

DESARROLLO DE APLICACIONES MULTIPLATAFORMA

DESARROLLO DE APLICACIONES WEB

Entornos de desarrollo

José Manuel Piñeiro Gómez



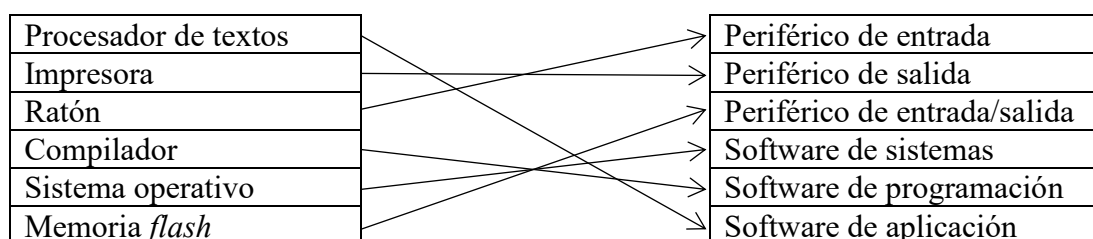
Paraninfo
ciclos formativos

Entornos de desarrollo

Solucionario Unidad 1 Desarrollo de software

Actividades propuestas

Actividad propuesta 1.1



Actividad propuesta 1.2

Todos los lenguajes indicados, excepto Pascal, son orientados a objetos. El lenguaje Pascal es un lenguaje estructurado, pero a partir de él se creó Object Pascal, que permite realizar programación orientada a objetos.

Actividad propuesta 1.3

En el modelo en espiral se va construyendo el software en base a incrementos o versiones cada vez más completas. Pues bien, si existe incertidumbre en relación con los requisitos, lo que se puede detectar en las primeras iteraciones en el cuadrante correspondiente al análisis de riesgo del modelo en espiral, se puede crear un prototipo en el cuadrante de ingeniería, dentro del cual se llevarían a cabo las tareas de modelado, construcción y despliegue del prototipo. La tarea de planificación del prototipo se correspondería con el cuadrante de planificación del modelo en espiral y la evaluación del prototipo y comunicación con el cliente, con el cuadrante de evaluación del modelo en espiral.

En caso de combinar estos dos modelos, dado que en el modelo en espiral se van construyendo versiones del software cada vez más completas, lo habitual es que el prototipo no se deseché y evolucione, siendo este ampliado y mejorado hasta convertirse en el producto final que se entrega al cliente.

ACTIVIDADES FINALES

Actividades de comprobación

1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	1.10	1.11	1.12
d)	c)	d)	b)	d)	b)	c)	a)	b)	d)	a)	c)

Actividades de aplicación

Actividad de aplicación 1.13

Su función es ejecutar las instrucciones contenidas en los programas, las cuales se encontrarán en memoria principal, junto con los datos necesarios.

Actividad de aplicación 1.14

Lenguajes de bajo nivel o lenguajes máquina, lenguajes ensambladores y lenguajes de alto nivel o evolucionados. Estos últimos son los lenguajes más empleados en la actualidad por su mayor facilidad de uso dada su mayor cercanía al lenguaje natural; además, permiten realizar tareas complejas en poco tiempo debido a que disponen de instrucciones potentes.

Actividad de aplicación 1.15

En el paradigma estructurado se presta más atención a los procesos que a los datos, mientras que en el orientado a objetos, es al revés.

En el paradigma estructurado una aplicación está formada por un conjunto de módulos de manera que desde unos se llaman a otros. En el paradigma orientado a objetos, una aplicación consta de un conjunto de objetos que tienen una serie de datos o atributos y un comportamiento definido por una serie de operaciones que puede realizar. Una aplicación está formada por un conjunto de objetos que se envían mensajes entre ellos.

Actividad de aplicación 1.16

Consiste en un conjunto de instrucciones escritas siguiendo las normas de un lenguaje de programación de alto nivel. Debido a que este lenguaje no es entendido directamente por el ordenador, se debe someter a este código a un proceso de traducción.

Actividad de aplicación 1.17

Sirven para convertir o traducir el código fuente de un programa en código objeto o ejecutable, según el caso. Se diferencian en la manera en que llevan a cabo el proceso de traducción: mientras

que los compiladores en un único proceso toman todo el código fuente y, si este es correcto, generan el código objeto correspondiente y lo almacenan, los intérpretes van traduciendo partes del código fuente y lo van ejecutando.

Actividad de aplicación 1.18

Existen las máquinas virtuales de sistema y las de proceso. Las primeras simulan a un ordenador completo de forma que se puede instalar en su interior otro sistema operativo y tiene los componentes de un ordenador como disco duro, memoria, etc. Las segundas no simulan a un ordenador, sino que ejecutan solo un proceso concreto que permite que un programa se ejecute de la misma manera en cualquier ordenador.

La máquina virtual de *Java* es una máquina virtual de proceso.

Actividad de aplicación 1.19

Su función es traducir los programas escritos en el lenguaje llamado *bytecode* que genera el compilador de *Java* en código binario ejecutable por el ordenador. Su existencia hace posible que un programa escrito en *Java* se pueda ejecutar en cualquier plataforma porque el programa no es realmente ejecutado por el ordenador, sino por la máquina virtual de *Java*.

Actividad de aplicación 1.20

El objetivo de esta disciplina es regular de alguna manera el desarrollo de *software*, de forma que en vez de desarrollar el *software* de manera artesanal, se establezcan de manera clara las tareas que es necesario llevar a cabo para llegar a obtener *software* fiable, rentable y que funcione de manera eficiente sobre máquinas reales.

Actividad de aplicación 1.21

Sirve para descubrir errores en el *software* antes de que este sea entregado al cliente, es decir, para garantizar su calidad. Como consecuencia de las pruebas puede ser necesario modificar el código o tareas previas de desarrollo, como análisis y diseño.

Actividad de aplicación 1.22

El modelo más antiguo es el modelo en cascada o ciclo de vida clásico, según el cual se concibe el proceso de desarrollo como un proceso secuencial en el que se llevan a cabo las siguientes tareas en este orden: análisis, diseño, programación, pruebas y mantenimiento. No obstante, puede ser necesario que en una fase se detecten fallos, como consecuencia de los cuales sea necesario realizar modificaciones sobre fases previas. Cuanto más tarde se detecten errores en etapas iniciales, más costosa será su corrección, porque será necesario corregir errores en todas las fases secuenciales hasta aquella en la que nos encontramos.

Actividad de aplicación 1.23

Cuando no es sencillo conocer todos los requisitos que debe cumplir el *software* o no se sabe qué forma debe adoptar la interfaz hombre-máquina o cuando hay dudas técnicas sobre la eficiencia de un algoritmo o la adaptabilidad de un sistema operativo, por ejemplo.

Actividad de aplicación 1.24

Una metodología incluye una serie de métodos porque dentro de una metodología se deben realizar una serie de tareas, para cada una de las cuales se debe aplicar un procedimiento para el cual se pueden emplear uno o varios métodos o técnicas.

Actividad de aplicación 1.25

Se trata de un elemento de información producido, modificado o usado en la metodología para el desarrollo de una aplicación. Las actividades pueden tener productos intermedios de entrada y de salida.

Actividad de aplicación 1.26

Se puede definir como el periodo de tiempo durante el cual se realiza el trabajo necesario para realizar una entrega al cliente.

Actividades de ampliación**Actividad de ampliación 1.27**

Hay disponible información sobre esta metodología en diversas webs, entre ellas https://administracionelectronica.gob.es/pae_Home/pae_Documentacion/pae_Metodolog/pae_Metrica_v3.html.

Se trata de una metodología empleada en la Administración Pública española. Como se puede leer en el documento de introducción, accesible desde la web indicada, los tres procesos principales son:

- Planificación de sistemas de información.
- Desarrollo de sistemas de información.
- Mantenimiento de sistemas de información.

Actividad de ampliación 1.28

Se puede encontrar información sobre el concepto de ingeniería inversa en <http://www.iiisci.org/journal/CV%24/risci/pdfs/X581YP.pdf>. Como se indica en este artículo, la ingeniería inversa en general consiste en descubrir los principios tecnológicos de un dispositivo, un objeto o un sistema mediante el análisis de su estructura, funcionamiento u operación. Aplicado al *software*, consiste en un proceso que recorre hacia atrás el ciclo de desarrollo del *software* partiendo del código fuente y llegando hasta la fase de análisis. El objetivo es construir documentación de la aplicación que no existe o es inadecuada para facilitar su mantenimiento.

Actividad de ampliación 1.29

Se puede encontrar información sobre el concepto de reingeniería en <http://www.iiisci.org/journal/CV%24/risci/pdfs/X581YP.pdf>. Como se indica en este artículo, la reingeniería tiene por objetivo entender una aplicación existente a nivel de análisis, diseño e implementación con el fin de hacer una reimplementación para aumentar y/o mejorar la funcionalidad del sistema, su desempeño o la propia implementación. Podemos deducir, por tanto, que la reingeniería es una tarea más ambiciosa que la ingeniería inversa en el sentido de que el objetivo del recorrido hacia atrás en el ciclo de desarrollo de *software* no es solo obtener una documentación inexistente o inadecuada, sino además realizar mejoras en la aplicación.

Actividad de ampliación 1.30

En la web <https://sites.google.com/site/adai6jfm/home/metodologas-de-desarrollo>, se indica que las características deseables de una metodología son las siguientes:

- Existencia de reglas predefinidas.
- Cobertura total del ciclo de desarrollo.
- Posibilidad de realizar verificaciones intermedias sobre los productos generados en cada fase.
- Planificación y control.
- Comunicación efectiva entre los desarrolladores y entre estos y los usuarios.
- Utilización sobre un amplio abanico de proyectos.
- Fácil formación.
- Debe estar soportada por herramientas CASE que mejoren la productividad de los desarrolladores y la calidad de los productos.
- Debe contener actividades que mejoren el proceso de desarrollo.
- Soporte al mantenimiento.
- Soporte a la reutilización del *software*.

Actividad de ampliación 1.31

En la web <https://sites.google.com/site/desarrollodesoftwareuba/reutilizacion> se indica que la reutilización de *software* consiste en el empleo de elementos de *software* u otros de nivel superior creados en desarrollos anteriores, para de este modo reducir los tiempos y simplificar el desarrollo de *software*, mejorando la calidad y reduciendo su coste.

Es importante fomentar la reutilización por las ventajas que aporta:

- Reducir tiempos de desarrollo.
- Reducir costes.
- Incrementar la productividad.
- Facilitar la compartición de productos del ciclo de vida.
- Mayor fiabilidad.
- Mayor eficiencia.
- Consistencia y familiaridad, lo que facilita el mantenimiento.

