diagrama de clases

Del análisis textual se obtuvieron las siguientes clases:

Brazo, Eslabon[1], Modelo, Controlador, RK, Friccion, Gravedad

En la figura 4.2 presentamos el diagrama de clases correspondiente. Las clases que se crearon para la implementación se muestran en la misma figura.

[1] Se omiten acentos ortográficos (eslabón y fricción) por compatibilidad con la mayoría de lenguajes de programación.

Podemos observar que se encapsula en un paquete el modelo dinámico, es decir, podemos verlo como un contenedor de los modelos de fricción, de gravedad, incluso el controlador. Es muy importante señalar que también se incluye una interface llamada RK, esta interface permite implementar el método de Runge-Kutta 4 para la solución de ecuaciones diferenciales, necesarias para llevar a cabo la simulación.

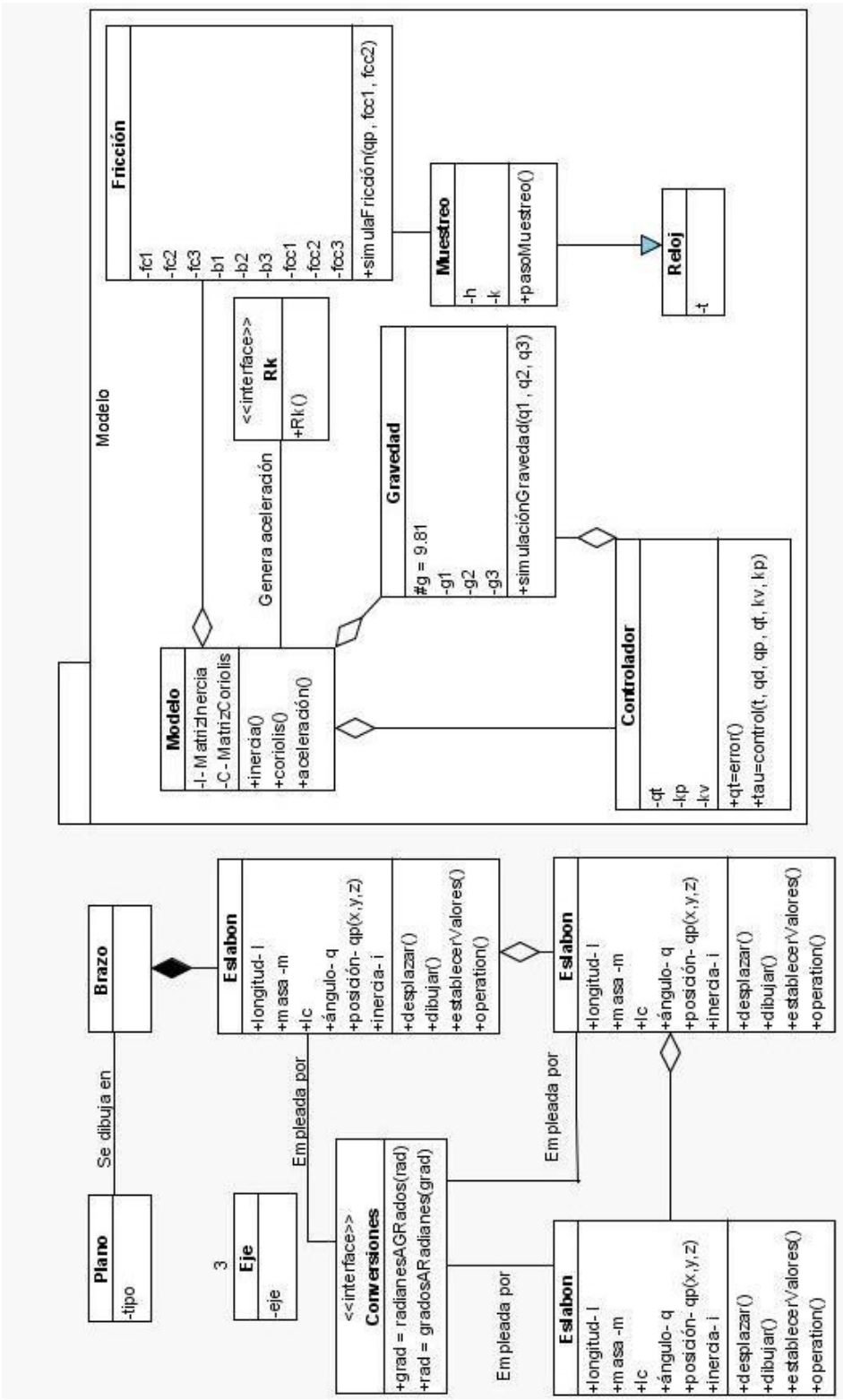

Figura 4.2: Diagrama de Clases

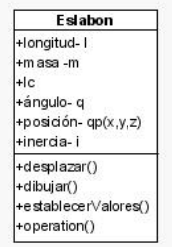

Figura 4.3: Clase Eslabon

La figura 4.3 nos muestra la clase Eslabon, la cual tiene como atributos, las características que tiene cada una de las articulaciones que conforman el robot, como son la longitud la masa, el ángulo, la inercia, etc. Como métodos de la clase tenemos las operaciones que nos ayudarán a dibujar el eslabón en el modo gráfico para mostrar los resultados en la pantalla.

En la figura 4.4 tenemos la clase Friccion, la cual tiene como atributos, los diferentes elementos que conforman la matriz de Coulomb y la matriz fricción viscosa, para formar el vector de fricciones. El método SimularFriccion() generarà la fricción para cada objeto de la clase eslabón que la utilice.

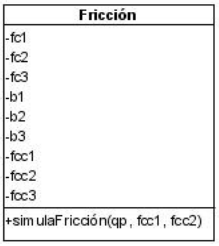

Figura 4.4: Clase Friccion

La clase Modelo que tenemos en la figura 4.4 muestra la clase que contendrá la matriz de inercia y la matriz de Coriolis, junto con sus métodos que nos ayudan a calcularlas.

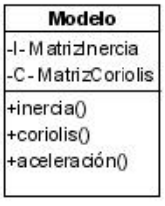

Figura 4.5: Clase Modelo

La clase de la figura 4.6 es la clase que nos proporciona la tau producto del uso de un controlador, como se verá mas adelante, este controlador es escogido por el usuario.

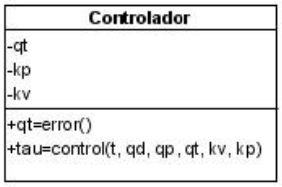

Figura 4.6: Clase Controlador

En la figura 4.6 tenemos la clase Gravedad con la cual se simulará la gravedad a utilizar en el modelo.

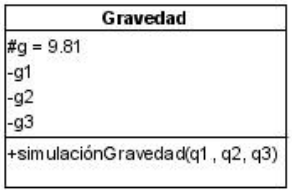

Figura 4.7: Clase Gravedad

La interface que se presenta en la figura 4.7, es la clase RK, la cual implementa el método Runge-Kutta 4, para la solución de ecuaciones diferenciales, que como se mencionó anteriormente, se utilizarán para la parte de la simulación.

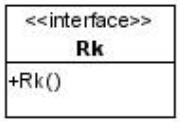

Figura 4.8: Clase RK

Todas las clases anteriores son agrupadas en un paquete, para su uso, el cual es mostrado en la figura 4.8. Este paquete es llamado modelo, dado que como su nombre lo indica, agrupa todas las clases que generan el modelo.

