Ciclo Formativo DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Módulo 5 Entornos de desarrollo

Unidad Formativa 3

INICIATIVA Y COORDINACIÓN
IFP Innovación en Formación Profesional

Supervisión editorial y metodológica:
Departamento de Producto de Planeta Formación

Supervisión técnica y pedagógica:
Departamento de Enseñanza de **IFP** Innovación en Formación Profesional

Módulo: Entornos de desarrollo
UF 3 / Desarrollo de Aplicaciones Multiplataforma

MÓDULO 5

Unidad Formativa 3

Entornos de desarrollo

Esquema de contenido

1. INTRODUCCIÓN A UML
2. DISEÑO DE CLASES EN UML
   - 2.1. CLASES, ATRIBUTOS Y MÉTODOS
   - 2.2. RELACIONES
3. HERRAMIENTAS
   - 3.1. HERRAMIENTAS DE MODELADO DE VS
   - 3.2. UMLPAD
4. TIPOS Y CAMPO DE APLICACIÓN
5. DIAGRAMAS DE ACTIVIDAD
6. DIAGRAMAS DE CASOS DE USO
7. DIAGRAMAS DE SECUENCIA
   - 7.1. INGENIERÍA INVERSA

OBJETIVOS

- Aprender lo que es el estándar UML y su importancia en el diseño de software.
- Manejar herramientas de modelado para crear diagramas UML.
- Comprender el concepto de los diagramas de clase y su utilidad.
- Saber diseñar una estructura de clases a partir de la descripción de un problema o requisitos de un software.

1. INTRODUCCIÓN A UML

Para realizar labores de diseño de software, es vital realizar modelados o diagramas representando gráficamente la estructura de la aplicación y su funcionalidad, de una manera fácil de entender y rápida de crear. Antiguamente, los diagramas de cada diseñador eran únicos, ya que, salvo un par de diagramas conocidos por todos (como los diagramas de flujo o los diagramas de entidad-relación), cada diseñador o analista realizaba los diagramas de la manera que mejor los entendía, por lo que podría suponer un problema si varias personas tenían que entender los diagramas que uno o varios diseñadores les presentaban para definir el software.

Con el fin de evitar ese problema se creó UMIL (Unified Modeling Language), un conjunto unificado de estándares para las diferentes necesidades y usos que un diseñador pudiera tener a la hora de plantear una representación gráfica de un programa. Son diagramas de propósito general que se presuponen conocidos por todos, con unas técnicas de notación conocidas, de modo que cualquiera pueda crear diagramas entendibles por todos.

UML es un lenguaje de modelado y, como hemos dicho, estandarizado, tiene su aplicación más importante en el desarrollo de software, siendo extremadamente útil para dar soporte a una gran cantidad de metodologías de software, pero no de modo restrictivo, es decir, un diagrama UML solo define una semántica mediante una serie de reglas y notaciones, pero no especifica cuál sería la metodología o procedimiento que hay que usar

UML ha pasado por diferentes versiones hasta convertirse en lo que es hoy en día, actualizándose y mejorando en cada paso, adaptando y soportando un cada vez más variado elenco de técnicas de diseño. Principalmente, podemos ver que hay tres etapas en el progreso de los diagramas UML:UML1.0, UML1.x y UML 2.0. La versión actual (UML 2.0) está siendo soportada y respaldada por OMG (Object Management Group), quienes deciden las características y notaciones del lenguaje.

En la versión 2.0 de UML se definió un completo árbol de superestructuras donde se relacionaban y complementaban todos los tipos de diagramas UML a la hora de definir un software por completo. Con esa nueva especificación de estructuras y una actualización en las notaciones de los diagramas, se observaba de manera más clara el funcionamiento y estructura de un modelado completo de UML para un software.

A pesar de prescribir unas notaciones y semánticas estandarizadas, UML sigue siendo criticado por algunos grupos al no tener unas reglas lo suficientemente estrictas como para evitar problemas de interpretación. Por muy bien especificadas que estén las notaciones y semántica del diagrama, muchos diagramas tendrán que ser interpretados, no todo el mundo podrá realizar la misma implementación partiendo de los mismos diagramas. No obstante, podemos confirmar que la aplicación resultante debería tener un funcionamiento y un ciclo de vida idéntico. No se puede pretender que un diagrama no deba ser interpretado, aunque ocasione interpretaciones diferentes, un diagrama no deja de ser una guía, un recurso utilizado en la fase de diseño de un software, y, como es de esperar, el software final no será una traducción directa de dichos diagramas por muy bien trabajados que estén.

Teniendo presente la superestructura de diagramas de UML, podríamos pensar que la tarea de definir y realizar dichos diagramas (y que por supuesto sean coherentes entre sí) sería una tarea abrumadora, pero no es necesario ni se suelen realizar todos los diagramas para modelar un software, por norma general se utilizan solo una serie de diagramas para modelarlo, lo más clásico sería la tríada diagrama de clases, de secuencia y de casos de uso.

En este capítulo vamos a ver el diagrama más básico y sencillo de todos, el diagrama de clases, que tiene un gran parecido con el diagrama clásico de entidad-relación. Además, es uno de los diagramas que sí tienen una traducción directa, es decir, cuando implementas un diagrama de clases, la implementación será idéntica a la realizada por otro programador.