Actividad de ampliación 1.32

Según se indica en la web [https://en.wikipedia.org/wiki/V-Model_\(software_development\)](https://en.wikipedia.org/wiki/V-Model_(software_development)), el modelo en V es una extensión del modelo de desarrollo de *software* en cascada en el que las tareas de pruebas (posteriores a las de programación) se colocan hacia arriba generando una V. De esta manera, se muestra la relación entre cada fase del ciclo de vida con su fase de pruebas asociada. Las fases de análisis y de diseño se dividen en dos pues a cada una de las dos fases le corresponde un tipo diferente de pruebas. Se muestra a continuación una representación de este modelo:

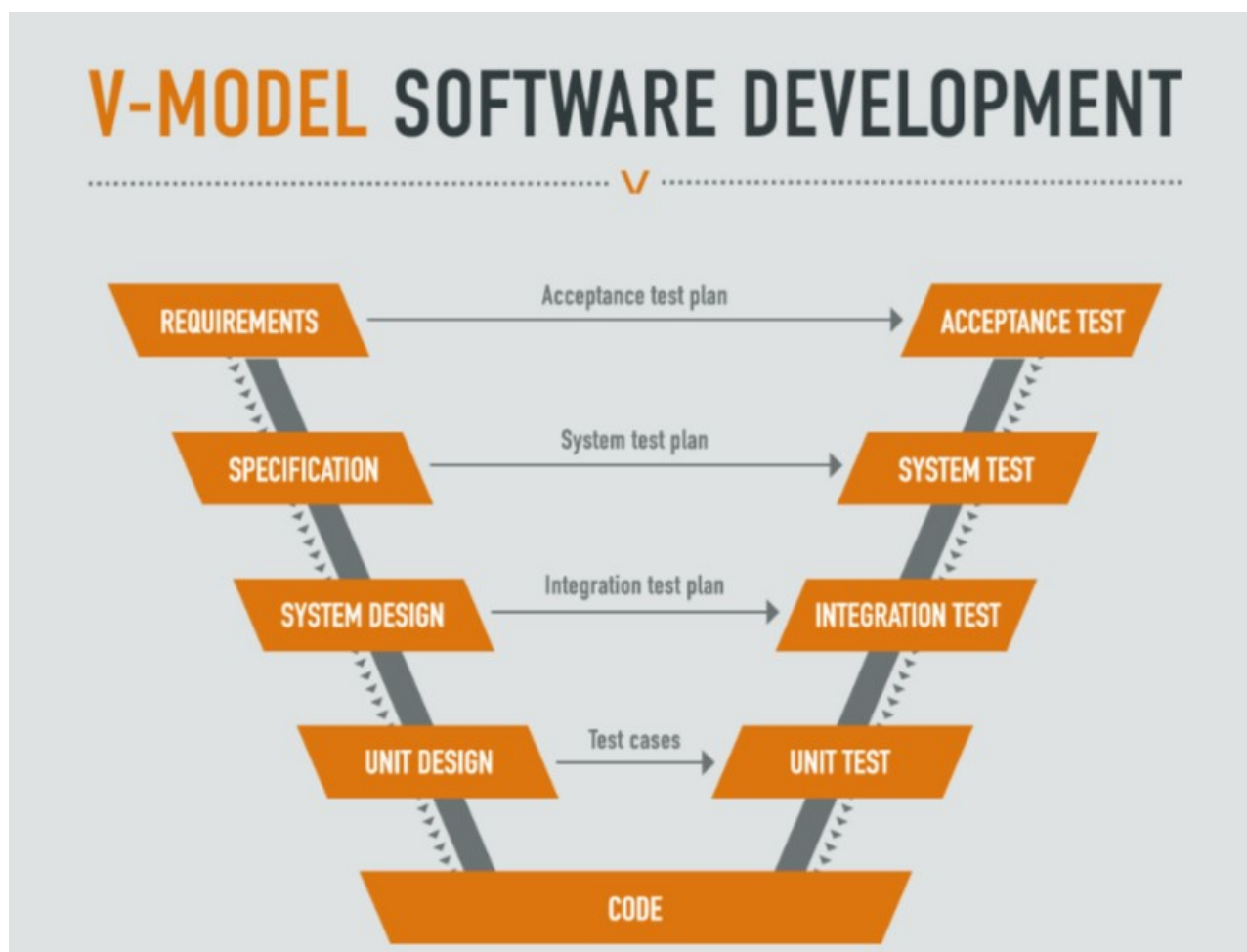


Figura 1.1. Representación del modelo en V, en el que las fases posteriores a la programación (pruebas) se dividen en varias y se hace corresponder con etapas del ciclo de vida en cascada, si bien las etapas de análisis y diseño se dividen en otras dos.

Entornos de desarrollo

Solucionario Unidad 2

Instalación y uso de entornos de desarrollo

Actividades propuestas

Actividad propuesta 2.1

Para instalar el módulo, se selecciona en Eclipse la opción de menú *Help* → *Eclipse Marketplace* y, en la ventana que aparece, se escribe Python en el cuadro de texto que hay a la derecha de la palabra *Find*. Entonces, se pulsa el botón *Go* a la derecha y nos aparecen varios módulos para programar en Python. Se puede elegir el módulo *PyDev – Python IDE for Eclipse 9.2.0*, como se muestra en la Figura 2.1, para lo que se debe clicar sobre el botón *Install*.

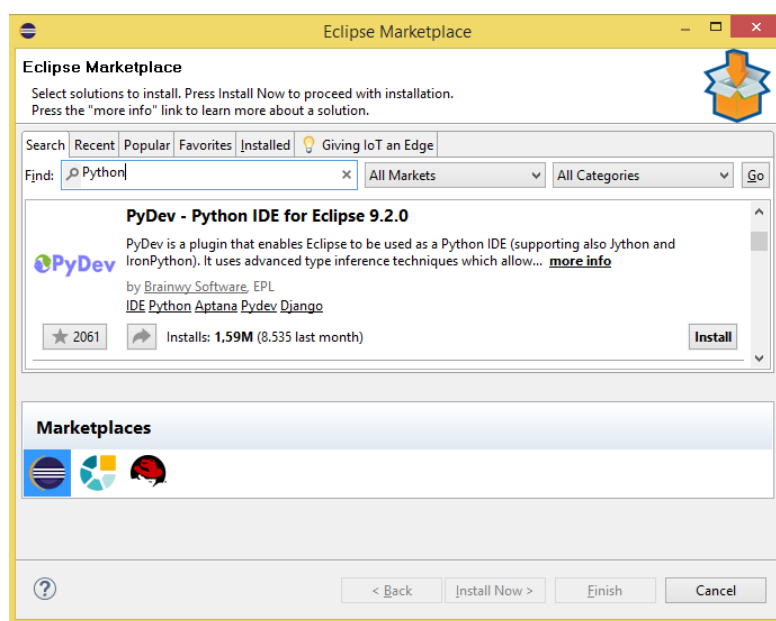


Figura 2.1. En la función *Eclipse Marketplace*, se muestra el módulo *Pydev – Python IDE for Eclipse 9.2.0* preparado para su instalación.

Se procederá a la instalación al hacer clic en el botón *Install*, tras lo cual habrá que reiniciar el IDE para que la instalación tenga efecto.

Actividad propuesta 2.2

Para desinstalar alguno de los elementos opcionales del módulo *PyDev – Python IDE for Eclipse 9.2.0*, en primer lugar, se selecciona en Eclipse la opción de menú *Help* → *Eclipse Marketplace* y se hace clic en la pestaña *Installed*, donde se muestran los módulos instalados. Entonces, se busca el

módulo *PyDev – Python IDE for Eclipse 9.2.0* y se hace clic en el botón *Change* y en la opción *Change*, como se muestra en la Figura 2.2.

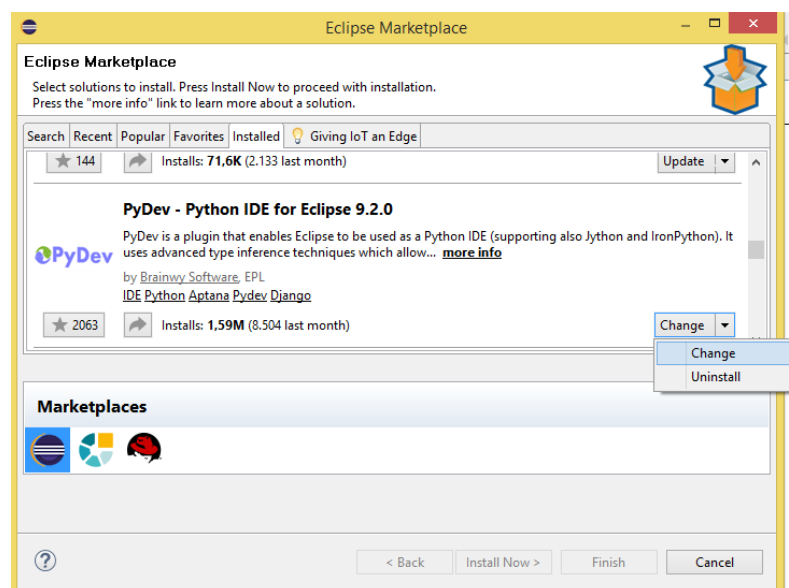


Figura 2.2. En la función *Eclipse Marketplace*, se muestra el módulo *Pydev – Python IDE for Eclipse 9.2.0* en la pestaña *Installed*, desde donde se pueden cambiar los elementos instalados seleccionando la opción *Change* para el botón *Change*.

En la siguiente ventana se pueden ver los elementos del módulo para que deseccionemos aquellos que se quiere desinstalar. Para los que se elija desinstalar, aparecerá una cruz de color rojo, como se puede ver en la Figura 2.3.

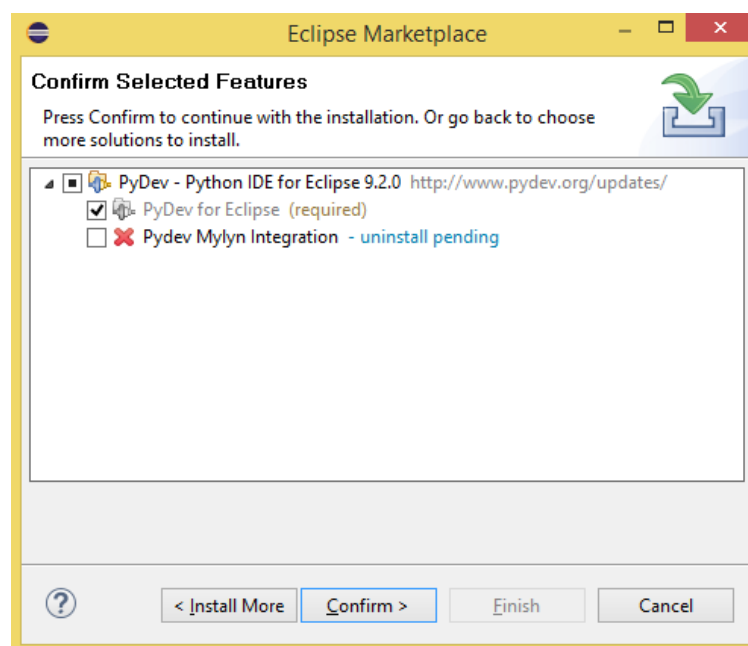


Figura 2.3. Se muestran los elementos instalados del módulo *Pydev – Python IDE for Eclipse 9.2.0* en la pestaña, desde donde se pueden deseccionar los que se desean desinstalar.

Una vez deseleccionados, se debe hacer clic en el botón *Confirm* para que se proceda a la desinstalación del elemento o de los elementos indicados.