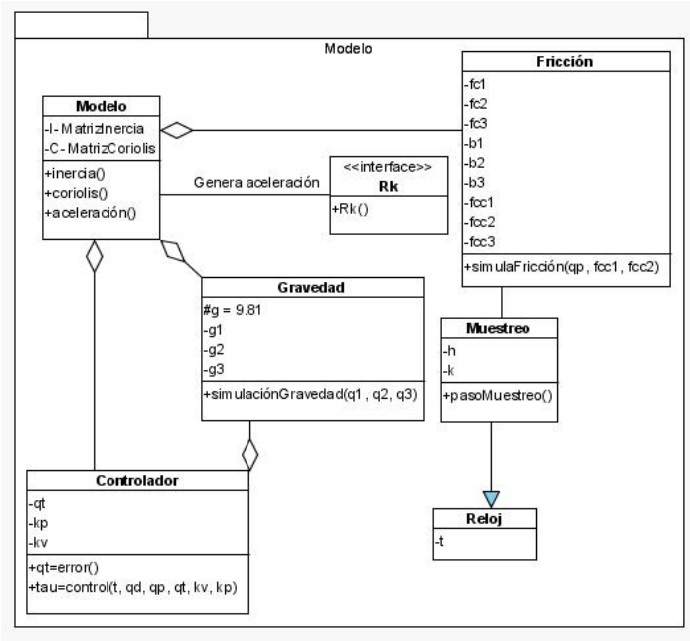

Figura 4.9: Paquete Modelo

Figura 4.10: Clase Conversiones

La figura 4.9 muestra la clase Conversiones, la cual es una interface que ayudará a realizar las conversiones de grados a radianes y de radianes a grados, las cuales son útiles al generar las tablas de resultados.

4.3 Diagrama de secuencias

Los diagramas de clase y de objetos representan información estática. Sin embargo en un sistema funcional los objetos interactúan entre sí y estas interacciones suceden conforme el tiempo transcurre. Un diagrama de secuencias UML muestra la mecánica de la interacción entre objetos y los mensajes que intercambian ordenados según su secuencia en el tiempo.

El tiempo se representa verticalmente, y horizontalmente se colocan los objetos y actores participantes en la interacción, sin un orden prefijado. El tiempo

fluye de arriba hacia abajo. Cada objeto o actor tiene una línea vertical discontinua, conocida como línea de vida. Véase la figura 4.11.

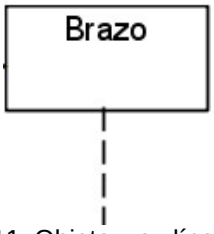

Figura 4.11: Objeto y su línea de vida.

El símbolo de activación del objeto, es un rectángulo vertical que reemplaza la línea de vida en el transcurso de la duración de la existencia de ese caso. Véase figura 4.12.

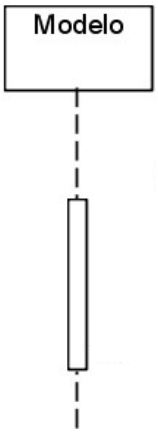

Figura 4.12: Objeto, su línea de vida y símbolo de activación del objeto.

Debemos tener en cuenta que un objeto se puede crear y destruir muchas veces y por ello se usa una línea de vida para representar todos los casos de esa clase en una secuencia.

4.4 Envío de mensajes

Los **mensajes** son líneas dirigidas que conectan líneas de vida. La línea se inicia en una línea de vida y la flecha apunta hacia aquella línea de vida que contenga el mensaje invocado. Véase la figura 4.13. El tipo de flecha permite diferenciar el tipo de mensaje.

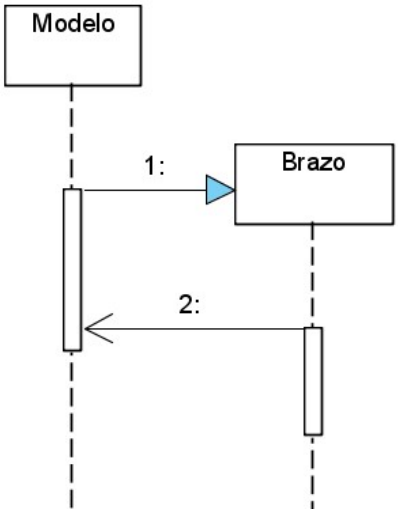

Figura 4.13: Envío de mensajes entre objetos.

4.4.1 Tipos de mensajes

Un mensaje puede ser simple, síncrono o asíncrono.

1. **Simple:** Se utiliza para transferir el control de un objeto a otro o cuando no se conocen los detalles del tipo de comunicación o no son relevantes en el diagrama. También se emplea para representar el retorno de un mensaje síncrono. Véase figura 4.14 a).
2. **Síncrono:** Representa la invocación de una operación en la cual el objeto invocante esperará la respuesta a tal mensaje antes de continuar con su trabajo. Véase figura 4.14 b).
3. **Asíncrono:** Este tipo de mensaje no esperará respuesta para continuar, es decir representa una invocación no bloqueante, cuando el objeto invocante continúa de inmediato su hilo de ejecución. Véase figura 4.14 c).

Figura 4.14: Envío de mensajes.

a) Simple. b) Síncrono. c) Asíncrono.