2. DISEÑO DE CLASES EN UML

Uno de los diagramas más básicos e importantes que realizaremos en nuestras labores de diseño de software serán los diagramas de clases. Se basan en ciertas reglas y notaciones sencillas para relacionar las clases y sus diferentes operaciones entre sí. En la programación orientada a objetos son un recurso básico y recurrente, usado para mostrar los bloques de construcción de cualquier sistema. Se utilizan principalmente para describir la capa del modelo y las relaciones entre las entidades del sistema.

2.1. CLASES, ATRIBUTOS Y MÉTODOS

Cada bloque dentro del diagrama representa a una clase; una clase, como sabemos, dispone de unos atributos y de unos métodos asociados. Representaríamos las clases como cajas en donde escribiríamos sus atributos y sus métodos.

Las clases representan a nuestros objetos, los atributos definen las propiedades y características del objeto y los métodos especifican las acciones que podemos realizar con dicho objeto.

Estos diagramas son particularmente útiles en el desarrollo de la capa del modelo, para mapear de un modo gráfico las entidades y sus relaciones en nuestra aplicación, ese tipo de diagramas tienen la particularidad de que no muestran las acciones que se realizan sobre cada objeto, ya que solo representan la entidad y se encuentran separadas las acciones de los datos, por lo que nuestro diagrama solo mostraría las clases y sus atributos. Dependiendo del lenguaje utilizado, las clases sí que tendrían operaciones, como por ejemplo los métodos set y get utilizados para obtener y establecer los atributos de los objetos siempre y cuando los atributos estén encapsulados. En C# realizaríamos esa operación definiendo los atributos como propiedades.

Vamos ahora a utilizar un ejemplo muy sencillo que iremos ampliando según nos surjan las necesidades en la aplicación. En este caso se trata de una libreta de contactos, donde podremos escribir el nombre, email, teléfono y dirección de cada contacto.

Observamos que, en este caso, tenemos dos clases, y que, por las características de nuestra aplicación, una de las clases tienen atributos y la otra solo tiene métodos. Eso es debido a que nuestra clase Libreta albergará los datos, y esos datos son de contactos, por lo que al tener una única libreta para nuestros contactos, y teniendo en cuenta que la libreta solo tiene datos de los contactos, la libreta no tiene información adicional de la contenida en los contactos y los contactos no tienen ninguna operación que realizar sobre ellos. Faltaría la creación de contactos, pero de ese asunto se encarga el constructor de la clase Contacto, por lo que no es necesario especificarlo, ya que es intrínseco a la propia clase.

2.1.1. Notación

Como hemos dicho al principio del capítulo, los diagramas UML responden a un estándar global prefijado de antemano, por lo que las notaciones son estandarizadas y es necesario conocerlas para que nuestros diagramas sean entendidos por todos.

En primer lugar, debemos percatarnos de que las clases se definen mediante cuadrados, y dichos cuadrados se dividen en tres segmentos horizontales.

Primero tendríamos el nombre de la clase y por tanto de nuestro objeto, seguidamente irían los atributos y, al final, los métodos de la clase. Es importante resaltar que si alguna clase no posee atributos propios o métodos propios, se deberá seguir dibujando el segmento que le corresponde, incluso si está vacío.

Vemos además que la especificación de los atributos y los métodos no se termina simplemente escribiendo su nombre, sino que llevan un carácter como prefijo y un tipo de dato separado por dos puntos. Además, los métodos pueden llevar un tipo de dato en los paréntesis. El carácter que precede al nombre del atributo o método se corresponde con su grado de comunicación.

- + Público, el elemento será visible tanto desde dentro como desde fuera de la clase.
- Privado, el elemento de la clase solo será visible desde la propia clase.
- # Protegido, el elemento no será accesible desde fuera de la clase, pero podrá ser manipulado por los demás métodos de la clase o de las subclases.

Después del nombre del método o atributo, tendríamos un tipo de dato de devolución separado por" ", de este modo podemos saber de qué tipo de dato estamos hablando y cuál será el valor de retorno, tanto de los atributos como de los métodos, es decir, cuando accedamos a alguno de los elementos de clase, obtendremos un tipo de dato (si procede, los métodos void no tienen valor de retorno), el tipo de dato bien puede ser propio del lenguaje o plataforma desde la que trabajamos o, como podemos ver en el ejemplo de la libreta, puede ser un objeto de nuestra propia aplicación.

Por último, tendríamos el tipo de dato que se encuentra entre paréntesis, como bien habréis podido adivinar, se trata del tipo de dato que espera nuestro método como parámetro, al igual que con los valores de retorno pueden ser de cualquier tipo de dato, ya sea del lenguaje o propio.

2.2. RELACIONES

Si analizamos con atención el diagrama de la libreta, podríamos percatarnos de que algo falla, a nuestro diagrama le falta algo, hemos especificado que la libreta contiene contactos, pero en ningún sitio lo hemos especificado, bien podríamos decir que la libreta tendría una lista de contactos, pero ésta no aparece por ningún lado. Realmente no se trata de un error, ni de un fallo en el diseño de la aplicación, se trata de evitar información redundante.

En el diagrama de clases, además de dibujar y representar las clases con sus atributos y métodos, también se representan las relaciones y, por ese motivo, esa lista quedaría implícita en la relación entre las clases.

Las relaciones se representan con flechas con una forma determinada, y, además, poseen una cardinalidad. La cardinalidad es un número o símbolo que representa al número de elementos de cada clase en cada relación, pudiendo usar el comodín "para definir un número indeterminado de elementos.