Actividad propuesta 2.3

Para desinstalar el módulo *PyDev – Python IDE for Eclipse 9.2.0*, en primer lugar, se selecciona en Eclipse la opción de menú *Help → Eclipse Marketplace* y se hace clic en la pestaña *Installed*, donde se muestran los módulos instalados. Entonces se busca el módulo *PyDev – Python IDE for Eclipse 9.2.0* y se hace clic en el botón *Change* y en la opción *Uninstall*, como se muestra en la Figura 2.4.

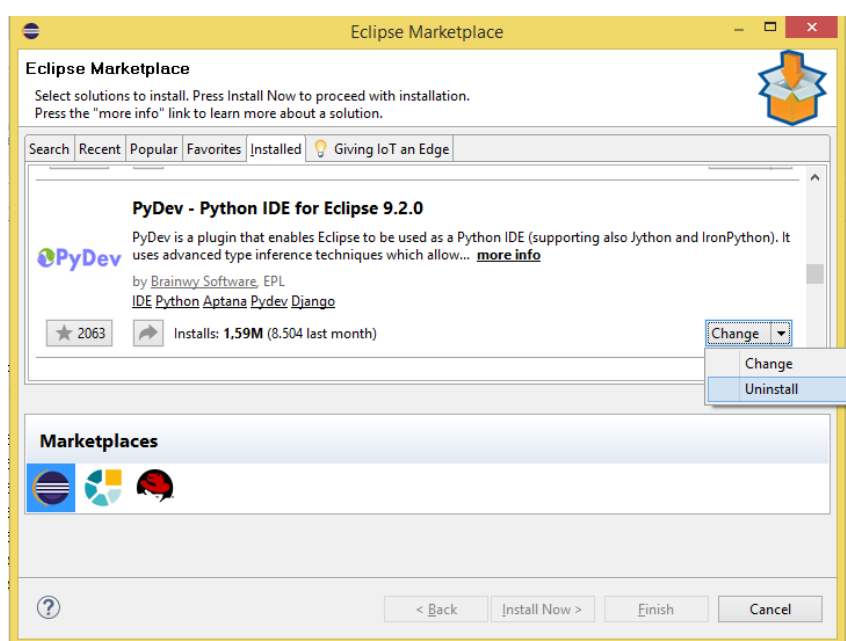


Figura 2.4. En la función *Eclipse Marketplace*, se muestra el módulo *Pydev – Python IDE for Eclipse 9.2.0* en la pestaña *Installed*, desde donde se puede desinstalar seleccionando la opción *Uninstall* para el botón *Change*.

En la siguiente ventana se pueden ver los elementos del módulo y desmarcar aquellos que se desea desinstalar. Si se desmarcan todos y se hace clic en botón *Confirm*, se procederá a la desinstalación.

ACTIVIDADES FINALES

Actividades de comprobación

2.1	2.2	2.3	2.4	2.5	2.6	2.7	2.8	2.9	2.10
a)	b)	d)	a)	a)	c)	a)	b)	d)	b)

Actividades de aplicación

Actividad de aplicación 2.11

En lugar de tener que ejecutar las diferentes herramientas implicadas en la creación y ejecución de programas (editor, compilador, etc.) de manera independiente, están todas integradas en el *IDE*, lo que agiliza la tarea de creación de programas.

Actividad de aplicación 2.12

El editor de un entorno de desarrollo proporciona ayudas para la creación de programas, ya que marca en diferentes colores diferentes elementos del programa, como palabras reservadas, variables, etc., señala los errores sintácticos, proporciona sugerencias a medida que se va escribiendo el programa, etc. Todo esto facilita la edición de programas y minimiza los errores de escritura.

Actividad de aplicación 2.13

Es necesario instalar tanto el *JDK*, como el *JRE* de *Java*, aunque el *JDK* tiene incorporado el *JRE*. El *JDK* (*Java Development Environment*) proporciona herramientas para la creación de programas en *Java*, como el compilador *javac*, mientras que el *JRE* (*Java Runtime Environment*) está formado por la máquina virtual de *Java* (*JVM*), un conjunto de bibliotecas de *Java* y otros componentes necesarios para poder ejecutar programas escritos en el lenguaje *Java*.

Actividad de aplicación 2.14

Los programas fuente en *Java* tienen extensión *.java* y los archivos resultado de la compilación tienen extensión *.class*.

Actividad de aplicación 2.15

Consiste en un componente que permite crear aplicaciones con una interfaz gráfica de usuario (*GUI*) que permite la creación de pantallas con distintos tipos de controles (etiquetas, cuadros de

texto, botones, etc.), con los que puede interactuar el usuario mediante el ratón o el teclado. Este componente lo incorpora *NetBeans* tras su instalación, pero no *Eclipse*, en cuyo caso es necesario instalar un módulo o *plug-in* para crear este tipo de aplicaciones.

Actividad de aplicación 2.16

En la consola de *Java*, que se muestra en la parte inferior derecha de la pantalla.

En la zona de proyectos o explorador de paquetes, que se muestra en la parte izquierda de la pantalla.

Actividad de aplicación 2.17

Los botones de opción

Los cuadros de contraseña

Actividad de aplicación 2.18

Sirve para ayudar al programador en la tarea de depuración, consistente en la localización y corrección de errores contenidos en los programas. El depurador permite detener la ejecución de un programa en una determinada instrucción, ejecutar el programa instrucción a instrucción, consultar los valores de las variables a lo largo de la ejecución, etc.

Actividad de aplicación 2.19

Permiten realizar tareas que no vienen incorporadas en el entorno de desarrollo instalado.

Actividad de aplicación 2.20

En *Eclipse* se debe elegir la opción de menú *Help – Eclipse Marketplace*, buscar el módulo escribiendo en la casilla *Find* su nombre y haciendo *clic* en el botón *Go*. Se busca el módulo deseado entre los que se muestran y se hace *clic* en el botón *Install*.

En *NetBeans* se debe elegir la opción de menú *Tools – Plugins*, buscar un módulo en la pestaña *Available Plugins* y una vez encontrado, seleccionarlo y hacer *clic* en el botón *Install*.

Actividad de aplicación 2.21

Sí es posible. Para hacerlo, es necesario elegir la opción de menú *Help – Eclipse Marketplace*, buscar el módulo en cuestión en la pestaña *Installed*, hacer *clic* en el botón *Change* y en la siguiente pantalla, donde se muestran los componentes del módulo, seleccionar o deseleccionar los que se deseen y hacer *clic* en el botón *Confirm*.

Actividad de aplicación 2.22

En *Java*, *C*, *C++*, *PHP*, *Javascript* y, aunque se trata de un lenguaje de marcas más que un lenguaje de programación, en *HTML5*.

Actividades de ampliación

Actividad de ampliación 2.23

En el proyecto creado en el epígrafe 2.5.1 en la vista diseño se selecciona, en primer lugar el *layout absolute* para colocar los controles de manera libre en la pantalla. Para ello, se hace doble *clic* en la paleta en la sección *Layouts* en *Absolute*. A continuación ya se puede ir añadiendo cada control. Por ejemplo, para añadir un control de tipo etiqueta, se hace clic en la sección *Components* de la paleta y en *JLabel*, como se muestra en la imagen y se arrastra hasta la ventana. Entonces nos aparecerá una etiqueta con el nombre *New Label*, que debemos cambiar.

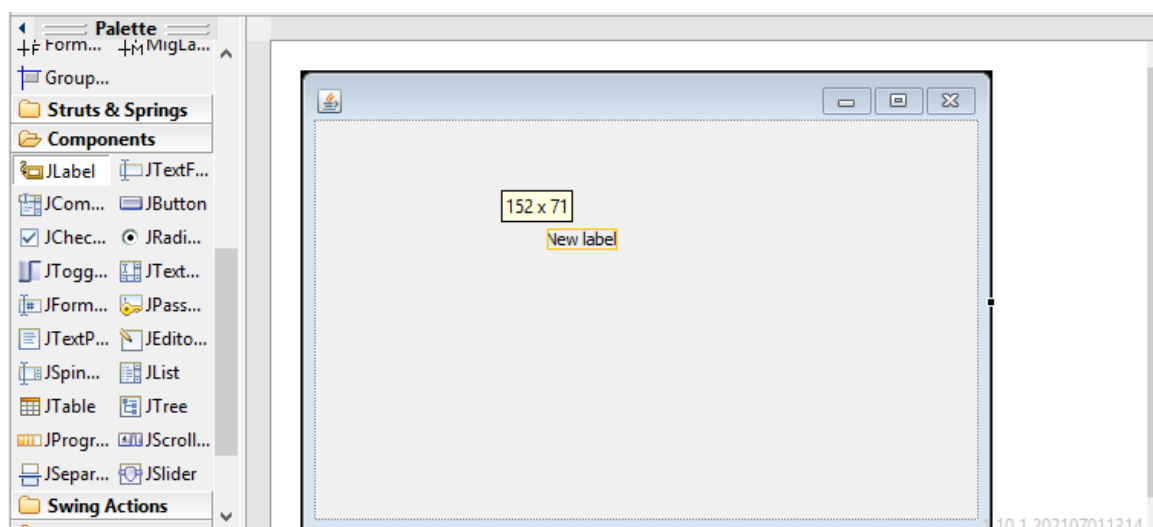


Figura 2.5. Se refleja la colocación del primer control de tipo etiqueta en la ventana. Se debe cambiar el nombre al control.

Para cambiar el nombre a cada control o componente hay que colocarse sobre él y modificar en la paleta de propiedades (en la parte izquierda de la pantalla) el nombre del componente, que aparece a la derecha de la palabra “Variable”. *Eclipse* asigna a cada componente un nombre genérico que comienza con tres letras que identifican el tipo de componente. Así, las etiquetas (componente *JLabel*) comienzan con “lbl” y los botones de opción (*JRadioButton*) con “rdb”. Pues bien, se deben asignar a los controles nombres significativos, por ejemplo, *lblNombre*, *txtNombre*, *lblContraseña*, etcétera.

También es necesario cambiar el texto visible de las etiquetas para que ponga los textos deseados (“Nombre” en la primera etiqueta, por ejemplo). Para ello, en la paleta de propiedades, habrá que cambiar el valor de la propiedad *text*.

Para la elección del idioma, debemos seleccionar el tipo de control *JComboBox*. Tras cambiarle el nombre por *cmbIdioma*, en la paleta en la propiedad *model* escribiremos los valores posibles, como se muestra en la Figura 2.6.

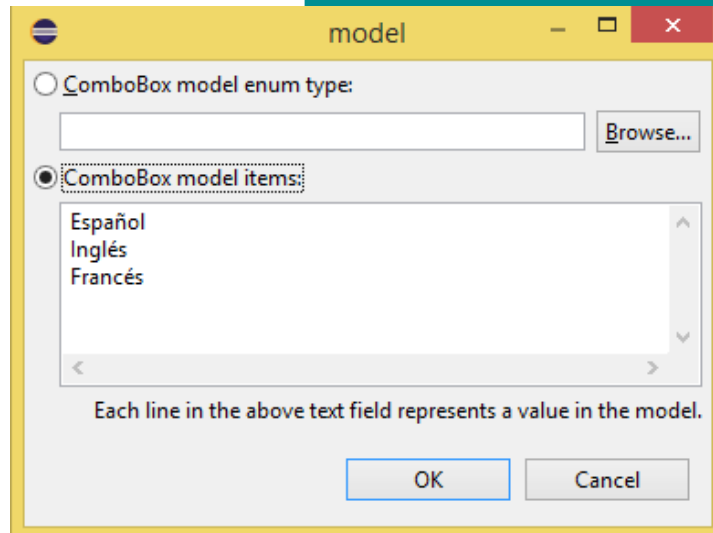


Figura 2.6. En la propiedad *model* de un control de tipo *JComboBox* se pueden indicar los valores posibles.

