Trabajo de Fin de Carrera

Fidelidad al modelo especificado en el código generado automáticamente por la herramienta CASE *Poseidon for UML*

Alejandro Sánchez León

ETIG

Anna Queralt Calafat

10 de Enero del 2005

DEDICATORIA Y AGRADECIMIENTOS

DEDICATORIA

Dedico este trabajo a mi mujer Montse, que me impulsó inicialmente a matricularme en la UOC y que ha soportado pacientemente todos los fines de semana sacrificados por tener que dedicarlos al estudio de las asignaturas de esta carrera así como otras muchas otras tardes y noches de ausencia.

Gracias por su incansable apoyo, por haber creído más que yo en mis propias posibilidades, por su afecto y atenciones, por estar siempre ahí.

AGRADECIMIENTOS

Este trabajo supone el último escalón que debo subir para terminar una aventura que comenzó unos pocos años atrás y, es a aquellos quienes me han ayudado a subir todos los escalones a los que quisiera agradecer profundamente su presencia.

A mi mujer, por buscar la información de la UOC, matricularme y darme ese empujón sin el que nunca hubiera comenzado la carrera, así como por su soporte permanente y su fe mis posibilidades.

A mi amigo Federico Alonso, que me ha sabido transmitir los conceptos matemáticos como ningún profesor supo, con paciencia y con cariño, con perseverancia y resolución, con claridad y sencillez. Sin su inestimable ayuda nunca hubiera superado las asignaturas matemáticas.

A mi amigo Andreu Mendoza, que me ayudo a superar el miedo a matricularme, a fracasar en el intento y supo aconsejarme en mis primeros pasos en la UOC, haciendo posible que el primer semestre fuera todo un éxito, de forma que quedé ‘enganchado’ rápidamente a la carrera.

A mi tutor Antoni Urpí, por haberme ‘recordado’ las fechas y acontecimientos claves en el momento correcto y por haber respetado mi forma de encarar la carrera, tan lejos de las recomendaciones típicas de matrícula.

A mi consultora, Anna Queralt, que ha sabido definir con claridad los objetivos de este trabajo y me ha guiado con maestría e interés en las acciones necesarias para finalizarlo con éxito.

Al resto de mis amigos y familia, por haber creído en mí, haberme alentado y haber soportado mis negativas a vernos, por estar ocupado con los estudios.

A los padres del concepto de la UOC que han permitido que gente como yo, con imposibilidad de asistir presencialmente a las aulas, hayamos podido hacer realidad el sueño de estudiar una carrera, con contenidos actuales y vivos, con métodos pedagógicos modernos y con la libertad que permite este tipo de aprendizaje.

A todos, de nuevo, gracias.

RESUMEN

Este trabajo constituye el **Trabajo de Fin de Carrera en el área de Generación Automática de Software** del plan de estudios de la carrera de **Ingeniería Técnica de Informática de Gestión**.

El objetivo principal de este trabajo es estudiar el grado de fidelidad al modelo especificado que mantiene el código generado automáticamente por la herramienta CASE *Poseidon Professional for UML*. Concretamente estudiaremos el código que se genera en *Java* y *SQL* a partir de un diagrama de clases dado.

Se ha elegido utilizar la especificación de una aplicación de venta de billetes de avión por Internet como punto de partida para la prueba de los generadores, por tanto, primero analizaremos la funcionalidad básica que subyace en varias páginas Internet actuales dedicadas a la venta de billetes de avión.

Una vez identificada la funcionalidad básica, se introducirán en *Poseidon for UML* el *diagrama de casos de uso* y el *diagrama de clases* correspondiente a la aplicación que queremos generar, aunque la generación de código dependerá exclusivamente del diagrama de clases.

Tras completar la especificación en *Poseidon* pasaremos a probar los generadores de código fuente indicados anteriormente, estableciendo una comparación entre los mismos, destacando los puntos fuertes y débiles y su capacidad de reflejar el modelo diseñado.

Finalmente, se indicará qué haría falta añadir al código Java y SQL resultante de la generación automática para completar la aplicación de forma que fuera totalmente funcional.

PALABRAS CLAVE

Trabajo de fin de carrera

Treball de fi de carrera

Ingeniería Técnica de Informática de Gestión

Enginyeria Tècnica d'Informàtica de Gestió

ETIG

Generación Automática de Software

Generació automàtica de programari

Poseidon Professional for UML

Comparación de generadores de código fuente

Comparació de generadors de codi font

CASE

Aplicación de compra de billetes de avión por Internet

Aplicació de compra de bitllets de avió per Internet

Fidelidad del código generado al modelo original

Fidelitat del codi generat al model original

ÍNDICE DE CONTENIDOS

DEDICATORIA Y AGRADECIMIENTOS 2

DEDICATORIA 2
AGRADECIMIENTOS 2

RESUMEN 3

PALABRAS CLAVE 3

ÍNDICE DE CONTENIDOS 4

ÍNDICE DE FIGURAS 6

INTRODUCCIÓN 7

JUSTIFICACIÓN 7
OBJETIVOS 7
ENFOQUE Y MÉTODO A SEGUIR 7
PLANIFICACIÓN 8
PRODUCTOS OBTENIDOS 8
ESTRUCTURA DE LA MEMORIA 8

PREPARACIÓN DEL TRABAJO 10

POSEIDON FOR UML 10
Descripción 10
Descarga 11
Instalación 11
Toma de contacto 11
Visión general 11

FUNCIONALIDAD BÁSICA COMÚN A LA VENTA DE BILLETES DE AVIÓN 12

ESTUDIO DE PÁGINAS INTERNET 12
El ejemplo de http://www.viajar.com/vuelos/busqueda 12
FUNCIONALIDAD COMÚN 15
Funciones 15
Datos 15

REQUISITOS 19

CONTEXTO 19
Alcance del sistema 19
REQUISITOS FUNCIONALES 20
REQUISITOS NO FUNCIONALES 20
REQUISITOS DE INTERFAZ 20
Interfaz con el usuario 20
Interfaz con otros sistemas 20
CASOS DE USO 21
Caso de uso: Reserva de billetes de avión 21
Caso de uso: Búsqueda de vuelo 22
Caso de uso: Selección de vuelo 22
Caso de uso: Compra de vuelo 23
Caso de uso: Confirmación de compra 23
Caso de uso: Clasificación de itinerarios 24
Caso de uso: Visualización de detalle de itinerario 24
Caso de uso: Verificación de los datos de compra 24
Caso de uso: Verificación disponibilidad de plazas 25
Caso de uso: Verificación validez tarjeta de crédito 25
Caso de uso: Compra de billete 26
Caso de uso: Cancelación de billete 26
Caso de uso: Imprimir detalles de la compra 27

Diagrama de casos de uso *27*
CLASES 27
Clases candidatas *28*
Clase Aerolínea *28*
Clase Billete de avión *28*
Clase Compra *28*
Clase Financiera *28*
Clase Impresora *28*
Clase Itinerario *28*
Clase Pasajero *28*
Clase Tarjeta de Crédito *28*
Clase Usuario *28*
Clase Vuelo *29*
Diagrama de clases del dominio *29*

ANÁLISIS **31**

CASOS DE USO 31
CLASES 31
Clase Aerolínea *31*
Clase Billete *31*
Clase Compra *32*
Clase Direccion *33*
Clase Financiera *34*
Clase Itinerario *34*
Clase Pasajero *34*
Clase Persona *35*
Clase Tarjeta *35*
Clase Usuario *36*
Clase Vuelo *36*
Diagrama de Clases del análisis *38*

ESTUDIO DEL GENERADOR **39**

GENERALIDADES 39
GENERACIÓN DE CÓDIGO FUENTE JAVA 40
Opciones *40*
Revisión general del código generado *41*
Clases *41*
Asociaciones *44*
GENERACIÓN DE CÓDIGO FUENTE SQL 50
Opciones *50*
Revisión general del código generado *50*
Clases *50*
Asociaciones *51*

CONCLUSIONES **54**

COMPARACIÓN DE LOS GENERADORES 54
CARENCIAS DE LA APLICACIÓN GENERADA 55
CONCLUSIONES FINALES 56

GLOSARIO **57**

BIBLIOGRAFÍA **59**

DOCUMENTOS 59
NOTAS 59

ANEXOS **60**

ÍNDICE DE FIGURAS

ILUSTRACIÓN 1 - WWW.VIAJAR.COM / BÚSQUEDA DE VUELO 12
ILUSTRACIÓN 2 - WWW.VIAJAR.COM / SELECCIÓN DE VUELO 13
ILUSTRACIÓN 3 - WWW.VIAJAR.COM / DETALLE DEL VUELO 13
ILUSTRACIÓN 4 - WWW.VIAJAR.COM / OPERACIÓN DE COMPRA 14
ILUSTRACIÓN 5 - FUNCIONALIDAD BÁSICA DE LA VENTA DE BILLETES DE AVIÓN POR INTERNET 15
ILUSTRACIÓN 6 - DIAGRAMA DE CONTEXTO DE LA APLICACIÓN 19
ILUSTRACIÓN 7 - DIAGRAMA DE CASOS DE USO INTRODUCIDO EN POSEIDON 27
ILUSTRACIÓN 8 - DIAGRAMA DE CLASES INTRODUCIDO EN POSEIDON 29
ILUSTRACIÓN 9 - DIAGRAMA DE CLASES DEL ANÁLISIS 38
ILUSTRACIÓN 10 - ÁREA DE TRABAJO DE POSEIDON FOR UML 39
ILUSTRACIÓN 11 - IDENTIFICACIÓN INTERNA DE LOS OBJETOS POSEIDON 39
ILUSTRACIÓN 12 - OPCIONES DEL GENERADOR JAVA 40
ILUSTRACIÓN 13 - PROPIEDADES DE LAS CLASES 41
ILUSTRACIÓN 14 - PROPIEDADES DE LOS ATRIBUTOS 42
ILUSTRACIÓN 15 - PROPIEDADES DE LOS MÉTODOS 43
ILUSTRACIÓN 16 - PROPIEDADES DE LAS ASOCIACIONES 44
ILUSTRACIÓN 17 - CÓDIGO GENERADO SEGÚN MULTIPLICIDAD DE LA ASOCIACIÓN 45
ILUSTRACIÓN 18 - ASOCIACIÓN UNIDIRECCIONAL 45
ILUSTRACIÓN 19 - ASOCIACIÓN BIDIECCIONAL 45
ILUSTRACIÓN 20 - CÓDIGO JAVA EN ITINERARIO DE LA ASOCIACIÓN BIDIRECCIONAL 45
ILUSTRACIÓN 21 - CÓDIGO JAVA EN VUELO DE LA ASOCIACIÓN BIDIRECCIONAL 45
ILUSTRACIÓN 22 - EXTREMOS ORDENADOS EN LAS ASOCIACIONES 46
ILUSTRACIÓN 23 - PROPIEDADES DE LOS EXTREMOS DE UNA ASOCIACIÓN 46
ILUSTRACIÓN 24 - AGREGACIÓN CURIOSA 46
ILUSTRACIÓN 25 - EJEMPLO CLASES ASOCIATIVAS 47
ILUSTRACIÓN 26 - EJEMPLO ASOCIACIÓN MÚLTIPLE MULTIDIRECCIONAL 48
ILUSTRACIÓN 27 - EJEMPLO HERENCIA MÚLTIPLE 49
ILUSTRACIÓN 28 - PERFIL PARA EL GENERADOR SQL 50
ILUSTRACIÓN 29 - EJEMPLO ASOCIACIÓN MÚLTIPLE UNIDIRECCIONAL 53

INTRODUCCIÓN

JUSTIFICACIÓN

Este TFC permite comprobar con qué grado de fidelidad la herramienta CASE *Poseidon for UML Professional Edition* es capaz de representar un modelo dado en la generación de su código.

Los resultados de este estudio pueden ser aprovechados como apoyo a la toma de decisiones respecto a utilizar o no este software en el desarrollo de aplicaciones de tamaño y características similares.

Puesto que *Poseidon for UML* se considera una de las herramientas CASE más avanzadas en éste área, este estudio nos dará también una idea del grado de evolución que han alcanzado las herramientas CASE.

OBJETIVOS

Como ya hemos comentado, el objetivo principal es **comprobar cuan fieles al modelo especificado son el código Java y SQL generado por *Poseidon for UML Professional* Edition.**

Para cubrir dicho objetivo será necesario realizar las siguientes tareas:

- **Análisis de funcionalidad estándar** ofrecida por las páginas Internet de venta de billetes aéreos: Viajar.com, Amadeus, EasyJet, RyanAir...
- **Recogida de requisitos y especificación de la aplicación a desarrollar**, basada en la funcionalidad estándar detectada en las páginas anteriormente mencionadas.
- **Diagramación UML**: casos de uso, clases del análisis (con restricciones de integridad)
- **Estudio de la fidelidad generatriz de los generadores para Java y SQL**.
- **Carencias.** Qué añadiríamos al código para que pudiéramos considerar completa la aplicación generada automáticamente por *Poseidon* (código Java y SQL).
- **Conclusiones.** Las resultantes de las tareas anteriores.

ENFOQUE Y MÉTODO A SEGUIR

El primer paso consistirá en estudiar brevemente la funcionalidad básica que subyace en varias aplicaciones de venta de billetes actualmente activas en Internet. Navegaré por las páginas seleccionadas seleccionando las funciones y datos más comunes o lógicos. Las funciones nos ayudarán a diseñar los casos de uso y los métodos de las clases. Los datos nos ayudarán a definir los atributos de éstas últimas.

Una vez recogidas las especificaciones básicas de la aplicación, procederé a crear los *diagramas de casos de uso* y de *clases de la aplicación* dentro del mismo *Poseidon*, utilizando para ello las técnicas diagramáticas de la metodología UML. Puesto que la generación de la propia aplicación no es el objetivo de este trabajo, seguiré una metodología cercana a la definida en *[EP1]* y al método de desarrollo propuesto por *[Larman99]*, pero recogiendo solo aquellos elementos vitales para cubrir el verdadero objetivo de este trabajo.

En *Poseidon* se puede llegar a generar código tan solo a partir del *diagrama de clases*. Aunque permite dibujar prácticamente todos los diagramas descritos en la metodología UML, tan solo el anterior es relevante para la generación de código y, consecuentemente, será sobre éste sobre el que focalicemos nuestro trabajo y, no se desarrollará ningún otro diagrama UML.

Finalmente, utilizaremos un método de *ensayo y error* para comprobar cómo se comportan los generadores de código al cambiar las especificaciones del modelo original (por ejemplo, añadiendo estereotipos, cambiando el tipo de asociación, etc.).

Todos los productos de este TFC estarán sujetos a un ciclo de desarrollo *Iterativo e Incremental*, hasta la fecha de finalización del mismo.

PLANIFICACIÓN

La planificación gruesa viene determinada por las fechas fijadas por la dirección de este TFC y son las siguientes:

Fecha	Evento	Detalles
16/09/2004	**Inicio del proyecto**	
22/09/2004	**PAC1**	Planificación del proyecto
02/11/2004	**PAC2**	Análisis de requisitos Especificación de la aplicación Introducción de la especificación en Poseidón
09/12/2004	**PAC3**	Estudio comparativo de los generadores de Poseidón
10/01/2005	**Fin del Proyecto**	Estudio carencias de la aplicación generada en Java y SQL Memoria Presentación Ficheros del proyecto

La planificación detallada puede verse en el documento adjunto:
asanchezl_planificación.xls

PRODUCTOS OBTENIDOS

Los productos obtenidos durante la realización del TFC son los siguientes:

Producto	Tipo	Descripción
Asanchezl_pladetreball.doc	Documento Word	Documento que contiene la planificación del proyecto.
Asanchezl_planificación.xls	Documento Excel	Documento que contiene la planificación detalla del proyecto (tareas, fechas y status):
Asanchezl_memoria.doc	Documento Word	Este documento. La memoria del proyecto.
Asanchezl_presentación.ppt	Documento PowerPoint	Presentación resumen del proyecto realizado.
Asanchezl_tfc.zuml	Documento Poseidon	Fichero de trabajo que contiene el diagrama de casos de uso y el diagrama de clases utilizado para la generación de código.

ESTRUCTURA DE LA MEMORIA

La memoria se desarrollará abriendo un capítulo por cada una de las fases o etapas que haya sido necesario cubrir para llevar a buen término el trabajo.

El primer capítulo estará dedicado a la **Preparación del trabajo**, donde describiremos las características básicas de la herramienta CASE Poseidon (elegida por la dirección del TFC como generador de la aplicación propuesta).

Continuaremos con el **estudio de la funcionalidad básica común** a las páginas de venta de billetes de avión analizadas, que servirá de base para el siguiente capítulo, la **especificación de requisitos**, donde se formalizarán dichas funciones utilizando la metodología UML, a muy alto nivel.

Una vez definidos los requisitos, pasaremos al **análisis** de la aplicación, donde modelaremos la aplicación cuyo código queremos generar mediante Poseidon, a un nivel más cercano a la implementación.

Seguiremos con el **estudio del generador**, donde comprobaremos los puntos fuertes y débiles de los generadores de código Java y SQL que provee Poseidon y cómo se ajustan al modelo especificado.

Después revisaremos las carencias de la aplicación generada, indicando qué modificaríamos, borraríamos o añadiríamos para que la aplicación pudiera considerarse robusta, funcional y eficiente. Compararemos también ambos generadores y recogeremos nuestras impresiones en las **conclusiones**.

PREPARACIÓN DEL TRABAJO

En este apartado queremos introducir la herramienta cuyo comportamiento pretendemos evaluar.

POSEIDON FOR UML

DESCRIPCIÓN

Según sus creadores, las siguientes características definen a *Poseidon for UML*:

- Desarrollado en Java por entero, independiente de plataforma.
- Soporta los 9 diagramas de UML.
- Guardado en formato compatible con el Estándar de Intercambio de Diagramas UML 2.0.
- Soporta XMI 1.2 como el formato de guardado estándar. Pueden cargarse tanto XMI 1.0 como 1.2.
- Exportación de diagramas como gif, ps, eps y svg.
- Soporta formatos gráficos jpeg y png para JDK 1.4
- Copiar/cortar/pegar dentro de la herramienta.
- Arrastrar y Soltar dentro de la herramienta.
- Internacionalización y localización al Inglés, Alemán, Ruso, Francés, Español y Chino.
- Generación de código Java.

community

- Mecanismo plug-in para cargar y descargar plug-ins de nuestros socios tecnológicos, incluso mientras está en ejecución.
- Impresión confortable con ajuste a página o division en múltiples páginas.
- Copia directa al clipboard de Windows, Arrastrar y Soltar directo a Word, Powerpoint, etc.
- Generación de documentación HTML dentro de UMLdoc.

standard

- Generación basada en plantilla, con pleno acceso.
- Ingeniería de ida y vuelta para Java.
- Importación de JAR para inlcuir librerías ya existentes.
- Importación de ficheros Rational Rose (.mdl).
- Generación de código en VB.net.
- Generación de código en C#.
- Generación de código en C++.
- Generación de código en CORBA IDL.
- Generación de código en Delphi.
- Generación de código en Perl.
- Generación de código en PHP.
- Generación de código en DDL.

professional

Como podemos observar, existen 3 versiones de éste software. A nosotros nos interesa evaluar la versión *professional* puesto que es la que tiene el conjunto mayor de generadores automáticos de código software, siendo necesario el generador *DDL* específicamente para este trabajo.