2.2.1. Asociación

La asociación es la más básica de las relaciones, no tiene un tipo definido y puede ser tanto una composición como una agregación, además una asociación podría implicar únicamente un uso del objeto asociado. Se representan mediante una flecha simple que también puede tener una cardinalidad.

2.2.2. Composición

La composición define los componentes de los que se compone otra clase, se define además que la clase que contiene la composición no tiene sentido de existencia si la(s) agregada(s) desaparece(n). Mediante esta relación, denotada por una flecha con un rombo relleno en una de sus puntas, se indica la presencia de las clases

origen en la clase destino, donde la clase destino es apuntada por el rombo de la relación.

Ahora ya tenemos representada en nuestra libreta a la lista de contactos. Como vemos, al utilizar la cardinalidad uno a muchos (1-o), donde ello se encuetra en la clase Contacto y el 1 en la clase Libreta, se especifica que una Libreta se compone de muchos Contactos.

La diferencia principal entre la agregación y la composición reside en que la agregación no implica que la clase agregada necesite una instancia de la otra clase, es decir, la cuenta no necesita tener una instancia de la clase Libreta y libreta tampoco necesita una instancia de la clase Cuenta. Sólo implica que la clase Cuenta accede a la clase Libreta. En ocasiones esa diferencia no resulta tan clara como debería, es por ello que en un primer boceto del diagrama de clases se suelen representar estas relaciones como relaciones de asociación, para más adelante "reforzarlas" mediante la relación apropiada, recordemos que las relaciones de composición y agregación son solo una versión más fuerte y específica de las relaciones de asociación.

2.2.3. Herencia

La herencia es un modo de representar clases y subclases, es decir, clases más específicas de una general, se representan mediante una flecha con una punta triangular vacía.

3. HERRAMIENTAS

A pesar de que la creación de los diagramas de clase sea una tarea sencilla, al ser realizados en la fase de diseño, se pierde mucho tiempo recolocando y pasando a limpio todas las ediciones que hemos realizado a nuestro boceto inicial, para evitar este tipo de problemas se utilizan programas que nos permiten mover las clases y crear las relaciones de una manera muy rápida, sencilla y por supuesto modificable.

3.1. HERRAMIENTAS DE MODELADO DE VS

En cualquier proyecto de Visual Studio podemos agregar un diagrama como se haría con cualquier otro componente, pero podemos crear en su defecto un proyecto específico para crear los diferentes modelados que necesitemos. Para aprender el funcionamiento de la herramienta de modelado utilizaremos un proyecto nuevo de modelado. Para ello, iremos a la entrada de menú Arquitectura > Nuevo Diagrama para que nos aparezca el cuadro de diálogo de creación de diagramas. Elegiremos la plantilla Diagrama de clases UML y haremos clic en Aceptar. Nos creará una pantalla en blanco con las instrucciones necesarias para empezar a trabajar

Tal y como bien nos indica Visual Studio, iremos primero al cuadro de herramientas y elegiremos el elemento Class.

Lo añadiremos en cualquier punto de nuestro cuadro de dibujo y veremos que nos aparece un pequeño recuadro con el rótulo Class. 1, que sería el nombre de nuestra clase, la llamaremos Familia, repetiremos la operación creando otra clase llamada Persona. Deberemos añadir los atributos que correspondan para que el diagrama tenga sentido, es decir, sabemos que una familia es un conjunto de personas.

Mientras añadimos los atributos, nos damos cuenta de que Visual Studio nos rellena automáticamente el acceso si no lo hemos definido. Lo mismo ocurre con las operaciones o métodos, que nos escribe los paréntesis si no lo hemos hecho nosotros, pero tenéis que tener cuidado, porque si no escribís los paréntesis pero sí definís el tipo de retorno, os pondrá los paréntesis después del tipo de dato, lo que sabemos que es una notación incorrecta.

Cuando añadáis las relaciones, debéis tener presente que se deben añadir desde la clase contenedora a la clase contenida, mientras que la herencia se hace desde la clase hija a la clase padre.

Vamos a realizar también una relación de herencia, ya que sabemos que una familia se compone de un padre, una madre e hijos, es decir, las personas incluidas en una familia están relacionadas entre sí.

Dejando de lado la poligamia, y la orfandad, suponemos que una familia debe tener un solo un padre y una sola madre, y que ellos pueden tener varios niños en conjunto.

Como vemos, el funcionamiento y manejo de la herramienta es muy sencillo, y basta con arrastrar y soltar para tener un diagrama de clases muy sencillo creado en un momento.

Basta con hacerdoble clic sobre los diferentes elementos (cardinalidad, rótulos, atributos...) para editarlos, y con un par de movimientos de ratón podemos moverlos a nuestro antojo sin que las relaciones se emborronen o distorsionen.

3.1.1. Generar código a partir de diagramas de clases

Ya hemos visto cómo utilizar la herramienta de modelado para crear proyectos de modelado y así diseñar nuestra aplicación. También hemos hecho notar al principio de este capítulo que los diagramas de clases

tienen una relación directa con el código, es decir, se puede crear una traducción exacta de un código a un diagrama de clases y de un diagrama de clases a un código. Esta funcionalidad nos la ofrece la herramienta de modelado, para ello crearemos un nuevo proyecto de aplicación de consola que llamaremos Clasesy Codigo. Nos generará la plantilla básica con una clase program y su método main. Esa clase no la vamos a tocar, en su lugar vamos a ir añadiendo las clases de nuestro programa, tal y como haríamos normalmente, pero en vez de crear las clases según el método tradicional lo vamos a hacer mediante un diagrama de clases, por ello, una vez creado el proyecto, haremos clic con el botón derecho en él y elegiremos la opción Agregar > Nuevo elemento. Elegiremos la plantilla diagrama de clases y nos aparecerá la misma vista de dibujo que hemos visto hasta ahora.

