Capítulo 5. Modelado Orientado a Objetos

En los siguientes apartados se realizará una introducción a la metodología orientada a objetos y a UML (Lenguaje Unificado de Modelado). A continuación se mostrará una descripción de los objetos y algoritmos implementados para la creación del sistema empleando UML.

5.1 Introducción a la programación orientada a objetos.

Para la implementación del modelo y posterior resolución se ha optado por el uso de un lenguaje orientado a objetos como es C++. Más concretamente se ha empleado el entorno de programación que proporciona la herramienta C++ Builder, la cual suministra un entorno de programación visual que facilita en gran medida el empleo de un lenguaje como C++, que se caracteriza por tener una gran cantidad de reglas y términos diferentes.

En este apartado se va a realizar una descripción muy breve de lo que es la programación orientada a objetos y de aquellos componentes fundamentales que forman parte de la misma.

La programación orientada a objetos (POO) se define como una técnica o estilo de programación que utiliza objetos como bloque esencial de construcción. Un *objeto* es una unidad que contiene datos y las funciones que operan sobre esos datos. Estos objetos se corresponden con los elementos que debe utilizar el programa, como pueden ser los tramos o los vehículos en el caso del simulador.

Una *clase* es una colección de objetos que poseen características y operaciones comunes. En una clase se encuentra contenida toda la información necesaria para crear nuevos objetos pertenecientes a esa misma clase.

Los objetos solamente pueden ser accedidos y manipulados por las operaciones que previamente se hayan definido sobre su clase. Para evitar que dichos objetos sean utilizados en otras operaciones, se utiliza una técnica denominada *encapsulación*, que esconde los datos y sólo permite acceder a ellos de forma controlada.

Otros conceptos de interés en la POO son el *polimorfismo* que permite utilizar un mismo nombre para un tipo genérico de acciones, y la *herencia*, mediante la cual un objeto puede adquirir las propiedades de otro objeto.

5.2 UML

En este apartado se va a hacer una revisión de cómo surge, qué es y por qué se utiliza el Lenguaje de Modelado Unificado (UML). Este lenguaje es una especificación de notación orientada a objetos y se emplea para modelar el sistema y representar las distintas relaciones entre los componentes del mismo.

5.2.1 Breve historia de UML

Los lenguajes de modelados orientados a objetos aparecieron a mediados de los 70 y finales de los 80, influidos por técnicas como los modelos de Entidad/Relación y el *Specifications & Description Language* (SDL, circa 1976, CCITT), como diversas metodologías que se aproximaban al análisis y diseño orientado a objetos.

El número de lenguajes de modelado sufrió un espectacular incremento entre los años 1989 y 1994, llegando a identificarse hasta 50 lenguajes diferentes con los problemas que esto conllevaba para los usuarios. A mediados de los 90 aparecieron nuevas interacciones entre los diferentes métodos, destacando el Booch'93, la evolución del OMT (Object Modeling Techniques) y Fusion. Los métodos empezaron a incluir técnicas unos de los otros, predominando un pequeño grupo de ellos, incluyendo el OOSE (Object Oriented Software Engineering), OMT-2 y el Booch'93. Cada uno de ellos era un método completo con sus puntos fuertes y flaquezas.

Simplificando, OOSE era un lenguaje que se orientaba hacia los casos de uso, dando un excelente soporte para los análisis de requerimientos en la ingeniería. OMT-2 era especialmente adecuado para el análisis de sistemas de información y bases de datos. Finalmente Booch'93 destacaba por su utilidad en las fases de diseño y construcción de proyectos, llegando a ser muy popular para el desarrollo de proyectos de ingeniería.

Con este panorama, en Octubre del 94, comenzó el desarrollo del UML cuando Grady Booch y Jim Rumbaugh, del grupo *Rational Software Corporation*, comenzaron un trabajo conjunto para unificar los métodos Booch y OMT. Se daba el caso de que los métodos Booch y OMT estaban conocidos en todo el mundo como

los líderes entre los métodos de modelado orientados a objetos, por lo que Booch y Rambaugh unieron sus esfuerzos para unificar sus trabajos. Un primer boceto (versión 0.8) del *Unified Method*, como fue llamado entonces, se dio a conocer en Octubre del 95. A finales de 1995, Ivar Jacobson y su compañía de unieron a *Rational* y a su esfuerzo por unificar el lenguaje, aportando lo ya desarrollado en el OOSE.

Los autores de los métodos Booch, OMT y OOSE (Grady Booch, Jim Rumbaugh e Ivar Jacobson), estaban por lo tanto decididos a crear un lenguaje unificado de modelado por tres razones:

- En primer lugar, los métodos se estaban desarrollando unos independientemente de los otros. Era de sentido común continuar la evolución conjuntamente mejor que por separado, eliminando la posibilidad de crear diferencias innecesarias que posteriormente podrían confundir a los usuarios.

- Por otro lado, unificar la nomenclatura y la semántica traería estabilidad al mercado de los lenguajes orientados a objetos, permitiendo que los proyectos se centraran en un solo lenguaje, dejando a los desarrolladores de herramientas crear mejor aplicaciones.

- Finalmente, ellos esperaban que su colaboración pudiera conllevar importantes mejoras en los tres métodos, de forma que se pudieran abarcar nuevos problemas que ninguno de los tres métodos originales resolvía adecuadamente.

Cuando se comenzó la unificación, se establecieron cuatro objetivos primordiales en los que centrar sus esfuerzos:

- Permitir el modelado de sistemas, y no solo de software, usando conceptos de la orientación hacia objetos.

- Modelar el sistema, desde el concepto hasta los artefactos ejecutables, utilizando técnicas orientadas a objetos.

- Cubrir las cuestiones relacionadas con el tamaño inherente a los sistemas complejos y críticos.

- Crear un lenguaje de modelado utilizable tanto por las personas como por las maquinas.

En Junio de 1996 se publicaron las versiones 0.9 y 0.91. Durante ese año los autores invitaron, y recibieron respuesta, de toda la comunidad internacional relacionada con el tema. Además, varias corporaciones vieron el UML como un punto estratégico en el desarrollo de sus actividades. Algunas de las empresas contribuyeron a la aparición de la versión 1.0 en Enero de 1997 fueron IBM, Hewlet-Packard, Dell, Texas Instruments... Esta versión fue ofrecida a la OMG (*Object Management Group*) para su estandarización, en respuesta a su solicitud de propuestas para un lenguaje estándar de modelado.

Entre Enero y Julio de 1997, el grupo inicial de colaboradores se amplió para incluir prácticamente a todas las organizaciones que habían aportado algo al proyecto, creándose un grupo de trabajo liderado por Cris Kobryn para trabajar en la semántica. Como resultado se le entregó a la OMG la versión 1.1 para su estandarización en Julio de 1997. Esta versión se aceptó el 14 de Noviembre de 1997 por la OMG.

El control del mantenimiento de UML fue asumido por la *OMG Revision Task Force*, dirigida por Cris Kobryn. La RTF publicó una revisión editorial, UML 1.2, en Junio de 1998. En otoño de 1998 la RTF publicó UML 1.3, que es la versión utilizada en este proyecto.

5.2.2 ¿Qué es UML? ¿Por qué usar UML?