Se deben ir colocando todos los controles y cambiándoles su nombre y la propiedad *text* en la sección *Components* de la paleta, como se ha indicado.

Actividad de ampliación 2.24

Se puede encontrar información sobre el entorno de desarrollo *BlueJ* en <https://es.wikipedia.org/wiki/BlueJ> y en la página web oficial de este entorno: <https://bluej.org/>. Como se indica en *Wikipedia*, *BlueJ* está pensado para ayudar a aprender a programar empleando el paradigma orientado a objetos. Debido a esto, difiere de otros entornos en que es más sencillo en cuanto a su interfaz y es más fácil de utilizar. Está pensado para el desarrollo de aplicaciones sencillas y para un uso educativo. Se puede utilizar solo el lenguaje *Java*.

Actividad de ampliación 2.25

Se puede encontrar información sobre el entorno de desarrollo *Microsoft Visual Studio* en https://es.wikipedia.org/wiki/Microsoft_Visual_Studio y en la página web oficial de este entorno: <https://visualstudio.microsoft.com/es/vs/>. Es un *IDE* para *Windows* y *MacOS* y se pueden emplear varios lenguajes de programación, como *C++*, *C#*, *Visual Basic.NET*, *F#*, *Java*, *Phyton*, *Ruby* y *PHP*. Se trata de un *software* de código cerrado propiedad de *Microsoft* y, aunque las versiones completas son de pago, ofrece versiones limitadas para determinados lenguajes de programación orientadas a estudiantes y programadores aficionados, que carecen de ciertas características avanzadas.

Actividad de ampliación 2.26

Se puede encontrar información sobre el entorno de desarrollo *Oracle Developer* en <https://es.wikipedia.org/wiki/JDeveloper> y en la página web oficial de este entorno: <https://www.oracle.com/application-development/technologies/jdeveloper.html>. Se pueden emplear lenguajes de programación del entorno de *Oracle*: *Java*, *HTML*, *XML*, *SQL*, *PL/SQL*, *Javascript*, *PHP*, *Oracle ADF*, y otros. Es *software* de código cerrado, pero es gratuito.

Actividad de ampliación 2.27

Se muestra la misma información que aparece en *Eclipse* haciendo clic en la opción de menú *Help – Eclipse Marketplace*, pero en forma de página web, por lo que se puede visualizar la información mejor. Se muestran, por tanto, todos los módulos o *plug-ins* que se pueden instalar en *Eclipse*. Al hacer clic en uno de ellos, se puede ver información detallada sobre él y en la pestaña *External Install Button*, hay un botón *Install* de manera que si arrastramos este botón a nuestro *Eclipse* abierto, se podrá proceder a la instalación del módulo.

Actividad de ampliación 2.28

Para crear una aplicación con interfaz gráfica de usuario en *NetBeans* podemos seleccionar en un proyecto creado la opción del menú contextual de un proyecto o un paquete *New – JPanel Form*. Nos aparecerá entonces una pantalla como la de la siguiente figura, con una ventana para colocar los controles que deseemos y, a la derecha, una paleta, muy similar a la de *Eclipse*:

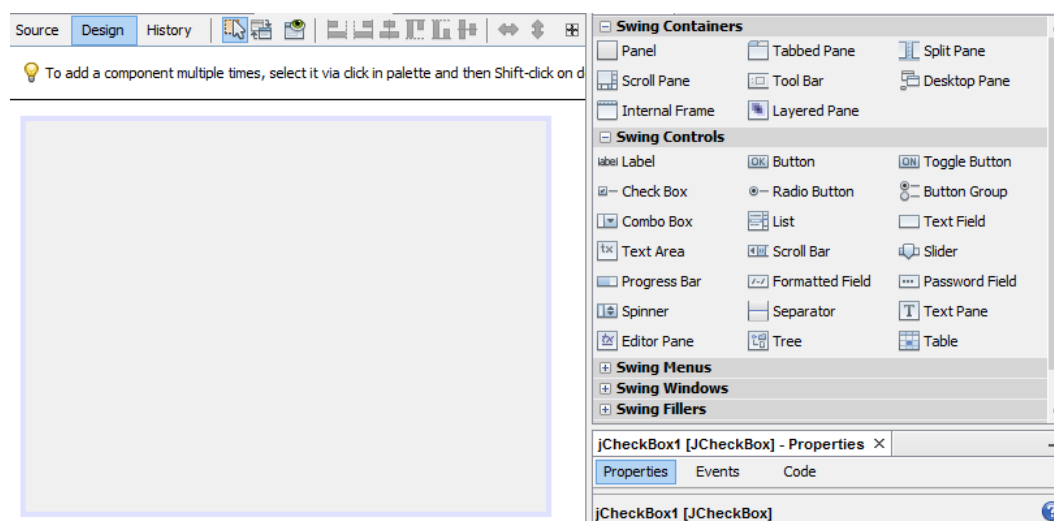


Figura 2.7. Se muestra en la parte izquierda la pantalla en la que se pueden colocar los controles y una paleta con los controles disponibles.

A continuación, habrá que llevar a cabo las mismas tareas que en *Eclipse*, esto es, colocar los controles deseados en la ventana, cambiarles el nombre y la propiedad *text* hasta construir una ventana como la solicitada. Las propiedades de cada control se pueden ver y modificar debajo de la paleta en la pestaña *Properties*.

Entornos de desarrollo

Solucionario Unidad 3 Diseño y realización de pruebas

Actividades propuestas

Actividad propuesta 3.1

Para realizar el grafo de flujo lo primero que hay que hacer es asignar un número único a cada sentencia o grupo de sentencias y a cada condición. Se muestra la asignación a continuación:

```
public static void main(String[] args) {  
    int num, divisor, sumadivisores;  
    divisor = 1;  
    sumadivisores = 0;  
    Scanner entrada = new Scanner(System.in);  
    System.out.print ("Introduzca un número mayor que 0: ");  
    num = entrada.nextInt();  
    while (divisor <= num/2) {  
        if (num % divisor == 0) {  
            sumadivisores = sumadivisores + divisor;  
            divisor++;  
        }  
        if (num == sumadivisores) {  
            System.out.println ("El número " + num + " es un número perfecto");  
        }  
        else {  
            System.out.println ("El número " + num + " no es un número perfecto");  
        }  
    }  
}
```

Diagrama de flujo de la asignación de números únicos:

- 1: Grupo de sentencias de inicialización y lectura de entrada.
- 2: Condición del bucle while.
- 3: Condición del bucle if.
- 4: Sentencia de suma.
- 5: Sentencia de incremento.
- 6: Condición del bucle if.
- 7: Sentencia de impresión.
- 8: Sentencia de impresión.
- 9: Fin del programa.

Ahora construimos el grafo de flujo:

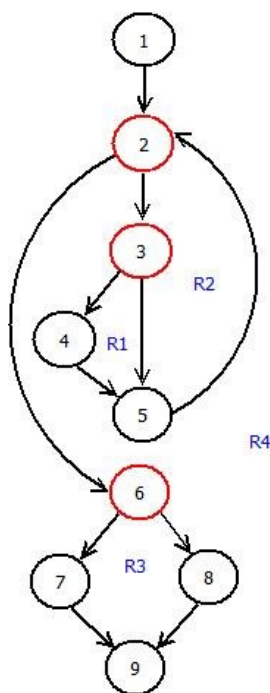


Figura 3.1. Representación del grafo de flujo.

Ahora ya podemos calcular la complejidad ciclomática $V(G)$ con las siguientes tres fórmulas:

$$V(G) = \text{nº regiones} = 4$$

$$V(G) = A - N + 2 = 11 - 9 + 2 = 4$$

$$V(G) = NP + 1 = 3 + 1 = 4$$

Como $V(G)$ toma el valor 4, hay 4 caminos independientes en el código examinado y, por tanto, el número de casos de prueba que hay que crear para cubrir todas las instrucciones y condiciones del programa es 4. Pues bien, encontremos estos cuatro caminos independientes. Para ello, hay que tener en cuenta que cada camino independiente incluye una ruta nueva en relación con los anteriores:

Camino 1: 1-2-6-8-9

Camino 2: 1-2-3-5-2-6-8-9

Camino 3: 1-2-3-4-5-2-6-8-9

Camino 4: 1-2-3-4-5-2-6-7-9

Para probar el camino 1, deberemos introducir el número 1. En este caso, se deberá indicar que el número no es un número perfecto. El camino 2 no lo podemos probar directamente porque todo

número es divisible entre 1, motivo por el cual hay que pasar obligatoriamente por el nodo 4 la primera vez que se pase por el bucle. Para probar el camino 3, deberemos introducir el número 2. En este caso, se deberá indicar que el número tampoco es un número perfecto. Para probar el camino 4, deberemos introducir un número perfecto. El primero de los números perfectos es el 6 porque $6 = 1+2+3$. Este camino 4, tal como está escrito, no se puede probar, pero al suministrar como entrada al programa el número 6, probaremos el paso por el nodo 7, que es la nueva ruta que incorpora este camino, ya que el número sí es perfecto. Esta debe ser, por tanto, la salida del programa. Con el número 5 probaremos la nueva ruta que incorpora el camino 3, que supone no pasar por el nodo 4, ya que la condición del nodo 3 debe ser falsa. La salida, en este caso, debe ser que el número no es perfecto. De esta manera, habremos probado todos los posibles caminos a través del código.

Actividad propuesta 3.2

El primer paso para aplicar esta técnica consiste en asignar un número a cada instrucción y condición del programa. Este paso ya está hecho en el ejercicio 1.

A continuación hay que calcular los conjuntos *DEF* y *USO* para la variable *sumadivisores*. Pues bien, podemos observar cómo se asigna valor a esta variable en el nodo 1. En el nodo 4 primero se usa (se consulta su valor) y luego se le asigna valor. Finalmente, se usa su valor en el nodo 6. A raíz de esto, tenemos los siguientes conjuntos *DEF* y *USO* para la variable *sumadivisores*:

$$DEF(1) = \{sumadivisores\}$$

$$USO(4) = \{sumadivisores\}$$

$$DEF(4) = \{sumadivisores\}$$

$$USO(6) = \{sumadivisores\}$$

Las cadenas *DU* para la variable *sumadivisores* son las siguientes:

$$DU [sumadivisores, 1, 4] (1)$$

$$DU [sumadivisores, 1, 6] (2)$$

$$DU [sumadivisores, 4, 4] (3)$$

$$DU [sumadivisores, 4, 6] (4)$$

Ahora vamos a encontrar el número mínimo de caminos que incluyan todas las cadenas *DU*. Tengamos en cuenta que las cadenas *DU* (1) y (2) son incompatibles, ya que si se pasa por el nodo 4, lo que es necesario para cubrir la cadena *DU* (1), ya no es posible cubrir la cadena *DU* (2) porque en el nodo 4 no solo se usa la variable *sumadivisores*, sino que luego se define. Podríamos definir los dos siguientes caminos: el primero de ellos para recorrer las cadenas *DU* (1), (3) y (4) y el segundo, para cubrir la cadena *DU* (2).