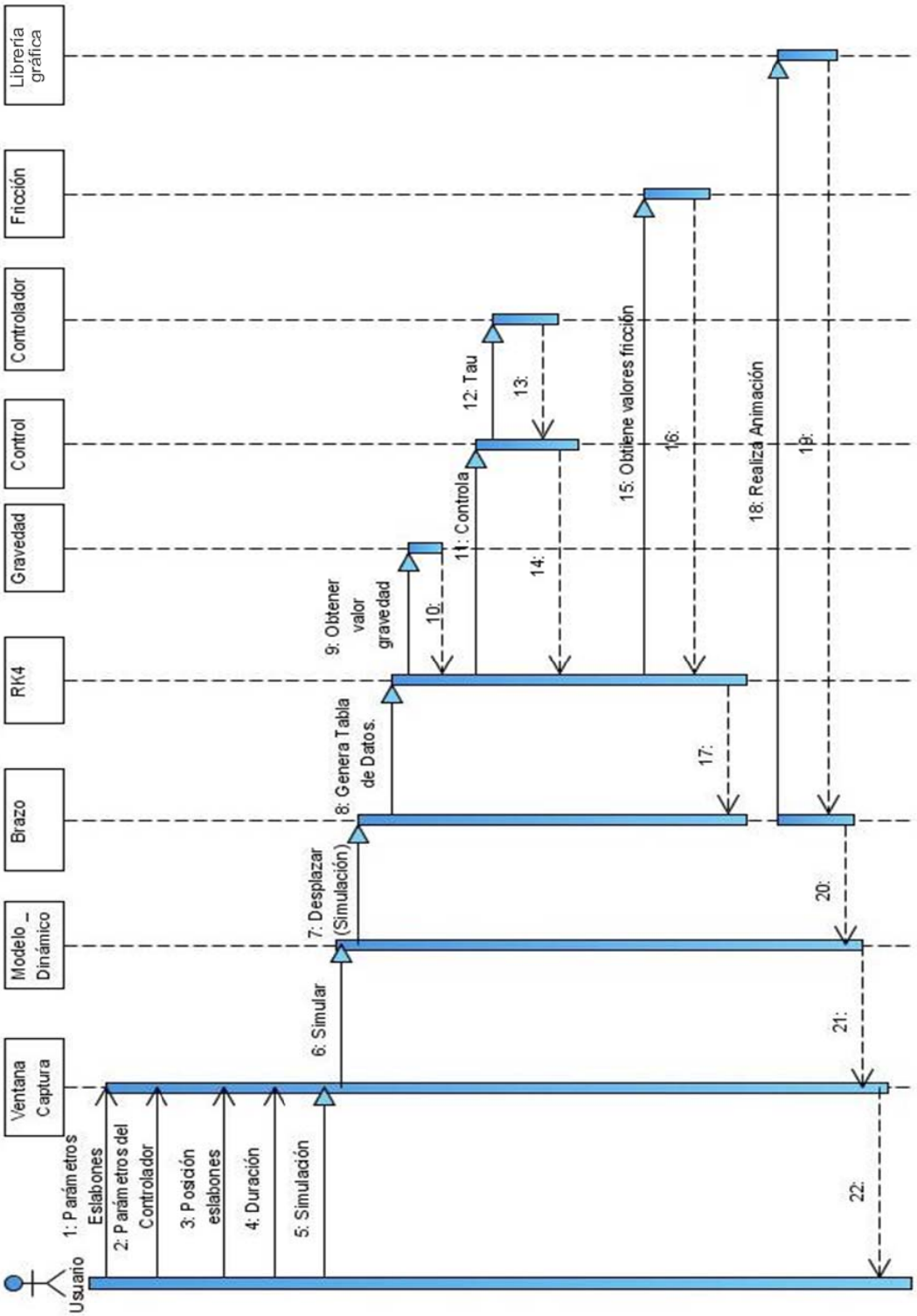

Figura 4.15: Diagrama de Secuencias

En nuestro caso de estudio, entre los componentes del simulador se encuentran el modelo, el brazo (compuesto por los eslabones), los simuladores de fricción, de gravedad, así como el controlador, etc.
En la figura 4.15 se presenta el diagrama de secuencias para el caso de uso desplazar.

4.5 Diagramas de actividades

Un diagrama de actividades se diseña para mostrar una visión simplificada de lo que ocurre durante una operación o proceso, es decir, muestra los pasos en dicho proceso.

A cada actividad se le representa por un rectángulo con las esquinas redondeadas. El procesamiento dentro de una actividad se lleva a cabo y al realizarse, se continúa con la siguiente actividad. Una flecha representa la transición de una actividad a otra. Se tiene un punto inicial (círculo relleno) y uno final (diana). Para representar un punto de decisión se hace por medio de un rombo con las rutas de decisión.

En este apartado presentamos varios diagramas de actividades que nos muestran los pasos a seguir de los objetos que conforman el software de simulación.

En primera instancia presentamos el diagrama de actividades principal. Véase la figura 4.16.

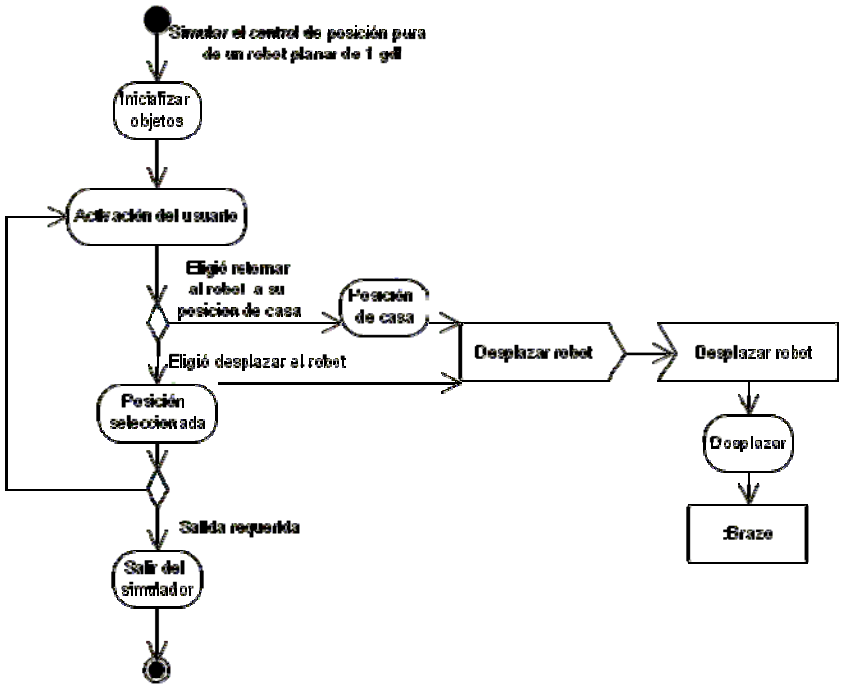

Figura 4.16: Diagrama de actividades principal.

Los siguientes diagramas de actividades muestran los pasos a seguir para llevar a cabo la conversión de radianes a grados (figura 4.17. a) y su recíproco (figura 4.17. b). Estos son parte de la interfaz ***Conversiones***.

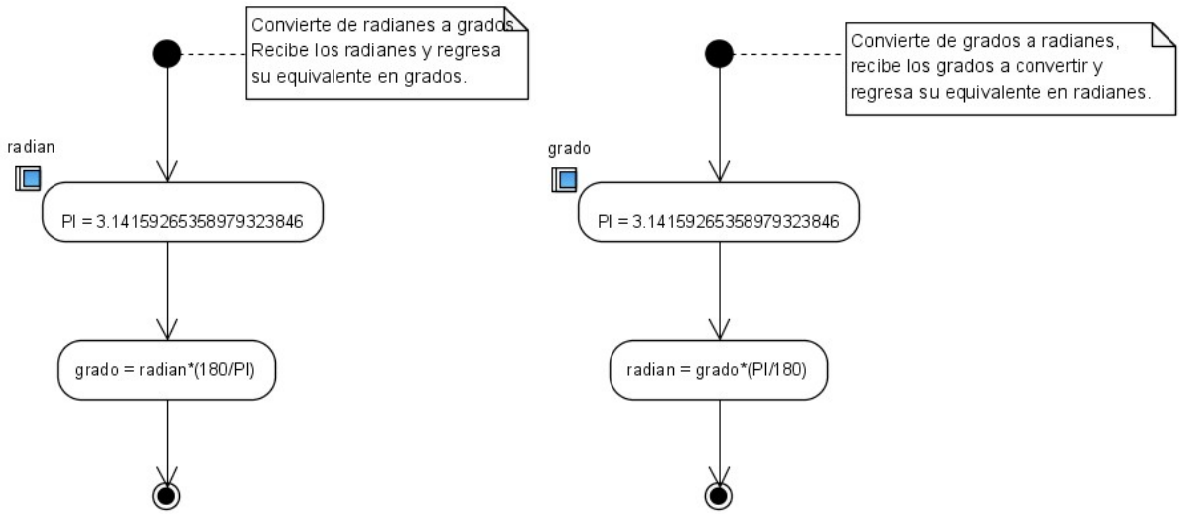

Figura 4.17: Diagrama de actividades ***Conversiones***.
a) Conversión de radianes a grados. b) Conversión de grados a radianes.