UML es un lenguaje estándar para escribir planos de software. Se puede utilizar para especificar, construir y documentar los artefactos de un sistema que involucran una gran cantidad de software.

UML es apropiado para modelar desde sistemas de información de empresas, hasta aplicaciones distribuidas basadas en la web, e incluso para sistemas

empotrados de tiempo real muy exigentes. Es un lenguaje expresivo, que cubre todas las vistas necesarias para desarrollar y luego desplegar tales sistemas.

UML es sólo un lenguaje, por lo tanto es tan sólo una parte de un método de desarrollo de software. Es independiente del proceso, aunque para utilizarlo óptimamente se debería usar en un proceso que fuese dirigido por los casos de uso, centrado en la arquitectura, iterativo e incremental.

En general, UML es un lenguaje para visualizar, especificar, construir y documentar los artefactos de un sistema con gran cantidad de software.

Como lenguaje, UML proporciona un vocabulario y unas reglas para combinar las palabras de dicho vocabulario con el objetivo de posibilitar la comunicación. Un lenguaje de modelado es un lenguaje cuyo vocabulario y reglas se centran en la representación conceptual y física del sistema. Un lenguaje de modelado como es UML es, por tanto, un lenguaje estándar para los planos de software.

El modelado proporciona una comprensión de un sistema. Nunca es suficiente un único modelo. Más bien, para comprender cualquier cosa, a menudo se necesitan múltiples modelos conectados entre sí, excepto en los casos más triviales. Para sistemas con gran cantidad de software, se requiere un lenguaje que cubra las diferentes vistas de la arquitectura de un sistema mientras evoluciona a través del ciclo de vida del desarrollo del software.

El vocabulario y las reglas de un lenguaje como UML indican cómo crear y leer modelos bien formados, pero no dicen qué modelos se deben crear ni cuándo se deberían crear. Ésta es la tarea del proceso del desarrollo del software. Un proceso bien definido guiará a su usuario a decidir qué artefactos producir, qué actividades y personal se emplean para crearlos y gestionarlos, y cómo usar estos artefactos para medir y controlar el proyecto de forma global.

Para muchos programadores, la distancia entre pensar en una implementación y transformarla en código es casi nula. Lo piensas, lo codificas. De hecho, algunas cosas se modelan mejor directamente en código. En este caso, el programador todavía está haciendo mentalmente algo de modelado, si bien lo hace de forma totalmente mental. No obstante, esta manera de proceder plantea algunos problemas:

- Primero, la comunicación de estos modelos conceptuales se torna complicada y está sujeta a errores, salvo que las dos partes hablen el mismo lenguaje. Normalmente, los proyectos y las organizaciones desarrollan su propio lenguaje, y es difícil comprender lo que está pasando para alguien ajeno al grupo.

- Segundo, hay algunas cuestiones en un sistema de software que no se pueden entender a menos que se construyan sobre modelos que trasciendan el lenguaje de programación textual. Por ejemplo, el significado de una jerarquía de clases puede inferirse, pero no capturarse completamente inspeccionando el código de todas las clases en la jerarquía.

- Tercero, si el desarrollador que escribió el código no dejó documentación escrita de los modelos que hacía de forma mental, esa información se perderá para siempre, o como mucho, será sólo parcialmente reproducible a partir de la implementación una vez que el desarrollador se haya marchado.

Al escribir modelos UML, se afronta el tercer problema: un modelo explícito para facilitar la comunicación.

Algunas cosas se modelan mejor textualmente, otras se modelan mejor de forma gráfica. En realidad, en todos los sistemas interesantes hay estructuras que trascienden lo que puede ser representado mediante un lenguaje de comunicación. UML es uno de esos lenguajes gráficos. Así afronta el segundo problema mencionado anteriormente.

UML es algo más que un simple compendio de símbolos gráficos. Más bien, detrás de cada símbolo en la notación UML hay una semántica bien definida. De manera que un desarrollador puede escribir un modelo en UML, y otro desarrollador, o incluso otra herramienta, pude interpretar ese modelo sin ambigüedad. Así afronta el primer problema mencionado con anterioridad.

En este contexto, especificar significa construir modelos precisos, no ambiguos y completos. En particular, UML cubre la especificación de todas las decisiones de

análisis, diseño e implementación que se deben realizar al desarrollar y desplegar un sistema con gran cantidad de software.

UML no es un lenguaje de programación visual, pero sus modelos pueden conectarse a gran variedad de lenguajes de programación. Esto significa que es posible establecer correspondencias desde un modelo UML a un lenguaje de programación como C++, Java o Visual Basic, o incluso tablas en una base de datos relacional o almacenamiento persistente en una base de datos orientada a objetos. Las cosas que se expresan mejor gráficamente también se representan gráficamente con UML, mientras que las cosas que se representan mejor textualmente se plasman con un lenguaje de programación.

Esta correspondencia permite la ingeniería directa: generación de código a partir de un modelo UML en un lenguaje de programación. Lo contrario también es posible: se puede reconstruir un modelo en UML a partir de una implementación. Pero este proceso no es automático, a menos que se codifique esa información en l a implementación, la información se pierde cuando se pasa del modelo al código. La ingeniería inversa requiere, por lo tanto, de herramientas que la soporten e intervención humana. La combinación de estas dos vías de generación de código y de ingeniería inversa produce una ingeniería "de ida y vuelta", entendiendo por eso la posibilidad de trabajar en una vista gráfica o textual, mientras que las herramientas mantienen la consistencia de las dos vistas.

Además de esta correspondencia directa, UML es lo suficientemente expresivo y no ambiguo como para permitir la ejecución directa de modelos, la simulación de sistemas y la instrumentación de sistemas en ejecución.

UML cubre la documentación de la arquitectura de un sistema y todos sus detalles. UML también proporciona un lenguaje para expresar requisitos y pruebas. Finalmente, UML proporciona un lenguaje para modelar las actividades de planificación de proyectos y gestión de versiones.

5.2.3 Diagramas en UML

A continuación se muestran los diagramas que forman parte del lenguaje de modelado unificado del proyecto, así como una breve introducción a los mismos, con el fin de que su interpretación sea de utilidad a la hora de entender el modelo aquí implementado.

5.2.3.1 Diagramas de casos de uso

Los casos de uso permiten describir el comportamiento de un sistema desde el punto de vista del usuario basándose en un conjunto de acciones y reacciones. Es por lo tanto una técnica que permite capturar los requisitos funcionales del sistema. De esta forma queda delimitado el alcance del sistema y cuál es su relación con el entorno.

En estos diagramas, el sistema queda reducido a una "caja negra", ya que no interesa cómo lleva a cabo sus funciones, sino simplemente qué acciones visibles desde el exterior son las que realiza.

Los casos de uso están basados en lenguaje natural, lo que los hace accesibles a cualquier usuario. Además, aquellos casos de uso que resulten muy complejos pueden descomponerse en nuevos casos de uso de un nivel inferior, hasta llegar a un nivel tal que resulten fáciles de analizar.

Los casos de uso guían todo el proceso de desarrollo del sistema, lo que quiere decir que en momentos determinados de dicho proceso, el sistema debe ser validado comprobando que se ajusta al diagrama de casos de uso.