Camino 1: 1-2-3-4-5-2-3-4-5-2-6-8-9. Cubre las cadenas *DU* (1), (3) y (4).

Camino 2: 1-6-8-9. Cubre la cadena *DU* (2).

Para recorrer el camino 1 podríamos suministrar como entrada el número 4 y el programa debe indicar que no es un número perfecto.

Para recorrer el camino 2 se puede suministrar como entrada el número 1. La salida esperada es también que el programa nos indique que no se trata de un número perfecto.

Actividad propuesta 3.3

- a) Se muestran en la siguiente tabla las condiciones de entrada del programa y, por cada una de ellas, las clases válidas y las no válidas, establecidas de acuerdo con las normas de la técnica de clases de equivalencia:

Tabla 3.1. Clases de equivalencia.

Condición de entrada	Clases Válidas	Clases no válidas
TIS	Un número entero de 8 dígitos (1)	< 8 dígitos (2) > 8 dígitos (3) Alguno de los 8 caracteres no es un número (4) No es un número entero (5)
Primer apellido	Cadena de entre 2 y 30 caracteres con letras y espacios en blanco (6)	< 2 caracteres (7) > 30 caracteres (8) Alguno de los caracteres no es una letra ni un espacio en blanco (9)
Año nacimiento	Número entero entre 1901 y el año actual (10)	Número < 1901 (11) Número > año actual (12) No es un número entero (13) No es un número (14)

- b) En las siguientes tablas se muestran los casos de prueba válidos (Tabla 3.2) y no válidos (Tabla 3.3). Por cada caso de prueba se indica en la última columna las clases de equivalencia incluidas. Recordemos que para el caso de las clases válidas se debe crear el número mínimo de casos de prueba que incluya todas las clases válidas; en este caso, con un caso de prueba es suficiente.

Tabla 3.2. Casos de pruebas con clases válidas.

TIS	Primer apellido	Año nacimiento	Clases incluidas
02348723	PÉREZ	1990	(1), (6), (10)

Para las clases no válidas, hay que crear un caso de prueba por cada una de ellas, de forma que el caso de prueba incluya la clase no válida y todas las demás deben ser válidas.

Tabla 3.3. Casos de pruebas con clases no válidas

TIS	Primer apellido	Año nacimiento	Clases incluidas
023487	Pérez	1990	(2), (6), (10)
0234872390	Pérez	1990	(3), (6), (10)
0987TT99	Pérez	1990	(4), (6), (10)
9087,1234	Pérez	1990	(5), (6), (10)
02348723	A	1990	(1), (7), (10)
02348723	Gil de Viedma García-Verdugo Maestro	1990	(1), (8), (10)
02348723	Gil?	1990	(1), (9), (10)
02348723	PÉREZ	1895	(1), (6), (11)
02348723	PÉREZ	2032	(1), (6), (12)
02348723	PÉREZ	199,9	(1), (6), (13)
02348723	PÉREZ	ABCD	(1), (6), (14)

ACTIVIDADES FINALES

Actividades de comprobación

3.1	3.2	3.3	3.4	3.5	3.6	3.7	3.8	3.9	3.10
c)	d)	d)	a)	b)	d)	b)	a)	c)	b)

Actividades de aplicación

Actividad de aplicación 3.11

Es un conjunto de entradas, condiciones de ejecución y resultados esperados desarrollados para un objetivo particular como, por ejemplo, ejercitar un camino concreto de un programa o verificar el cumplimiento de un determinado requisito. Al ejecutar un caso de prueba, se obtienen unos resultados, de manera que si estos no coinciden con los esperados, se considera que el caso de prueba ha tenido éxito y se deberá localizar y corregir el error, proceso que recibe el nombre de depuración.

Actividad de aplicación 3.12

En las pruebas de la caja blanca es necesario conocer los detalles procedimentales del *software* que se prueba para generar los casos de prueba, pues hay que fijarse en los diferentes caminos a través

del código fuente, las variables, los bucles, etc. En las pruebas de la caja negra solo es necesario conocer las funciones que realiza el *software* para generar los casos de prueba.

Actividad de aplicación 3.13

Para realizar el grafo de flujo lo primero que hay que hacer es asignar un número único a cada sentencia o grupo de sentencias y a cada condición. Se muestra la asignación a continuación:

```
public static void main(String[] args) {
    int num;
    int sumapares = 0;
    int sumaimpares = 0;
    Scanner entrada = new Scanner(System.in);
    System.out.print ("Introduzca un número (0 para terminar): ");
    num = entrada.nextInt();
    while (num != 0) {
        if (num % 2 == 0) {
            sumapares = sumapares + num;
        } else {
            sumaimpares = sumaimpares + num;
        }
        System.out.print ("Introduzca un número (0 para terminar): ");
        num = entrada.nextInt();
    }
    System.out.println ("La suma de los números pares es " + sumapares);
    System.out.println ("La suma de los números impares es " + sumaimpares);
}
```

Ahora construimos el grafo de flujo:

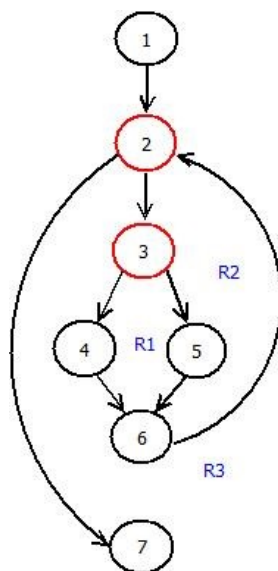


Figura 3.2. Representación del grafo de flujo.

Ahora ya podemos calcular la complejidad ciclomática $V(G)$ con las siguientes tres fórmulas:

$$V(G) = n^{\circ} \text{ regiones} = 3$$

$$V(G) = A - N + 2 = 8 - 7 + 2 = 3$$

$$V(G) = NP + I = 2 + 1 = 3$$

Como $V(G)$ toma el valor 3, hay 3 caminos independientes en el código examinado y, por tanto, el número de casos de prueba que hay que crear para cubrir todas las instrucciones y condiciones del programa es 3. Pues bien, encontremos estos tres caminos independientes. Para ello, hay que tener en cuenta que cada camino independiente incluye una ruta nueva en relación con los anteriores:

Camino 1: 1-2-7

Camino 2: 1-2-3-4-6-2-7

Camino 3: 1-2-3-5-6-2-7

Para probar el camino 1, deberemos introducir el número 0 únicamente. En este caso se deberá indicar que la suma de los números positivos y negativos es 0. Para probar el camino 2, deberemos introducir un número par, por ejemplo, el 10, y luego un 0. En este caso, se deberá indicar que la suma de los números pares es 10, y la de los impares, 0. Para probar el camino 3, deberemos introducir un número impar, por ejemplo, el 5, y luego un 0. En este caso, se deberá indicar que la suma de los números pares es 0, y la de los impares, 5.

Actividad de aplicación 3.14

Para realizar el grafo de flujo lo primero que hay que hacer es asignar un número único a cada sentencia o grupo de sentencias y a cada condición. Se muestra la asignación a continuación:

```
public static void main(String[] args){
    int num, i=2;
    boolean esPrimo = true;
    Scanner entrada = new Scanner(System.in);
    System.out.print ("Introduzca un número entero: ");
    num = teclado.nextInt();

    while ( i <= num/2 && esPrimo ) {
        if ( num%i == 0 ) {
            esPrimo = false;
        }
        i++;
    }
    if (esPrimo) {
        System.out.println ("El número " + num + " es primo.");
    }
    else {
        System.out.println ("El número " + num + " no es primo.");
    }
}
```

Diagrama de flujo de la asignación de números:

- 1: Inicio del método main.
- 2: Declaración de variables.
- 3: Creación del Scanner.
- 4: Condición de la while.
- 5: Condición de la if.
- 6: Incremento de i.
- 7: Condición de la if.
- 8: Condición de la if.
- 9: Sentencia de la if.
- 10: Sentencia de la else.
- 11: Fin del método main.

Ahora construimos el grafo de flujo:

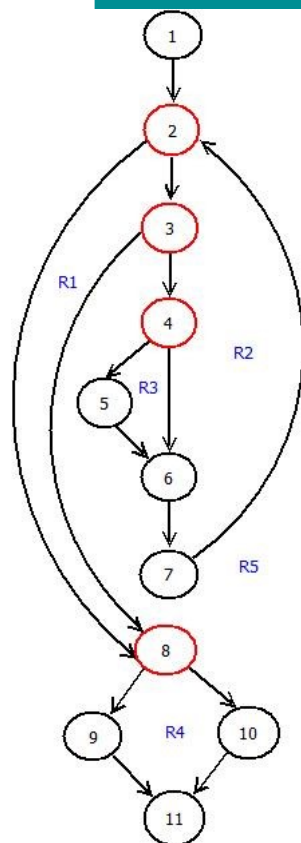


Figura 3.3. Representación del grafo de flujo.

Ahora ya podemos calcular la complejidad ciclomática $V(G)$ con las siguientes tres fórmulas:

$$V(G) = \text{nº regiones} = 5$$

$$V(G) = A - N + 2 = 14 - 11 + 2 = 5$$

$$V(G) = NP + 1 = 4 + 1 = 5$$

Como $V(G)$ toma el valor 5, hay 5 caminos independientes en el código examinado y, por tanto, el número de casos de prueba que hay que crear para cubrir todas las instrucciones y condiciones del programa es 5. Pues bien, encontremos estos 5 caminos independientes. Para ello, hay que tener en cuenta que cada camino independiente incluye una ruta nueva en relación con los anteriores:

Camino 1: 1-2-8-9-11

Camino 2: 1-2-3-4-6-7-2-8-9-11

Camino 3: 1-2-3-4-5-6-7-2-8-9-11

Camino 4: 1-2-3-4-5-6-7-2-3-8-9-11

Camino 5: 1-2-3-4-5-6-7-2-3-8-10-11

Para probar el camino 1, deberemos introducir el número 1 y la salida debe indicar que el número es primo. Para el camino 2 podemos introducir el número 5 y la salida debe indicar que el número es

primo. El camino 3 tal cual lo hemos diseñado es imposible probarlo porque habría que introducir un número par para que no se cumpla la condición del nodo 4 y, por tanto, el número no sería primo, por lo que no sería posible pasar por el nodo 9. Introduciendo un número par superior a 4, probaremos el camino 5, en cuyo caso la salida debe ser que el número no es primo. Con este camino ya probamos todas las rutas del programa adicionales a las incluidas en los caminos 1 y 2 (el paso por el nodo 5 y el paso del nodo 3 al 8).

Actividad de aplicación 3.15

El primer paso para aplicar esta técnica consiste en asignar un número a cada instrucción y condición del programa. Este paso ya está hecho en el ejercicio 3.13.