DESCARGA

La versión descargada es **Poseidon for UML Professional Edition versión 2.5.**

Esta herramienta CASE se puede descargar desde la página web:

http://www.gentleware.com/products/download.php4

y la clave de evaluación puede obtenerse en la siguiente:

http://www.gentleware.com/shop/eval.php4?item=300

INSTALACIÓN

Existe también un manual de instalación disponible en la misma página de descarga o en el siguiente enlace:

http://www.gentleware.com/products/documentation/installguide/InstallGuide.pdf

TOMA DE CONTACTO

He descargado los **viewlets** disponibles en la siguiente página, para hacerme una idea global de la potencia y facilidad de uso del producto, como estrategia para preparar mejor la información con la que hay que alimentar a la herramienta.

http://www.gentleware.com/products/demo.php4

Los *viewlets* están en inglés, por lo que he preferido las versiones de texto, más fáciles de seguir que las de audio.

VISIÓN GENERAL

Aunque *Poseidon for UML* se declara como una herramienta *CASE* es, realmente, una herramienta que soporta cómodamente las técnicas diagramáticas *UML*. Su potencia generatriz de aplicaciones está limitada y se basa fundamentalmente en la generación de código a partir de la información contenida en el diagrama de clases introducido en la herramienta. El resto de diagramas que soporta la aplicación tienen función documental.

No existe tampoco la posibilidad de generar estructuras de aplicaciones a partir de patrones o modelos estandarizados incluidos en la herramienta.

Sin embargo, para el generador Java la herramienta facilita la tecnología de *ida y vuelta,* que permite incorporar en el diagrama los cambios introducidos manualmente en el código que previamente se había generado automáticamente.

FUNCIONALIDAD BÁSICA COMÚN A LA VENTA DE BILLETES DE AVIÓN

ESTUDIO DE PÁGINAS INTERNET

He realizado un estudio de varias páginas de Internet dedicadas a la venta de billetes de avión, con el objetivo de detectar la funcionalidad básica que subyace a todas ellas.

Las páginas analizadas han sido: www.viajar.com, www.amadeus.net, www.easyjet.com y www.ryanair.com.

Después de realizar las operaciones de reserva de billetes en cada una de ellas, considero que www.viajar.com es la más racional, en cuanto a uso, de todas ellas. Por ello, he centrado el estudio de la funcionalidad común basándome en ésta página.

EL EJEMPLO DE HTTP://WWW.VIAJAR.COM/VUELOS/BUSQUEDA

Este es el punto de entrada para el usuario que desea realizar una reserva de vuelos de avión. Como podemos ver, el primer paso consiste en rellenar ciertos criterios de búsqueda, en un formulario, y solicitar al sistema un listado de posibilidades mediante la pulsación del botón *buscar la mejor tarifa*.

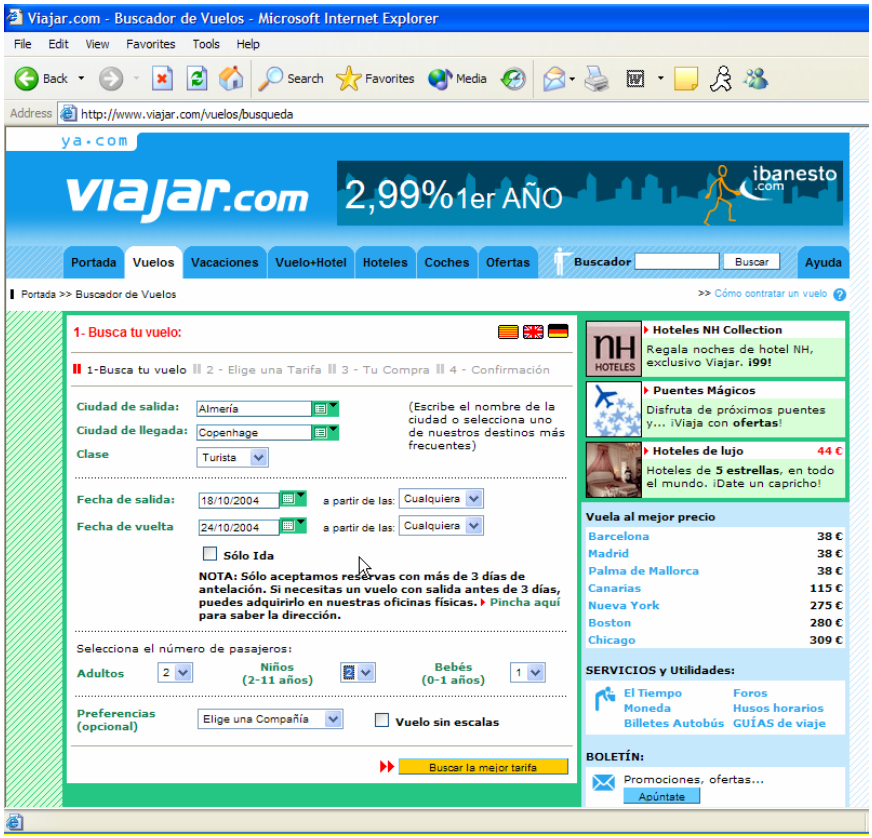

Ilustración 1 - www.viajar.com / búsqueda de vuelo

El sistema nos ofrece un listado de itinerarios que cumplen con los criterios de búsqueda indicados anteriormente. El listado está ordenado ascendentemente por precio del pasaje.

Existe la posibilidad de que el sistema realice búsqueda de tarifas más baratas dejando flexible la fecha de salida y la de llegada pero esta funcionalidad no es común al resto de páginas consultadas y, por tanto, no será tenida en cuenta en los requisitos.

En otras páginas hemos visto la posibilidad de volver a ordenar este listado de vuelos por otros criterios que, a mi entender, son también importantes: duración del vuelo y número de escalas. Esto será tenido en cuenta en la especificación de los requisitos.

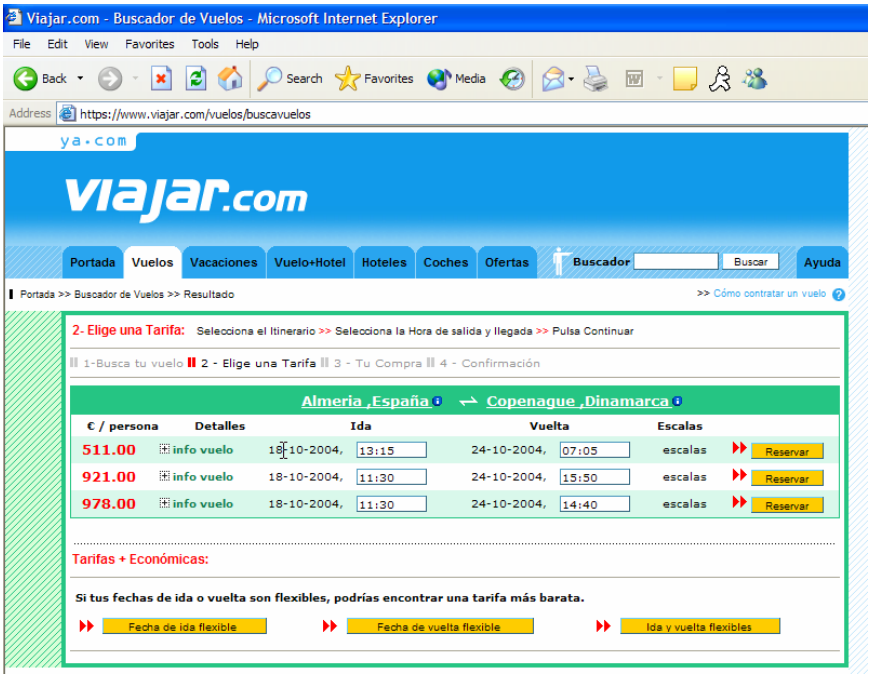

Ilustración 2 - www.viajar.com / selección de vuelo

Podríamos ver el detalle de cierto itinerario., pulsando sobre *+info vuelo* en uno de los itinerarios propuestos, como podemos ver a continuación:

Ilustración 3 - www.viajar.com / detalle del vuelo

Tras pulsar el botón *reservar* en la línea del listado deseada, el sistema nos presenta la pantalla de la compra, en la que constan los datos ya conocidos del vuelo, así como un formulario que permite introducir los nombres de los pasajeros, los datos de la dirección de envío de los billetes, la forma de pago, datos de la tarjeta de crédito, etc.

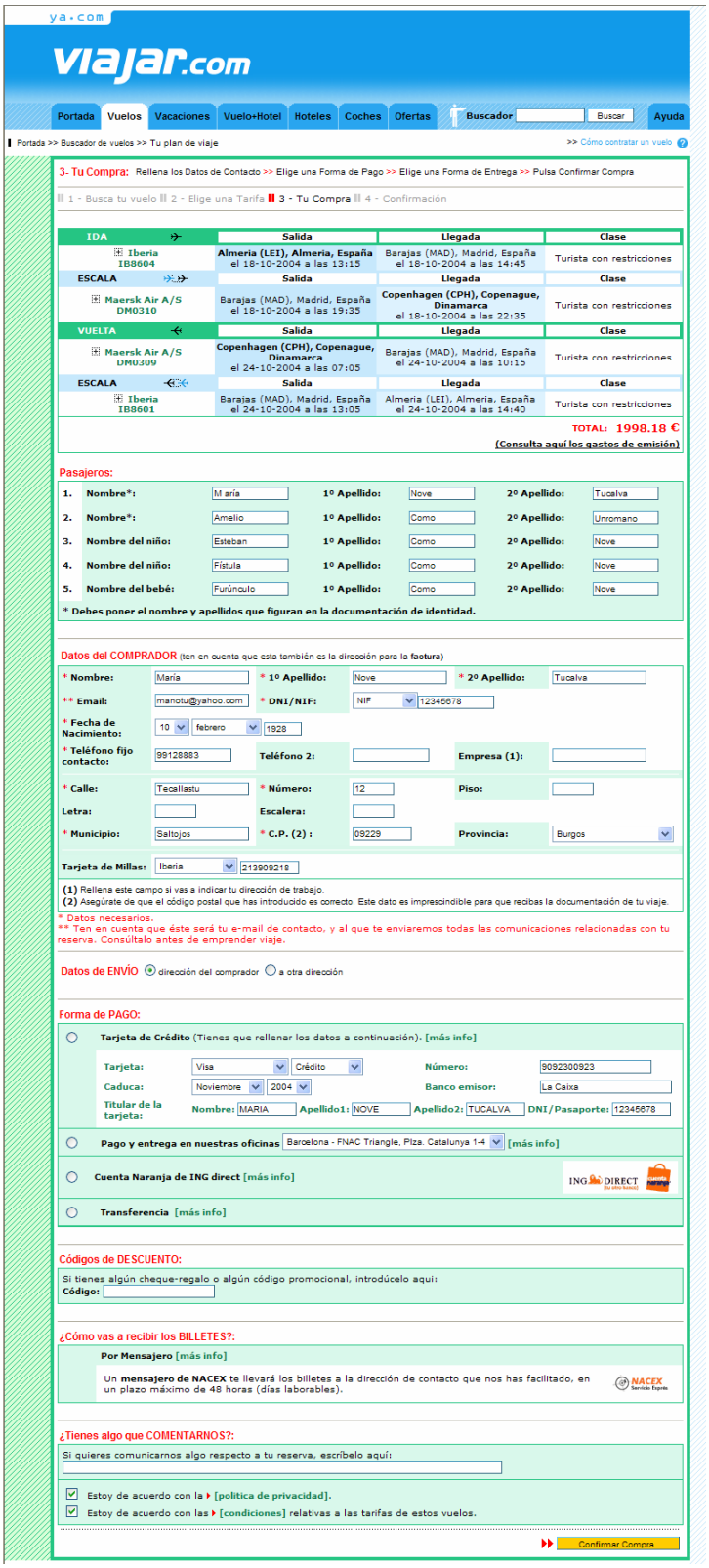

Ilustración 4 - www.viajar.com / operación de compra

Finalmente, al pulsar el botón *Confirmar compra*, el sistema realizaría los chequeos relativos a la bondad de los datos, a la disponibilidad de pasaje y validez de la tarjeta de crédito y, nos conduciría a una pantalla que recogería el resumen de la operación de compra (esta pantalla no ha sido capturada), realizándose todas las operaciones reales (débito en tarjeta, reserva de asiento...) al confirmar la compra.

FUNCIONALIDAD COMÚN

FUNCIONES

Podríamos recoger las funciones detectadas en la siguiente tabla:

Función
Obtención de un listado de itinerarios de vuelo, que responda a nuestros criterios de búsqueda.
Selección de uno de los itinerarios del listado, discrecionalmente
Compra de los billetes, mediante tarjeta de crédito.
Confirmación de la compra, una vez validados los datos y comprobada la disponibilidad de asientos.
Consultar datos detallados de un itinerario, incluso pueden llegar a detallarse los datos de las distintas escalas del vuelo y hasta los datos del avión.
Ordenar listado de vuelos por distintos criterios, como: precio, duración y número de escalas.
Validar datos de la tarjeta de crédito
Validar disponibilidad del vuelo (asientos para pasajeros en la clase seleccionada).

El siguiente esquema, ilustra muy genéricamente el flujo de las acciones habituales durante la compra de billetes de avión por Internet:

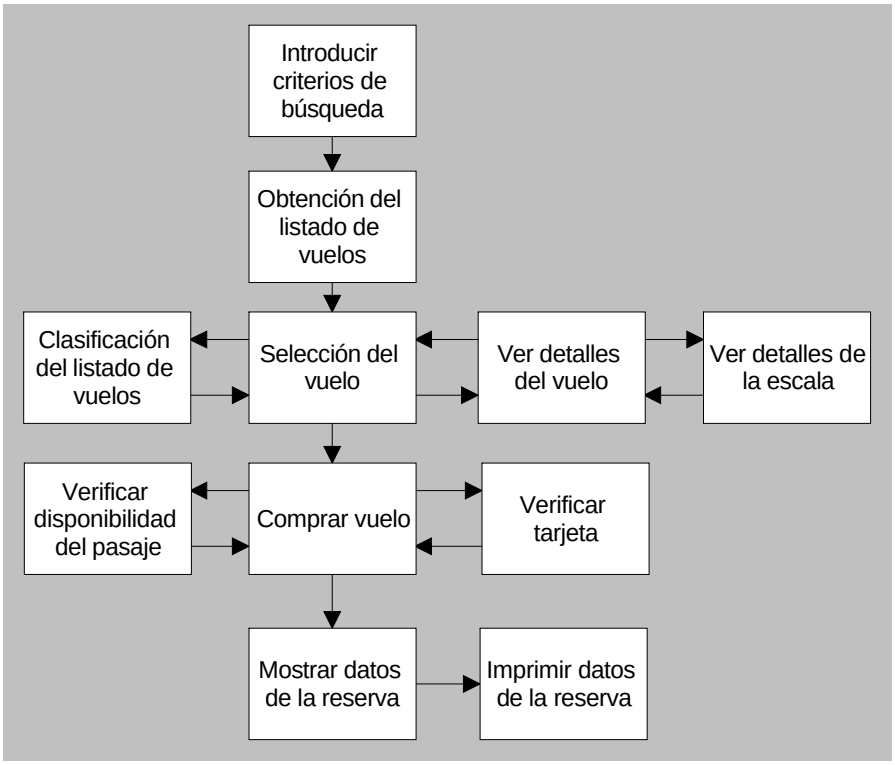

Ilustración 5 - Funcionalidad básica de la venta de billetes de avión por Internet

DATOS

Así como las funciones detectadas son importantes para definir los casos de uso, los datos son fundamentales para modelar clases y atributos. Indicamos a continuación los datos identificados como necesarios.

Criterios de búsqueda de vuelos

Como primer paso, el usuario debe indicar los criterios que servirán al sistema para realizar una búsqueda acotada de posibles vuelos.

La columna *Ob* indica la obligatoriedad del dato.

Dato	Comentario	Ob
Ciudad de Salida	Determina el o los aeropuertos de origen	X
Ciudad de Llegada	Determina el o los aeropuertos de destino	
Fecha de Salida		X
Fecha de Vuelta		
Hora salida a partir de	Hora a partir de la que se solicita el vuelo de ida	
Hora vuelta a partir de	Hora a partir de la que se solicita el vuelo de vuelta	
Clase	Clases: Primera, Business y Turista	
Flag Solo Ida	Indica que solo interesa el vuelo de ida	
Número de adultos	Número de pasajeros mayores de 12 años	
Número de niños	Número de pasajeros entre 2 y 12 años	
Número de bebés	Número de pasajeros con menos de 2 años.	
Compañía preferente	Compañía aérea preferente. Limita la selección de vuelos.	
Flag vuelo sin escalas	Indica que solo está interesado en vuelos directos entre las ciudades de origen y destino.	

Listado de itinerarios encontrados

El sistema ofrecerá al usuario un listado de itinerarios (secuencia de vuelos necesarios para unir un determinado aeropuerto de origen con otro de destino) que cumplen con los criterios de búsqueda indicados en el paso anterior.

Dato	Comentario
Aeropuerto de ida	
Fecha de salida	
Hora de salida	
Duración vuelo Ida	
Aeropuerto de vuelta	
Fecha de vuelta	
Hora de vuelta	
Duración vuelo vuelta	
Importe total vuelo	Suma de los precios por persona, aplicados según tipo de pasajero y tarifas vigentes.
Detalles del itinerario	Link que permite visualizar todos los detalles de las distintas conexiones (escalas) que forman parte del vuelo de ida y del vuelo de vuelta.

Detalles del Itinerario

Por cada uno de los vuelos o conexiones, el sistema presenta la siguiente información:

Dato	Comentario
Compañía aérea	
Vuelo	Código de vuelo
Hora salida	
Aeropuerto salida	
Ciudad salida	
País salida	
Hora llegada	
Aeropuerto llegada	
Ciudad llegada	
País llegada	

Detalles de la compra

Finalmente, una vez seleccionado el itinerario elegido, debemos formalizar la compra. Los datos de la compra se pueden separar en varios grandes grupos.