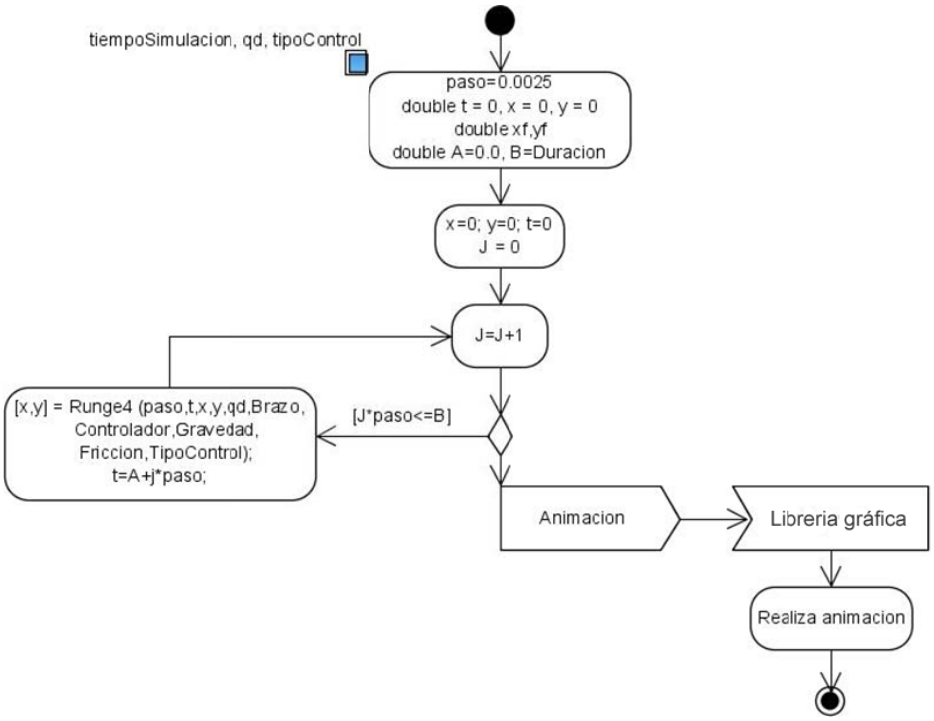

Figura 4.18. Diagrama de actividades simulación.

La figura 4.18 nos muestra el diagrama de actividades para realizar la simulación. Como podemos observar se generan los datos empleando la función Runge4 la cual es la implementación del método de Runge Kutta 4 y en seguida se llevaría a cabo la animación gráfica. La figura 4.19 nos ilustra el diagrama de actividades que permite simular la fricción. Como se observa esta se logra por medio de la función signo.

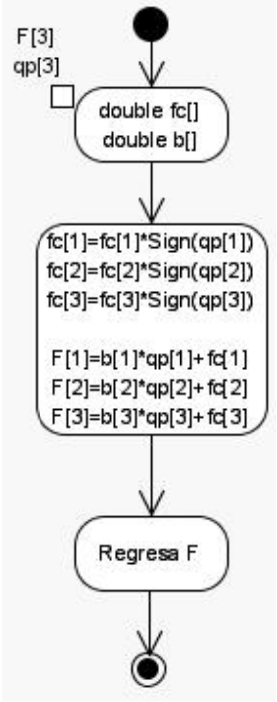

Figura 4.19: Simulación fricción

La simulación de gravedad podemos verla representada en el diagrama de actividades de la figura 4.20.

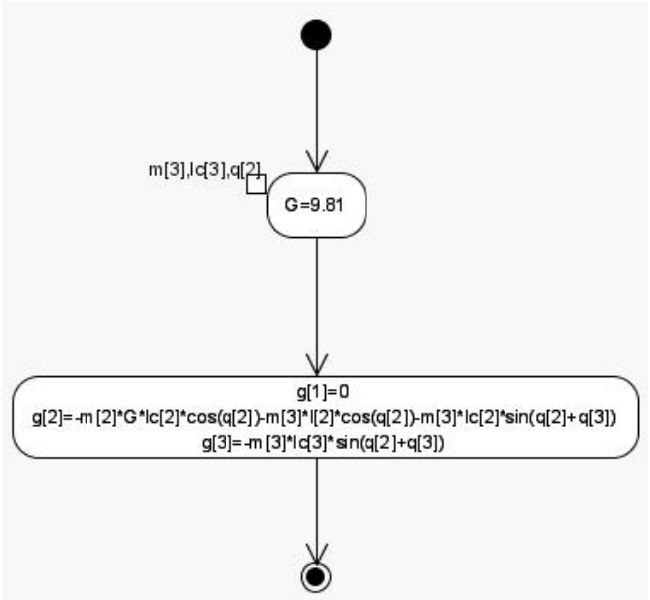

Figura 4.20: Simulación gravedad.

La figura 4.21 ilustra el diagrama de actividades del control. Como podemos ver este algoritmo emplea el simulador de gravedad, y posteriormente selecciona el controlador elegido con antelación por parte del usuario.

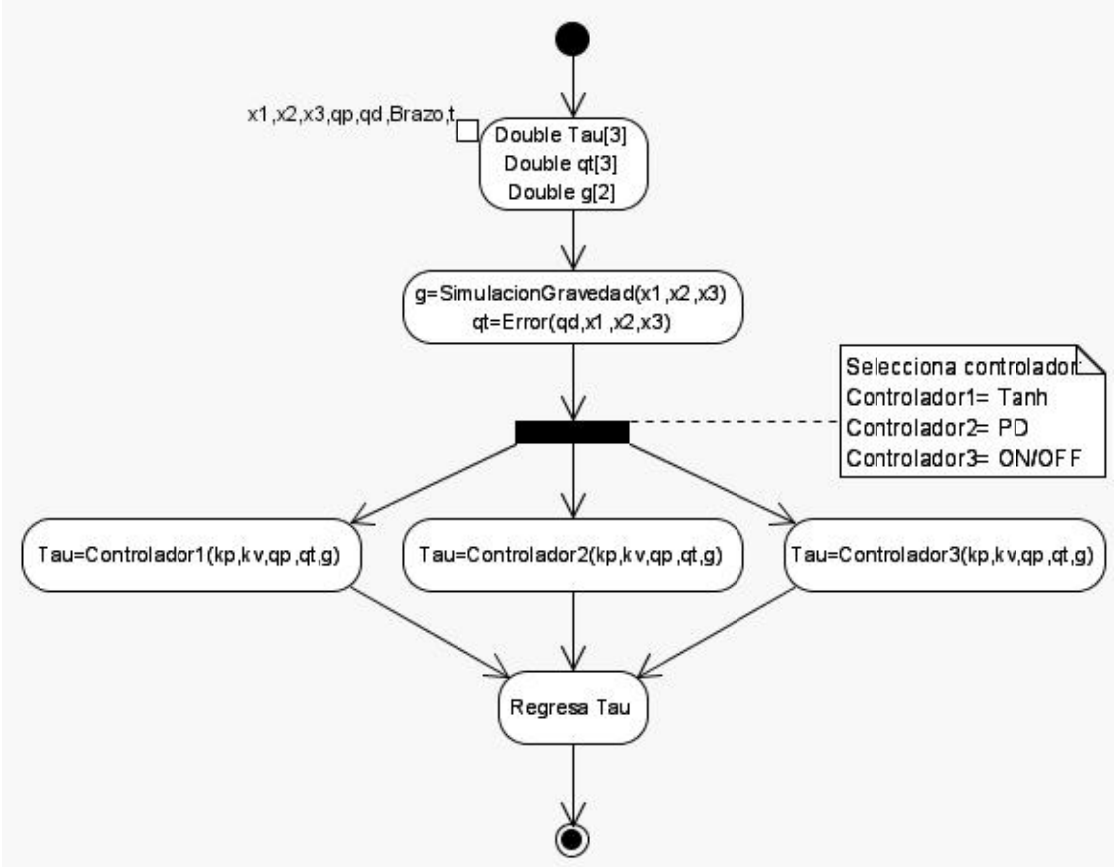

Figura 4.21: Diagrama de actividades control.