Los diagramas de casos de uso están formados por tres elementos fundamentales:

- **Actores.** Los actores son los participantes de os casos de uso, se corresponden con los usuarios que interactúan con el sistema. Estos actores pueden ser humanos, dispositivos externos que interactúen con el sistema, o incluso temporizadores que envíen eventos al mismo. Un actor se caracteriza por la forma de interaccionar con el sistema, por lo que un mismo usuario puede ejercer de varios actores, y un actor puede representar a varios usuarios. Los actores se representan de la siguiente manera en los diagramas de casos de uso:

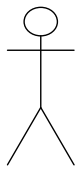

Figura 5.1: Actor

- **Casos de uso.** Son los escenarios de interacción de los actores. Representan el comportamiento del sistema en relación con los usuarios. De esta forma, un caso de uso define la secuencia de interacciones entre uno o más usuarios y el sistema. Los casos de uso se representan de la siguiente manera en los diagramas de casos de uso:

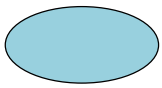

Figura 5.2: Representación de un caso de uso en UML

- **Relaciones.** Representan el flujo de información intercambiada entre los actores y los casos de uso, o entre diferentes casos de uso. Normalmente, se emplean para que un caso de uso obtenga la información necesaria para llevar a cabo alguna acción, o para que el proceso proporcione algún resultado. Estas relaciones pueden ser unidireccionales o bidireccionales. La relación entre un actor y un caso de uso se representa de la siguiente manera:

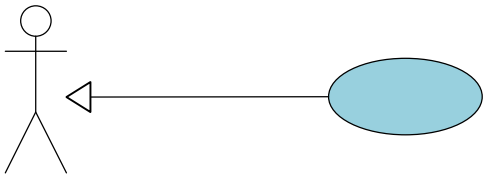

Figura 5.3: Relación entre un actor y un caso de uso

Figura 5.4: Flujo de información intercambiada entre casos de uso

- **Límite de un sistema.** Sólo aparece en el diagrama de más alto nivel, se representa por un rectángulo que contiene los casos de uso, uno por cada servicio de alto nivel que el sistema ofrezca a los actores.

Los diagramas de casos de uso se clasifican en diferentes niveles, en función del grado de detalle con el que se represente el funcionamiento del sistema. De esta

forma, los diagramas de *nivel 0* o *contexto* representan el sistema completo con un nivel de detalle muy bajo, mientras que al aumentar el nivel, el grado de detalle va incrementándose.

5.2.3.2 Diagramas de paquetes

El objetivo de este tipo de diagramas es obtener una visión mucho más clara del sistema de información orientado a objetos, organizándolo en diferentes subsistemas, agrupando los elementos del análisis, diseño o construcción y detallando las relaciones de dependencias entre ellos. El mecanismo de agrupación utilizado se denomina *paquete.*

Estrictamente hablando, los paquetes y sus dependencias son elementos de los diagramas de casos de uso, de clases y de componentes, por lo que se podría decir que el diagrama de paquetes es una extensión de éstos.

En estos diagramas de pueden diferenciar dos tipos de elementos:

- **Paquetes.** Un paquete es la agrupación de elementos, bien sea casos de uso, clases o componentes. Los paquetes pueden contener a su vez otros paquetes anidados que en última estancia contendrán alguno de los elementos anteriores. Un paquete es representado mediante un símbolo como el que se muestra a continuación, colocándose el nombre del paquete en la pestaña, y el contenido debajo. En los casos en que no sea visible el contenido del paquete se podrá colocar en su lugar el nombre.

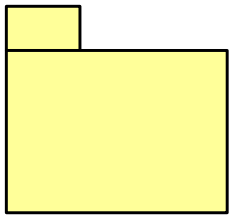

Figura 5.5: Representación de un paquete en UML

- **Dependencia entre paquetes.** Existe una dependencia cuando un elemento de un paquete requiere de otro que pertenece a un paquete distinto. Es importante resaltar que las dependencias no son transitorias. Las dependencias entre paquetes se representan con una flecha discontinua con inicio en el paquete que depende del otro.

Figura 5.6: Dependencia entre paquetes

5.2.3.3 Diagramas de clases

El objetivo principal de este diagrama es la representación de los objetos estáticos del sistema, utilizando diversos mecanismos de abstracción (clasificación, generalización, agregación). Este diagrama permite por tanto modelas los aspectos estructurales del sistema a modelar, y los elementos que no dependen del tiempo.

El diagrama de clases recoge las clases de objetos y sus asociaciones. En este diagrama se representa la estructura y el comportamiento de cada uno de los objetos del sistema y sus relaciones con los demás objetos, pero no muestra información temporal. Con el fin de facilitar la comprensión del diagrama, se pueden incluir paquetes como elementos del mismo, donde cada uno de ellos agrupa un conjunto de clases.

- **Notación.** Describe como se representan las clases y las relaciones que pueden aparecer entre ellas.

  - **Clases.** Una clase se representa como una caja, separada en tres zonas por líneas horizontales, como se muestra en la siguiente figura:

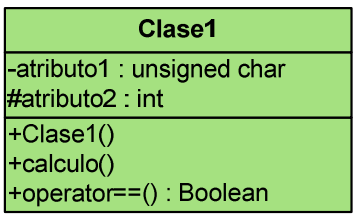

Figura 5.7: Representación de una clase en UML

    - En la zona superior se muestra el nombre de la clase y propiedades generales como el estereotipo. El nombre de la clase aparece centrado y si la clase es abstracta se representa en cursiva. El estereotipo, si se muestra, se sitúa sobre el nombre y entre el símbolo: <<…>>.

- La zona central contiene una línea de atributos, uno en cada línea. La notación utilizada para representarlos incluye, dependiendo del detalle, el nombre del atributo, su tipo y su valor por defecto, con el formato:

  Visibilidad nombre: tipo = valor-inicial {propiedades}

  La visibilidad será en general pública (+), privada (-) o protegida (#), aunque puede haber otros tipos de visibilidad dependiendo del lenguaje de programación empleado.

- En la zona inferior se incluye una lista con las operaciones que proporciona la clase. Cada operación aparece en una línea con formato:

  Visibilidad nombre (lista-de-parámetros): tipo-devuelto {propiedad}

Para el modelado de los datos del sistema solo indicaremos el nombre de la clase ya que la especificación de todos los atributos y métodos repercute en una peor legibilidad del modelo.

- **Relaciones.** Una relación de asociación se representa como una línea discontinua entre las clases asociadas. En una relación de asociación, ambos extremos de la línea pueden conectar con la misma clase, lo que se conoce como *asociación reflexiva*. La relación puede tener un nombre y un estereotipo, que se colocan junto a la línea. El nombre suele corresponderse con expresiones verbales presentes en las especificaciones, y define la semántica de la asociación. Los estereotipos permiten clasificar las relaciones en familias y se escribirán entre el símbolo: <<…>>. Las diferentes propiedades de la relación se pueden representar con la siguiente notación:

  - **Multiplicidad.** La multiplicidad puede ser un número concreto, un rango o una colección de números. La letra ´n´ y el símbolo ´*´ representan cualquier número. En la figura siguiente se representa que

la *Clase-1* tiene desde 1 hasta n instancias de la *Clase-2*, mientras que cada instancia de la *Clase-2* tiene una instancia de la *Clase-1*.