- **Detalles del itinerario**: contiene la misma información descrita en el apartado del mismo nombre.
- **Datos del comprador**: contiene los datos personales del comprador
- **Datos de los pasajeros**: contiene los datos necesarios para emitir los billetes a los pasajeros finales de la reserva
- **Datos de forma de pago**: contiene los datos relevantes a la forma en la que se va a realizar el pago (consideramos aquí solo el pago mediante tarjeta de crédito).
- **Datos de recogida de billetes**: contiene las directrices para que el usuario pueda recibir los billetes objeto de su reserva.

Datos de los pasajeros

Por cada uno de los pasajeros, deben indicarse los siguientes datos:

Dato	Comentario
Nombre	Obligatorio
Apellido 1	Obligatorio
Apellido 2	Obligatorio

Datos del comprador

Dato	Comentario	Ob.
Nombre		X
Apellido 1		X
Apellido 2		X
Fecha de nacimiento		X
NIF / DNI		X
Email		X
Teléfono de contacto 1		X
Teléfono de contacto 2		

Dato	Comentario	Ob.
Empresa		
Tipo de tarjeta de fidelización		
Tarjeta de fidelización		
{DIRECCION HABITUAL}	Ver Datos de dirección	X
Modo de envío	Modos: envío por mensajero a dirección de envío, recogida en oficinas, recogida en mostrador aeropuerto.	X
{DIRECCION DE ENVIO}	Ver *Datos de dirección*	
{DATOS DE PAGO}	Ver *Datos del pago*.	X

Datos de dirección

Para cada dirección, deben indicarse los siguientes datos:

Dato	Comentario	Ob.
Calle		X
Número		X
Piso		
Letra		
Escalera		
Municipio		X
Código Postal		X
Provincia		
País		X

Datos del pago

Dato	Comentario	Ob.
Tipo de tarjeta de crédito	Seleccionar una: VISA, American Express, MasterCard.	X
Tipo de operación	Seleccionar: crédito o débito	X
Tarjeta de crédito	Número de tarjeta de crédito	X
Nombre titular	Nombre del titular	X
Apellido 1 titular	Primer apellido del titular	X
Apellido 2 titular	Segundo apellido del titular	X
Mes caducidad		X
Año caducidad		X
Banco emisor		
DNI / Pasaporte		X

REQUISITOS

CONTEXTO

Creo necesario recordar aquí que la aplicación a especificar no es el objetivo principal de este TFC, sino una excusa para poder comprobar la fidelidad generatriz de *Poseidon*, por ello, no entraremos a fondo en el conocimiento de la operación real y presupondremos un contexto determinado que explicaremos a continuación.

Alejándonos de UML, utilizaremos el *Diagrama de Contexto* de la metodología de *Análisis estructurado moderno* de Yourdon *[Yourdon93]* vista en *[EP2]*, para obtener una visión global de los flujos del sistema y de su interacción con otras entidades y otros sistemas externos. Esto nos ayuda rápidamente a comprender las actividades principales del sistema.

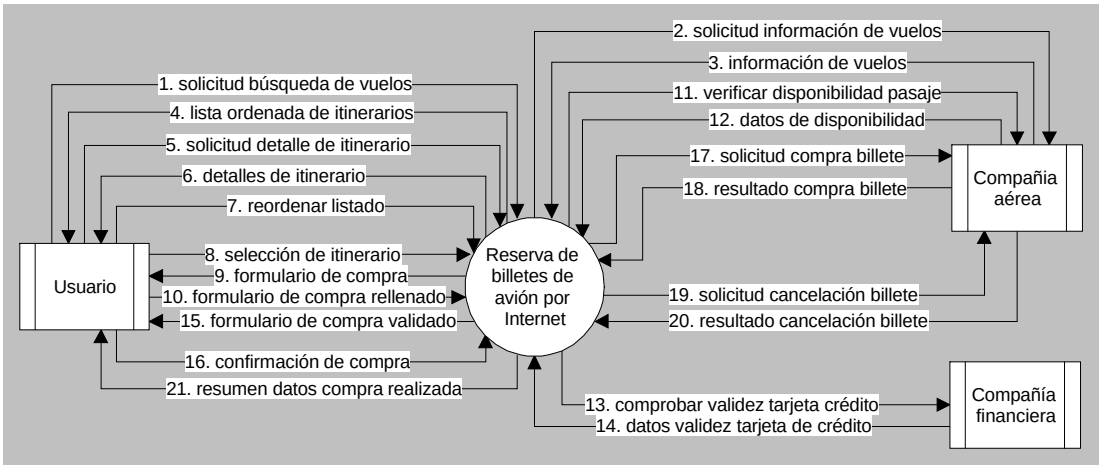

Ilustración 6 - diagrama de contexto de la aplicación

Como podemos ver, existen básicamente 3 entidades que interactúan con el sistema: el **usuario** y las entidades externas **compañía aérea** y **compañía financiera**.

Definimos aquí la **compañía aérea** como una caja negra a la que se solicita información de vuelos, se comprueba disponibilidad de pasajes y se realiza la reserva real de pasajes. Para este TFC no importa si conectamos con varias entidades externas o con una sola.

También definimos aquí como **compañía financiera** una caja negra que simboliza la entidad o entidades capaces de validar la tarjeta de crédito. Accedemos a ella para validar la tarjeta de crédito. Se supone que la compañía aérea también accederá a ella para realizar los cargos y abonos pertinentes por la compra de billetes.

ALCANCE DEL SISTEMA

Para facilitar las tareas de modelado nos parece conveniente establecer las siguientes suposiciones:

1. Los datos maestros (los no transaccionales) serán generados y mantenidos por otras aplicaciones. El sistema consultará los mismos pero no será responsable de su actualización.
2. El sistema solicitará la lista de posibles vuelos a la entidad denominada **compañía aérea**, pasando como parámetros de búsqueda la *ciudad,* las *fechas* límite y las *horas* límite. La **compañía aérea** retornará una lista de vuelos que cumplen dichas condiciones.
3. El sistema delegará el control de disponibilidad de pasaje a la **compañía aérea**, recibiendo de esta la respuesta de los asientos reservados para cada uno de los pasajeros indicados.

4. El sistema delegará la reserva real de plazas en los vuelos a la **compañía aérea**. Esta última retornará una lista de billetes que confirman dichas reservas.
5. El sistema delegará la verificación de la validez de la tarjeta de crédito a la **compañía financiera**, obteniendo como resultado un si o un no.
6. El sistema delegará la operación financiera de cargo o abono a la **compañía aérea**, como parte del proceso de compra de billete, recibiendo como respuesta el número de operación obtenido de la compañía financiera.
7. El sistema guardará exclusivamente los datos directamente relacionados con la operación de reserva.

REQUISITOS FUNCIONALES

La mayoría de los requisitos funcionales ya han sido determinados durante el análisis de la funcionalidad básica de las páginas de Internet. Los recogemos aquí y completamos.

Requisito	Descripción
RF01	El sistema presentará al **usuario** un *listado de itinerarios de vuelo* que corresponda con ciertos criterios de búsqueda que el mismo **usuario** habrá indicado previamente. El listado estará ordenado ascendentemente por importe de compra.
RF02	El **usuario** podrá solicitar al sistema ordenar el *listado de itinerarios de vuelo* por tres conceptos: *importe*, *duración del vuelo* y *número de escalas*.
RF03	El **usuario** podrá ver el *detalle* de cualquiera de los itinerarios del *listado*.
RF04	El **usuario** podrá solicitar la compra de uno de los *itinerarios* propuesto en el *listado*.
RF05	El sistema verificará la validez de los *datos de compra*, apoyándose en la **compañía aérea** y en la **entidad financiera**.
RF06	El sistema realizará la compra efectiva de billetes de avión para los pasajeros y vuelos indicados por el usuario, delegando en la **compañía aérea**,

REQUISITOS NO FUNCIONALES

Se supone que la aplicación será robusta, eficiente, amigable y fácil de mantener.

- La interfaz en la que el usuario desarrolle su trabajo será una página Internet.
- La única forma de pago que soportará el sistema será mediante tarjeta de crédito.

REQUISITOS DE INTERFAZ

INTERFAZ CON EL USUARIO

La interfaz con el usuario serán páginas de Internet que contendrán partes estáticas y dinámicas, así como formularios para llevar a cabo las distintas operaciones descritas. El usuario necesitará al menos un teclado y se aconseja también la utilización de un ratón.

No existen restricciones respecto al tipo de páginas dinámicas o al servidor Internet a utilizar.

INTERFAZ CON OTROS SISTEMAS

Como ya hemos indicado, serán necesarias al menos las interfaces:

- Con la **compañía aérea**, para la obtención de vuelos, la verificación de disponibilidad de plazas y la compra de billetes (con cargo y abono real a la tarjeta de crédito).
- Con la **compañía financiera**, para la validación de los datos de la tarjeta de crédito.
- Con el **sistema gestor de datos maestros**, para la obtención de los datos maestros que utilizará la aplicación y que habrán sido introducidos y mantenidos por este otro sistema.

Suponemos todas interfases definidas y accesibles a partir de una clase definida específicamente para cada sistema.

CASOS DE USO

Un caso de uso "*es un documento narrativo que describe la secuencia de eventos de un actor (un agente externo) que usa un sistema para completar un proceso [Jackobson92]* "[Grau]. Vamos a describir textualmente cada caso de uso y, finalmente, recogeremos todos ellos en el *diagrama de casos de uso de los requisitos*.

Podríamos decir que el caso de uso *Reserva de billetes de avión* es el núcleo de control del resto de casos de uso, que podrían considerarse subcasos de éste.

CASO DE USO: RESERVA DE BILLETES DE AVIÓN

Nombre del caso de uso	Reserva de billetes de avión
Actores	**Usuario** (iniciador)
Propósito	Reservar billetes de avión (ida o ida y vuelta) para uno o más vuelos y para uno o más pasajeros determinados.
Visión general	El **usuario** introduce los criterios de búsqueda de un vuelo y recibe del sistema una lista de posibles itinerarios ordenada ascendentemente por importe del vuelo, que cumplen dichas condiciones. El **usuario** puede clasificar la lista según distintos criterios: importe, duración del vuelo o número de escalas....) El **usuario** puede consultar información detallada de un itinerario. El **usuario** selecciona uno de los itinerarios y solicita iniciar el proceso de compra. El sistema proporciona un formulario para realizar la compra, el **usuario** lo rellena y realiza la compra de los billetes para los pasajeros indicados. El sistema confirma al **usuario** la operación de compra y la reserva de los pasajes. Este caso de uso es la suma de los casos de uso señalados en las referencias.
Tipo	Primario y esencial
Referencias	Búsqueda de vuelo Selección de vuelo Compra de vuelo Confirmación de compra
Curso típico de eventos	La operación se inicia con la conexión del usuario a Internet. Generalmente, la acción se desarrolla siguiendo el flujo de los casos de uso indicados en las referencias: 1. búsqueda de vuelo 2. selección de vuelo 3. compra de vuelo 4. confirmación de compra Para mayor información, revisar dichos casos de uso.
Cursos alternativos	En cualquier momento se puede interrumpir este flujo, sin que sea necesario completar todos los pasos (decisión de abortar la operación por parte del **usuario**).

CASO DE USO: BÚSQUEDA DE VUELO

Nombre del caso de uso	Búsqueda de vuelo
Actores	**Usuario** (iniciador) **Compañía aérea**
Propósito	Retornar una lista de itinerarios de vuelos que cumplan los criterios de búsqueda introducidos por el usuario.
Visión general	El **usuario** facilita los criterios de búsqueda al sistema. El sistema solicita información de vuelos a la **compañía aérea**, con los que puede construir distintos itinerarios que satisfagan las condiciones de búsqueda del **usuario**. El sistema proporciona al **usuario** un *listado de itinerarios* que cumplen los requisitos de búsqueda, ordenado ascendentemente por importe de compra. Se inicia el caso de uso selección de vuelo.
Tipo	Primario y esencial
Referencias	Reserva de billetes de avión Selección de vuelo
Curso típico de eventos	1. el **usuario** proporciona los criterios de búsqueda 2. el sistema solicita información a la **compañía aérea** y construye itinerarios posibles. 3. el sistema devuelve al **usuario** un listado de itinerarios que cumplen los criterios de búsqueda, ordenado por importe.
Cursos alternativos	El usuario podría abandonar la aplicación sin iniciar la Selección de vuelo.

CASO DE USO: SELECCIÓN DE VUELO

Nombre del caso de uso	Selección de vuelo
Actores	**Usuario** (iniciador)
Propósito	Revisar la lista de itinerarios propuesta por el sistema y seleccionar un vuelo para iniciar el proceso de compra.
Visión general	Una vez se ha obtenido el listado mediante el caso de uso búsqueda de vuelo, el usuario revisa la información y selecciona uno de los itinerarios, acción que iniciará el proceso de compra.
Tipo	Primario y esencial
Referencias	Reserva de billetes de avión Búsqueda de vuelo Compra de vuelo Clasificación de itinerarios Visualizar detalle de itinerario
Curso típico de eventos	1. El **usuario** puede solicitar clasificar el listado por otro criterio: precio, duración del vuelo, número de escalas..., lo que activaría el caso de uso: Clasificación de itinerarios 2. El **usuario** podría solicitar ver más detalles de uno de los itinerarios, para lo cual se utilizaría el caso de uso Visualizar detalle de itinerario. 3. El **usuario** seleccionará uno de los itinerarios del listado 4. El **usuario** solicitará iniciar la compra del itinerario seleccionado
Cursos alternativos	El **usuario** podría estar insatisfecho con el listado obtenido, volviendo al caso de uso: Búsqueda de vuelo).

CASO DE USO: COMPRA DE VUELO

Nombre del caso de uso	Compra de vuelo
Actores	**Usuario** (iniciador)
Propósito	Permite al **usuario** indicar los datos relevantes para poder realizar la compra de los billetes deseados para los pasajeros y vuelos deseados.
Visión general	Una vez el **usuario** ha seleccionado un itinerario, el **usuario** debe poder introducir todos los datos relevantes a la operación de compra de billetes de avión para los pasajeros y conexiones determinadas en pasos anteriores. El sistema deberá realizar una comprobación de la validez de los datos de dicho formulario y solicitará a la **compañía aérea** la verificación de disponibilidad de plazas y, a la **compañía financiera** la validación de la tarjeta de crédito proporcionada por el **usuario.** Si el resultado de las validaciones es correcto, el usuario podrá continuar con el caso Confirmación de la compra**.**
Tipo	Primario y esencial
Referencias	Reserva de billetes de avión Selección de vuelo Confirmación de compra Verificación de los datos de compra
Curso típico de eventos	1. El sistema presenta al **usuario** los datos ya conocidos del itinerario seleccionado en el caso selección de vuelo**.** 2. El **usuario** introduce el resto de datos relevantes para la compra de billetes. 3. El **usuario** solicita iniciar el proceso de compra. 4. El sistema comprueba la validez de los datos introducidos, llamando al caso de uso Verificación de los datos de compra
Cursos alternativos	El sistema puede detectar falta de información obligatoria en el formulario o bien, que no existen plazas suficientes en algún vuelo, o bien que los datos de la tarjeta no son válidos. En caso de error, será mostrado el error y solicitada la corrección de los datos introducidos. Opcionalmente, el usuario podrá abortar la operación y volver al caso búsqueda de vuelo.

CASO DE USO: CONFIRMACIÓN DE COMPRA

Nombre del caso de uso	Confirmación de compra
Actores	**Usuario** (iniciador)
Propósito	Una vez verificada la validez de la información proporcionada por el **usuario** se procede a la reserva real efectiva de los billetes solicitados, cargando la tarjeta de crédito.
Visión general	Los datos relevantes para la compra que ha indicado el **usuario** habrán sido validados y dados por correctos, en el caso de uso compra de vuelo. El sistema solicitará uno a uno la compra de los billetes de avión a la **compañía aérea**. La **compañía aérea** retornará los detalles del billete de avión comprado y de la operación del cargo bancario realizado a la tarjeta de crédito.
Tipo	Primario y esencial
Referencias	Reserva de billetes de avión Compra de vuelo Compra de billete Cancelación de billete Imprimir detalles de la compra
Curso típico de eventos	1. El sistema pedirá a la **compañía aérea** la compra de cada uno de los billetes que intervienen en la compra del itinerario que ha solicitado el **usuario**, llamando al caso de uso Compra de billete.

	2. El sistema recogerá los datos de los distintos billetes comprados y mostrará al **usuario** el resultado final de la compra. 3. El usuario podrá imprimir los detalles de la operación de compra ya finalizada, mediante el caso de uso: imprimir detalles de la compra.
Cursos alternativos	Si la **compañía aérea** no es capaz de emitir un billete (por falta de plazas o por fallo en la operación de cargo en la tarjeta de crédito), entonces el sistema solicitará a la **compañía aérea** la cancelación de cada uno de los billetes que haya emitido hasta el momento, mediante el caso de uso cancelación de billete**.** El **usuario** podrá elegir cambiar los datos de compra o bien volver a la búsqueda de vuelos.

CASO DE USO: CLASIFICACIÓN DE ITINERARIOS

Nombre del caso de uso	Clasificación de itinerarios
Actores	**Usuario** (iniciador)
Propósito	Clasificar una lista de itinerarios dada por un concepto determinado: precio, duración del vuelo, número de escalas y, retornar la lista nuevamente clasificada.
Visión general	Recibe como entrada una lista de itinerarios y un criterio de clasificación, devolviendo la misma lista clasificada por dicho criterio.
Tipo	Secundario y esencial
Referencias	Selección de vuelo
Curso típico de eventos	1. El **usuario** solicita la clasificación de la *lista de itinerarios* por cierto criterio. 2. El sistema clasifica la lista y la vuelve a mostrar en el orden solicitado
Cursos alternativos	No existen

CASO DE USO: VISUALIZACIÓN DE DETALLE DE ITINERARIO

Nombre del caso de uso	Visualización de detalle de itinerario
Actores	**Usuario** (iniciador)
Propósito	Visualizar los detalles de un itinerario de la lista propuesta por el sistema.
Visión general	El **usuario** selecciona ver los detalles de uno de los itinerarios de la lista propuesta por el sistema y el sistema muestra dicha información
Tipo	Secundario y esencial
Referencias	Selección de vuelo
Curso típico de eventos	1. El **usuario** solicita visualizar detalles de cierto itinerario de la lista de itinerarios propuesta por el sistema. 2. El sistema muestra todos los detalles disponibles de dicho itinerario
Cursos alternativos	No existen

CASO DE USO: VERIFICACIÓN DE LOS DATOS DE COMPRA

Nombre del caso de uso	Verificación de los datos de compra
Actores	**Usuario** (iniciador)
Propósito	Una vez que el usuario ha indicado los datos relevantes para la compra, el usuario debe verificar la corrección de la información proporcionada, antes de iniciar la compra real de billetes.
Visión general	Este caso de uso es llamado por el caso de uso Compra de vuelo, una vez que el usuario ha solicitado la acción de iniciar la compra.