Las figuras 4.22, 4.23 y 4.24 muestran los diagramas de actividades de los distintos controladores empleados. En la figura 4.14 tenemos diagrama de actividades que corresponde al controlador: Tangente hiperbólica, la figura 4.15 la correspondiente al controlador: PD con compensación de gravedad y la figura 4.16 al controlador: ON / OFF.

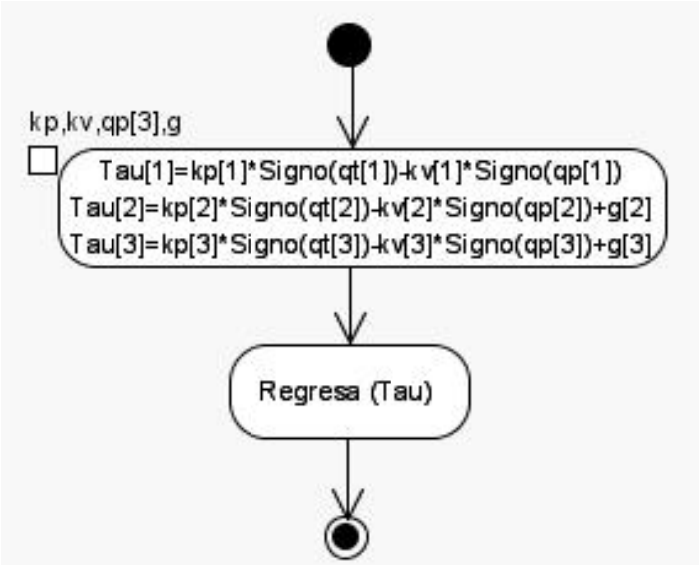

Figura 4.22: Diagrama de actividades controlador Tangente hiperbólica.

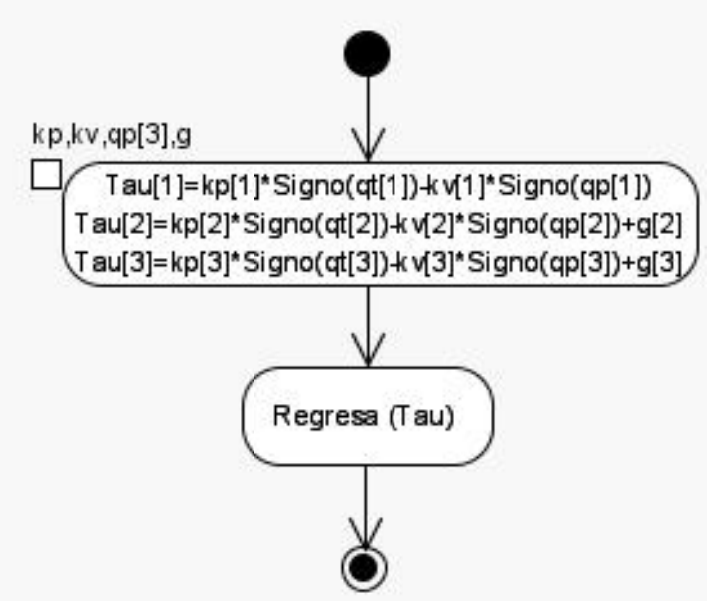

Figura 4.23: Diagrama de actividades controlador PD con compensación de gravedad.

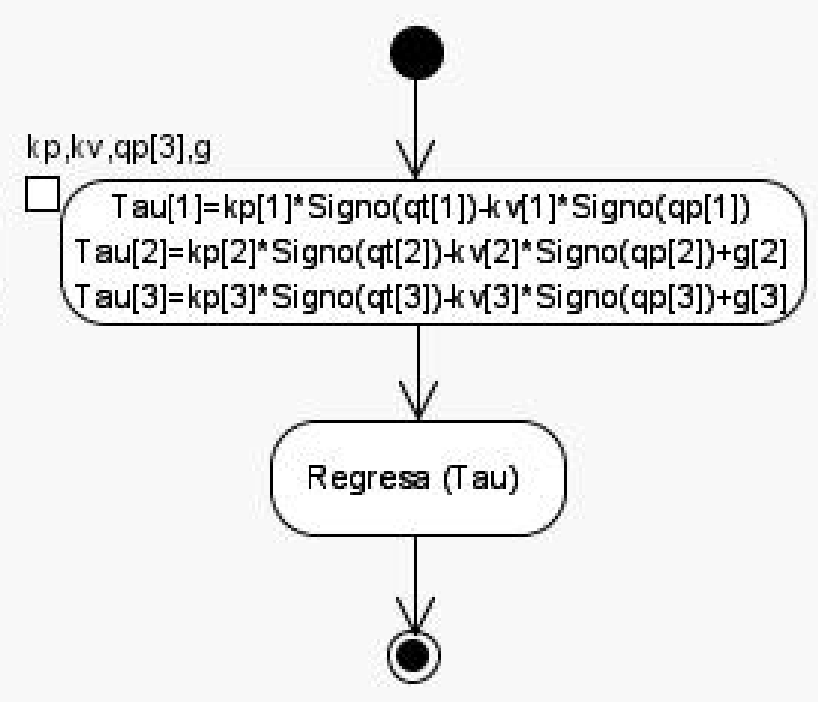

Figura 4.24: Diagrama de actividades controlador TODO/NADA.

Como ya hemos comentado anteriormente, para la resolución del sistema de ecuaciones diferenciales involucradas en el modelo dinámico del brazo robótico, se emplea el método de Runge Kutta, dicho método hace uso de dos funciones importantes: f1 y f2. Los diagramas de actividad ilustrados en las figuras 4.25 y 4.26, nos muestran los pasos a seguir para estas funciones en ese orden.
Para finalizar con los diagramas de actividades la figura 4.27 nos muestra los pasos para resolver el método de Runge Kutta.

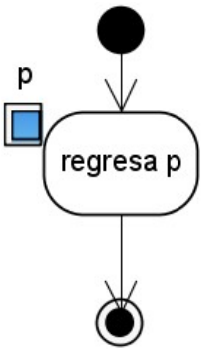

Figura 4.25: Diagrama de actividades f1.