Vamos a modelar una aplicación muy sencilla, donde tendremos solamente dos clases: una clase Operario y una clase Trabajos, donde representamos los trabajos que puede desempeñar un operario. Primero vamos a crear nuestra clase Operario, para ello iremos al cuadro de herramientas del diagrama y arrastraremos el elemento "Clase" hasta el cuadro de dibujo.

Nos aparecerá un cuadro de diálogo donde rellenar los datos necesarios para la clase, en este caso solo tendremos que añadir el nombre de la clase, darle a Aceptar y ya tendríamos nuestra clase tanto en el diagrama como en el proyecto. Si abrimos la clase que nos acaba de crear automáticamente, veremos que es una plantilla que solo contiene un constructor vacío, pero nuestro operario sin duda tiene unos atributos. Volvamos al diagrama de clases para añadirle los atributos necesarios, si hacemos clic con el botón derecho en la clase del diagrama, y nos vamos a Agregar, veremos que nos aparecen una serie de opciones para agregar a a la clase, un atributo es un campo, por lo que elegiremos la opción campo, nos saldrá entonces una entrada en la clase llamada "field" donde podremos escribir el nombre, escribiremos nombre y presionaremos Intro.

Si ahora nos vamos a la clase Operario, veremos que ha creado un atributo privado llamado Nombre y vemos también que ese campo es de tipo int y eso no es correcto, bien podríamos arreglarlo desde el archivo de la clase o desde el diagrama, vamos a hacerlo desde el diagrama. Si seleccionamos el campo en nuestro diagrama, vemos que en la ventana de propiedades nos aparecen todas las características del campo. Entre ellas se encuentra la propiedad tipo, que podemos modificar a nuestro antojo, en este caso escribiremos string, podemos ver como al cambiar el tipo del campo en las propiedades también se ha cambiado en el diagrama. Nuestro código se encuentra ahora completamente enlazado con el diagrama, cualquier cambio en el diagrama modifica el código y cualquier cambio en el código modifica el diagrama. Añadiremos de este modo el atributo dini, que será un entero, por lo que no tendremos que cambiar el tipo. En el campo dini añadiremos también un comentario, escribiremos en la propiedad Comentario lo siguiente: "Sólo los números, no se tiene en cuenta la letra", y veremos que dicho comentario se ha convertido en un comentario justo encima del atributo dini. Haremos lo propio con nuestra clase Trabajo con los atributos nombre y descripción, los dos strings.

Una vez con nuestras dos clases y antes de relacionarlas, vamos a añadir un método a la clase Operario que se llame Tiene El Trabajo, donde recibirá un trabajo como parámetro y devolverá un booleano como respuesta indicando si tiene o no tiene el trabajo. Al igual que con los atributos, podemos ponerle el nombre y su tipo de devolución desde la ventana de propiedades, pero no podemos añadirle los parámetros. Para añadirle los parámetros vamosa hacer uso de una ventana extremadamente útil en la herramienta de modelado, haremos clic con el botón derecho en el método y elegiremos la opción Detalles de clase, veremos una ventana con toda la información de la clase de una manera muy compacta y útil, además de permitirnos añadir o modificar cualquier información sobre nuestros elementos, utilizaremos esta ventana de ahora en adelante para realizar todas estas operaciones. Si expandimos los detalles del atributo haciendo clic en la flecha del método, podremos añadir los parámetros en la caja de texto rotu-

lada como <agregar parámetro», llamaremos al parámetro trabajo y elegiremos como tipo nuestra clase Trabajo.

Si nos fijamos en nuestra clase Operario, veremos que nos ha creado un método vacío que simplemente lanza una excepción indicando que no se ha implementado la creación automática del método. Bien, parece que lo tenemos todo listo, solo nos quedaría definir la relación entre operarios y trabajos.

Vamos a definir la relación indicando que un operario puede realizar uno o varios trabajos, por lo que elegiremos la herramienta de asociación y la arrastraremos desde la clase Operario a la clase Trabajo. Nos aparecerá una propiedad relacionando las dos clases, por desgracia no podemos definir una cardinalidad, pero podemos hacer otra cosa. Desde la ventana Detalles de clase de la clase Operario vemos nuestra propiedad Trabajo, la vamosa renombrara Trabajos y vamos a cambiarle el tipo (cuando modificamos un tipo en una relación, nos aparece el espacio de nombres al completo) definiendo que es una lista, por lo que cambiaremos el tipo de Trabajo a Lista. Trabajo>. Al hacerlo, vemos que la propiedad que representaba la relación ahora se encuentra en una sección del diagrama que se llama Propiedades; realmente, la relación existe y es la misma a efectos prácticos, pero no es eso lo que queremos ver, es menos claro, pero podemos arreglarlo. Haremos clic con el botón derecho en la propiedad y elegiremos la opción Mostrar cómo relación de colecciones y volverá a mostrarse automáticamente como la asociación que queremos representar