	El sistema verifica las posibles inconsistencias de cada dato individual y de datos relacionados entre si. El sistema llama al caso de uso verificación disponibilidad de plazas por cada uno de los billetes que comprende el itinerario seleccionado por el usuario. El sistema utiliza también el caso de uso Verificación de tarjeta de crédito para comprobar la validez de la misma. Si las verificaciones son correctas, el sistema inicia el caso de uso Confirmación de la compra.
Tipo	Primario y esencial
Referencias	Compra de vuelo Verificación disponibilidad de plazas Verificación validez tarjeta crédito Confirmación de la compra
Curso típico de eventos	1. El sistema verifica datos generales del formulario. 2. El sistema llama al caso de uso *Verificación disponibilidad de plazas* para comprobar que hay plazas en todos los vuelos. 3. El sistema llama al caso de uso *Verificación validez tarjeta de crédito* para asegurarse de que la tarjeta está activa y es reconocida por el sistema financiero.
Cursos alternativos	Si alguna de las comprobaciones anteriormente mencionada falla, el sistema debe indicar el problema y derivar de nuevo al caso de uso Compra de vuelo.

CASO DE USO: VERIFICACIÓN DISPONIBILIDAD DE PLAZAS

Nombre del caso de uso	Verificación disponibilidad de plazas
Actores	**Compañía aérea**
Propósito	Comprobar que, en el momento de la consulta, existe plaza para el pasajero indicado, en el vuelo y clase determinados.
Visión general	Este caso es activado por el caso de uso Verificación de los datos de compra para comprobar que hay plazas suficientes en la lista de vuelos que forman parte del itinerario que ha elegido el usuario. Debe verificarse cada uno de los billetes que finalmente será necesario emitir para todos los pasajeros y vuelos seleccionados en los datos de compra.
Tipo	Primario y esencial
Referencias	Verificación disponibilidad de datos de compra
Curso típico de eventos	1. El sistema solicitará la verificación de plazas a la **compañía aérea**, indicando el vuelo y clase requeridos. 2. La **compañía aérea** responderá afirmativamente o negativamente a dicha petición.
Cursos alternativos	No existen

CASO DE USO: VERIFICACIÓN VALIDEZ TARJETA DE CRÉDITO

Nombre del caso de uso	Verificación validez tarjeta de crédito
Actores	**Compañía financiera**
Propósito	Verificar la validez de los datos de la tarjeta de crédito indicada para hacer la compra de billetes.
Visión general	Este caso es activado por el caso Verificación de los datos de compra. El sistema solicita a la **compañía financiera** comprobar la validez de la tarjeta de crédito cuyos datos ha proporcionado el **usuario**.

	La **compañía financiera** retorna el resultado de la verificación.
Tipo	Primario y esencial
Referencias	Verificación de los datos de compra
Curso típico de eventos	1. El sistema solicita a la **compañía financiera** la comprobación de la validez de la tarjeta de crédito cuyos datos ha proporcionado el usuario. 2. La **compañía financiera** realiza la comprobación y retorna al sistema el resultado de la misma.
Cursos alternativos	No existen

CASO DE USO: COMPRA DE BILLETE

Nombre del caso de uso	Compra de billete
Actores	**Compañía aérea**
Propósito	Realizar la compra real de uno de los billetes implicados en el itinerario seleccionado previamente por el **usuario**.
Visión general	Este caso es activado por el caso de uso Confirmación de la compra. El sistema solicita la compra de billete por cada combinación de pasajero / vuelo que forman parte de la operación de compra que ha iniciado el **usuario**. Este caso de uso representa cada una de las compras individuales de los billetes relacionados.
Tipo	Primario y esencial
Referencias	Confirmación de la compra
Curso típico de eventos	1. El sistema solicita a la **compañía aérea** la compra de un billete para un pasajero y vuelo determinados, indicando la clase de billete. 2. La **compañía aérea** realiza la reserva real de la plaza para el pasajero y el cargo a la tarjeta de crédito proporcionada, retornando la información del billete generado y de la transacción comercial realizada
Cursos alternativos	La compra del billete podría fallar por no existir suficientes plazas en el momento de la compra o bien por no haber podido realizar el cargo a la tarjeta de crédito.

CASO DE USO: CANCELACIÓN DE BILLETE

Nombre del caso de uso	Cancelación de billete
Actores	**Compañía aérea**
Propósito	Anular un billete que ha sido comprado previamente durante el proceso de compra de billetes de avión.
Visión general	Este caso es activado por el caso *Confirmación de la compra*, solo en el caso de que no haya sido posible comprar alguno de los billetes que componen el itinerario seleccionado por el **usuario**. El sistema solicita a la **compañía aérea** la cancelación de un billete que previamente había solicitado.
Tipo	Primario y esencial
Referencias	Verificación de los datos de compra
Curso típico de eventos	1. El sistema solicita a la **compañía aérea** la cancelación de un billete que previamente había solicitado. 2. La **compañía aérea** retorna el resultado de la operación.
Cursos alternativos	Es posible que no sea posible cancelar el billete de avión

CASO DE USO: IMPRIMIR DETALLES DE LA COMPRA

Nombre del caso de uso	Imprimir detalles de la compra
Actores	**Usuario** (iniciador) **Impresora**
Propósito	Permite imprimir todos los detalles de la compra en un formato adecuado a una impresora.
Visión general	Una vez confirmada la compra, el sistema visualiza todos los detalles de la misma. El **usuario** puede, entonces, decidir imprimir los datos para su uso partícular.
Tipo	Secundario y esencial
Referencias	Confirmación de la compra
Curso típico de eventos	1. El **usuario** selecciona imprimir los datos de compra. 2. El sistema imprime por la **impresora** por defecto del **usuario** los datos de detalle de la compra realizada.
Cursos alternativos	No existen

DIAGRAMA DE CASOS DE USO

Los casos de uso descritos anteriormente pueden verse reflejados en el siguiente diagrama UML.

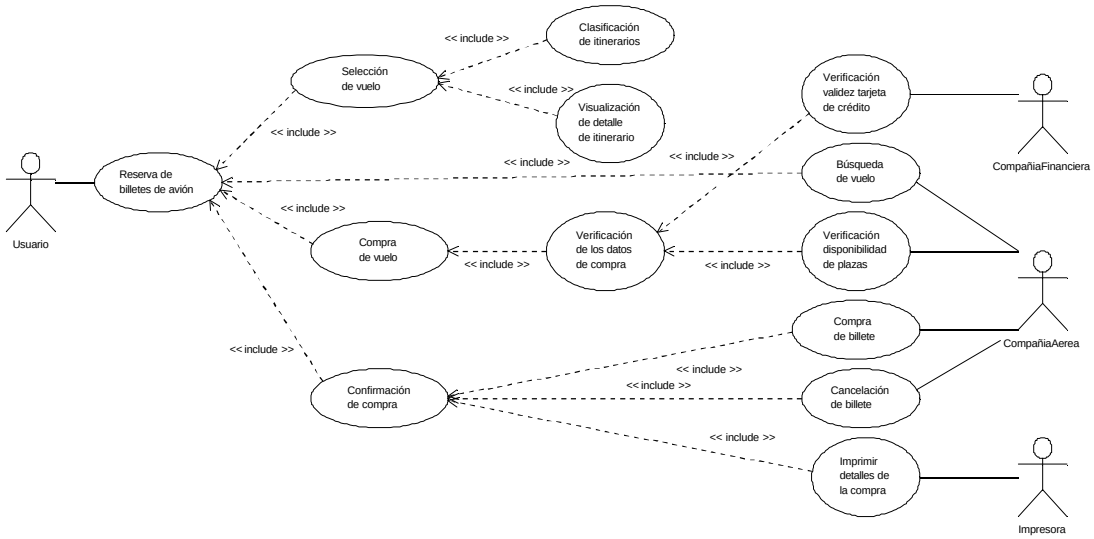

Ilustración 7 - Diagrama de casos de uso introducido en Poseidon

CLASES

Del análisis de las páginas Internet y de los casos de uso definidos en el apartado anterior, se pueden extraer las clases que se indicarán a continuación.

En este apartado, las clases se especifican a alto nivel, no indicando ni atributos ni métodos. Éstos serán añadidos en la fase de análisis.

Además, aquí se reflejan las clases que corresponden a las entidades que son visibles desde el negocio, no teniendo en cuenta clases de implementación (gestores de BBDD, gestores de comunicaciones, clases auxiliares...), las cuales serían visibles en la fase de diseño.

CLASES CANDIDATAS

Las siguientes entidades parecen candidatas a clases:

Aerolínea	Billete	Compra	Financiera
Impresora	Itinerario	Pasajero	Tarjeta
Usuario	Vuelo		

A continuación describimos someramente cada clase.

CLASE AEROLÍNEA

Para la aplicación, una aerolínea es aquella entidad que pone a disposición del usuario vuelos.

CLASE BILLETE DE AVIÓN

Un billete de avión justifica la operación de compra de un pasaje para un pasajero en un vuelo y una clase determinados.

CLASE COMPRA

Una compra es el conjunto de billetes de vuelo que ha reservado el usuario para uno o más pasajeros, utilizando cierta tarjeta de crédito y para un determinado itinerario de los propuestos por el sistema.

CLASE FINANCIERA

Para la aplicación, una financiera es una entidad financiera capaz de validar los datos de una tarjeta de crédito con la que se efectuará el pago de la reserva de vuelos.

CLASE IMPRESORA

Una impresora es un dispositivo de impresión por el que podrán imprimirse los detalles

CLASE ITINERARIO

Un itinerario es un grafo de uno o más vuelos que unen la ciudad de origen y destino que ha solicitado el usuario. Cada vuelo se considera una escala o conexión en el itinerario y requiere un billete específico.

CLASE PASAJERO

Un pasajero es una persona que adquiere un billete para reservar una plaza en un vuelo determinado, en una categoría determinada.

CLASE TARJETA DE CRÉDITO

Representa a cualquiera de las tarjetas de crédito comerciales habituales (VISA, American Express, MasterCard...) con las que se pagará la compra de los billetes de vuelo.

CLASE USUARIO

La clase usuario representa a cualquier usuario conectado a Internet que desea realizar una reserva de billetes de avión.

Un usuario puede reservar uno o más vuelos, para una o más personas (pasajeros).

CLASE VUELO

Un vuelo es un trayecto que une dos aeropuertos y que es realizado mediante el uso de un avión y es ofertado por una determinada compañía aérea.

DIAGRAMA DE CLASES DEL DOMINIO

En el siguiente diagrama podemos ver las clases detectadas como candidatas junto con sus asociaciones.

No hemos indicado ningún atributo ni ningún método y, es posible que en el apartado de análisis se eliminen ciertas clases y surjan otras nuevas.

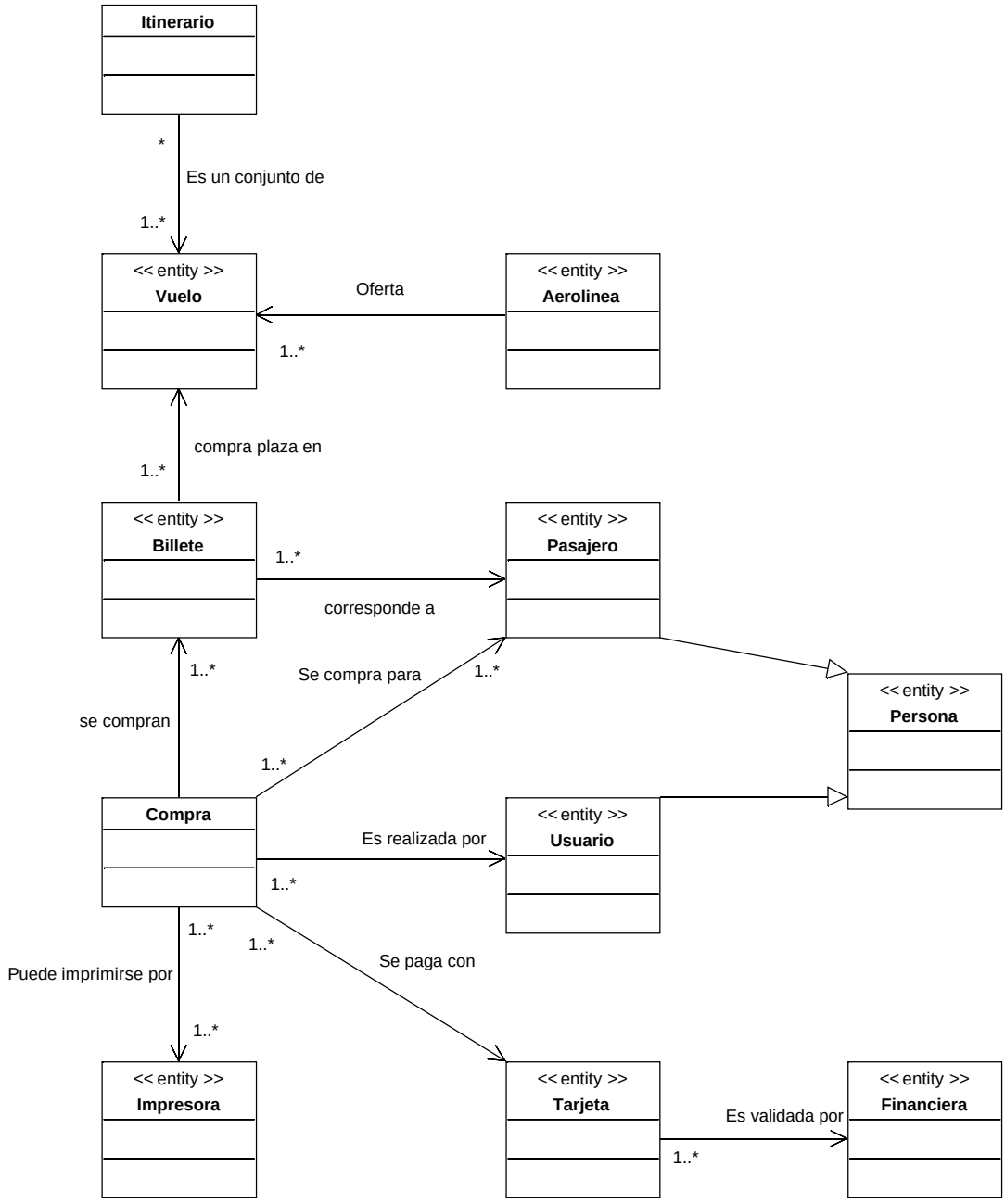

Ilustración 8 - Diagrama de clases introducido en poseidon

Nota: En el diagrama anterior, todo extremo de relación que no tenga asignado un cardinal, simboliza cardinalidad 1. Aunque han sido explícitamente introducidos en *Poseidon*, éste no los representa, si no se fuerza a ello.

ANÁLISIS

En este apartado pasamos a un nivel de mayor detalle en cuanto a la especificación del problema.

CASOS DE USO

Utilizaremos los mismos casos de uso indicados en los requisitos. No se ha detectado ningún dato adicional de interés para el problema.

CLASES

Partiremos de las clases candidatas detectadas en los Requisitos y, empezaremos a pensar en cómo interactúan entre ellas. Definiremos aquí los principales atributos y métodos previstos.

Pueden desaparecer o aparecer clases respecto a las definidas en los requisitos, así como relaciones entre las mismas.

CLASE AEROLÍNEA

La clase Aerolínea representa a cualquiera de las compañías aéreas que ofertan vuelos en esta aplicación. Son responsables de las operaciones de compra y cancelación de billetes y del control de disponibilidad de plazas en los vuelos.

Atributos

-**nombre**: String	Nombre de la compañía aérea
-**codigo**: String	Código identificador de la compañía aérea

Métodos

Aerolinea(nombre:String, codigo:String): Aerolínea	Instancia un objeto Aerolínea.
+**getNombre**(): String	Retorna el atributo *nombre*
+**getCodigo**(): String	Retorna el atributo *codigo*
+**getVuelos**(fecha:Fecha, ciudad:Ciudad): Vuelo[]	Retorna un conjunto de vuelos con origen o destino en la ciudad *ciudad* y con fecha *fecha*.
+**hayPlazas**(vuelo:Vuelo, numero:int, clase:String): bolean	Retorna *true* si hay suficientes plazas (*numero*) de clase *clase* en el vuelo *vuelo*
+**comprarBillete**(vuelo:Vuelo, pasajero:Pasajero, clase:String): Billete	Retorna el *Billete* que corresponde al comprobante de compra de una plaza de clase *clase* para el pasajero *Pasajero* en el vuelo *Vuelo*. En caso de no haber sido posible la compra, retornará un valor *null*.
+**cancelarBillete**(billete:Billete): bolean	Cancelará un billete previamente comprado. Retornará *true* en caso de que la cancelación haya sido posible y *false* en caso contrario.

CLASE BILLETE

La clase billete representa el contrato de compra de una plaza de una determinada clase para un determinado pasajero y en un determinado vuelo. Un billete puede comprarse y cancelarse. La compra implica una reserva inmediata de la plaza afectada y su cancelación, la liberación de la misma.

Atributos

-**pasajero**: Pasajero	Pasajero
-**vuelo**: Vuelo	Vuelo
-**numero**: long	Número de billete de vuelo
-**plaza**: String	Plaza reservada

Métodos

Billete(pasajero:Pasajero, vuelo:Vuelo, numero:int, plaza:String): Billete	Instancia un objeto Billete
+**getNumero**(): long	Retorna el atributo *número*
+**getPasajero**(): Pasajero	Retorna el atributo *pasajero*
+**getVuelo**(): Vuelo	Retorna el atributo *vuelo*
+**getPlaza**(): String	Retorna el atributo *plaza*.

CLASE COMPRA

Una compra representa al conjunto de datos que refleja la operación de reserva de billetes de avión que ha llevado a cabo un usuario de la aplicación en un momento dado, para un grupo de pasajeros y vuelos determinados.