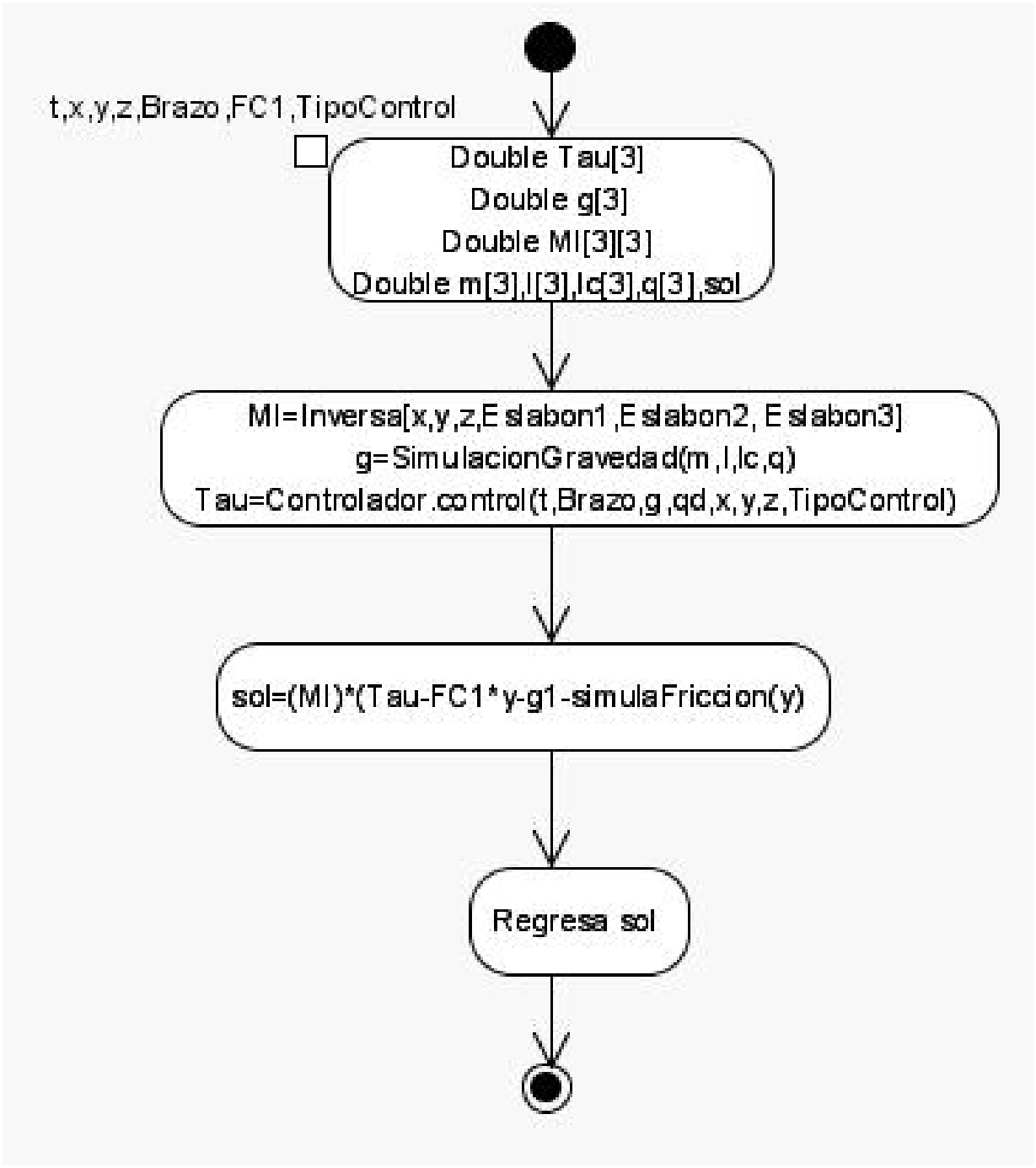

Figura 4.26: Diagrama de actividades f2.

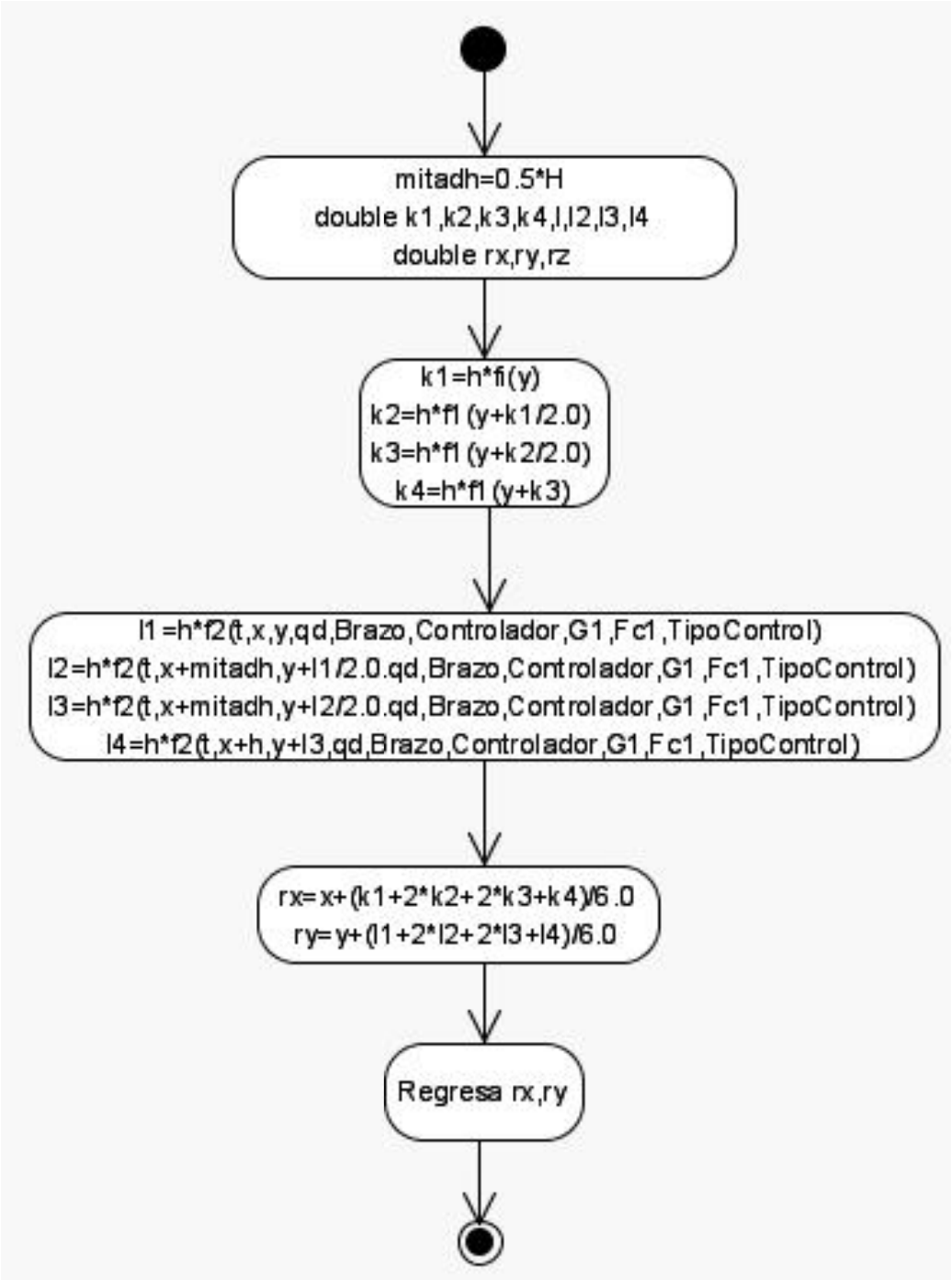

Figura 4.27: Diagrama de actividades RK4.