Y ya tendríamos nuestra estructura de clases prácticamente terminada, solo nos queda un detalle más. Tenemos nuestros atributos como privados, como tiene que ser, pero no hemos realizado la encapsulación ni nada por el estilo, por lo que no serán accesibles, no pasa nada, es normal, tenemos la estructura, pero no hemos implementado el código para hacerlo funcional, pero tenemos a nuestra disposición una funcionalidad adicional que hace a esta herramienta más útil de lo que aparenta. Podemos utilizar todas las refactorizaciones automáticas de Visual Studio que vimos en el tema anterior directamente desde el diagrama, por lo que si hacemos clic con el botón derecho en un campo y nos vamos al menú Refactorizar, vemos que tenemos la opción de encapsular el campo directamente, haremos clic en esa opción y nos aparecerá una ventana para que escribamos el nombre de la propiedad, escogemos la que más nos interesa (por convenio se suele utilizar el mismo nombre que el atributo pero con letra capital) y aplicamos los cambios. Vemos como nuestro campo está perfectamente encapsulado como propiedad. De este modo ya tendríamos nuestro diagrama y nuestra estructura de clases perfectamente realizada, y todo mientras realizábamos el diagrama.

Las posibilidades de esta herramienta son infinitas, aunque sea algo más restrictiva que si simplemente creamos un proyecto de modelado, podemos crear la estructura de clases al mismo tiempo que realizamos el diagrama, lo cual sin duda nos ahorra una gran cantidad de tiempo.

3.1.2. Ingeniería inversa

Visual Studio nos permite realizar una ingeniería inversa de un proyecto con su herramienta de modelado, pero, ¿qué es la ingeniería inversa en el contexto de los diagramas de clase? La respuesta es bien sencilla: consiste en obtener de forma automática el diagrama de clases de un proyecto mediante su código.

El procedimiento de esta poderosa herramienta es muy sencillo, lo único que tenemos que hacer es seleccionar un proyecto de los que hemos realizado hasta ahora, donde hayamos utilizado en código relaciones de asociación o herencia. Una vez abierto, haremos clic con el botón derecho en el proyecto y elegiremos la opción Ver diagrama de clases, automáticamente, Visual Studio nos creará un diagrama con las clases utilizadas en el proyecto, aunque sin relacionar, nosotros solo tendremos que establecer correctamente las relaciones entre las clases para dejarlo correctamente.

Es también un dato interesante la posibilidad de ver la clase base de una clase, como la de un WebService o de un formulario Winforms, solo tenemos que hacer clic con el botón derecho en la clase de la que queremos sacar la derivada y elegir la opción Mostrar clase base, automáticamente, nos creará y relacionará la clase base en nuestro diagrama de clases.

3.2. UMLPAD

UMLPad es una herramienta muy sencilla y ligera que nos permite crear diagramas de clases. Es una herramienta externa y portable y su manejo es muy sencillo, muy similar al de la herramienta de modelado de Visual Studio.

El primer paso sería descargar el programa, el cual podemos encontrar en su web oficial: http://web.tiscali.it/ggbhome/

Una vez descargado, no necesita instalación, es una carpeta comprimida que contiene el ejecutable del programa. Cuando lo ejecutamos, nos aparece una sencilla ventana dividida en dos. A la izquierda tenemos nuestro árbol de diagramas, que se actualizará según vayamos creando diferentes diagramas. En la parte derecha, tenemos nuestro cuadro de dibujo, donde incluiremos los elementos de nuestro diagrama. En la parte superior, tenemos además una barra de herramientas donde podemos elegir los elementos que vamos a añadir.

Para añadir una clase, bastaría con hacer clic en el elemento Clase y luego hacer clic donde queramos para que se posicione, y del mismo modo que hemos hecho hasta ahora, relacionar las clases seleccionando y arrastrando.

Sin embargo, vemos que no reacciona a nuestros dobles clics para editarlo, para editar un elemento y agregarle los valores, seleccionaremos un elemento (suborde se volverá de color rojo) y haremos clic con el botón derecho eligiendo la opción Edit Object en el menú contextual que aparece.

En la ventana que nos aparece, podemos darle nombre a nuestra clase, establecer si es virtual o si es una interfaz, y la lista desplegable Constraint nos permite establecer si dispone de algún tipo de restricción. En esa ventana, vemos además otras dos pestañas: Attributes y Operations. Desde esas pestañas añadimos o eliminamos los atributos y métodos a nuestras clases. El procedimiento es muy sencillo, primero escribimos el nombre y hacemos clic en el botón marcado con una V de color rojo, que nos añadirá el atributo a la lista y nos dejará rellenar el resto de datos al atributo. Podemos definirle el tipo de dato mediante una lista desplegable, su valor por defecto, sus restricciones y su nivel de acceso, entre otras opciones.

Para crear las operaciones, el método es muy similar, solo que en este caso podemos definir el valor de retorno y además tenemos una pestaña para añadir los parámetros que serán definidos del mismo modo que lo hicimos con los atributos.

4. TIPOS Y CAMPO DE APLICACIÓN

Existen diversos tipos de diagramas de comportamiento y, al igual que en el caso de los diagramas de clase del capítulo anterior, también son diagramas basados en el estándar unificado UML. Los diagramas de comportamiento son un modo de representar gráficamente los procesos y formas de uso de un programa, con ellos visualizamos los aspectos dinámicos de un sistema, como el flujo de mensajes a lo largo del tiempo, el movimiento físico de los componentes en una red o los diferentes estados y operaciones que transcurren en el ciclo de vida de un programa.