- **Navegabilidad.** La navegación desde una clase a otra se representa poniendo una flecha sin relleno en el extremo de la línea, indicando el sentido de la navegación. En caso de no especificar nada el sentido es bidireccional. En la figura siguiente se muestra un ejemplo en el que la *Clase-2* contiene un listado de instancias de la *Clase-1*, mientras que la *Clase-1* no contiene referencias a la *Clase-2*.

- **Rol o nombre de la asociación.** Este nombre se coloca junto al extremo de la línea que está unida a la clase, para expresas cómo esa clase hace uso de la otra clase con la que mantiene la asociación.

Además existen notaciones específicas para los otros tipos de relación, como son:

- **Agregación.** Se representa como un rombo hueco en la clase cuya instancia es una agregación de las instancias de la otra.

- **Composición.** Se representa como un rombo lleno en la clase cuya instancia contiene las instancias de la otra clase.

- **Dependencia.** Una línea discontinua con una flecha apuntando a la clase cliente. La relación puede tener un estereotipo que se coloca junto a la línea, y entre el símbolo: <<…>>.

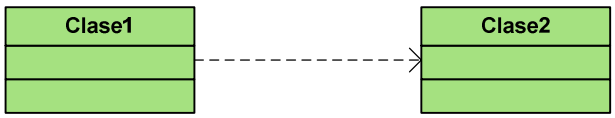

Figura 5.8: Relación de dependencia entre dos clases en UML

- **Herencia.** Esta relación se representa como una línea continua con una flecha hueca en el extremo que apunta a la superclase.

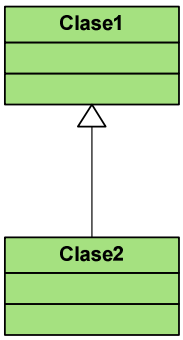

Figura 5.9: Herencia de una clase en UML

5.2.3.4 Diagrama de flujo

Un diagrama de flujo es un esquema para representar gráficamente un algoritmo. Se basan en la utilización de diversos símbolos para representar operaciones específicas. Se les llama diagramas de flujo porque los símbolos utilizados se conectan por medio de flechas para indicar la secuencia de operación. Se usan para mostrar la secuencia de operaciones que se siguen en un programa.

No es indispensable usar un tipo especial de símbolos para crear un diagrama de flujo, pero existen algunos ampliamente utilizados, en el presente proyecto se utilizarán los estandarizados según la *ISO 5807*, que se explican a continuación:

- **Flecha.** Indica el sentido o trayectoria del proceso de información o tarea.

- **Rectángulo.** Se usa para representar un evento o proceso determinado. Éste es controlado dentro del diagrama de flujo en que se encuentra. Es el símbolo más comúnmente utilizado.

- **Rectángulo redondeado.** Se usa para representar un evento que ocurre de forma automática y del cual se sigue una secuencia determinada.

- **Rombo.** Se utiliza para representar una condición. Normalmente el flujo de información entra por arriba y sale por un lado si la condición se cumple o sale por el lado opuesto si la condición no se cumple. Lo anterior hace que a partir de éste el proceso tenga dos caminos posibles.

- **Círculo.** Representa un punto de conexión entre procesos, se utiliza cuando en necesario dividir un diagrama de flujo en varias partes, por ejemplo por razones de espacio o simplicidad. Una referencia debe de darse dentro para distinguirlo de otros. La mayoría de las veces se utilizan números en los mismos.

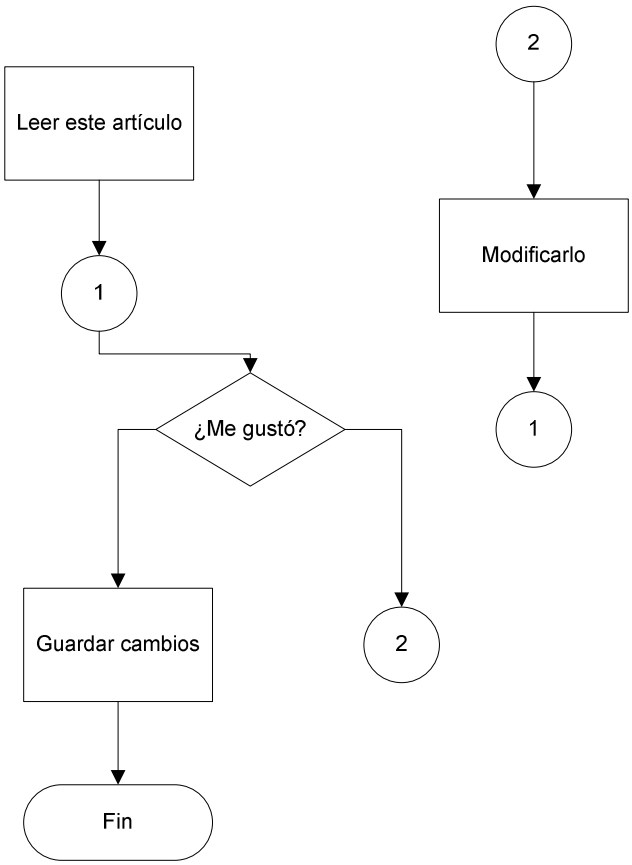

Figura 5.10: Diagrama de flujo con los símbolos más comunes

5.2 Diagramas aplicados al proyecto

5.2.1 Diagrama de casos de uso

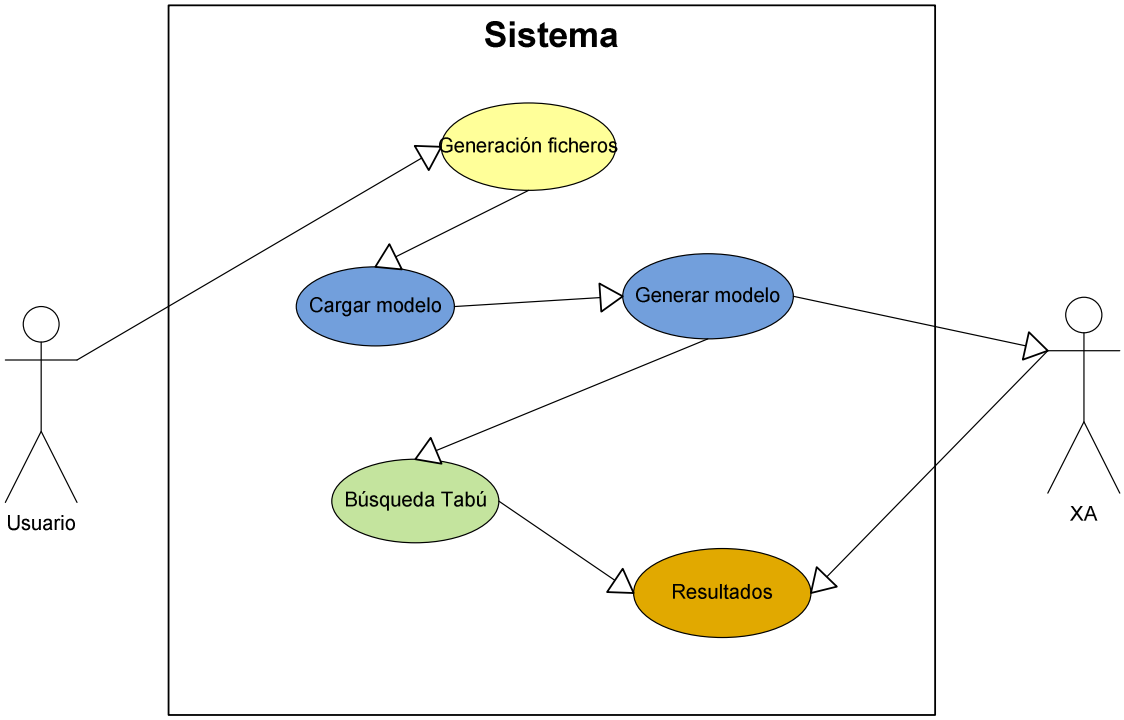

Figura 5.11: Diagrama de casos de uso del sistema

Se explican a continuación cada uno de los usuarios y casos de uso que aparecen en la figura 5.11.