CONCLUSIONES

CONCLUSIONES

El empleo de un método que nos permita modelar de forma simple, fácil, completa y entendible todas las etapas que conforman el desarrollo de un sistema, cualquiera que este sea, es muy importante durante la etapa de desarrollo y de implementación.

Este tipo de características pueden ser obtenidas mediante el empleo de la metodología UML, ya que es producto de un conjunto de técnicas empleadas por expertos, que dan como resultado un conjunto de reglas y diagramas que nos permiten plasmar de forma que sean entendibles para cada involucrado en las distintas etapas que un sistema conlleva.

Los diagramas que se obtuvieron, son muy útiles para una posterior implementación del sistema, ya que describen el modelo matemático que forma un robot de 3 grados de libertad, así como las clases que conformarían el sistema, la comunicación entre las clases y los distintos diagramas de actividades que se llevan a cabo.

La herramienta que se escogió para llevar a cabo el desarrollo de los distintos diagramas (Visual Paradigm For UML 5.0) es una herramienta útil y sencilla para implementar la metodología UML, y los resultados son visiblemente fáciles de entender, así como su relación entre cada uno de los elementos que forman los diagramas.

Todo este conjunto de diagramas, nos ayudan a tener una idea más clara de lo que será la implementación final del sistema, nos ayuda a simplificarlo y además obtenemos un sistema escalable, una característica muy importante en cuanto al desarrollo de software se refiere.

El análisis con todo el conjunto de diagramas permite identificar de igual forma errores de diseño, puesto que al realizar los diagramas de casos de uso, el usuario puede identificar si sus requerimientos serán cubiertos de manera adecuada, sin necesidad que el usuario conozca ni de programación ni de UML, ya que estos diagramas son muy fáciles de entender.

Los diagramas de actividades resultan potencialmente útiles dado que muestran los pasos para cada tarea, esto nos permite codificar de manera ampliamente satisfactoria todas y cada una de estas tareas.

El análisis textual es muy conveniente puesto que al representar con palabras y por escrito las ideas y el funcionamiento, se facilita la comunicación con el usuario, además al analista y diseñador le permite identificar con facilidad los objetos, las clases y sus métodos, esto es posible gracias a que los verbos nos proporcionan los métodos, los sustantivos las clases y sus características las propiedades.

Otro de los beneficios de los diagramas es la oportunidad de identificar elementos que pueden ser reutilizados, por ejemplo, en nuestro caso de estudio el eslabón, y la simulación de gravedad son objetos reutilizables, puesto que se utilizan si necesidad de redefinir.

AP NDICE A

OBTENCIÓN DEL MODELO DIN MICO BAJO LA METODOLOG A EULER LAGRANGE

APÉNDICE A. *OBTENCIÓN DEL MODELO DINÁMICO BAJO LA METODOLOGÍA DE EULER - LAGRANGE*.

Metodología Euler-Lagrange

En el presente apéndice se mostrará la metodología de Euler - Lagrange, para la obtención del modelo dinámico de un robot manipulador de 2 grados de libertad, el cual puede ser seguido también para obtener el modelo para un robot de tres grados de libertad.

Las ecuaciones dinámicas de un robot manipulador pueden obtenerse a partir de las ecuaciones de movimiento de Newton. El inconveniente que presenta este método es que el análisis se complica notablemente cuando se aumenta el número de articulaciones del robot. En estos casos, es conveniente emplear las ecuaciones de movimiento de Lagrange[1] (también conocidas como ecuaciones de Euler-Lagrange).

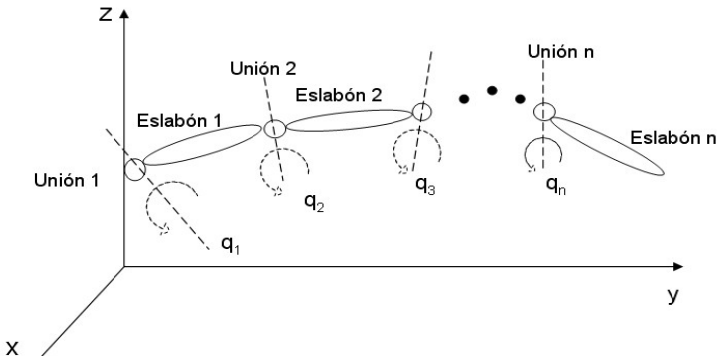

Figura A.1: Diagrama abstracto de un robot manipulador de n grados de libertad (g.d.l.).

Consideremos un robot manipulador que se compone de *n* eslabones mostrado en la figura A.1. La energía total de un robot manipulador de n grados de libertad es la suma de sus energías cinética y potencial

(A.1)

[1] Reciben el nombre de Lagrange, debido a que fue el primero que las dio a conocer en 1788.

Donde

Llamado el vector de posiciones.

Como podemos observar, la energía potencial depende únicamente de la posición, mientras que la energía cinética depende tanto de la posición como de la velocidd.

El *lagrangiano* de un robot manipulador de n grados de linertad es la diferencia entre su energía cinética y su energía potencial

(A.2)

Considerando aquí que la energía potencial se debe a fuerzas conservativas como la fuerza de gravedad y a fuerzas de resortes.

Las ecuaciones de movimiento de Euler - Lagrange para un manipulador de n grados de libertad, vienen dadas por:

(A.3)

o de forma equivalente:

(A.4)

Donde son las fuerzas y pares ejercidos externamente (por accionadores) en cada articulación asi como fuerzas no conservativas. es el vector de pares de fricción presentes en las articulaciones así como fuerzas no conservativas, como la de resistencia al movimiento y en general las que dependen del tiempo y de la velocidad.

De esta forma, tendremos tantas ecuaciones como grados de libertad tenga el robot manipulador.

El modelado dinámico de robots manipuladores por medio de las ecuaciones de Euler-Lagrange, podemos resumirlo en cuatro etapas:

1. Cálculo de la cinemática directa.
2. Cálculo de la energía cinética y potencial: y .
3. Cálculo del Lagrangiano:
4. Desarrollo de las ecuaciones de Euler-Lagrange.

Modelo Dinámico de un robot de n grados de libertad.

Considérese un robot manipulador de n grados de libertad, formado por eslabones rígidos, por simplicidad, libres de fricción y elasticidad. La energía cinética , asociada tal dispositivo articulado podemos expresarla como sigue:

– (A.5)

Donde *M(q)* es una matriz simétrica definida positiva de n x n, denominada *matriz de inercia*. La energía potencial no tiene una forma específica como el caso de la energía cinética, pero se sabe que depende del vector de posiciones articulares ***q.***

El lagrangiano dado por la ecuación (A.2), en este caso:

– (A.6)

Con esta forma para el lagrangiano, la ecuación de movimiento de Lagrange (A.4) puede expresarse como:

(A.7)

Por otro lado puede verificarse que:

(A.8)

(A.9)

Considerando las ecuaciones anteriores, la ecuación de movimiento toma la forma:

(A.10)

O de modo compacto:

(A.11)

Donde:

(A.12)

La ecuación (A.11) es la ecuación dinámica para los robots de n grados de libertad. Cabe notar que es una ecuación diferencial vectorial no lineal en el estado . es un vector de n x 1 llamado *vector de fuerzas centrífugas y de Coriolis,* es un vector de n x 1 de *fuerzas o pares gravitacionales* y es un vector de n x 1 llamado el *vector de fuerzas externas*, siendo generalmente los pares y fuerzas aplicadas por los accionadores en las articulaciones.