Si en los diagramas de clases veíamos cómo se definían y especificaban las diferentes clases, sus operaciones y relaciones, los diagramas de comportamiento nos dirán cómo interactúan a lo largo del tiempo dichas clases y operaciones.

Al tratarse de un tipo totalmente diferente de diagramas, no es necesario que lleven una relación directa con el diagrama de clases de un sistema, es más, podría no aparecer ningún nombre del diagrama de clases en un diagrama de comportamiento. No son diagramas estructurales, son diagramas conceptuales de manejo, de flujo y de la secuencia de un programa, por lo que es muy posible que dependiendo de la profundidad y extensión de un diagrama no aparezcan algunas de las entidades definidas en el diagrama de clases.

Cabe destacar que todos los diagramas de comportamiento se utilizan de un modojerárquico, es decir, en principio, sea el diagrama que sea, se realiza un diagrama sencillo con pocas "etapas", donde vemos el funcionamiento del sistema, separándolo en diferentes secciones que posteriormente tendrán un diagrama más detallado y específico. Dependiendo de las circunstancias que rodeen el desarrollo del diagrama, podría ser más práctico realizar un diagrama detallado de las pequeñas partes y luego unirlas en un diagrama más general, no obstante, esta práctica no es muy habitual, ya que se puede perder la coherencia del sistema al desarrollar las partes pequeñas sin que sean parte de un todo mientras se realizan.

Durante este capítulo iremos viendo los diferentes tipos de diagramas de comportamiento más relevantes, explicando cuál sería su correcto uso.

5. DIAGRAMAS DE ACTIVIDAD

Los diagramas de actividad son los diagramas de comportamiento más sencillos y fáciles de comprender. Representan los flujos de trabajo del sistema desde su inicio hasta el fin con las operaciones y componentes del sistema.

Este tipo de diagramas tienen un gran parecido con los clásicos diagramas de flujo que seguramente hayáis visto con anterioridad y con una notación muy similar. Los diagramas de actividad tienen unas características muy concretas y restrictivas, se componen de tres elementos: estados, transiciones y nodos.

Las reglas son muy sencillas, siempre debe haber un único estado inicial y un único estado final, todas las operaciones, transiciones y procesos ocurren entre esos dos puntos. Las transiciones se realizan entre estados y pueden tener nodos de por medio. Para aprender tanto su notación como su utilidad, vamos a mostrar un ejemplo sencillo en donde utilizamos todos los elementos de los que se puede componer nuestro diagrama de actividad.

Como vemos, en el ejemplo hemos utilizado un nodo inicial(representado por un punto), un nodo final(representado con un punto rodeado por una circunferencia), unas flechas de transición entre estados, un nodo de decisión (representado por un rombo) y dos barras de sincronización (representadas por una línea gruesa de color negro). Al igual que los nodos de bifurcación, las barras de sincronización pueden unir transiciones y separarlas, en este caso hemos usado los dos tipos de barras, ya que después de desayunar y leer el periódico debemos unir nuestro flujo de transición al realizarse de manera asíncrona. Para modelar estos diagramas podemos utilizar la herramienta de modelado de Visual Studio, esta herramienta dispone de una plantilla específica para los diagramas de actividad con sus herramientas personalizadas. Para *"Brear un diagrama de actividad, lo haremos como hicimos con el diagrama de clases, iremos a la entrada de menú Arquitectura > Nuevo Diagrama y elegiremos en este caso la plantilla Diagrama de actividades UML.

Veremos un cuadro de herramientas distinto al que conocemos donde se encuentran los elementos que vamos a utilizar en los diagramas de actividad.

Utilizaremos el elemento rotulado como Initial Node para poner el nodo inicial desde donde empezará nuestro flujo de trabajo, como vemos, tenemos también el nodo final rotulado como Activity Final Node que, como hemos visto, utilizaremos para definir el final de nuestra actividad.

El elemento Action dibuja nuestros estados, y luego tendríamos las bifurcaciones. Las bifurcaciones las tenemos de dos tipos, las que separan y las que unen, básicamente tendríamos un Decision node, que recibe una flecha de transición y pueden salir una o más de una, representando una decisión. Lo contrario ocurre con el Merge Node, donde recibe más de una transición y solo sale una transición de él.

6. DIAGRAMAS DE CASOS DE USO

Los diagramas de casos de uso representan cómo interactúan los diferentes actores en un sistema para cada caso de uso. Es decir, definen qué acciones puede realizar cada actor dentro de un sistema. Cada acción está representada de un modo muy simple por un rótulo que representa el caso de uso de la operación en cuestión.

Un modo de ver los casos de uso dentro de una aplicación serían los diferentes roles o permisos de los usuarios que tienen acceso a dicha aplicación, de este modo, dentro de un banco, los casos de uso no son los mismos (al menos no todos) si el usuario es el gerente o es un cajero.

Las relaciones que se pueden realizar entre casos de uso y los actores son realmente muy parecidas, son del mismo tipo pero restrictivas entre sí, es decir, una relación de extensión es una asociación de comunicación específica, al igual que todas las demás, por lo que podríamos generar un diagrama de casos de uso utilizando solamente la asociación de comunicación. En concreto, las relaciones de extensión e inclusión tienen una particularidad añadida: ese tipo de relaciones solo pueden darse entre casos de uso, es decir, es una interacción independiente del actor, que se produce por otro caso de uso.