A continuación hay que calcular los conjuntos *DEF* y *USO* para la variable *sumapares*. Pues bien, podemos observar que se asigna valor a esta variable en el nodo 1. En el nodo 4 primero se usa (se consulta su valor) y luego se le asigna valor. Finalmente, se usa su valor en el nodo 7. A raíz de esto, tenemos los siguientes conjuntos *DEF* y *USO* para la variable *sumapares*:

$$DEF(1) = \{sumapares\}$$

$$USO(4) = \{sumapares\}$$

$$DEF(4) = \{sumapares\}$$

$$USO(7) = \{sumapares\}$$

Las cadenas *DU* para la variable *sumapares* son las siguientes:

$$DU [sumapares, 1, 4] (1)$$

$$DU [sumapares, 1, 7] (2)$$

$$DU [sumapares, 4, 4] (3)$$

$$DU [sumapares, 4, 7] (4)$$

Ahora vamos a encontrar el número mínimo de caminos que incluyan todas las cadenas *DU*:

Camino 1: 1-2-3-4-6-2-3-4-6-2-7. Cubre las cadenas *DU* (1), (3) y (4).

Camino 2: 1-2-3-5-6-2-7. Cubre la cadena *DU* (2).

En este caso no es posible recorrer en un solo camino todas las cadenas *DU* porque las cadenas *DU* (1) y (2) son incompatibles, ya que si se pasa por el nodo 4, lo que es necesario para cubrir la cadena *DU* (1), ya no es posible cubrir la cadena *DU* (2) porque en el nodo 4 no solo se usa la variable *sumapares*, sino que luego se define.

Para cubrir el camino 1 es necesario suministrar al programa dos números pares y luego un cero. Por ejemplo, se podrían introducir el 6 y el 10, en cuyo caso el programa deberá indicar que la suma de los números pares es 16, y la de los impares, 0.

Para cubrir el camino 2 es necesario suministrar el programa un número impar y luego un cero. Por ejemplo, se podría introducir el número 9, en cuyo caso el programa deberá indicar que la suma de los números pares es 0, y la de los impares, 9.

Actividad de aplicación 3.16

El primer paso para aplicar esta técnica consiste en asignar un número a cada instrucción y condición del programa. Este paso ya está hecho en el ejercicio 3.14.

A continuación, hay que calcular los conjuntos *DEF* y *USO* para la variable *i*. Se asigna valor a esta variable en el nodo 1. En los nodos 2 y 4 se usa *y*, finalmente, en el nodo 6 se usa *y* luego se le asigna valor. Por todo ello, tenemos los siguientes conjuntos *DEF* y *USO* para la variable *i*:

$$DEF(1) = \{i\}$$

$$USO(2) = \{i\}$$

$$USO(4) = \{i\}$$

$$USO(6) = \{i\}$$

$$DEF(6) = \{i\}$$

Las cadenas *DU* para la variable *i* son las siguientes:

$$DU[i, 1, 2] (1)$$

$$DU[i, 1, 4] (2)$$

$$DU[i, 1, 6] (3)$$

$$DU[i, 6, 2] (4)$$

$$DU[i, 6, 4] (5)$$

$$DU[i, 6, 6] (6)$$

Ahora vamos a encontrar el número mínimo de caminos que incluya todas las cadenas *DU*. En este caso es posible recorrer todas las cadenas *DU* mediante el siguiente camino:

Camino: 1-2-3-4-6-2-3-4-6-2-8-9-11.

Para cubrir este camino podemos suministrar el programa como entrada el número 7. La salida que se debe obtener es la indicación de que el número 7 es primo.

Actividad de aplicación 3.17

- a) Se muestran en la siguiente tabla las condiciones de entrada del programa y, por cada una de ellas, las clases válidas y las no válidas, establecidas de acuerdo con las normas de la técnica de clases de equivalencia:

Tabla 3.4. Clases de equivalencia.

Condición de entrada	Clases Válidas	Clases no válidas
NIF	Una cadena de 9 caracteres compuesta por 8 números y una letra correcta (1)	< 9 caracteres (2) > 9 caracteres (3) Alguno de los 8 primeros caracteres no es un número (4) El último carácter no es una letra (5) La letra no es correcta (6)
Nº tarjeta	16 números (7)	< 16 caracteres (8) > 16 caracteres (9) Alguno de los caracteres no es un número (10)
Marca tarjeta	Visa (11) Mastercard (12) Maestro (13)	Otra distinta (14)

- b) En las siguientes tablas se muestran los casos de prueba válidos (Tabla 3.5) y no válidos (Tabla 3.6). Por cada caso de prueba se indica en la última columna las clases de equivalencia incluidas. Recordemos que para el caso de las clases válidas se debe crear el número mínimo de casos de prueba que incluyan todas las clases válidas.

Tabla 3.5. Casos de prueba con clases válidas.

NIF	Nº tarjeta	Marca tarjeta	Clases incluidas
20175988C	2345678901212222	Visa	(1), (7), (11)
20175988C	2345678901212222	Mastercard	(1), (7), (12)
20175988C	2345678901212222	Maestro	(1), (7), (13)

Para las clases no válidas, hay que crear un caso de prueba por cada una de ellas, de forma que el caso de prueba incluya la clase no válida y todas las demás deben ser válidas.

Tabla 3.6. Casos de prueba con clases no válidas.

NIF	Nº tarjeta	Marca tarjeta	Clases incluidas
2017598C	2345678901212222	Visa	(2), (7), (11)
201759889C	2345678901212222	Mastercard	(3), (7), (12)
20k75988C	2345678901212222	Maestro	(4), (7), (13)
201759889	2345678901212222	Visa	(5), (7), (11)
20175988L	2345678901212222	Mastercard	(6), (7), (12)
20175988C	234567890121221	Maestro	(1), (8), (13)
20175988C	23456789012122200	Visa	(1), (9), (11)
20175988C	Agua122229876542	Mastercard	(1), (10), (12)
20175988C	2345678901212222	Tarjeta	(1), (7), (14)

Actividad de aplicación 3.18

- a) Se muestran en la siguiente tabla las condiciones de entrada del programa y, por cada una de ellas, las clases válidas y las no válidas, establecidas de acuerdo con las normas de la técnica de clases de equivalencia:

Tabla 3.7. Clases de equivalencia.

Condición de entrada	Clases válidas	Clases no válidas
Matrícula	Una cadena de 7 caracteres, siendo los 4 primeros números y los 3 últimos letras (1)	< 7 caracteres (2) > 7 caracteres (3) Alguno de los 4 primeros caracteres no es número (4) Alguno de los 3 últimos caracteres no es letra (5)
Nº puertas	3 (6) 4 (7) 5 (8)	Otro (9)
Potencia	Número entre 40 y 300 (10)	< 40 (11) > 300 (12) No es un número entero (13) No es un número (14)

- b) En las siguientes tablas se muestran los casos de prueba válidos (Tabla 3.8) y no válidos (Tabla 3.9).

Tabla 3.8. Casos de prueba con clases válidas.

Matrícula	Nº puertas	Potencia	Clases
2666FFF	3	110	(1)(6)(10)
2666FFF	4	110	(1)(7)(10)
2666FFF	5	110	(1)(8)(10)

Tabla 3.9. Casos de prueba con clases no válidas.

Matrícula	Nº puertas	Potencia	Clases
2666FF	3	110	(2)(6)(10)
2666FFGT	3	110	(3)(6)(10)
2ª66FFF	3	110	(4)(6)(10)
2666F3F	3	110	(5)(6)(10)
2666FFF	7	110	(1)(9)(10)
2666FFF	3	23	(1)(6)(11)
2666FFF	3	498	(1)(6)(12)
2666FFF	3	110,23	(1)(6)(13)
2666FFF	3	A	(1)(6)(14)

Actividad de aplicación 3.19

Consiste en generar una lista de errores que suelen cometer los desarrolladores y de las situaciones propensas a dichos errores, para luego generar un conjunto de casos de prueba basándose en esta lista. Se trata de errores que se cometen comúnmente no relacionados con aspectos funcionales.

Actividad de aplicación 3.20

Se pueden complementar porque la técnica de análisis de valores límite exige elegir de entre los valores a probar aquellos que se encuentran justo en los límites de la clase de equivalencia. Por ejemplo, si tenemos que una clase válida detectada mediante la técnica de clases de equivalencia que nos indica que suministremos un dato entre 10 y 30, ambos incluidos, la técnica de análisis de valores límite nos indicará que debemos escoger justo los valores en el límite de la clase (10 y 30). Además, si detectamos dos clases no válidas (valor menor que 10 y valor mayor que 30), la técnica de análisis de valores límite nos indica que probemos justamente con los valores que están en el límite (9 y 31, respectivamente).

Actividad de aplicación 3.21

```
import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;
```

```
import org.junit.jupiter.api.BeforeEach;

class PuntoTest {

    private Punto p1;

    private Punto p2;

    @BeforeEach

    public void nuevoPunto() {

        p1 = new Punto (1,0);

        p2 = new Punto (4,4);

    }

    @Test

    void testDistancia() {

        double d = p1.distancia(p2);

        assertEquals (d, 5);

    }

    @Test

    void testCompara() {

        boolean resul = p1.compara(p2);

        assertEquals (resul, false);

    }

}
```

Actividad de aplicación 3.22

Para que sus instrucciones se ejecuten siempre después de la ejecución de cada método de prueba.


Actividad de aplicación 3.23

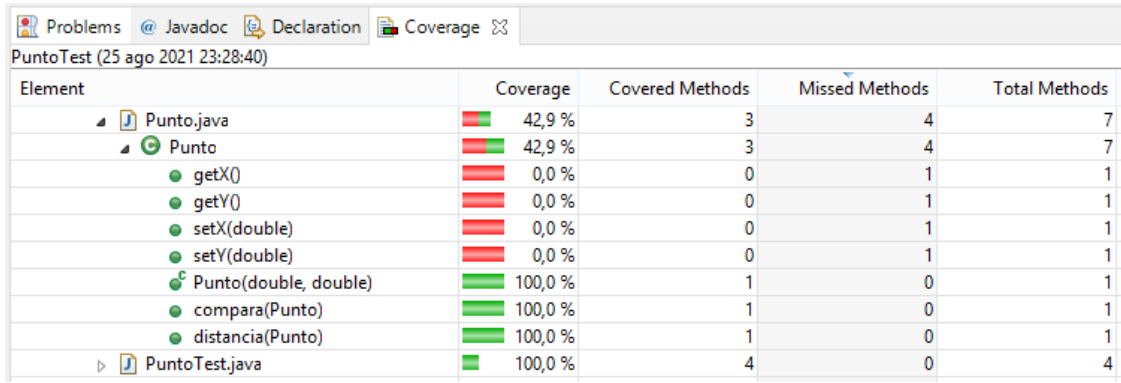
Se toma del explorador de proyectos la clase *PuntoTest* y se selecciona la opción de menú *Run – Coverage*. Se obtiene el siguiente resultado, donde se nos muestra la cobertura de instrucciones.

Element	Coverage	Covered Instruccio...	Missed Instructions	Total Instructions
PuntoTest (25 ago 2021 23:28:40)				
└ Punto.java	60,7 %	34	22	56
└ Punto	60,7 %	34	22	56
└ compara(Punto)	50,0 %	8	8	16
└ setX(double)	0,0 %	0	4	4
└ setY(double)	0,0 %	0	4	4
└ getX()	0,0 %	0	3	3
└ getY()	0,0 %	0	3	3
└ Punto(double, double)	100,0 %	9	0	9
└ distancia(Punto)	100,0 %	17	0	17
└ PuntoTest.java	100,0 %	40	0	40

Figura 3.4. Se nos muestra la cobertura de instrucciones para la clase *Punto*, que es del 60,7%

Se nos indica que se ha cubierto el 60,7% de las instrucciones. Se han cubierto completamente el método constructor y el método *distancia()*, no se han cubierto en absoluto los métodos *setX()*, *setY()*, *getX()* y *getY()* y se ha cubierto parcialmente (el 50% de las instrucciones) el método *compara()*.