La matriz llamada la *matriz centrífuga y de Coriolis* puede no ser única, pero el vector sí lo es. Una manera de obtener es a través de los coeficientes o símbolos de Christoffel[2] .

[2] Definidos como donde denota el *ij-ésimo*elemento de la matriz de inercia .

(A.13)

Como se mencionó al principio de esta sección la ecuación genérica (3.15) supone eslabones rígidos, es decir, eslabones que no presentan torsión, ni flexión, por otro lado, debemos considerar que las uniones entre eslabones no están libres de fricción, al tomarla en cuenta el modelo dinámico queda finalmente como:

(A.14)

Donde es el vector de fricción.

Modelo Dinámico de un robot de 2 grados de libertad

El modelo dinámico se obtendrá de acuerdo a la metodología de Euler-Lagrange, siguiendo los cuatro mencionados anteriormente.

1. Cálculo de la Cinemática Directa.

El modelo cinemático directo de un robot, describe la relación entre la posición articular q y la posición y orientación x del dispositivo terminal del robot. En otras palabras el modelo cinemático directo de un robot es una relación de la forma:

El tópico del modelado cinemático directo de robots manipuladores se plantea en los siguientes términos. Considérese un robot de n grados de libertad colocado en una superficie fija. Defínase un marco de referencia también fijo en algún lugar de la superficie. El problema de la determinación del modelo cinemático directo del robot consiste en expresar la posición y orientación de un marco de referencia sólidamente colocado en la parte terminal del último eslabón del robot. La solución de este problema se reduce, desde el punto de vista matemático, a la solución de un problema geométrico (figura 1).

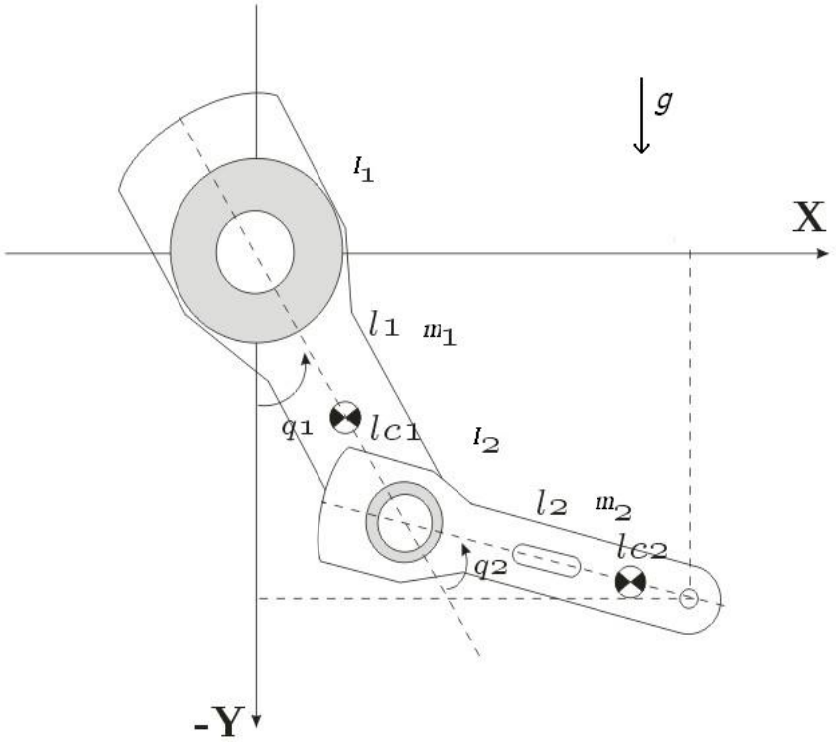

Figura 2: Robot planar de 2 grados de libertad.

Como se puede observar, ambas coordenadas cartesianas *x* e *y* dependen de las coordenadas articulares y . La relación entre ellas define el modelo cinemático propiamente dicho:

Donde

De la figura podemos obtener que el modelo cinemático directo viene dado por:

2. Cálculo de la energía cinética y potencial.

La energía cinética esta dada por:

La energía cinética para este sistema puede descomponerse en la suma de dos partes:

Que es la suma de las energías asociadas a cada masa.

La energía cinética para el primer eslabón, esa definida como:

Encontramos el centro de masa en el plano x-y como:

De donde obtenemos la velocidad:

Lo mismo para el segundo eslabón:

Finalmente:

(A.15)

De forma similar obtenemos la energía potencial del mecanismo con la suma de las energías potenciales de los eslabones:

donde[3]:

y

Por tanto la energía potencial es:

(A.16)

3. Cálculo del Lagrangiano

A partir de las ecuaciones (A.15) y (A.16) podemos obtener el Lagrangiano :

4. Desarrollo de las ecuaciones de Euler-Lagrange

Para obtener las ecuaciones de movimiento de (3.6) tenemos:

Para :

(3.33)

[3] En este punto se considera que la energía potencial es nula en y=0.

Calculamos las derivadas parciales:

La derivada total:

Calculamos las derivadas parciales:

+

La derivada total:

Ahora aplicando las ecuaciones de movimiento de Lagrange (3.5) y resumiendo, se tiene:

Y

Para tenerlo en la representación:

Matriz de inercia:

Matriz centrifuga y de Coriolis:

Vector de gravedad:

En variables de Estado

REFERENCIAS

[1] Booch, G; 1998, "*Análisis y Diseño Orientado a Objetos con Aplicaciones*"; Adison Wesley Longman.

[2] Booch, G., Rumbaugh, J., Jacobson I., 2004, "*El Lenguaje Unificado de Modelado*", Adison Wesley.

[3] Fairley, R., Agosto 1994, "*Ingeniería de Software*", McGraw-Hill.

[4] Kimmel, P., 2007, *"Manual de UML, guía de aprendizaje",* Mc Graw Hill.

[5] Vilalta J.,2001, *"UML Guía Visual, Cómo crear formas de vida",* Vilalta Consultors

[6] Kelly, R., Santibañez, V., 2003 *"Control de Movimientos de Robots Manipuladores"*, Pearson Prentice Hall.

[7] Vico open modeling, s.l., "UML guía visual". http://www.vico.org/UMLguiavisual/ Consulta: Septiembre 2007

[8] Alberto LaCalle, *"Diseño Orientado a Objetos" http://albertolacalle.com/diseno-uml.htm* Consulta: Octubre 2007

[9] La web de física, *"Ecuaciones de Lagrange" http://www.lawebdefisica.com/dicc/lagrange/* Consulta: Diciembre 2007

[10] Stevens, P.,Pooley R., 2002, *"Utilizacion de UML en ingeniería del software con objetos y componentes",* Adison Wesley

[11] Academia de Ciencias Luventivus, "Inercia". *http://www.luventicus.org/articulos/02N002/propge_inercia.html* Consulta: Enero 2008

[12] Sears F.W., Zemansky M.W., 1971 *"Fisica general",* Aguilar S.A. de ediciones