Atributos

-**ultimaCompra**: long	Ultimo número de compra asignado
-**numero**: long	Número de operación de compra
-**fecha**: Date	Fecha de la compra
-**status**: int	Estado de la operación de compra (a medias, finalizada, finalizada con error...)
-**itinerario**: Itinerario	Itienerario elegido
-**pasajeros**: Pasajero[]	Lista de pasajeros a los que se les compra billete
-**billetes**: Billete[]	Lista de billetes comprados
-**envio**: int	Modo de envío
-**direccion**: Direccion	Dirección de envío de los billetes comprados
-**usuario**: Usuario	Usuario que ha realizado la reserva
-**tarjeta**: Tarjeta	Tarjeta de crédito empleada

Métodos

Compra(fecha,itinerario): Compra	Instancia un objeto Compra
+**getNumero**(): long	Retorna el atributo *numero*
+**getFecha**(): Date	Retorna el atributo *fecha*
+**getStatus**(): int	Retorna el atributo *status*
+**getItinerario**(): Itinerario	Retorna el atributo *itinerario*
+**getPasajeros**(): Pasajero[]	Retorna el atributo *pasajeros*
+**getBilletes**(): Billete[]	Retorna el atributo *billetes*
+**getEnvio**(): int	Retorna el atributo *envio*
+**getDireccion**(): Direccion	Retorna el atributo *direccion*
+**getUsuario**(): Usuario	Retorna el atributo *usuario*.
+**getTarjeta**(): Tarjeta	Retorna el atributo *tarjeta*

+**setStatus**(status:int)	Fija un nuevo valor para el atributo *status*
+**addPasajero**(pasajero:Pasajero)	Añade un pasajero a la lista de pasajeros
+**removePasajero**(pasajero:Pasajero)	Elimina un pasajero de la lista de pasajeros
+**comprarBilletes**(): bolean	Intenta la compra de billetes para los pasajeros de la lista *pasajeros* para el itinerario *itinerario*. Retorna *true* si ha sido posible comprar todos los billetes. Retorna *false* si ha fallado la compra de algún billete. Todas las operaciones quedan anuladas y la compra finaliza sin éxito.
+**setEnvio**(envio:int)	Fija el tipo de envío.
+**setDireccion**(direccion:Direccion)	Fija la dirección de envío a la que se enviarán los billetes comprados
+**setUsuario**(usuario:Usuario)	Fija los datos del usuario que realiza la compra
+**setTarjeta**(tarjeta:Tarjeta)	Asigna la tarjeta de crédito con la que se realizará la compra.
+**imprimir**()	Imprime los datos relativos a la operación de compra realizada.

CLASE DIRECCION

Esta clase representa los datos necesarios para identificar una dirección (dirección de envío, dirección de residencia, etc).

Atributos

-**calle**: String	Calle, carretera o vía
-**numero**: String	Número de la calle
-**piso**: String	Piso del número
-**puerta**: String	Puerta del piso
-**escalera**: String	Escalera del número
-**codigoPostal**: String	Código postal
-**poblacion**: String	Población, ciudad, pueblo, aldea....
-**provincia**: String	Provincia
-**pais**: String	Pais

Métodos

Direccion(calle:String, numero:String, escalera:String, codigoPostal:String, población:String, provincia:String, pais:String): Direccion	Instancia un objeto Direccion.
+**getCalle**(): String	Retorna el atributo *calle*
+**getNumero**(): Stirng	Retorna el atributo *numero*
+**getPiso**(): String	Retorna el atributo *piso*
+**getPuerta**(): String	Retorna el atributo *puerta*.
+**getEscalera**(): String	Retorna el atributo *escalera*
+**getCodigoPostal**(): String	Retorna el atributo *codigoPostal*.
+**getPoblacion**(): String	Retorna el atributo *población*
+**getProvincia**(): String	Retorna el atributo *provincia*.
+**getPais**(): String	Retorna el atributo *pais*

CLASE FINANCIERA

La clase Financiera representa a cualquiera de las compañías financieras capaces de comprobar y certificar la validez de los datos de una determinada tarjeta de crédito.

Atributos

-**nombre**: String	Nombre de la empresa financiera

Métodos

Financiera(nombre:String): Financiera	Instancia un objeto Financiera.
+**getNombre**(): String	Retorna el atributo *nombre*
+**verificarTarjeta**(tarjeta:Tarjeta): boolean	Retorna *true* si los datos de la tarjeta de crédito *tarjeta* son válidos. *False* en caso contrario.

CLASE ITINERARIO

La clase Itinerario representa una lista de vuelos que son necesarios para unir dos aeropuertos determinados (origen y destino).

Atributos

-**vuelos**: Vuelo[]	Lista ordenada de vuelos del itinerario
-**origen**: String	Ciudad de origen
-**destino**: String	Ciudad destino
-**importeTotal**: float	Importe total del itinerario
-**duracionTotal**: long	Duración total del itinerario
-**escalas**: int	Número de escalas del itinerario

Métodos

Itinierario(origen:String, destino:String): Itinerario	Instancia un objeto Itinerario
+**getOrigen**(): String	Retorna la ciudad de origen del itinerario
+**getDestino**(): String	Retorna la ciudad destino del itinerario
+**getVuelos**(): Vuelo[]	Retorna la lista de vuelos del itinerario
+**getImporteTotal**(): float	Retorna el atributo *importeTotal*
+**getDuracionTotal**(): long	Retorna el atributo *duracionTotal*
+**getEscalas**(): int	Retorna el atributo *escalas*
+**calcularItinerario**()	Intenta calcular un itinerario que una las ciudades de origen y destino, actualiza la lista de vuelos, el importe total, duración total y número de escalas del itinerario.

CLASE PASAJERO

Un pasajero corresponde a cada una de las personas que ocupan una de las plazas de uno de los vuelos ofertados en la aplicación. La clase Pasajero deriva de la clase Persona, de la que hereda atributos y métodos.

Atributos

-**tipo**: String	Tipo de pasajero (adulto, niño, bebé)

-**clase**:String	Clase de viajero (turista, bussiness, primera)

Métodos

Pasajero(nombre:String, apellido1:String, apellido2:String, dni:String, tipo:String, clase:String): Pasajero	Constructor
+**getTipo**(): String	Retorna el atributo *tipo*
+**getClase**():String	Retorna el atributo *clase*

CLASE PERSONA

Una persona es un individuo relacionado con la aplicación. Puede ser usuario o pasajero o ambos a la vez, dependiendo del rol que desempeñe en cada momento.

Atributos

-**nombre**: String	Nombre de la Persona
-**apellido1**: String	Apellido 1 de la persona
-**apellido2**: String	Apellido 2 de la persona
-**dni**: String	DNI o pasaporte de la persona

Métodos

Persona(nombre:String, apellido1:String, apellido2:String, dni:String): Pasajero	Instancia un objeto Persona
+**getNombre**(): String	Retorna el atributo *nombre*
+**getApellido1**(): String	Retorna el atributo *apellido1*
+**getApellido2**(): String	Retorna el atributo *apellido2*
+**getDni()**: String	Retorna el atributo *dni*

CLASE TARJETA

Esta clase representa a cualquiera de las tarjetas de crédito donde se anotarán los cargos y abonos resultantes de las compras y cancelaciones de billetes.

Atributos

-**tipo**: int	Tipo de tarjeta de crédito (visa, american express, ...)
-**numero**: String	Número de tarjeta de crédito
-**titular**: String	Nombre del titular
-**mes**: int	Mes de caducidad
-**ano**: int	Año de caducidad

Métodos

Tarjeta(tip:int, numero:String, titular:String, mes:int, ano:int): Tarjeta	Instancia un nuevo objeto Tarjeta
+**getTipo**(): int	Retorna el atributo *tipo*
+**getNumero**(): String	Retorna el atributo *numero*
+**getTitular**(): String	Retonra el atributo *titular*
+**getMes**(): int	Retorna el atributo *mes*
+**getAno**(): int	Retorna el atributo *ano*

CLASE USUARIO

Esta clase representa a cualquier usuario potencial de la aplicación. El usuario deriva de la clase Persona, heredando atributos y métodos.

Atributos

-**nacimiento**: Fecha	Fecha de nacimiento
-**email**: String	Email
-**telefono1**: String	Teléfono de contacto 1
-**telefono2**: String	Teléfono de contacto 2
-**empresa**: String	Nombre de la empresa para la que trabaja
-**tipoTarjeta**: int	Tipo tarjeta de fidelización
-**numero**: String	Numero de tarjeta de fidelización
-**direccion**: Direccion	Direccion habitual

Métodos

Usuario(nombre:String, apellido1:String, apellido2:String, dni:String, nacimiento:Fecha, email:String, telefono1:String, telefono2:String, empresa:String, tipo: int, numero: String, direccion:Direccion): Usuario	Constructor
+**getNacimiento**(): String	Retorna el atributo *nacimiento*
+**getEmail**(): String	Retorna el atributo *email*
+**getTelefono1**(): String	Retorna el atributo *telefono1*
+**getTelefono2**(): String	Retorna el atributo *Telefono2*
+**getEmpresa**(): String	Retorna el atributo *empresa*
+**getTipoTarjeta**(): int	Retorna el atributo *tipo*
+**getNumero**(): int	Retorna el atributo *numero*
+**getDireccion**(): Direccion	Retorna el atributo *direccion*

CLASE VUELO

Esta clase representa cualquiera de los vuelos que es ofertado por cualquiera de las compañías aéreas relacionadas con esta aplicación. Nota: esta clase debe afinarse una vez conocida la interfaz con las compañías aéreas, quienes serán las que proporcionarán los datos relevantes de los vuelos.

Atributos

-**numero**: String	Número de vuelo
-**origen**: String	Aeropuerto de origen
-**fechaSalida**: Fecha	Fecha de salida de origen
-**horaSalida**: Hora	Hora de salida de origen
-**destino**: String	Aeropuerto destino
-**fechaLlegada**: Fecha	Fecha de llegada a destino

-**horaLlegada**: Hora	Hora de llegada a destino
-**precioPrimeraAdulto**: float	Precio en dólares de un pasaje para primera clase, pasajero adulto
-**precioPrimeraNino**: float	Precio en dólares de un pasaje para primera clase, pasajero niño
-**precioPrimeraBebe**: float	Precio en dólares de un pasaje para primera clase, pasajero bebe
-**precioBussinessAdulto**: float	Precio en dólares de un pasaje para clase Bussiness, pasajero adulto
-**precioBussinessNino**: float	Precio en dólares de un pasaje para clase Bussiness, pasajero niño
-**precioBussinesBebe**: float	Precio en dólares de un pasaje para clase Bussiness, pasajero bebe
-**precioTuristaAdulto**: float	Precio en dólares de un pasaje para clase Turista, pasajero adulto
-**precioTuristaNino**: float	Precio en dólares de un pasaje para clase Turista, pasajero niño
-**precioTuristaBebe**: float	Precio en dólares de un pasaje para clase Turista, pasajero bebe

Métodos

Vuelo(numero:String, origen: String, fechaSalida:Fecha, horaSalida:Hora, destino:String, fechaLlegada:Fecha, horaLlegada:Hora, ppa:float, ppn:float, ppb:float, pba:float, pbn:float, pbb:float, pta:float, ptn:float, ptb:float): Vuelo	Instancia un nuevo objeto Vuelo.
+**getNumero**(): String	Retorna el atributo *numero*
+**getOrigen**(): String	Retorna el atributo *origen*
+**getFechaSalida**(): Fecha	Retorna atributo *fechaSalida*
+**getHoraSalida**(): Hora	Retorna atributo *horaSalida*
+**getDestino**(): String	Retorna atributo *destino*
+**getFechaLlegada**(): Fecha	Retorna atributo *fechaLlegada*
+**getHoraLlegada**(): Hora	Retorna atributo *horaLlegada*
+**getPrecio**(clase:int, tipoPasajero:int): float	Retorna el precio dependiendo de la clase de billete y tipo de pasajero.

DIAGRAMA DE CLASES DEL ANÁLISIS

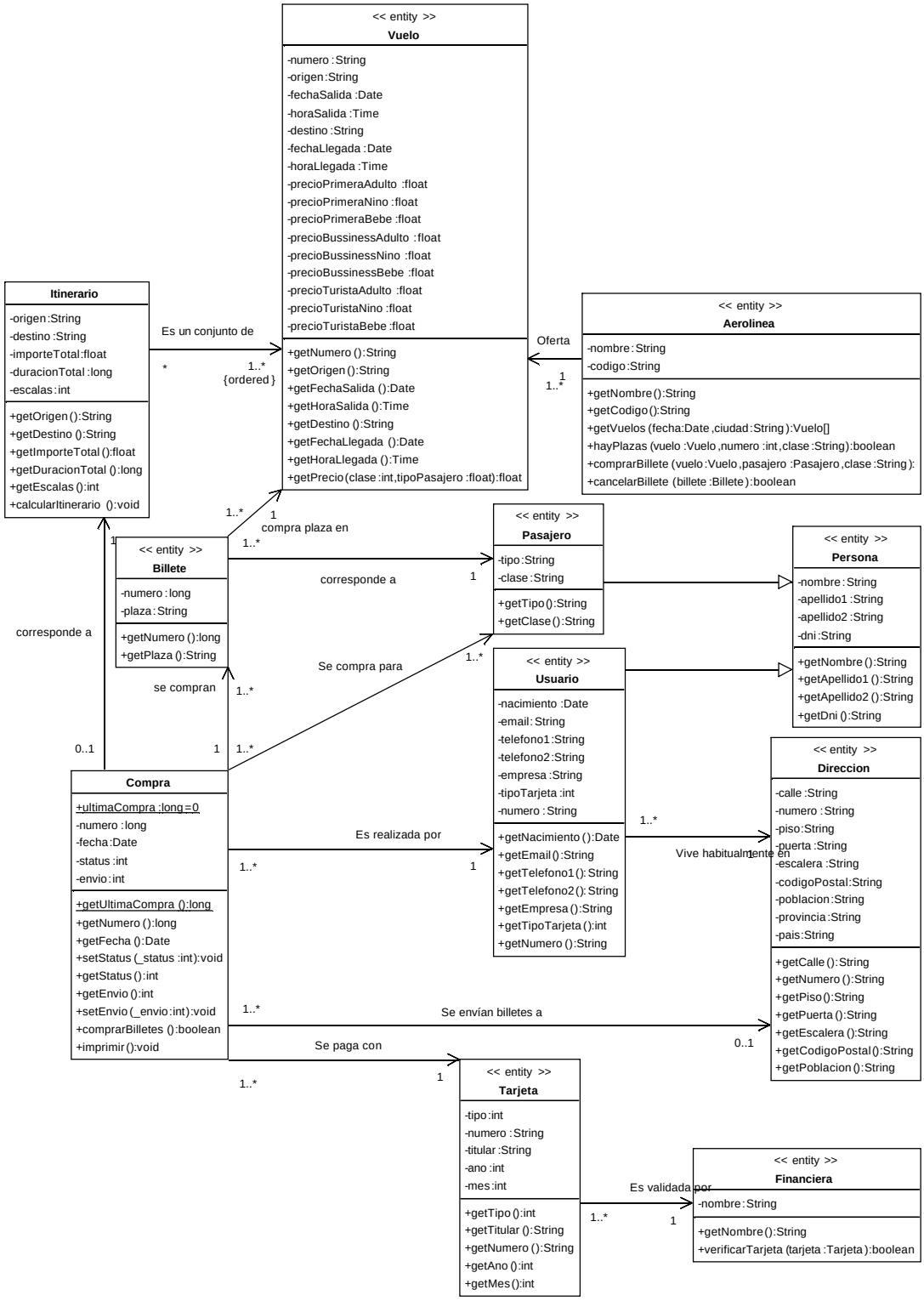

Ilustración 9 - diagrama de clases del análisis

Nota: Los constructores de las clases no se han introducido porque ensanchan demasiado cada clase, impidiendo un diagrama cuyo tamaño y visibilidad se ajusten correctamente a este documento.

ESTUDIO DEL GENERADOR

GENERALIDADES

Mientras estamos diseñando el diagrama de clases, *Poseidon for UML* se encarga de ir generando el código fuente en tiempo real. La siguiente captura ilustra este hecho:

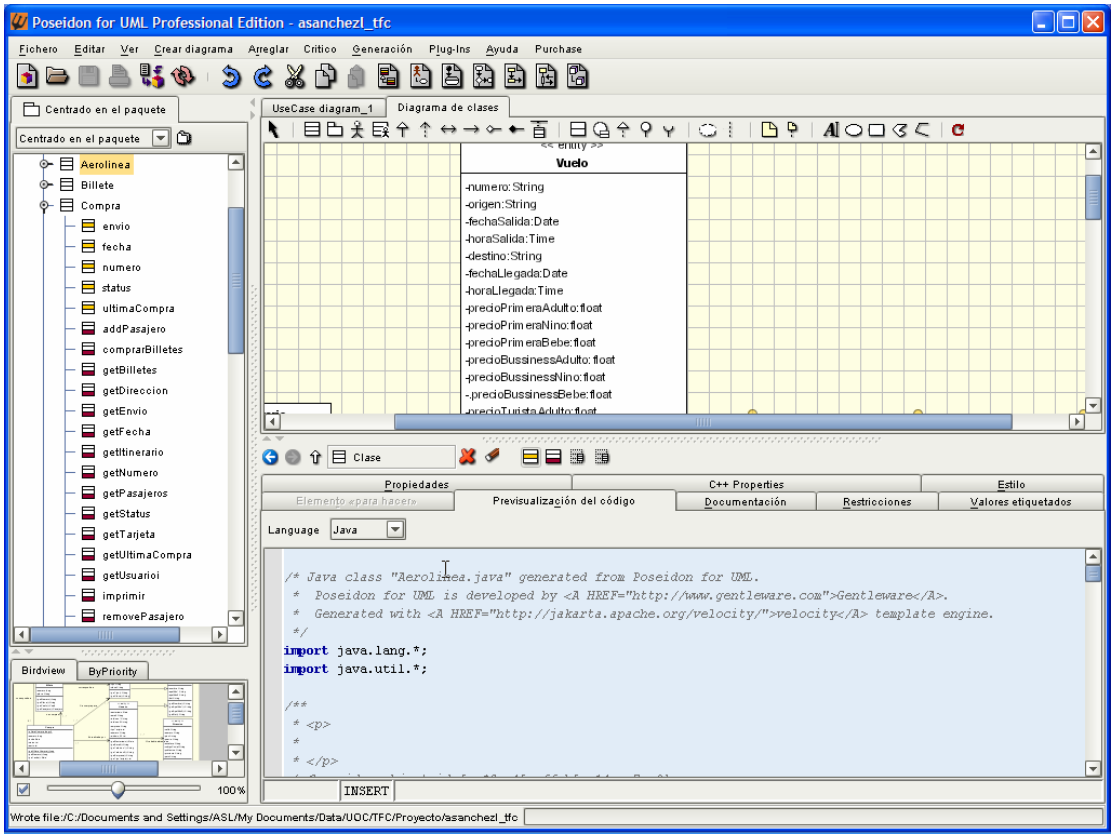

Ilustración 10 - Área de trabajo de Poseidon for UML

Si seleccionamos una clase y accedemos a la pestaña de **previsualización de código**, veremos inmediatamente el código que la aplicación está generando en tiempo real, conforme a las especificaciones de atributos y métodos indicados en dicha clase. También podemos seleccionar el lenguaje de programación en el que queremos ver el código generado.