Extensión: esta relación implica que un caso de uso puede extender a otro, es decir, que el comportamiento del caso extendido se utiliza en otro caso de uso.

Inclusión: similar a la extensión, esta relación implica que un caso de uso se incluye en otro, pero con una dirección determinada. Generalmente, este caso de uso suele provenir de un caso de uso de otro actor, mientras que el de extensión puede no venir de un actor concreto.

Generalización: es un modo de representar la herencia de casos de uso, es decir, un caso de uso puede ser genérico, pero tener formas más específicas del mismo (como el ejemplo que vimos de Formas y las diferentes formas: cuadrado, rectángulo y círculo).

Límite de un sistema: se utiliza para separar los diferentes sistemas dentro de un diagrama de casos de uso. En caso de que solo haya un sistema, no es necesario establecer el límite de un sistema.

Si utilizamos estos conceptos para modelar los casos de uso de una máquina expendedora de cafés, tendremos dos actores: el cliente que va a comprar a la máquina y la propia máquina, podemos especificar diversos casos de uso para cada uno de los actores que intervienen en el sistema. Sin duda, varios de los casos de uso estarán relacionados entre sí, sobre todo los que son propios de la máquina y los que son del cliente.

Intente realizar por su cuenta el diagrama de casos de uso de la máquina expendedora de café con los casos de uso “Escoger producto” y “Entregar producto”.

Visto de otro modo, la máquina necesita que se haya escogido el producto y la cantidad de azúcar para preparar el producto, y la máquina devuelve el cambio e imprime el recibo cuando entrega el producto.

En la herramienta de modelado de Visual Studio podríamos crear este mismo diagrama utilizando los elementos de su cuadro de herramientas. Tal y como hemos hecho en anteriores ocasiones, crearíamos un nuevo diagrama de casos de uso utilizando la plantilla disponible en el IDE añadiendo sus elementos al cuadro de dibujo de la herramienta.

7. DIAGRAMAS DE SECUENCIA

Los diagramas de secuencia modelan la secuencia lógica a través del tiempo de los mensajes entre instancias. Se podría definir como la pila de llamadas resultante de realizar las diferentes operaciones, forman un mapeado de la traza de llamadas que se realizan cuando un participante realiza una acción.

El diagrama de secuencia se estructura mediante líneas de vida, que a su vez representan a un participante en el sistema, ya sea un actor o cualquier otro elemento que participe en el transcurso secuencial de nuestro programa. La parte más importante del diagrama es la línea de vida, esa línea representa una secuencia, una cronología que nos permite visualizar con un rápido vistazo las interacciones, mensajes y participantes, así como el número de ellos a través de toda la vida del programa, desde que se ejecuta hasta que se cierra.

Como ya hemos comentado, en el diagrama de secuencia se visualizan los diferentes mensajes que se realizan entre los objetos, éstos pueden ser de dos tipos:

Síncronos: se corresponden con llamadas a métodos del objeto que recibe el mensaje. El objeto que envía el mensaje queda bloqueado hasta que termina la llamada. Este tipo de mensajes se representan con flechas con la cabeza llena.

Asíncronos: estos mensajes terminan inmediatamente, y crean un nuevo hilo de ejecución dentro de la secuencia. Se representan con flechas con la cabeza abierta. Al igual que los mensajes síncronos, también se representa la respuesta con una flecha discontinua.

Vemos en el ejemplo anterior la notación de las líneas de vida, los participantes y los hilos de ejecución. Cada participante debe tener su propia línea de vida y cada hilo de ejecución se representa como un rectángulo sombreado a lo largo de la línea de vida.

Este ejemplo es muy sencillo y básicamente hemos aprendido la notación y poco más, pero vayamos con algo más complejo y entendible conceptualmente, utilicemos el ejemplo de la máquina expendedora de los casos de uso.

Vemos en el ejemplo anterior la notación de las líneas de vida, los participantes y los hilos de ejecución. Cada participante debe tener su propia línea de vida y cada hilo de ejecución se representa como un rectángulo sombreado a lo largo de la línea de vida.

Este ejemplo es muy sencillo y básicamente hemos aprendido la notación y poco más, pero vayamos con algo más complejo y entendible conceptualmente, utilicemos el ejemplo de la máquina expendedora de los casos de uso.

7.1. INGENIERÍA INVERSA

Al igual que con los diagramas de clases, podemos conseguir el diagrama de secuencia de un método concreto de manera automática mediante técnicas de ingeniería inversa disponibles en la herramienta de modelado de Visual Studio, para ello, vamosa realizar una sencilla aplicación que añade productos a un carrito de la compra. Para realizar nuestra estructura de clases, vamos a utilizar el mismo método que vimos en el Apartado 5.3.1 y crearemos las clases desde la herramienta de modelado, tendremos dos clases, Carrito y Producto, primero crearemos nuestro proyecto (una aplicación de consola por ejemplo) y crearemos las clases desde el diagrama, donde los atributos de Carrito serían num Pedido y los atributos de Producto serían nombre y precio.

Ahora añadiremos los métodos y funcionalidades que necesitamos para que sea funcional, añadiremos un contador en nuestro método main para llevar un contador de los carritos de compra que creemos y crearemos los métodos CrearCarrito, CrearProducto y Listar Productos.