- **Usuario del sistema.** Es el actor principal del sistema, la persona que usa la aplicación para la finalidad que se diseñó, obtener la mejor solución posible con el método Búsqueda Tabú. Debe introducir los datos que caracterizan el modelo, así como seleccionar el modelo que se desea utilizar. Estos datos representan los números de viajes y de autobuses existentes en el modelo, con los costes asociados a cada uno de ellos. Evidentemente, el usuario será por tanto el encargado de recolectar los diferentes resultados, de interpretarlos, etc.

- **XA.** El sistema generará los ficheros necesarios, que serán enviados al programa XA, el cual obtendrá la solución óptima.

- Generación ficheros. Es el módulo que genera los diferentes ficheros, deben mantener una estructura determinada.

- Cargar modelo. Módulo que se encarga de leer los fichero creados e introducir en el sistema las características de los modelos.

- Generar modelo. Cuando ya han sido cargadas las características en el sistema, este módulo es el encargado de generar los ficheros de tipo lp, que el programa XA ejecutará.

- Búsqueda Tabú. Será el módulo que obtenga la solución aproximada a nuestro problema.

- Resultados. Muestra los resultados obtenidos tanto con la búsqueda tabú como con el programa XA, estos resultados serán mostrados detalladamente en el capítulo 7.

5.2.2 Diagrama de clases

En la figura 5.12 aparece el diagrama de clases general del sistema, que recoge todas las clases de objetos definidas para este proyecto y las asociaciones existentes para las mismas.

A continuación se describe cada clase, detallando sus métodos y atributos.

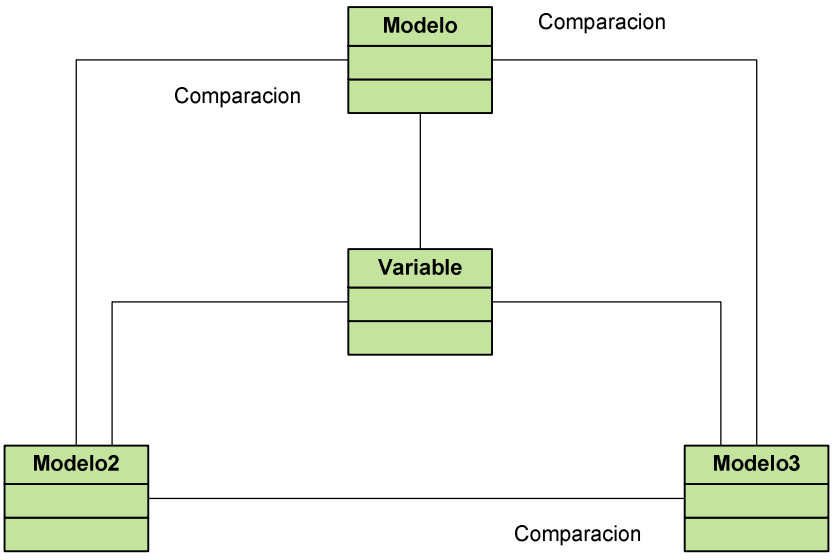

Figura 5.12: Diagrama de clases general

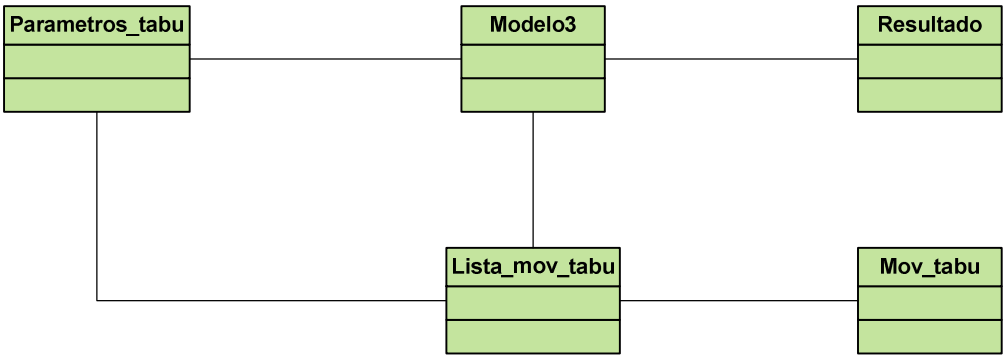

Figura 5.13: Diagrama de clases búsqueda tabú

5.2.2.1 Variable

Esta clase es utilizada para convertir dos o más índices en un único índice, en nuestro caso lo máximo que se utilizan son dos índices, la función principal de esta clase es para evitar confusiones y que se puedan mezclar los índices.

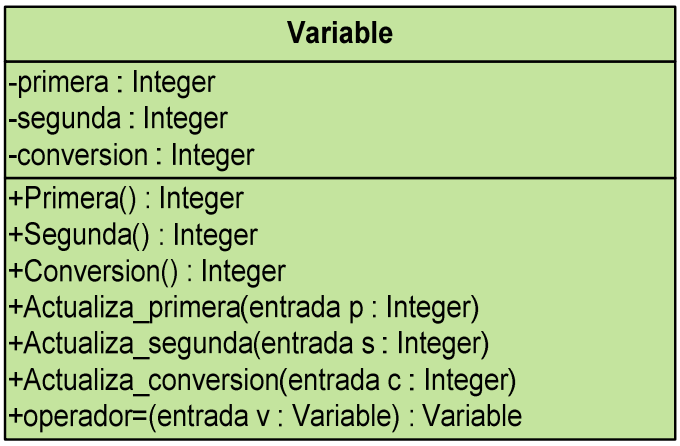

Figura 5.14: Especificación de la clase Variable

Esta clase define los siguientes atributos:

- **primera**: primer índice de la variable.
- **segunda**: segundo índice de la variable.
- **conversion**: resultado de convertir los dos índices, será el índice definitivo.

También se definen los métodos que se detallan a continuación:

- **Primera**: devuelve el atributo primera.
- **Segunda**: devuelve el atributo segunda.
- **Conversion**: devuelve el atributo conversión.

- **Actualiza_primera**: para actualizar el atributo primera.
- **Actualiza_segunda**: para actualizar el atributo segunda.
- **Actualiza_conversion**: para actualizar el atributo conversión.

Existe además sobrecarga de operadores:

- **operator=**: operador para la igualdad de dos Variables.

A continuación y ya definida esta clase que será utilizada en los tres modelos, se comienza con la explicación del modelo básico (primer modelo).