El código generado queda un poco 'sucio' debido a la identificación de los distintos objetos que lleva a cabo *Poseidon*. Cada atributo y método es identificado internamente por la herramienta, dejando dicha información como comentarios en el código generado, como puede verse en la siguiente captura de pantalla:

```
/**
 * <p>
 *
 * </p>
 * @poseidon-object-id [sm$8ae45a:ffab5ee14e:-7e33]
 */
public class Vuelo {
```

Ilustración 11 - Identificación interna de los objetos Poseidon

El comportamiento de *Poseidon* puede modificarse ligeramente haciendo uso de la opción de menú *Editar→Ajustes*, que permite configurar la herramienta, Para el modelado nos interesa

especialmente la solapa *Modeling,* donde podremos fijar que *Poseidon* genere automáticamente los métodos *accesors* (*getter* y *setter*) y, que elimine automáticamente los métodos accesors si eliminar un atributo, lo que es bastante útil. Una vez generados automáticamente los métodos accesors, podemos borrar cualquiera de ellos sin que eso afecte ni al atributo ni a ningún otro método.

Como podemos ver, existen pocos parámetros para incidir en el modelado de los diagramas de clases. Personalmente, noto a faltar que la herramienta no proponga de forma automática un constructor que contenta los atributos definidos para la clase, así como el constructor por defecto (sin atributos).

GENERACIÓN DE CÓDIGO FUENTE JAVA

Estudiaremos aquí el comportamiento particular del generador *Java*.

OPCIONES

Esta captura muestra las opciones del generador *Java*:

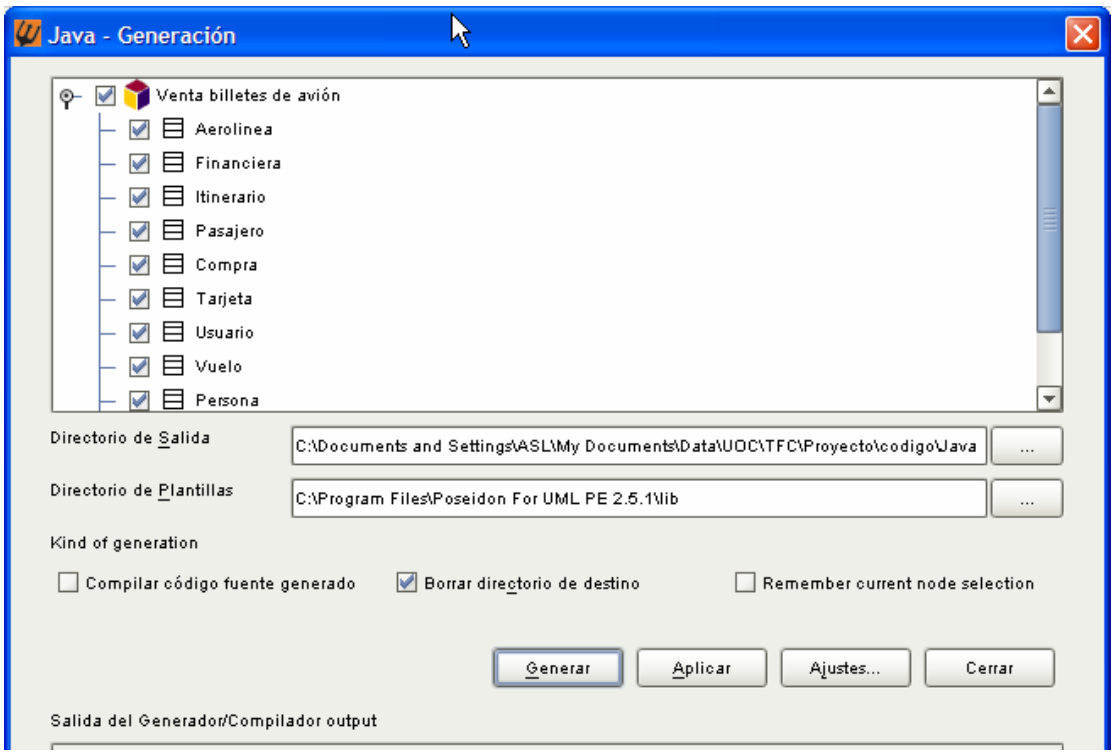

Ilustración 12 - Opciones del generador Java

El primer punto de interés es comprobar que sólo se genera código a partir del modelo especificado en el diagrama de clases correspondiente. Ningún otro diagrama influye en la generación de código.

El generador permite elegir las clases cuyo código quiere generarse y ciertas opciones como compilar el código fuente generado o borrar el directorio de destino previamente. Adicionalmente, permite los siguientes ajustes:

- Generar automáticamente los métodos accessors para las asociaciones de esta clase con otras clases, que encuentro particularmente útil.
- Clases con las que definirá aquellos atributos que implementarán las asociaciones multitudinarias con otras clases. Permite especificar una clase distinta para las asociaciones multitudinarias ordenadas y desordenadas (por defecto: *ArrayList, TreeSet*, respectivamente).

Otra posibilidad interesante del generador Java la constituye la posibilidad de generar código de ida y vuelta, opción que se activa antes de la generación. La generación de ida y vuelta permite que la herramienta incorpore en nuestro modelo aquellos cambios introducidos manualmente en el código generado automáticamente.

REVISIÓN GENERAL DEL CÓDIGO GENERADO

En una revisión rápida del código generado, se observa lo siguiente:

- No se genera ningún constructor. Ni tan siquiera el constructor por defecto.
- Si hemos elegido que genere automáticamente el código para los accessors de las asociaciones y, manualmente habíamos definido algún método para ello, el generador no detecta la duplicidad, dando pie a errores en compilación. Además, al generarse dichos métodos sólo en tiempo de generación, el diagrama de clases no representa dichos métodos, lo que es confuso.
- Los atributos definidos con tipo *Time* no son reconocidos. El problema se debe a que el generador cree que esta clase está en el paquete *java.util*, estando realmente en el paquete *java.sql.* Si se incluyen ambos, se genera un conflicto con la clase *Date* que existe en ambos paquetes. Por lo que conviene hacer la importación de la clase en concreto (*java.util.Date* o *java.sql.*Date y *java.sql.Time*).

CLASES

General

Propiedades

Para alterar la definición de la clase, tenemos la solapa propiedades, donde podemos especificar los modificadores de visibilidad y otras características, como podemos ver en la captura de pantalla:

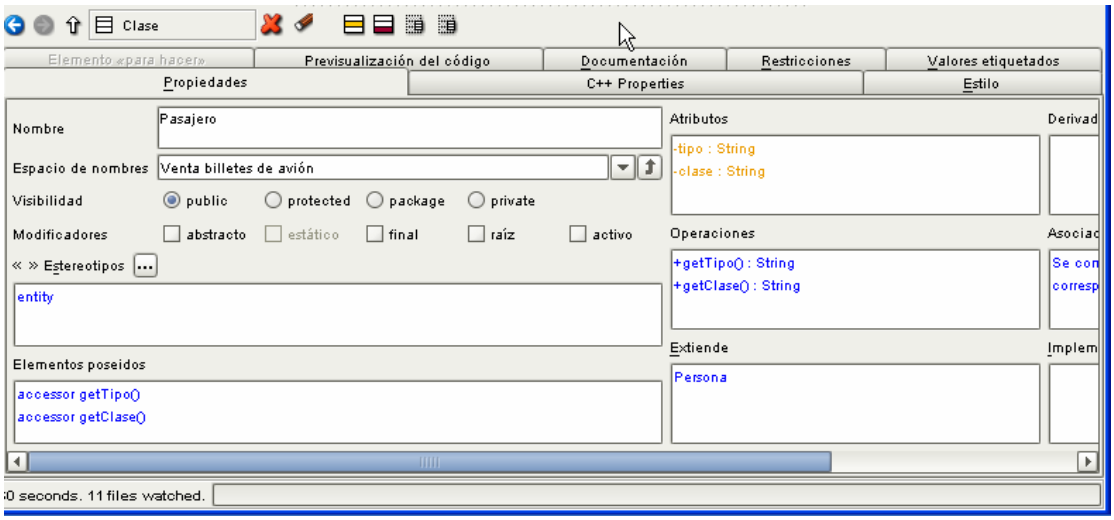

Ilustración 13 - Propiedades de las clases

Nos llama la atención los modificadores *raíz* y *activo*: así como el resto de modificadores tiene efecto inmediato en la definición Java de la clase, ninguno de estos dos modificadores parece afectar al código resultante.

Los estereotipos tienen cierto interés, puesto que definen mejor la naturaleza de cada clase pero, en principio, ningún estereotipo tiene efecto sobre el código Java generado. Por ejemplo:

- El estereotipo *entity* (entidad) no añade código para asegurar la persistencia de la clase.

- El estereotipo *boundary* no añade código de interfaz de usuario, como forzar que la clase derive de una clase como *JFrame*, por ejemplo.

En definitiva, los estereotipos añaden claridad al diagrama pero no inciden en la generación del código.

Restricciones

Las restricciones (*constraints*) parecen seguir la misma pauta que los estereotipos. Se pueden indicar las restricciones que se crean oportunas, las cuales se incorporan automáticamente al diagrama, pero no parecen afectar al código generado (precondiciones, postcondiciones...).

Atributos

En las propiedades de los atributos podemos ver todas las opciones para alterar las características del atributo.

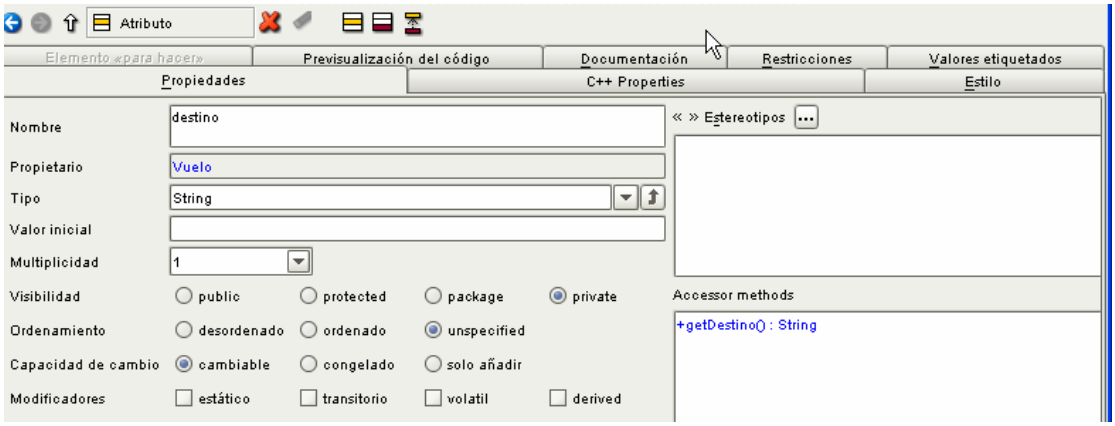

Ilustración 14 - Propiedades de los atributos

Cabe comentar que no hay ningún estereotipo definido por defecto.

Multiplicidad

Si cambiamos la multiplicidad al atributo, cambia la definición del dato en el código Java generado. Por ejemplo, el atributo *destino* de *Vuelo* está definido como un *String* con multiplicidad 1.

- Si cambiamos la multiplicidad a **1..*** o *****, *destino* se define como un atributo de clase *Collection* (vector de elementos).
- Si cambiamos la multiplicidad a **0..1**, la declaración de *destino* no varía (elemento simple).

Orden

Si no especificamos nada, el atributo se considera desordenado. Si indicamos que el atributo está ordenado, sólo el diagrama se ve afectado, no teniendo impacto en el código generado.

Capacidad de cambio

Si fijamos el atributo como *congelado*, el generador añade el modificador *final* al atributo, impidiendo que su valor inicial sea modificado con posterioridad.

Si indicamos *solo añadir* no ocurre nada con el código. Por ejemplo, podríamos haber definido el atributo con multiplicidad ***** (*Collection*); el generador añade los métodos accesors, como puede verse en el siguiente ejemplo (nota: los comentarios han sido eliminados, para mayor claridad):

```
private Collection destino; // of type String

public Collection getDestinos() {
    return destino;
} // end getDestinos

public void setDestinos(Collection _destino) {
    destino = _destino;
} // end setDestinos

public void addDestino(String _destino) {
    if (! destino.contains(_destino)) destino.add(_destino);
} // end addDestino

public void removeDestino(String _destino) {
    destino.remove(_destino);
} // end removeDestino
```

A mi entender, el modificador *solo añadir* debería impedir que se generara el método *remove...*, por ejemplo, pero no hay ninguna alteración visible en el código.

Modificadores de persistencia

El único modificador que no afecta al código es *derived*. El resto funcionan como podría preverse.

Métodos

En las propiedades de los métodos, podemos ver también dónde podemos incidir para variar las características de los mismos.

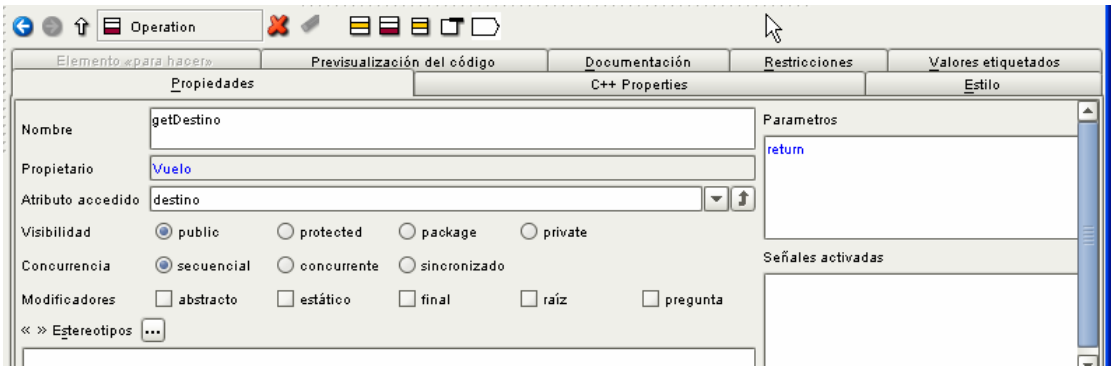

Ilustración 15 - Propiedades de los métodos

Concurrencia

Por defecto, los métodos se definen como secuenciales. Modificar la concurrencia no afecta ni al diagrama ni al código generado.

Modificadores

Los modificadores *abstracto, estático* y *final* se comportan como es previsible. Los modificadores *raíz* y *pregunta* no tienen ningún efecto ni sobre el código generado ni sobre el diagrama de clases.

Estereotipos

El estereotipo *create* se especifica para indicar que el método es un constructor de la clase. Esto fuerza que el generador suprima el retorno en la definición del método, como podemos ver en este ejemplo:

```
public  getPrecio(int clase, float tipoPasajero) {        /** lock-end */
    return precioTuristaBebe;
} // end getPrecio        /** lock-begin */
```

El método estaba definido previamente como *public float getPrecio(...)*. Este ejemplo muestra uno de los problemas del generador, que permite asignar el estereotipo *create* a un método con nombre distinto al de la clase.

El estereotipo *destroy* afecta al diagrama pero no al código (Java no tiene destructores).

El estereotipo *initializer* convierte el método en un bloque de código no asociado a ningún método, como podemos ver en este ejemplo:

```
{       /** lock-end */
    return precioTuristaBebe;
} // end initializer        /** lock-begin */
```

ASOCIACIONES

General

En las propiedades de la asociación están los elementos que nos permiten alterar las características globales de las mismas.

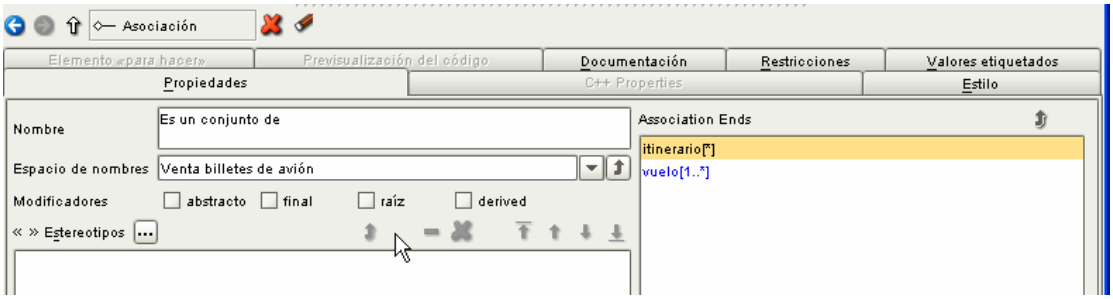

Ilustración 16 - Propiedades de las asociaciones

Sin embargo, para poder examinar cómo refleja el modelo el código generado, debemos examinar cada uno de los extremos de la asociación, donde se ejerce el control real de todos los atributos de la asociación.

Multiplicidad

Dependiendo de la multiplicidad de cada extremo, la asociación se implementa de una u otra forma. Si un extremo tiene multiplicidad **1..*** o *****, se implementa en la clase que está al otro extremo como un atributo de clase *Collection*, como podemos ver en la siguiente captura, correspondiente a la clase *Itinerario*:

```
private Collection vuelo = new ArrayList(); // of type Vuelo
```

Ilustración 17 - código generado según multiplicidad de la asociación

En caso de multiplicidad **0..1** o **1**, se implementa con tipo de dato igual a la clase que está en dicho extremo de la asociación.

Direccionabilidad

La direccionabilidad influye en la accesibilidad entre clases a través de una asociación. La presencia de una flecha en uno de los dos extremos convierte la accesibilidad en unidireccional. Por ejemplo:

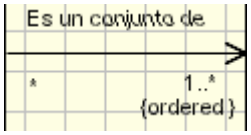

Ilustración 18 - asociación unidireccional

Aquí vemos la asociación **es un conjunto de**, establecida entre las clases **Itinerario** y **Vuelo**. La clase *Itinerario* utiliza la clase *Vuelo*, pero no al revés. Al ser unidireccional, el extremo en el que está la punta se incluye como atributo en la clase origen de la asociación. Como hemos visto un poco más arriba, al tener multiplicidad mayor que 1, *Vuelo* se incorpora como una colección en la clase *Itinerario*.

En el caso de que la asociación sea bidireccional, cada clase se incluye como atributo en la otra. El tipo de dato depende de la multiplicidad: la propia clase para multiplicidades iguales o menores a 1 y *Collection* para multiplicidades mayores.

Ejemplo, si cambiamos a bidireccional la asociación anterior:

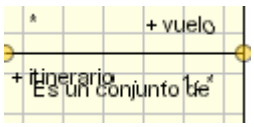

Ilustración 19 - asociación bidieccional

Vemos que en *Itinerario*, se añade como atributo una colección de objetos tipo *Vuelo*:

```
public Collection vuelo_1 = new TreeSet(); // of type Vuelo
```

Ilustración 20 - código java en Itinerario de la asociación bidireccional

Y que en *Vuelo* se añade como atributo una colección de objetos de tipo *Itinerario*:

```
public Collection itinerario_1 = new TreeSet(); // of type Itinerario
```

Ilustración 21 - código java en Vuelo de la asociación bidireccional

El tipo de dato con el que se inicializa la variable depende de si el extremo está o no ordenado, usándose *TreeSet* si no está ordenado y *ArrayList* en caso contrario (o las clases elegidas en las opciones, antes de generar el código, como hemos visto mas arriba).