No hay que olvidar que una vez creados y encapsulados los campos, y creada nuestra colección de asociación, deberemos eliminar el lanzamiento de la excepción NotImplementedException y deberemos inicializar la colección en el constructor de la clase Carrito para que la propiedad Productos sea una lista vacía y no una referencia nula.

Ejemplo

CREARCARRITO

```
private static Carrito CrearCarrito (int numPedido) {
    Carrito carrito = new Carrito ();
    carrito.NumPedido = num Pedido;
    return carrito;
}
```

Ejemplo

CREARPRODUCTO

```
private Static Producto CrearProducto (String nombre, double precio)
{
    Producto producto = new Producto ();
    producto.Nombre = nombre;
    producto.Precio = precio;
    return producto;
}
```

Ejemplo

LISTARPRODUCTOS

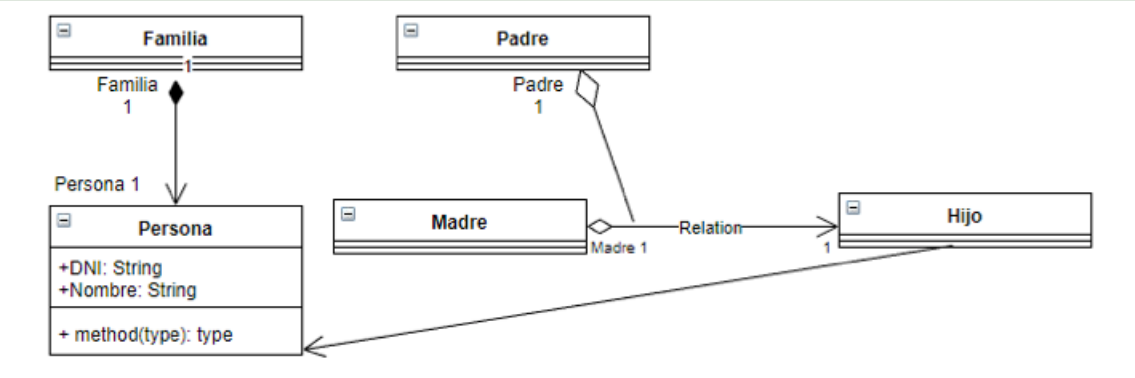
```
private static string Listar Productos (Carrito carrita)
{
string resultado = "Productos del carrito con N° de pedido: " + carrito.NumPedido + "\n";
resultado + = carrito.Productos.Select (p =» p.Nombre + " Precio: " + p.Precio.ToString ()). Aggregate
( (a,b) => a+"\n"+b);
return resultado;
}
```

Con nuestra aplicación funcionando correctamente, ha llegado el momento de obtener nuestro diagrama de secuencia. Podemos crear el diagrama de secuencia de cualquier método de nuestro código, para observar las diferencias en cada método, vamos a generar un diagrama por cada uno de los métodos implementados en nuestra clase principal excluyendo el método main. Lo único que tenemos que hacer es botón derecho en cualquier método y seleccionar la opción Generar diagrama de secuencia, una vez seleccionada, nos saldrá un cuadro de diálogo con las opciones de nuestro diagrama de secuencia, el cual dejaremos con los valores por defecto.

Actividad

3.1 Nuestra libreta se complica, ahora se quiere poder añadir grupos de contactos. Los grupos de contactos contendrán contactos i la libreta contendrá tanto grupo de contactos como contactos. Los contactos, además, no necesitan estar en un grupo de contactos para estar en la libreta. Dibuje y represente los cambios mencionados, usando como base el diagrama de clases de ejemplo de Libreta.

3.2 Intenta por tu cuenta realizar el diagrama de los que se ha explicado de las relaciones familiares antes de proseguir.



3.3 Antes de continuar, razona sobre la relación entre operarios y trabajos. ¿Un operario tendrá muchos trabajos para realizar o un trabajo tendrá muchos operarios que lo pueden realizar? Es decir, tenemos dos opciones, que se a el operario quien tenga una lista e operarios, o el trabajo el que tenga una lista de operarios que pueden realizarlo. ¿Qué diferencias habría? ¿Hay alguna otra opción?

3.4 Realiza diagramas de clases de los proyectos anteriores en los que hemos trabajado sin utilizar ingeniería inversa con cualquiera de las herramientas propuestas.

3.5 Realiza ahora ingeniería inversa en dichos proyectos, compara los resultados, analiza las diferencias que encuentre.

3.6 Mediante la opción mostrar base clase saca toda la jerarquía de clases de un formulario WinForms.

Índice

Objetivos 5
1. INTRODUCCIÓN A UML 7
2. DISEÑO DE CLASES EN UML 8
2.1. CLASES, ATRIBUTOS Y MÉTODOS 8
2.1.1. Notación 8
2.2. RELACIONES 9
2.2.1. Asociación 9
2.2.2. Composición 9
2.2.3. Herencia 10
3. HERRAMIENTAS 11
3.1. HERRAMIENTAS DE MODELADO DE VS 11
3.1.1. Generar código a partir de diagramas de clases 11
3.1.2. Ingeniería inversa 13
3.2. UMLPAD 14
4. TIPOS Y CAMPO DE APLICACIÓN 15
5. DIAGRAMAS DE ACTIVIDAD 16
6. DIAGRAMAS DE CASOS DE USO 17
7. DIAGRAMAS DE SECUENCIA 18
7.1. INGENIERÍA INVERSA 18

Índice 21