5.2.2.2 Modelo

Modelo
-conductores : Integer -lista_variables : Variable* -lineas : Integer -autobuses : Integer -turnos : Integer -costes : float* -cpa : float*
+Inicializa_vectores() +Condcutores() : Integer +Lineas() : Integer +Autobuses() : Integer +Turnos() : Integer +Libera_costes() +Costes() : float* +Costes(entrada i : Integer) : float +Cpa() : float* +Cpa(entrada i : Integer) : float +Actualiza_conductores(entrada i : Integer) +Actualiza_lineas(entrada l : Integer) +Actualiza_autobuses(entrada a : Integer) +Actualiza_turnos(entrada t : Integer) +Actualiza_costes(entrada n : float*) +Actualiza_cpa(entrada cpa : float*) +operator=(entrada f : Modelo) : Modelo

Figura 5.15: Especificación de la clase Modelo

Se comentan a continuación los atributos de la clase:

- **conductores**: número de conductores existentes en la plantilla.

- **lista_variables**: lista donde se almacenan todos los objetos de la clase variable, utilizadas en el modelo con su respectiva conversión.
- **líneas**: número de líneas de autobús.
- **autobuses**: número de autobuses.
- **turnos**: número de turnos.
- **costes**: coste que tiene cada conductor.
- **cpa**: número de conductores asignados a cada uno de los autobuses, de los cuales se elegirá el conductor más apropiado para cada autobús.

A continuación se comentan los métodos de la clase:

- **Inicializa_vectores**: se reserva memoria para los atributos costes y cpa.
- **Conductores**: devuelve el atributo conductores.
- **Lineas**: devuelve el atributo líneas.
- **Autobuses**: devuelve el atributo autobuses.
- **Turnos**: devuelve el atributo turnos.
- **Libera_costes**: libera el atributo costes.
- **Costes**: devuelve el atributo costes.
- **Costes**: devuelve el valor determinado que se le pasa por parámetro del atributo costes.
- **Cpa**: devuelve el atributo cpa.
- **Cpa**: devuelve el valor determinado que se le pasa por parámetro del atributo cpa.
- **Actualiza_conductores**: para actualizar el atributo conductores.
- **Actualiza_lineas**: para actualizar el atributo líneas.
- **Actualiza_autobuses**: para actualizar el atributo autobuses.
- **Actualiza_turnos**: para actualizar el atributo turnos.
- **Actualiza_costes**: para actualizar el atributo costes.
- **Actualiza_cpa**: para actualizar el atributo cpa.
- **operator=**: operador para la igualdad de dos Modelos.

5.2.2.3 Modelo2

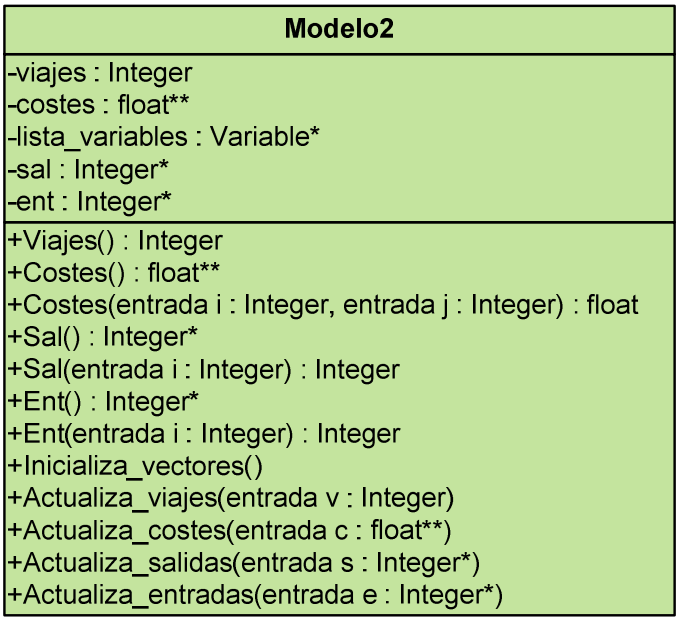

Figura 5.16: Especificación de la clase Modelo2

Los atributos de esta clase son los siguientes:

- **viajes**: número de viajes existentes en el modelo
- **costes**: es la matriz de costes de un viaje a otro.
- **lista_variables**: lista donde se almacenan todos los objetos de la clase variable, utilizadas en el modelo con su respectiva conversión.
- **sal**: para cada viaje conocemos a partir de que viaje de salida es compatible por tiempo, los viajes que nos compatibles significan que han comenzado antes de que haya terminado el actual.
- **ent**: para cada viaje conocemos a partir de que viaje de entrada es compatible, los viajes que no sean compatibles significan que su trayecto terminará después de que comience el actual.

Los métodos de la clase son:

- **Viajes**: devuelve el atributo viajes.
- **Costes**: devuelve el atributo costes.
- **Costes**: devuelve el valor determinado de la matriz de costes cuyos índices se le pasan por parámetros.
- **Sal**: devuelve el atributo sal.
- **Sal**: devuelve el valor determinado que se le pasa por parámetro del atributo sal.

- **Ent**: devuelve el atributo ent.
- **Ent**: devuelve el valor determinado que se le pasa por parámetro del atributo ent.
- **Inicializa_vectores**: se reserva memoria para los atributos costes, sal y ent.
- **Actualiza_viajes**: para actualizar el atributo viajes.
- **Actualiza_costes**: para actualizar el atributo costes.
- **Actualiza_salidas**: para actualizar el atributo sal.
- **Actualiza_entradas**: para actualizar el atributo ent.

5.2.2.4 Modelo3

Modelo3
-viajes : Integer -lista_variables : Variable* -costesV : float** -sal : Integer* -ent : Integer* -conductores : Integer -autobuses : Integer -turnos : Integer -costesC : float* -cpa : float* -s : Resultado -SM : Resultado -parametros_tabu : Parametros_tabu
+Viajes() : Integer +CostesV() : float** +CostesV(entrada i : Integer, entrada j : Integer) : float +Sal() : Integer* +Sal(entrada i : Integer) : Integer +Ent() : Integer* +Ent(entrada i : Integer) : Integer +Conductores() : Integer +Autobuses() : Integer +CostesC() : float* +CostesC(entrada i : Integer) : float +Cpa() : float* +Cpa(entrada i : Integer) : float +Inicializa_vectores() +Actualiza_viajes(entrada v : Integer) +Actualiza_costesV(entrada c : float**) +Actualiza_salidas(entrada s : Integer*) +Actualiza_entradas(entrada e : Integer*) +Actualiza_conductores(entrada c : Integer) +Actualiza_lineas(entrada l : Integer) +Actualiza_autobuses(entrada a : Integer) +Actualiza_costesC(entrada n : float*) +Actualiza_cpa(entrada cpa : float*) +Obtener_variable(entrada primera : Integer, entrada segunda : Integer) +Imprimir_variable(entrada primera : Integer, entrada segunda : Integer) : Integer +Grabar_variables(entrada cadena : char*) +Cargar_modelo3(entrada cad : char*) +Grabar_modelo3(entrada cad : char*) +Escribir_titulo3(entrada F1 : string*) +Escribir_objetivo3(entrada F1 : string*) +Escribir_restricciones3(entrada F1 : string*) +Obtener_restricciones(entrada F1 : string*) +Vecindad() +Parametros() : Parametros_tabu +Obtener_fo() +Solucion_inicial() +Busqueda_tabu(entrada numero_iteraciones : uint, entrada tamano_lista_tabu : uint) +Generar_fichero_solucion() +Asignacion_conductores() +Diversificacion() +Solucion_aleatoria() +Intensificacion()