Para calcular la cobertura de métodos, hacemos clic en el botón  situado en la parte derecha de la pestaña *Coverage* de la consola *Java*. Así, si seleccionamos la opción *Method Counters*, el resultado es el siguiente:



Element	Coverage	Covered Methods	Missed Methods	Total Methods
PuntoTest (25 ago 2021 23:28:40)				
Punto.java	42,9 %	3	4	7
Punto	42,9 %	3	4	7
getX()	0,0 %	0	1	1
getY()	0,0 %	0	1	1
setX(double)	0,0 %	0	1	1
setY(double)	0,0 %	0	1	1
Punto(double, double)	100,0 %	1	0	1
compara(Punto)	100,0 %	1	0	1
distancia(Punto)	100,0 %	1	0	1
PuntoTest.java	100,0 %	4	0	4

Figura 3.5. Se nos muestra la cobertura de métodos para la clase *Punto*, que es del 42,9%

En este caso se nos indica que se ha cubierto el 42,9% de los métodos porque se han probado 3 de los 7 métodos de la clase *Punto*. La cobertura es inferior a la de instrucciones porque solo se considera si el método ha sido probado o no sin tener en cuenta el número de instrucciones de este que han sido probadas.

Actividades de ampliación

Actividad de ampliación 3.24

No son válidas porque en el *software* convencional una aplicación no consta de clases, sino de módulos (procedimientos y funciones) que se llaman entre sí. En <https://manuel.cillero.es/doc/metodologia/metrica-3/tecnicas/pruebas/integracion> se nos indica que existen dos posibilidades para la integración: la no incremental, en la que se prueban todos los componentes por separado y luego se integran todos de una sola vez; o la incremental, en la que se prueba un nuevo componente o un grupo pequeño de componentes con los previamente probados, incrementando progresivamente el alcance de la prueba. Dentro de la integración incremental hay, a su vez, tres posibilidades:

- De arriba a abajo o integración incremental descendente; se comienza probando el módulo de la parte superior de la jerarquía y se va descendiendo progresivamente.
- De abajo a arriba o integración incremental ascendente; se comienza probando los módulos de los niveles más bajos y se va ascendiendo progresivamente hasta incorporar el módulo de la parte superior.
- Estrategias combinadas o sándwich; se aplican las dos estrategias anteriores.

Actividad de ampliación 3.25

En <https://www.tutorialcup.com/es/testing/types-of-testing/alpha-testing.htm> se indica que estos dos tipos de pruebas forman parte de la prueba de aceptación o validación como fase final de la tarea de pruebas. En cuanto a las diferencias:

- Las pruebas alfa se realizan en el sitio del desarrollador y las pruebas beta, en el sitio del cliente.
- Los clientes llevan a cabo ambos tipos de pruebas, pero las pruebas alfa se realizan en presencia del desarrollador, mientras que las pruebas beta, no.
- El objetivo de las pruebas alfa es que los errores no sean detectados por el cliente, mientras que el objetivo de las pruebas beta es establecer si el producto funciona correctamente en el entorno del usuario y está preparado para su explotación.
- En las pruebas alfa se pueden emplear pruebas de caja blanca y de caja negra, mientras que en el caso de las pruebas beta solo se usan pruebas de caja negra.

Actividad de ampliación 3.26

En https://es.wikipedia.org/wiki/Pruebas_de_regresión se indica que este tipo de pruebas se deben llevar a cabo con cada nueva liberación del producto con el fin de asegurarse de que los cambios introducidos no hayan producido regresiones en el software, es decir, no se hayan introducido defectos inexistentes antes de realizar los cambios. El propósito de este tipo de pruebas es asegurarse de que los casos de prueba que ya habían sido probados y no encontraron defectos permanezcan así, de lo que deducimos que se deben emplear las mismas técnicas de diseño de casos de prueba que se usaron antes. Este tipo de pruebas se pueden automatizar y, además, es aconsejable que se haga así.

Actividad de ampliación 3.27

Podemos encontrar información sobre las pruebas de usabilidad en la web <https://sg.com.mx/revista/29/pruebas-usabilidad-web>, donde se indica que mediante estas pruebas se evalúa la usabilidad de una aplicación entendiendo esta como “la capacidad de un producto de *software* para ser entendido, aprendido, utilizado y atractivo para el usuario, cuando se usa bajo condiciones específicas”. Se indica asimismo que estas pruebas se suelen llevar a cabo observando a las personas usar el producto objeto de prueba y que este grupo de personas es seleccionado considerando ciertas características según las condiciones que se deseen valorar, categorizándolos, por ejemplo, como usuarios expertos, medios o inexpertos.

Entre los aspectos de usabilidad que se suelen valorar se encuentran la facilidad de aprendizaje, la accesibilidad, la flexibilidad, el tiempo de respuesta, la reducción de carga cognitiva, la recuperabilidad y la buena imagen y estética.

Actividad de ampliación 3.28

Podemos encontrar información sobre las pruebas de humo en la web <https://www.javiergarzas.com/2014/06/smoke-test-en-menos-de-10-min.html>, donde se indica que estas pruebas incluyen casos de prueba que solo verifican ciertas funcionalidades básicas del *software*. Cada vez que se realizan cambios o adiciones en un producto *software*, se deben realizar este tipo de pruebas de tal forma que si se detectan fallos en los elementos básicos, se deben

corregir, y en caso contrario, se continúa con otro tipo de pruebas que analicen la aplicación en profundidad.

Actividad de ampliación 3.29

Podemos encontrar información sobre las pruebas de portabilidad en la web https://en.wikipedia.org/wiki/Portability_testing, donde se indica que estas pruebas tienen por objetivo establecer la facilidad o dificultad con que un producto de *software* se puede mover de un entorno a otro, lo que se refiere a la medida en que el producto se puede seguir utilizando al realizar cambios en cuanto a sistema operativo, hardware, base de datos, navegador, etc. Por este motivo, este tipo de prueba se realiza frecuentemente con productos que se transfieren con frecuencia entre entornos. Por ejemplo, si una aplicación web se ha desarrollado en el sistema operativo *Windows* y para el navegador *Chrome*, sería recomendable saber si la misma página web se puede visualizar en el navegador *Safari* de un usuario que utilice *MacOS* o en el navegador *Firefox* de un usuario de *Linux*. Por tanto, las páginas web son muy buenas candidatas para la realización de pruebas de portabilidad. Las pruebas de portabilidad se corresponden con las pruebas de despliegue, que son un tipo de pruebas del sistema que se explicaron en el Apartado 3.2.3.

Entornos de desarrollo

Solucionario Unidad 4

Optimización y documentación

Actividades propuestas

Actividad propuesta 4.1

Se selecciona en el método indicado la porción de código entre *case 2* y la correspondiente instrucción *break*, se activa la opción de menú contextual *Refactor* → *Extract Method* y se hace clic en el botón *Preview*, tras lo que se puede ver la sustitución. La sustitución se llevará a cabo tras clicar sobre el botón *OK*.

Se hace lo mismo para la porción código entre *case 3* y la correspondiente instrucción *break*.

Actividad propuesta 4.2

Se selecciona el atributo *lado1* de la clase *Triángulo* y se activa la opción *Refactor* → *Encapsulate Field* del menú contextual. Eclipse indicará los nombres de los métodos de consulta y acceso que se usarán para el encapsulamiento, que serán los presentes en la clase, en caso de que ya estén declarados. Como siempre, al hacer clic en el botón *Preview*, es posible visualizar los cambios que supondría llevar a cabo esta refactorización, la cual se activará al pulsar el botón *OK*.

Se debe hacer lo mismo para los atributo *lado2* y *lado3* de la clase *Triángulo*.

Actividad propuesta 4.3

Añadimos comentarios en la clase *Punto* dando lugar a un archivo con el siguiente contenido:

```
package figuras;

/**
 * <h3> Clase Punto, para contener la información de cada punto </h2>
 * @version 02-2022
 * @author Jose Piñeiro
 * since 24-02-2022
 */
public class Punto {
    private double x;    //Coordenada x del punto
    private double y;    //Coordenada y del punto
```

```
/**
 * Constructor de la clase Punto sin parámetros. Crea un punto con coordenadas (0,0)
 */
public Punto() {
    setX(0);
    setY(0);
}

/**
 * Constructor de la clase Punto con las coordenadas x e y del punto que se quiere crear
 * @param x Coordenada x del punto
 * @param y Coordenada y del punto
 */
public Punto(double x, double y) {
    this.setX(x);
    this.setY(y);
}

/**
 * Constructor de la clase Punto a partir de otro punto
 * @param p Punto a partir del que se desea crear uno nuevo
 */
public Punto(Punto p) {
    setX(p.getX());
    setY(p.getY());
}

/**
 * Método de selección de la coordenada x del punto
 * @return Coordenada x del punto
 */
public double getX() {
    return x;
}
```

```
/**
 * Método de selección de la coordenada y del punto
 * @return Coordenada y del punto
 */
public double getY() {
    return y;
}

/**
 * Método de acceso a la coordenada x del punto
 * @param x Coordenada x que se desea asignar al punto
 */
public void setX(double x) {
    this.x = x;
}

/**
 * Método de acceso a la coordenada y del punto
 * @param y Coordenada y que se desea asignar al punto
 */
public void setY(double y) {
    this.y = y;
}

/**
 * Método que calcula la distancia con el punto recibido como parámetro
 * @param p Punto hasta el que se desea calcular la distancia
 * @return Distancia hasta el punto recibido como parámetro
 */
public double distancia(Punto p) {
    return Math.sqrt (Math.pow(p.getX() - this.getX(), 2) + Math.pow(p.getY() - this.getY(), 2));
}

/**
```

```
* Método que devuelve el punto simétrico
* @return Punto simétrico
*/
public Punto simétrico() {
    return new Punto (this.getX() * -1, this.getY());
}

/**
* Método que realiza una comparación con el punto recibido como parámetro
* @param p Punto con el que se desea realizar la comparación
* @return Devuelve true si las coordenadas de los dos puntos son iguales y false en caso
*         contrario
*/
public boolean compara(Punto p) {
    if (p.getX() == getX() && p.getY() == getY())
        return true;
    else
        return false;
}

/**
* Método que devuelve en una cadena de caracteres con la forma (x,y) la información de un
* punto (sus coordenadas x e y)
* @return Cadena de caracteres con los datos del punto
*/
public String toString() {
    return "(" + getX() + ", " + getY() + ")";
}
}
```

Se selecciona la opción de menú *Project* → *Generate Javadoc* y se siguen los pasos indicados al final del Apartado 4.4.2.