Orden

Para especificar el orden en una asociación, ésta debe seleccionarse en el diagrama y después, elegir el extremo que debe estar presente de forma ordenada, como puede verse en la siguiente relación (*es un conjunto de*) entre *Itinerario* y *Vuelo* (extremo *Vuelo*).

Ilustración 22 - extremos ordenados en las asociaciones

Esta es la única asociación de nuestro ejemplo que requiere orden. El orden en este caso es necesario para construir el grafo ordenado de vuelos (itinerario) que unen el aeropuerto origen con el aeropuerto destino indicados por el usuario.

Extremos

Accediendo al extremo, se ejerce un mayor control a través de la pestaña de propiedades:

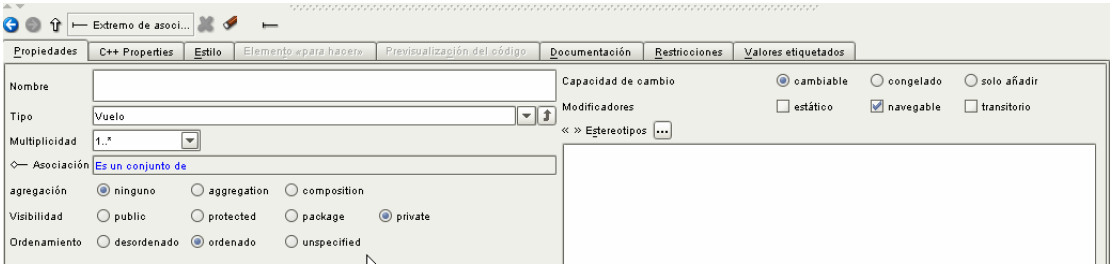

Ilustración 23 - Propiedades de los extremos de una asociación

Las opciones son similares a las vistas en atributos, con excepción del apartado *Agregación*, que describimos a continuación. Los estereotipos no tienen efecto sobre el código.

Agregación

Puede indicarse que la asociación específicamente es una agregación o una composición. Esto tiene un efecto inmediato sobre el diagrama, pero no afecta al código generado.

En principio, el extremo que debe escogerse es el del elemento superior de la agregación o composición y, éste puede llegar a ser la clase accedida, en vez de la que accede, como puede verse en la siguiente imagen:

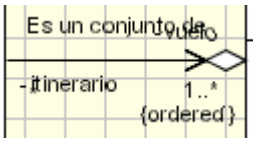

Ilustración 24 - Agregación curiosa

Clases Asociativas

No he incluido ninguna en mi modelo, sin embargo, tiene interés comprobar qué tal reacciona el programa con este tipo de objetos.

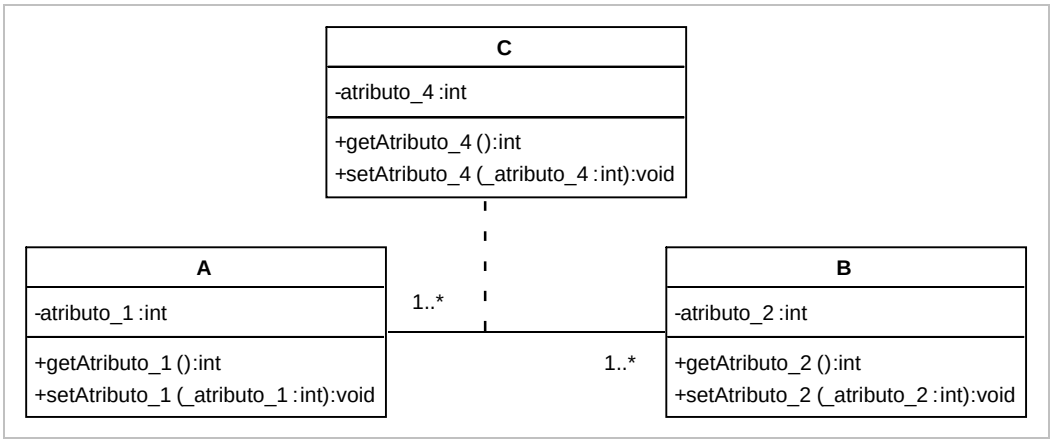

Ilustración 25 - Ejemplo clases asociativas

En el esquema podemos ver que la clase asociativa C es una instancia de la asociación establecida entre A y B. Por defecto, el programa deja los extremos de la asociación con multiplicidad unaria pero, en este ejemplo hemos forzado multiplicidad multitudinaria.

Podemos ver en el código generado en la clase C, que se incluyen dos atributos, de clase A y B, respectivamente, y con multiplicidad 1:

```
public A a; // of type A
public B b; // of type B
```

y en las clases A y B, se genera un atributo de clase C y sus métodos accessors:

```
public Collection c = new TreeSet(); // of type C
public Collection getCs() {
    return c;
}
public void addC(C c) {
    if (! this.c.contains(c)) {
        this.c.add(c);
        c.setA(this);
    }
}
public void removeC(C c) {
    boolean removed = this.c.remove(c);
    if (removed) c.setA(null);
}
```

Modificar la multiplicidad en los extremos de la asociación no afecta al código generado en ninguna de las tres clases.

Asociaciones múltiples

Aunque no estaba presente en mi modelo, me ha parecido interesante probar una asociación múltiple, concretamente el ejemplo siguiente

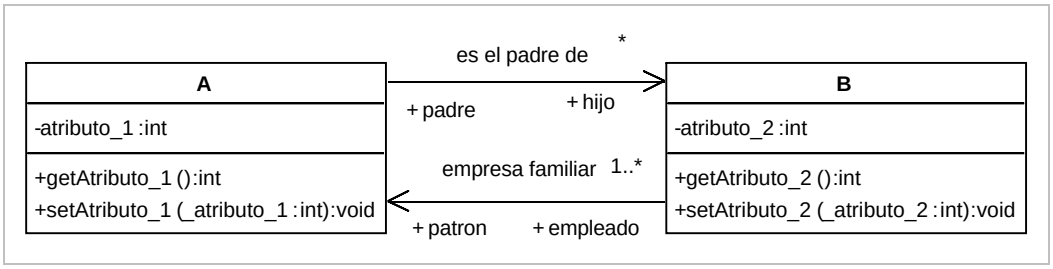

Ilustración 26 - Ejemplo asociación múltiple multidireccional

en el que se establecen dos asociaciones entre las clases A y B, adoptando un rol distinto cada clase en cada asociación.

En la clase A se implementa la relación “es el padre de” añadiendo el atributo *hijo* y sus métodos *accesors*, tal y como era previsible:

```
public Collection hijo = new TreeSet(); // of type B
public Collection getHijos() {
    return hijo;
}
public void addHijo(B b) {
    if (! this.hijo.contains(b)) this.hijo.add(b);
}
public void removeHijo(B b) {
    this.hijo.remove(b);
}
```

Por otro lado, en la clase B, se implementa la relación “empresa familiar” a través del atributo “patron”, tal y como era previsible:

```
public A patron;
public A getPatron() {
    return patron;
}
public void setPatron(A a) {
    this.patron = a;
}
```

He probado también una doble asociación con la misma dirección (de A a B en ambos casos) y el generador la interpreta correctamente.

Herencia

La herencia está presente en el modelo en dos casos. La clase *Pasajero* y las clase *Usuario* derivan, ambas, de la superclase *Persona*, de la que heredan atributos y métodos, especializándola.

El generador *Java* es capaz de representar la herencia simple correctamente, definiendo la clase con el modificador *extends*, como puede verse en el siguiente ejemplo:

```
public class Pasajero extends Persona {
```

donde la clase *Pasajero* es una subclase de *Persona*.

Herencia Múltiple

Tampoco se da la herencia múltiple en mi modelo, sin embargo tiene cierto interés conocer cómo resuelve este problema *Poseidon*.

En el siguiente esquema, podemos ver que la clase C deriva simultáneamente de la clase A y la clase B.

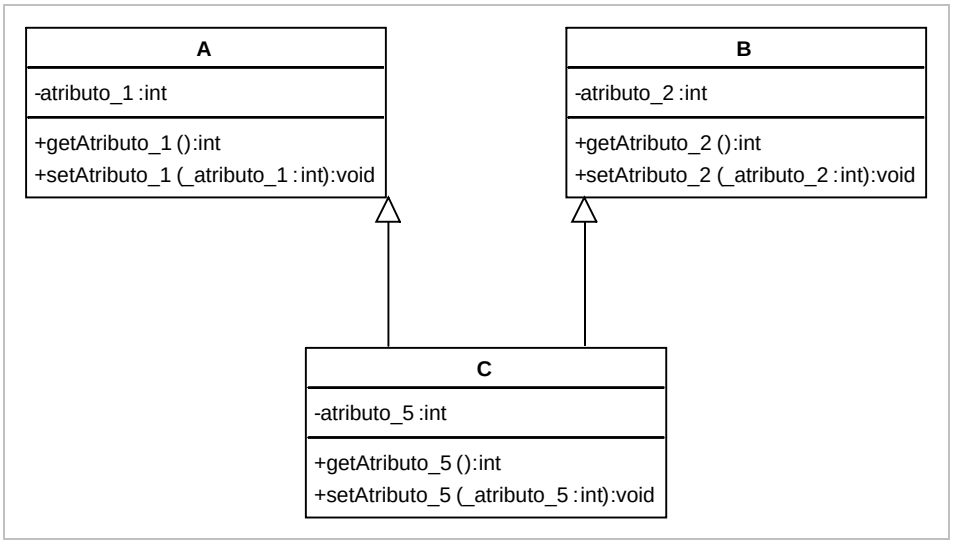

Ilustración 27 - Ejemplo herencia múltiple

El código de la clase C, está mal generado, como puede verse a continuación:

```
public class C extends A, B {
```

puesto que la herencia múltiple no es posible en Java. La clase C daría un claro mensaje de error en tiempo de compilación. Debería resolverse la herencia por los métodos habituales empleados en Java, pero el programa no parece capaz de idear un sistema concreto.

GENERACIÓN DE CÓDIGO FUENTE SQL

OPCIONES

El menú del generador SQL es prácticamente igual que el del generador Java, con las excepciones de que no se puede realizar ningún ajuste adicional ni compilar el código generado. Además, para evitar problemas al generar código SQL, antes debemos activar el perfil SQL. En el menú **Plug-ins→Profiles Panel** seleccionaremos el perfil de SQL y lo activaremos, como puede verse en la siguiente imagen:

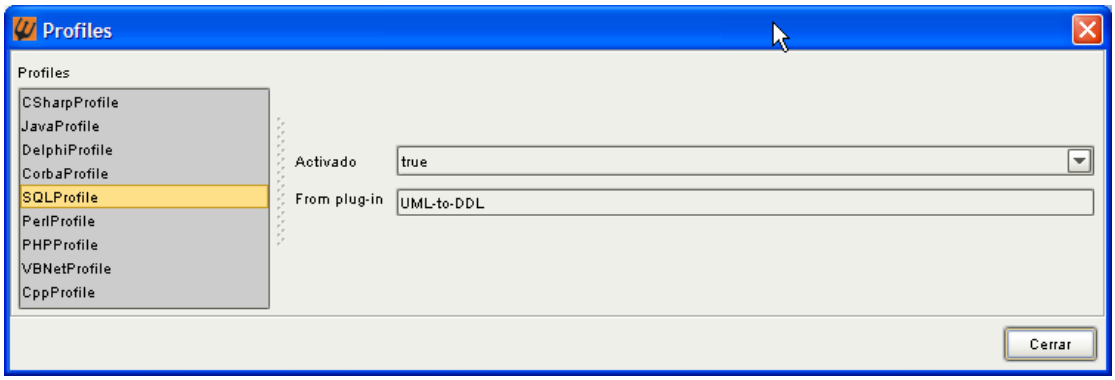

Ilustración 28 - Perfil para el generador SQL

En caso contrario, no podremos asignar los estereotipos de *Primary Key, Not Null*, etc a los atributos, lo que acarreará un montón de problemas al intentar generar el código SQL.

De todas formas, la generación da problemas debido a que la herramienta no es capaz de mapear los tipos de datos Java en tipos de dato SQL, por lo que el resultado no es directamente utilizable en ningún caso.

REVISIÓN GENERAL DEL CÓDIGO GENERADO

El código generado se basa en un fichero con extensión *sql* por cada una de las clases seleccionadas. En el siguiente ejemplo (los comentarios han sido eliminados):

```
CREATE TABLE Itinerario
(
origen String    ,
destino String    ,
importeTotal float    NOT NULL ,
duracionTotal long    NOT NULL ,
escalas int    NOT NULL ,
numero String

, PRIMARY KEY (origen, destino)
, FOREIGN KEY (numero) REFERENCES Vuelo
);
```

podemos ver el resultado de la generación SQL para la clase *Itinerario*. Observemos que ha incluido los atributos (con las restricciones SQL indicadas mediante el estereotipo), así como la clave primaria definida y, como clave foránea ha incorporado la clave primaria de aquellas otras clases con las que mantiene una relación.

CLASES

Si una clase no tiene definida clave primaria, el generador no la tendrá en cuenta al generar las claves foráneas de otras clases con las que mantiene cierta relación. Cualquier estereotipo asignado a la clase impedirá que se genere el código SQL para la tabla

correspondiente a dicha clase, incluso utilizando el estereotipo <<*entity*>>, que define a la clase inequívocamente como una entidad (y por tanto, que requiere persistencia).

Atributos

Los atributos definidos en la clase son incorporados en la definición de las columnas de la tabla SQL. El programa no es capaz de mapear correctamente los tipos de dato Java con los tipos de datos SQL (String, long, ...). Cuando no sabe cómo definir un atributo, lo define con tipo de dato *int*.

Los atributos con estereotipos *PRIMARY KEY* son incorporados a la cláusula PRIMARY KEY de la tabla, correctamente. Los atributos con estereotipo *NOT NULL* son definidos con dicha cláusula SQL.

Atributos de clase

A mi entender, la traducción de los atributos de clase está mal resuelta. Por ejemplo, la clase *Compra* incorpora un atributo de clase llamado *ultimaCompra* que se incrementa con cada objeto *Compra* instanciado y que, no deja de ser un contador. El generador produce el siguiente código (dentro de la tabla *COMPRA*):

```
ultimaCompra long    DEFAULT 0 NOT NULL ,
```

Al ser un atributo permanente, que debería existir independientemente de si la tabla COMPRA tiene o no entradas, este atributo debería ser incorporado en otra tabla (de parámetros, de sistema o cualquier otro artificio similar).

Métodos

No se genera ningún código relacionado con los métodos de la clase.

ASOCIACIONES

En general, las asociaciones que no son del tipo M:N se incorporan bien en la definición de las tablas, utilizando como clave foránea aquellas columnas que constituyen clave primaria en la otra tabla pero, las relaciones M:N no se están reflejando en el modelo SQL.

Como ejemplo, podemos comprobar la relación M:N (modelo E/R de *[Yourdon]*) de la asociación *se compra para*, establecida entre las clases *Compra* y *Pasajero*. Según la teoría relacional de *[Codd]*, la relación M:N debería traducirse en una tabla que contuviera como clave primaria las claves primarias de todas las tablas que entran en la relación. Así se han implementado las relaciones que mantiene la clase *Compra*:

```
CREATE TABLE Compra
(
numero long    ,
ultimaCompra long   DEFAULT 0  NOT NULL ,
fecha Date    NOT NULL ,
status int    NOT NULL ,
envio int    NOT NULL ,
tipo int    ,
numero String    ,
origen String    ,
destino String    ,
numero long
, FOREIGN KEY (tipo, numero) REFERENCES Tarjeta
, FOREIGN KEY (origen, destino) REFERENCES Itinerario
, FOREIGN KEY (numero) REFERENCES Billete
);
```

En ninguna de las pruebas se ha conseguido que una relación M:N se implemente como una nueva tabla.

En el código generado, el primer problema observado es que la clave primaria no ha sido definida (el atributo *numero* tiene el estereotipo *PRIMARY KEY* definido). En cuanto a las asociaciones de *Compra* con otras clases, la siguiente tabla describe cómo se han implementado.

Asociación OO **Compra → otra clase**	**Otra clase**	**Relación E/R** **Compra → otro clase**	**Implementada?**
0..1 → 1	**Itinerario**	0,1 : 1	Correcto. Incluida FK a tabla *Itinerario* con columnas *tipo* y *numero* en tabla *Compra*.
1 → 1..*	**Billete**	1 : N	Incorrecto. Incluida FK a tabla *Billete* con columna *numero* en tabla *Compra*. No se ha definido la cláusula *NOT NULL* en *numero*.
1..* → 1..*	**Pasajero**	M : N	**NO**
1..* → 1	**Usuario**	M : 1	**NO**
1..* → 1	**Tarjeta**	M : 1	Incorrecto. Incluida FK a tabla *Tarjeta* con columnas *tipo* y *numero* en tabla *Compra*. No se ha definido la cláusula *NOT NULL* ni en *tipo* ni en *numero*.

Como podemos observar, existen varios casos donde no se ha definido adecuadamente la relación:

- En el caso de la asociación con *Pasajero*, el programa parece tener problemas a la hora de representar como tabla las asociaciones tipo M : N.
- En el caso de la asociación con *Usuario* el problema debe provenir de que no existe clave primaria definida para usuario y, no existe dicha clave primaria porque no se ha incorporado la clave primaria de persona, clase de la que deriva. La herencia no se resuelve satisfactoriamente, como detallamos más abajo, en el apartado *Herencia*.
- La asociación entre *Compra* y *Billete* se ha implementado al revés. Debería haberse definido la clave foránea en *Billete*, apuntando a la clave primaria de *Compra*.
- Adicionalmente, el generador no distingue las relaciones 0,1:N de 1:N. En las relaciones 1:N debería incluir la cláusula *NOT NULL* en el atributo que constituye la clave foránea a la otra tabla.
- La implementación de las claves foráneas está directamente relacionada con la accesibilidad de la asociación. La PK se implementa en la clase accesora, apuntando a la clase accedida. Si convertimos la asociación en bidireccional, crea erróneamente una FK en cada clase, sin tener en cuenta la multiplicidad de los extremos de la asociación.