Figura 5.17: Especificación de la clase Modelo3

Se definen a continuación los atributos correspondientes a esta clase:

- **viajes**: número de viajes existentes en el modelo.
- **lista_variables**: lista donde se almacenan todos los objetos de la clase variable, utilizadas en el modelo con su respectiva conversión..
- **costesV**: es la matriz de costes de un viaje a otro.
- **sal**: para cada viaje conocemos a partir de que viaje de salida es compatible por tiempo, los viajes que nos compatibles significan que han comenzado antes de que haya terminado el actual.
- **ent**: para cada viaje conocemos a partir de que viaje de entrada es compatible, los viajes que no sean compatibles significan que su trayecto terminará después de que comience el actual.
- **conductores**: número de conductores.
- **autobuses**: número de autobuses.
- **turnos**: número de turnos.
- **costesC**: coste que tiene cada conductor.
- **cpa**: número de conductores asignados a cada uno de los autobuses, de los cuales se elegirá el conductor más apropiado para cada autobús.
- **s, SM**: se trata de las dos variables de tipo Resultado que se utilizarán, s se utiliza para los resultados provisionales, y SM es la mejor solución obtenida.
- **parametros_tabu**: son los parámetros que se utilizan para la búsqueda tabú.

Los métodos que llevan asociados son los siguientes:

- **Viajes**: devuelve el atributo viajes.
- **CostesV**: devuelve el atributo costesV.
- **CostesV**: devuelve el valor determinado de la matriz de costes cuyos índices se le pasan por parámetros.
- **Sal**: devuelve el atributo sal.
- **Sal**: devuelve el valor determinado que se le pasa por parámetro del atributo sal.
- **Ent**: devuelve el atributo ent.
- **Ent**: devuelve el valor determinado que se le pasa por parametro del atributo ent.
- **Conductores**: devuelve el atributo conductores.
- **Autobuses**: devuelve el atributo autobuses.
- **CostesC**: devuelve el atributo costesC.

- **CostesC**: devuelve el valor determinado que se le pasa por parametro del atributo costesC.
- **Cpa**: devuelve el atributo cpa.
- **Cpa**: devuelve el valor determinado que se le pasa por parametro del atributo cpa.
- **Inicializa_vectores**: se reserva memoria para los atributos costesV, sal, ent, costesC, cpa y lista_variables.
- **Actualiza_viajes**: para actualizar el atributo viajes.
- **Actualiza_costesV**: para actualizar el atributo costesV.
- **Actualiza_salidas**: para actualizar el atributo sal.
- **Actualiza_entradas**: para actualizar el atributo ent.
- **Actualiza_conductores**: para actualizar el atributo condcutores.
- **Actualiza_lineas**: para actualizar el atributo lineas.
- **Actualiza_autobuses**: para actualizar el atributo autobuses.
- **Actualiza_costesC**: para actualizar el atributo costesC.
- **Actualiza_cpa**: para actualizar el atributo cpa.
- **Obtener_variable**: obtiene un elemento de la clase variable a partir de los dos parámetros que se le pasan.
- **Imprimir_variable**: busca en la lista de variables la correspondiente a los parámetros que se le pasan, y devuelve el argumento conversion.
- **Grabar_variables**: graba en un fichero toda la lista de variables.
- **Cargar_modelo3**: se encarga de cargar del fichero que se le pasa los datos del modelo.
- **Grabar_modelo3**: graba en el fichero que se le pasa los datos del modelo.
- **Escribir_titulo3**: se escribe en el fichero la parte del titulo.
- **Escribir_objetivo3**: se escribe en el fichero la parte de los objetivos.
- **Escribir_restricciones3**: se escribe en el fichero la parte de las restricciones.
- **void Obtener_restricciones3**: se obtienes las restricciones que después se escriben en el fichero.
- **Vecindad**: en la búsqueda tabú cada una de las vecindades que se realizan, donde se busca en cada una de ellas el mejor valor posible y se compara con la mejor solución total.
- **Parametros**: devuelve los atributos de Parametros_tabu.
- **Obtener_fo**: se obtiene la función objetivo de la solución actual, también se obtiene si la solución es admisible.
- **Solucion_inicial**: se obtiene una solución inicial y a partir de ella se irán haciendo vecindades para obtener la mejor solución.

- **Busqueda_tabu**: se obtiene la búsqueda tabú, se le pasan el numero de veces que se realizan las vecindades y el tamaño de la lista tabú que se utilizara.
- **Generar_fichero_solucion**: se genera un fichero solución con la mejor solución encontrada y su función objetivo.
- **Asignacion_conductores**: se obtiene para los autobuses utilizados los conductores con menos coste.
- **Diversificacion**: se trata de una función que utilizamos cuando en la búsqueda tabú llevamos un número grande de vecindades sin mejorar la función objetivo, entonces lo que se realiza es buscar soluciones completamente diferentes para intentar por esa zona conseguir mejores soluciones.
- **Solucion_aleatoria**: función que se utiliza dentro de la Diversificación para obtener una solución aleatoria y a partir de ella buscar mejores soluciones que las que tenemos hasta el momento.
- **Intensificacion**: se utiliza después de haber realizado una serie de Diversificaciones y se trata de otro método diferente también para intentar mejorar la función objetivo, en éste método utilizamos una cualquiera de las mejores soluciones del pasado y a partir de ella cogemos una solución que no fuera la mejor.

A diferencia con los modelos anteriores, en este también están incluidas las funciones que serán utilizadas en la búsqueda tabú, ya que siempre se realizará la búsqueda para modelos de tipo tres.

5.2.2.5 Resultado

Para ir guardando los resultados que se van obteniendo en la búsqueda tabú se utiliza la clase Resultado:

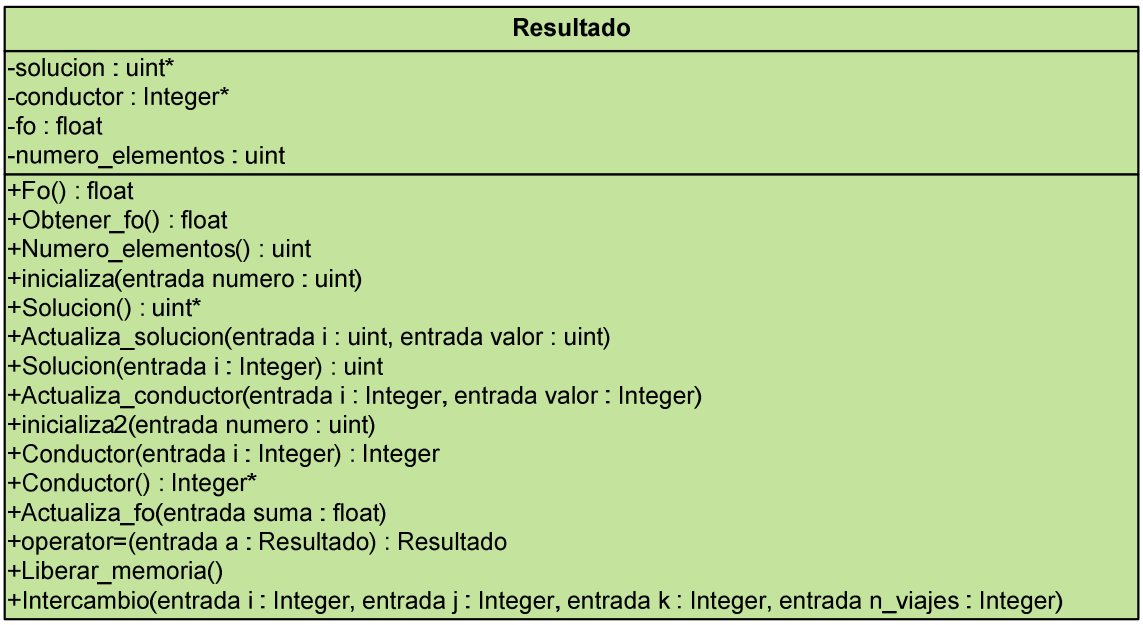

Figura 5.18: Especificación de la clase Resultado

Atributos que componen la clase:

- **solucion**: se trata de la solución del modelo, una tabla de tamaño numero_elementos compuesta por unos y ceros, donde cada uno nos indica que servicio es asignado a cada autobús.
- **conductor**: se trata de los conductores asignados a cada autobús.
- **fo**: la función objetivo de la solución encontrada.
- **numero_elementos**: el número de elementos de la solución, es la multiplicación de los servicios por los autobuses.

Y los métodos que se utilizan son los siguientes:

- **Fo**: devuelve el atributo fo.
- **Obtener_fo**: función que obtiene la función objetivo de la solución.
- **Numero_elementos**: devuelve el atributo numero_elementos.
- **Inicializa**: reserva memoria en la solución para el número de elementos que halla.
- **Solucion**: devuelve el atributo solución.
- **Actualiza_solucion**: actualiza un valor determinado de la solución con un valor que le pasamos.
- **Solucion**: devuelve el elemento de la solución que le pasamos.

- **Actualiza_conductor**: actualiza un valor determinado de los conductores con un valor que le pasamos.
- **inicializa2**: reserva memoria en los conductores para el número de elementos que halla.
- **Conductor**: devuelve el elemento de los conductores que le pasamos.
- **Conductor**: devuelve el atributo conductor.
- **Actualiza_fo**: actualiza el atributo fo.
- **operator=**: asignación a una clase Resultado los valores de otra ya existente.
- **Liberar_memoria**: libera el atributo solucion.
- **Intercambio**: función que se encarga de intercambiar en la solución los elementos i y k que se le pasan, se utiliza para realizar la vecindad.

5.2.2.6 Parámetros_tabú

Para la búsqueda tabú y antes de comenzarla se le tienen que pasar dos parámetros importantes como son el número de vecindades que se van a realizar y el tamaño que tendrá la lista de movimientos tabú, para estos parámetros se utilizan las siguientes clases.

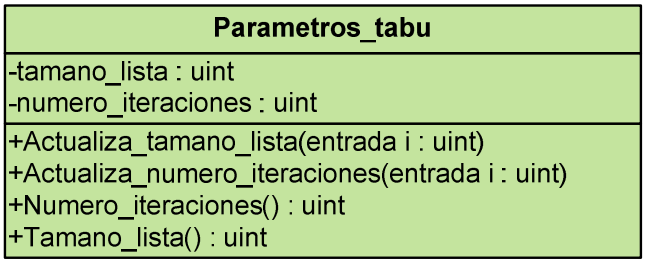

Figura 5.19: Especificación de la clase Parámetros_tabu

La clase incluye estos atributos:

- **tamano_lista**: tamaño de la lista de movimientos tabú.
- **numero_iteraciones**: número de veces que se repetirán las vecindades.

Los métodos que lleva asociados esta clase son:

- **Actualiza_tamano_lista**: actualiza el atributo tamano_lista.
- **Actualiza_numero_iteraciones**: actualiza el atributo numero_iteraciones.
- **Numero_iteraciones**: devuelve el atributo numero_iteraciones.
- **Tamano_lista**: devuelve el atributo tamano_lista.

5.2.2.7 Mov_tabu

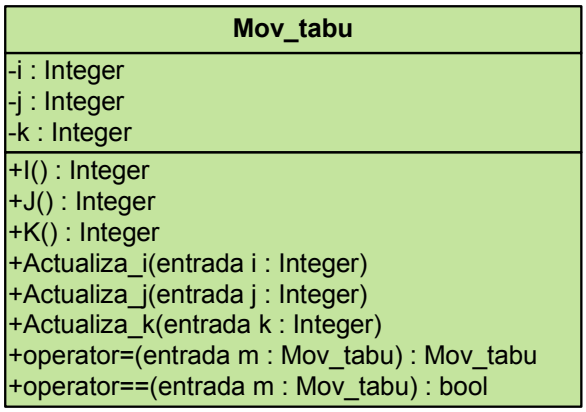

Figura 5.20: Especificación de la clase Mov_tabu

La clase está formada por estos atributos:

- **i**: es el número de autobús
- **j**: es el número del servicio.
- **k**: es por el autobús que se ha realizado el cambio un determinado servicio.

- Estos son los métodos que pertenecen a la clase:
- **I**: devuelve el atributo i.
- **J**: devuelve el atributo j.
- **K**: devuelve el atributo k.
- **Actualiza_i**: actualiza el atributo i.
- **Actualiza_j**: actualiza el atributo j.
- **Actualiza_k**: actualiza el atributo k.
- **operator=**: asignación a una clase Mov_tabu los valores de otra ya existente.
- **operator==**: compara dos clases Mov_tabu.

5.2.2.8 Lista_mov_tabu

Hay que resaltar que existe una lista de movimientos tabú, que nos sirve para no realizar movimientos que deshagan movimientos que se han realizado

previamente, por lo que se van incluyendo en esta lista y siempre antes de realizar cualquier movimiento se comprueba que no esté incluido en la lista.

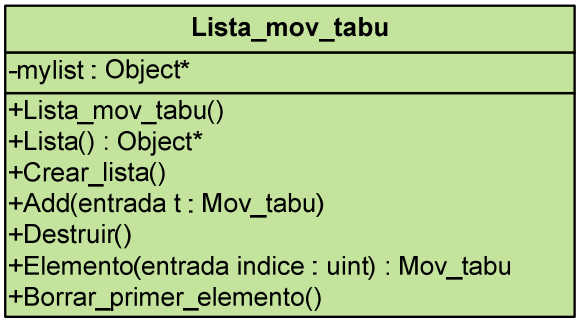

Figura 5.21: Especificación de la clase Lista_mov_tabu

El único atributo utilizado en la clase es:

- **mylist**: lista de movimientos tabú.

- Los métodos que compone la clase son:
- **Lista_mov_tabu**: se pone mylist a null.
- **Lista**: devuelve mylist.
- **Crear_lista**: crea una nueva lista.
- **Add**: se añade un movimiento tabú a la lista.
- **Destruir**: se borra mylist.
- **Elemento**: busca y devuelve el elemento de la lista tabú que le pasamos
- **Borrar_primer_elemento**: borra el primer elemento de la lista, se utiliza cuando la lista está completo y queremos insertar un nuevo movimiento tabú.