Asociaciones múltiples

En asociaciones múltiples entre clases (como el ejemplo visto en el apartado del mismo nombre, en el estudio del generador Java), el generador añade como clave foránea la clave primaria de la clase en el otro extremo de la asociación, siempre y cuando no exista más de una relación en la misma dirección entre dos clases, como podemos ver a continuación:

Si la asociación múltiple es bidireccional, en la clase *A* se define el atributo *atributo_2* como clave foránea a la clase *B* y, en la clase *B* sucede lo mismo, también se define el atributo *atributo_2* como clave foránea a la clase *A*.

```
CREATE TABLE A
(
atributo_1 int    PRIMARY KEY NOT NULL ,
atributo_2 int

, FOREIGN KEY (atributo_2) REFERENCES B
);
```

En el caso de asociación múltiple unidireccional entre dos clases, como la que puede verse en la figura:

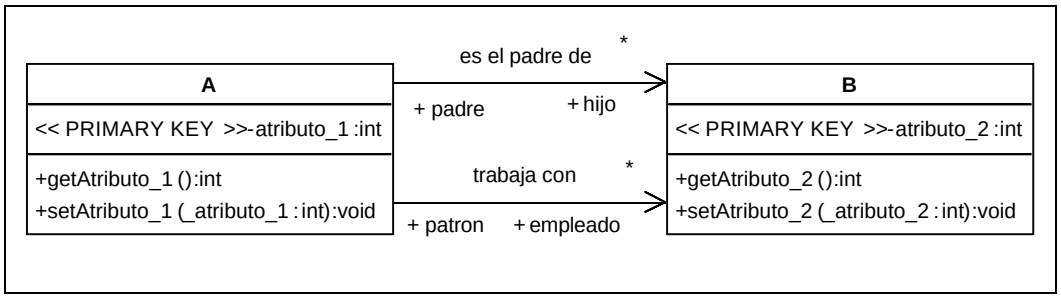

Ilustración 29 - Ejemplo asociación múltiple unidireccional

el generador se hace un autentico lío, definiendo dos veces el atributo *atributo_2* en la clase *A* como columna y como clave foránea a la clase *B*. Para mayor claridad, presento el código generado:

```
CREATE TABLE A
(
atributo_1 int    PRIMARY KEY NOT NULL ,
atributo_2 int    ,
atributo_2 int

, FOREIGN KEY (atributo_2) REFERENCES B
);
```

Como podemos ver, ni tan siquiera genera dos atributos distintos, con el nombre del rol que juega la clase en la relación (*hijo* y *empleado*, por ejemplo), sino que se limita a inventar dos veces un atributo con el mismo nombre (*atributo_2*).

Herencia

La herencia no ha sido resuelta satisfactoriamente. Las clases *Usuario* y *Pasajero* derivan de *Persona*. En el código SQL generado para las clases *Usuario* y *Pasajero* debería, por lo menos, haberse incluido la clave primaria a *Persona*. Podemos ver en el código generado para la tabla *Pasajero* que no se ha incluido como clave primaria la clave de la superclase *Persona*:

```
CREATE TABLE Pasajero
(
tipo String    NOT NULL ,
clase String
);
```

Herencia múltiple

En el caso de herencia múltiple, el programa se limita a definir el código de creación de la clase derivada (estrictamente sus propios atributos), no incluyendo tampoco en ningún momento las claves primarias de las superclases de las que deriva.

CONCLUSIONES

COMPARACIÓN DE LOS GENERADORES

La potencia y precisión generatriz de *Poseidon* es bastante mayor generando código Java que generando código SQL. Adicionalmente, el generador Java permite la opción de tecnología de *ida y vuelta*, de forma que, si actualizamos el código fuente, los cambios se reflejan en el diagrama de clases que se ha utilizado como modelo para el generador.

En cuanto a la fidelidad al modelo UML original, el generador Java es capaz de incorporar prácticamente todos los elementos del diagrama (de clases), notando a faltar la generación automática de constructores. Sería también deseable que facilitara la generación del código correspondiente al mantenimiento básico de los datos persistentes de cada clase, así como GUIs en Java y/o HTML, por ejemplo. Por otro lado, la opción de generar los métodos accessors de forma automática tanto para atributos como para asociaciones es realmente práctica, aunque podría haberse incluido la generación automática del código para el método *equals()*, derivado de la clase *Object*, basado en atributos con el estereotipo <<*PRIMARY KEY*>>, por ejemplo.

El generador SQL es bastante más pobre y presenta serios problemas para realizar las conversiones entre los tipos de datos definidos en las clases y los tipos de datos esperados por SQL. También es incapaz de generar tablas que representen relaciones tipo M:N y, por supuesto, no genera ningún código relacionado con los métodos de clase.

Otro aspecto muy mal resuelto en SQL es el de la herencia, el generador no es capaz de crear las tablas derivadas con una clave que las relacione directamente con la tabla de la superclase. Por otro lado, el generador Java actúa correctamente en la representación de las herencias entre clases, siempre y cuando no exista herencia múltiple, caso en el que la herencia está mal representada, puesto que indica que cierta clase extiende las superclases de las que deriva, no siendo posible dicha notación en el lenguaje Java.

Tabla comparativa

En el siguiente cuadro podemos ver una comparativa resumida de las distintas características analizadas y cómo las soluciona cada generador.

Grupo	Elemento	Java	SQL
Clase	Estereotipos	No	Afectan negativamente: no se define la tabla.
	Atributos de clase	Si.	No sabe qué hacer con ellos.
	Restricciones	No.	No
Atributos	Múltiples: vectores, tablas, etc.	Si, a través de clases como *Collection*.	No sabe qué hacer con ellos, los define también como *Collection*.
	Orden	Si, a través de inicializar con las clases *TreeSet* y *ArrayList*	No
	Control de la manipulación	Si, mediante la palabra reservada *final* o *constant*	No
	Persistencia	Si, mediante la palabra reservada *final*	No

Grupo	Elemento	Java	SQL
Métodos	Concurrencia	No	No
	Modificadores	Si, los estándares de Java: *abstract, static* y *final*.	No
	Estereotipos	No	No
Asociaciones	Relación 0,1:N	Si	Si
	Relación 1:N	Si	Si, pero incorrectamente, depende de la accesibilidad y no incluye la cláusula *NOT NULL*.
	Relación M:N	Si	No
	Direccionabilidad	Si.	Si.
	Orden	Si, mediante las clases *TreeSet* y *ArrayList*, para atributos múltiples.	No.
	Agregaciones, Composiciones	No.	No
	Clases Asociativas	Si.	No.
	Asociaciones múltiples en la misma dirección	Si.	No, se hace un lío.
Herencia	Simple	Si	No
	Múltiple	No, código mal generado.	No.

CARENCIAS DE LA APLICACIÓN GENERADA

Por supuesto, la aplicación generada dista mucho de ser una aplicación comercial.

Noto a faltar, especialmente, la posibilidad de generar de forma automática clases *frontera* (interfaz de usuario) y de *control* (interacción con SQL a través de JDBC/ODBC, por ejemplo), que propicien el mantenimiento estándar (un bastidor de aplicación) de datos, a partir de una clase con estereotipo *entidad*. Una herramienta de este tipo aceleraría el desarrollo de la parte más tediosa (mantenimiento de datos maestros) y permitiría que nos centráramos en el núcleo del problema.

En cuanto al código generado, podemos decir que en el caso de SQL el código es prácticamente inaprovechable, debido principalmente al problema de conversión de tipos de datos, entre los tipos de datos seleccionados en el diagrama y los generados finalmente para SQL, que no son compatibles. De todas formas, el código solo modela los datos, sin que exista ninguna lógica de actuación sobre ellos, que debería implementarse aparte.

Descartando todos los elementos directamente relacionados con el servidor Internet y la gestión del site), desde mi punto de vista, la aplicación requeriría los siguientes elementos para poder tener carácter comercial:

- Interfaz de usuario: menús, páginas web activas, ventanas Java o cualquier otro interfaz deseado.

- Lógica: El código generado sólo afecta a los métodos accesors de los atributos definidos, sin embargo, no existe ninguna lógica generada a partir de los diagramas (como el de colaboración, por ejemplos).
- Interfases con otros sistemas: interfaz con el servidor Internet, con el servidor donde residan los datos y con las entidades representadas por las clases *Aerolínea* y *Financiera*.
- Clases de utilidad: el modelo representa la información del dominio, sin embargo, en las fases de diseño aparecerán muchas otras clases necesarias para implementar la ya comentada lógica de la aplicación.
- Arreglar la herencia múltiple: en caso de que se haya definido la herencia múltiple en algún caso, ésta deberá ser resuelta utilizando los trucos habituales en el desarrollo con Java.

CONCLUSIONES FINALES

Las reflexiones incluidas en este apartado hacen mención exclusiva a las pruebas realizadas con los dos generadores elegidos: Java y SQL, no siendo extensibles al resto.

Poseidon es una gran herramienta diagramática, que permite representar los 9 diagramas de UML. Su potencia generatriz, sin embargo, se reduce a la mínima información que puede extraerse del diagrama de clases de la aplicación modelada, no teniendo ningún impacto los diagramas mas relacionados con la propia lógica del sistema (colaboración, estados...).

Teniendo en cuenta esta limitación, su generación de código Java se ciñe con bastante fidelidad al modelo representado en el diagrama de clases, con la clara excepción de la herencia múltiple, que no es capaz de resolver con algún artificio específico.

Otra ventaja del generador Java reside en la posibilidad de utilizar la tecnología de *ida y vuelta*, que permite al programador modificar directamente las clases generadas y que dichos cambios se reflejen automáticamente en *Poseidon*.

En contra, no existe la posibilidad de generar el código Java de mantenimiento de los datos maestros incluidos como atributos en las tablas, ni en formato web (HTML, CGI, PHP, ...) ni en formato tradicional (Jframe, Jmenu....), ni la generación automática de gestores de persistencia (puente JDBC/ODBC), que incrementen la productividad.

En cuanto a SQL, su utilidad es prácticamente nula si partimos de un modelo que no esté directamente orientado a SQL. Especificando para cada atributo un tipo de dato compatible con SQL, podríamos llegar a obtener el código DDL que permitiera generar las tablas correspondientes a las clases del modelo.

En mi opinión, para SQL, sería más útil un generador que estuviera basado en un diagrama E/R de *Chenn* o *Young* ([EP2]) y que reflejara mejor el modelo relacional. Sería también deseable algún generador que permitiera añadir *triggers* y *procedures* asociados a las tablas que pudieran ser directamente incorporados como parte de definición de la base de datos en el sistema.

Resumiendo, considero el producto bastante adecuado para implementar una aplicación en Java, siempre y cuando tengamos la precaución de modelar teniendo en cuenta las limitaciones reales del lenguaje elegido y, entendamos las limitaciones que se derivan del hecho de generar código tan solo a partir de la definición UML de una clase o de sus asociaciones.

Respecto a SQL, no creo que sea un producto que tenga mucha utilidad, sobre todo si el grueso de la aplicación va a ser prácticamente código SQL.

Por último, habría sido interesante que *Poseidon* dispusiera de un tipo de datos genéricos que supiera mapear correctamente al tipo de dato más adecuado a cada uno de los lenguajes cuyo código es capaz de generar, de esta forma se evitaría tener que realizar un modelado muy dependiente del lenguaje de implementación finalmente elegido.

GLOSARIO

Término	Significado
CASE	Computer Aided Software Engineering (Inglés). Ingeniería de Software Asistida por Ordenador (Español). Herramientas de programación que permiten generar software de forma automática a equipos de desarrollo. Puede incluir herramientas para: sumarizar requisitos iniciales, desarrollar diagramas de flujo, planificar las tareas de desarrollo, preparar documentación, controlar las versiones de software y desarrollar el código del programa. Varias empresas ofrecen programas CASE capaces de soportar alguna o todas de dichas actividades. Aunque algunos sistemas CASE proveen soporte específico para programación OO, el término CASE se puede aplicar a cualquier tipo de entorno de desarrollo de software.
CORBA	Common Object Request Broker Architecture (Inglés). Arquitectura de Intermediario Común para Peticiones de Objetos (Español). Arquitectura que permite a piezas de programas, llamados objetos, comunicarse con otros independientemente del lenguaje de programación en el que fueron escritos y del sistema operativo en el que están ejecutándose. Fue desarrollado por el OMG. Hay varias implementaciones de CROBA, las más usadas son las arquitecturas de IBM: SOM y DSOM. Otros modelos en competencia son: COM y DCOM de Microsoft y RMI de Sun Microsystems.
DDL	Data Definition Language (Inglés). Lenguaje de Definición de Datos (Español). Es un sublenguaje de la especificación SQL empleado para la definición de datos en un entorno de Bases de Datos Relacionales.
E/R	Entity / Relationship (Inglés). Entidad / Relación (Español).
FK	Foreign Key (Inglés). Clave foránea (Español). Columna de una tabla que apunta a la clave primaria de otra tabla, con la que está relacionada por alguna razón determinada definida en el modelo de datos.
GUI	Graphic User Interface (Inglés). Interfaz Gráfica de Usuario (Español). Interfaz de programación que permite al usuario interactuar con el sistema a través de elementos gráficos: menús, iconos, ventanas, ...
HTML	HyperText Markup Language (Inglés). Lenguaje de Marcaje de HiperTexto (Español). Lenguaje usado para crear documentos en la WWW. Es similar a SGML, aunque no es un subconjunto estrictamente hablando. HTML define la estructura y aspecto de un documento Web usando una gran variedad de etiquetas y atributos.
IDE	Integrated Development Environment (Inglés). Entorno de Desarrollo Integrado. (Español). Herramienta que facilita la programación, documentación y depuración del software. Suele comprender: un generador de GUI, editor de código fuente, compilador y/o intérprete y un depurador.
Metadato	Datos sobre otros datos. Un metadato describe como, cuando y para qué se recoge un determinado dato y como debe formatearse. Los metadatos son esenciales para entender la información almacenada en Almacenes de Datos y su importancia ha crecido por las aplicaciones Web basadas en XML.
MOF	Meta-Object Facility (Inglés). Lenguaje abstracto y estructura para especificar, construer y manejar metamodelos de tecnología neutral. Un metamelo es en efecto un lenguaje abstracto para algún tipo de metadato.
OMG	Object Management Group (Inglés). Grupo de Gestión de Objetos (Español). Es un consorcio de 700 empresas. El objetivo de la organización es proveer una estructura común para desarrollar aplicaciones usando técnicas OO. Es responsable de la especificación *CORBA*.
OO	Object Oriented (Inglés). Orientado a Objetos (Español).
PK	Primary Key (Inglés). Clave primaria (Español). Clave que identifica inequívocamente cada fila de una determinada tabla relacional.

Término	Significado
SGBDR	Sistema Gestor de Bases de Datos Relacionales. Motor de base de datos relacional.
SGML	Standard Generalized Markup Language (Inglés). Lenguaje de Marcado Generalizado eStandard (Español). Un sistema para organizar y etiquetar elementos de un documento, desarrollado por ISO en 1986. SGML no especifica ningún formato particular; en vez de ello, especifica las reglas para etiquetar los elementos. Estas etiquetas pueden ser interpretadas para formatear los elementos de distintas formas. SGML se usa ampliamente para manejar documentos grandes que están sujetos a revisiones frecuentes y necesitan ser impresos en formatos distintos. Debido a que es un sistema complejo y grande, no se utiliza mucho en PCs. Sin embargo, el crecimiento de Internet y especialmente el WWW, ha renovado el interés en SGML porque WWW utiliza HTML, un camino para definir e interpretar etiquetas conforme a las reglas SGML.
TFC	Trabajo de Fin de Carrera
UML	Unified Modelling Language (Inglés). Lenguaje de Modelado Unificado (Español). Lenguaje notacional de propósito general para la especificación y visualización de proyectos de software OO complejo y especialmente grande. Se basa en métodos notacionales previos como Booch, OMT y OOSE.
Viewlet	Presentación incrustada en una página de Internet.
Web **World Wide Web** **WWW**	World Wide Web (Inglés). Red de Área Mundial (Español). Es un sistema de servidores Internet que soportan documentos especialmente formateados con HTML. No todos los servidores Internet forman parte de la World Wide Web (Red de Área Mundial).
XMI	XML Metadata Interchange. Una aplicación XML que facilita el intercambio estandarizado de modelos de objetos y *metadatos* en Internet para grupos en entornos de desarrollo que utilizan aplicaciones de varios proveedores. XMI también puede ser usado para intercambiar información en *Almacenes de Datos*. XMI está basado en tres estándares industriales: XML, UML y MOF. La arquitectura permite a las herramientas compartir metadatos usando interfaces XML o CORBA especificadas en los estándares UML o MOF.
XML	eXntensible Markup Language (Inglés). Lenguaje de Marcado eXtensible (Español). Es una versión recortada de *SGML*, diseñada específicamente para documentos Web. Permite a los diseñadotes crear sus etiquetas personalizadas, activando la definición, transmisión, validación e interpretación de datos entre aplicaciones y entre organizaciones.

BIBLIOGRAFÍA

DOCUMENTOS

Reseña	Documento	Autor	Editorial
[BD1]	Bases de Dades I	Jaime Sistac Planas	Universitat Oberta de Catalunya
[Codd]	The Relational Model for Database Management: Version 2	E.F. Codd	Addison Wesley Publishing Company
[EP1]	Enginyeria del Programari 1	Benet Campderrich Falgueras	Universitat Oberta de Catalunya
[EP2]	Enginyeria del Programari 2	M. Jesús Marco Galindo	Universitat Oberta de Catalunya
[Grau]	Desarrollo Orientado a Objetos con UML V 2.0 http://www.tevasoft.com/UMSA/ADOO/doo_uml.pdf	Xavier Ferrer Grau	Facultad de Informática - UPM
[Larman99]	UML y Patrones	C. Larman	Prentice Hall
[Mendez]	Trabajo de fin de carrera Aplicación del método Larman a la construcción de entornos virtuales http://bermudas.ls.fi.upm.es/~gonzalo/docs/publicaciones/tfc.pdf	Gonzalo Méndez Pozo	Facultad de Informática - UPM
[Yourdon]	Análisis estructurado moderno	Edward Yourdon	Prentice Hall

NOTAS

Del documento de [Grau] he tomado los conceptos que hay tras el método de [Larman99] para realizar el análisis de la aplicación, al estar en formato PDF y disponible en Internet, me resulta más fácil de manejar y consultar que el original.

Del documento de [Méndez] he tomado la idea de la realización práctica de dicho método.

El resto de documentos han sido citados como fuentes originarias de otros elementos que he utilizado (diagrama de contexto, por ejemplo).

ANEXOS

Este documento constituye la memoria del TFC, sin embargo el trabajo completo es el conjunto de una serie de ficheros que se adjuntan a este documento, los cuales pasamos a relacionar:

Asanchezl_planificación.xls	Hoja de cálculo que contiene la planificación día a día del TFC.
Asanchezl_platreball.doc	Documento que indica el compromiso en cuanto a planificación para el desarrollo del TFC. Contiene la planificación gruesa, así como las directrices del proyecto.
Asanchezl_tfc.zuml	Fichero de la aplicación *Poseidon for UML professional Edition*, que contiene los diagramas resultantes del modelado de la aplicación de este TFC.
***.zuml**	Otros archivos utilizados para realizar pruebas y documentar distintas situaciones particulares.
***.java**	Ficheros resultantes de la generación automática de código Java a partir del diagrama de clases introducido en *Poseidon*
***.sql**	Ficheros resultantes de la generación automática de código SQL a partir del diagrama de clases introducido en *Poseidon*