ACTIVIDADES FINALES

Actividades de comprobación

4.1	4.2	4.3	4.4	4.5	4.6	4.7	4.8	4.9	4.10	4.11	4.12
d)	d)	b)	b)	d)	b)	d)	c)	b)	c)	c)	b)

Actividades de aplicación

Actividad de aplicación 4.13

Podemos definir la refactorización como la realización de modificaciones en el código con el objetivo de mejorar su estructura interna sin alternar su comportamiento externo.

Actividad de aplicación 4.14

Es mejor aplicar la refactorización continua, que consiste en ir aplicando pequeñas refactorizaciones a medida que se desarrolla el *software* y normalmente tras la incorporación de cada nueva funcionalidad. Esto es debido a que cuanto mayor sea la porción de *software* a la que afecta la refactorización, mayor es la posibilidad de que como consecuencia de la refactorización la aplicación deje de funcionar correctamente.

Actividad de aplicación 4.15

Consiste en sustituir la expresión seleccionada por una nueva variable de manera que cualquier referencia a esa expresión es sustituida por dicha variable en ese ámbito. Esta modificación solo afecta al método en el que se realiza la refactorización.

Actividad de aplicación 4.16

Consiste en sustituir todas las referencias a un atributo de una clase por los correspondientes métodos de consulta (*get*) y de acceso (*set*).

Actividad de aplicación 4.17

Se debe seleccionar en el explorador de proyectos la clase *Triángulo* y elegir del menú contextual la opción *Refactor – Rename*. Hay que escribir en *New name* el nuevo nombre, esto es, *Triangulo* y al hacer *clic* en el botón *Next* aparecen todos los lugares de la aplicación en los que se propone realizar el cambio. Se hace clic en el botón *Finish* para que se realicen las modificaciones.

Para los métodos abstractos *área()* y *perímetro()* de la clase *Figura*, se deben seleccionar y elegir también la opción del menú contextual *Refactor – Rename* y se llevarán a cabo los cambios de nombre en cuatro clases en cada caso (*Figura*, *PruebaFigura*, *Triangulo* y *Rectangulo*).

Actividad de aplicación 4.18

Se debe seleccionar cada uno de los elementos cuyo nombre se desea cambiar y elegir del menú contextual la opción de menú *Refactor – Rename*.

Actividad de aplicación 4.19

Se debe seleccionar la variable *opcion* en el método *main* y elegir del menú contextual la opción *Refactor – Convert Local Variable to Field*.

Actividad de aplicación 4.20

Las herramientas de refactorización no detectan los defectos del *software*, que en el ámbito de la refactorización se suelen llamar malos olores o *bad smells*, pero sí proporcionan medios para su eliminación. Así, en *Eclipse* por medio de la opción de menú *Refactor* se puede aplicar sobre el elemento seleccionado un determinado patrón de refactorización.

Los analizadores de código, por el contrario, sí detectan los defectos del *software* de acuerdo con las reglas definidas, pero no los corrigen de manera automática.

Actividad de aplicación 4.21

Sí, se pueden consultar eligiendo la opción de menú *Windows – Preferences* y seleccionando en la parte izquierda *PMD – Rule Configuration*.

Actividad de aplicación 4.22

Sí, se pueden consultar eligiendo la opción de menú *Windows – Preferences* y seleccionando en la parte izquierda *PMD – Rule Configuration*. Si se activa la casilla de verificación de la parte superior *Use global rule management*, se pueden modificar las reglas, e incluso eliminar reglas o añadir nuevas reglas. Para modificar una regla, al seleccionarla, en la parte inferior en la pestaña *Rule* se muestran las propiedades de la regla y se pueden modificar.

Actividad de aplicación 4.23

En *Eclipse* vamos a la perspectiva *Git* seleccionando la opción de menú *Window – Perspective – Open Perspective – Other – Git*. Se elige en la parte izquierda de la pantalla la opción de crear un nuevo repositorio local de *Git*, tras lo cual se nos pedirá la ruta de dicho repositorio y el nombre. Se elige la ruta del repositorio *RepositoriosGit* y le añadiremos *\Ejemplo2*.

Para crear un nuevo proyecto en *Eclipse*, primero es necesario mostrar la perspectiva *Java*, a la que se accede seleccionando la opción de menú *Window – Perspective – Open Perspective – Other – Java (default)*. Creamos el proyecto en la carpeta *EjemplosCtrlVersionesGit* y le asignamos el nombre *Proyecto2*. A continuación vamos a asociar este proyecto al repositorio local *Ejemplo2*.

creado anteriormente, para lo que debemos seleccionar del menú contextual del proyecto la opción *Team/Share Project...* En la pantalla que se nos muestra hemos de seleccionar en *Repository* el repositorio que hemos creado y pulsar el botón *Finish*.

Tras crear la clase que muestra un mensaje por pantalla, seleccionamos para el proyecto la opción del menú contextual *Team – Commit*, escribimos un mensaje para el *commit* y hacemos clic en el botón *Commit*. Los cambios están confirmados en el repositorio local.

Creamos en *GitHub* el repositorio remoto y para almacenar los cambios en este repositorio seleccionamos para el proyecto la opción del menú contextual *Team – Remote – Push...* Proporcionamos la *URL* del repositorio obtenida desde *GitHub* y las credenciales e indicamos que se suba la cabeza del proyecto (*HEAD*).

Para crear la rama *rama1*, en la ventana de repositorios *Git* se elige para el elemento *Branches* la opción del menú contextual *Switch To – New Branch* y se le da un nombre a la rama (*rama1*). Se crea a continuación una clase dentro de la rama con un método *main* que muestre dos mensajes. Para confirmar los cambios en el repositorio local, se selecciona para el proyecto la opción del menú contextual *Team – Commit* y se escribe un mensaje para el *commit*. Se añaden dos nuevos mensajes en la clase y se hace otro *commit*. Para subir esta rama al repositorio remoto, se selecciona la rama *rama1* en la ventana de los repositorios *Git* y la opción del menú contextual *Push Branch...*

Para fusionar la rama *rama1* con el tronco, en la ventana de repositorios *Git* se hace doble clic sobre la rama *master* con el fin de movernos a esa rama (hacer *checkout*). A continuación, se debe seleccionar del menú contextual de la rama *master* la opción *Merge* para indicar que la queremos fusionar con la rama *rama1*. Para subir esto al repositorio remoto se debe seleccionar del menú contextual para la rama *master* la opción *Push Branch*.

Se crea a continuación la rama *rama2* como se creó la rama *rama1* y se confirma mediante la opción del menú contextual *Team – Commit*.

Para visualizar las diferentes versiones, se selecciona para el proyecto la opción de menú *Team – Show in History*, se pueden ver todos los *commits* que se han realizado y que para la rama *rama2* aparece el indicador *HEAD*, pues se trata de la versión más reciente. Aparecen los cuatro *commits* que hemos realizado.

Para subir la rama *rama2* al repositorio remoto se debe seleccionar del menú contextual para la rama la opción *Push Branch*. Se fusiona la rama *rama2* con la rama *master* de igual forma que se hizo para fusionar la rama *rama1*. Para subir los cambios al repositorio remoto, se debe seleccionar del menú contextual para la rama *master* la opción *Push Branch*.

Actividad de aplicación 4.24

ERS hace referencia a especificación de requisitos del *software*, que es el resultado de la fase de análisis.

Actividades de ampliación

Actividad de ampliación 4.25

Como se puede ver en la página <https://blog.ahierro.es/tipos-de-mantenimiento-de-software/>, el mantenimiento preventivo consiste en modificar el *software* proactivamente con el objetivo de evitar su deterioro o de mejorarlo sin alterar su funcionamiento.

Podemos deducir de esta definición que uno de los medios para realizar un mantenimiento preventivo es la refactorización continua, por lo que podemos decir que una de las técnicas que se puede emplear durante la tarea de mantenimiento preventivo es la refactorización. El proceso de corregir los “malos olores” del código, que es en lo que consiste básicamente la refactorización, mejora el código sin modificar su funcionamiento y facilita su posterior mantenimiento. Pero, además de corregir los “malos olores” del código, el mantenimiento preventivo puede implicar la realización de otras tareas, como una correcta documentación del código fuente.

Actividad de ampliación 4.26

En la página web <https://refactoring.guru/es/smells/shotgun-surgery> se indica que la razón de este “mal olor” es que una responsabilidad individual se ha dividido en muchas clases. Se puede solucionar moviendo métodos y atributos de varias clases a una única clase; si no hubiera una clase a donde realizar estos movimientos, se debe crear una nueva.

Actividad de ampliación 4.27

En la página web <https://refactoring.guru/es/smells/divergent-change> se indica que la razón de este “mal olor” es una pobre estructuración del programa o programación de copia y pega. Se debe a que una clase tiene más responsabilidades de las que debería, por lo que habría que crear una nueva clase donde situar varios atributos y métodos de la clase con excesivos cometidos. Si ocurre que varias clases tienen el mismo comportamiento, entonces se debería crear una jerarquía con esas clases.

Actividad de ampliación 4.28

La página web <https://git-scm.com/> es la página oficial de *Git*. Las siglas *SCM* hacen referencia a la gestión de la configuración del *software* (*Software Configuration Management*). Haciendo clic en el enlace *Downloads* de la página anterior o directamente en la página web <https://git-scm.com/downloads> aparecen las diferentes opciones de descarga de *Git* en modo línea de comandos para los sistemas operativos *Mac OS*, *Windows* y *Linux/Unix*.

Actividad de ampliación 4.29

Sí es posible, desde la página web <https://git-scm.com/downloads/guis> se pueden descargar dos clientes *GUI* para sistemas operativos *Mac OS*, *Windows* y *Unix/Linux*: se trata de *git-gui*, para confirmar cambios (*committing*) y *gitk* para navegar por el repositorio *Git*. Además, es posible descargarse herramientas de terceros. Algunas de ellas, como *Git Extensions* o *SmartGit* se pueden emplear en varios sistemas operativos, mientras que otras son exclusivas de un sistema operativo en concreto, como *giggle*, que es exclusiva para *Linux*; *Pocket Git*, que solo se puede usar en *Android*; y *GitDrive*, que se puede usar con *iOS*.

Actividad de ampliación 4.30

Desde la web <https://git-scm.com/docs> se puede acceder al manual de referencia de *Git* y desde https://git-scm.com/docs/git#_git_commands a la lista completa de comandos que se pueden emplear en *Git*. La utilidad de los comandos solicitados es:

- *git init*; crea un repositorio *Git* local vacío o reinicializa uno existente. Una vez creada una carpeta en el equipo local, crea un repositorio para la carpeta actual.
- *git diff*; muestra las diferencias existentes entre lo que hay en el área de trabajo y en el área de preparación en relación con lo que está confirmado en el repositorio.

Actividad de ampliación 4.31

Sí es posible. En la página web <https://github.com/Apache/NetBeans-website/blob/master/NetBeans.Apache.org/src/content/kb/docs/IDE/git.asciidoc> se proporcionan instrucciones sobre el uso de *Git* en *Apache NetBeans*. No es necesario instalar ningún módulo, pues estas funciones ya vienen incluidas de serie en *Apache NetBeans*.

Entornos de desarrollo

Solucionario Unidad 5

Elaboración de diagramas de clases

Actividades propuestas

Actividad propuesta 5.1



Figura 5.1. Diagrama de clases que refleja los supermercados de la cadena y las ciudades donde estos están ubicados.

Actividad propuesta 5.2

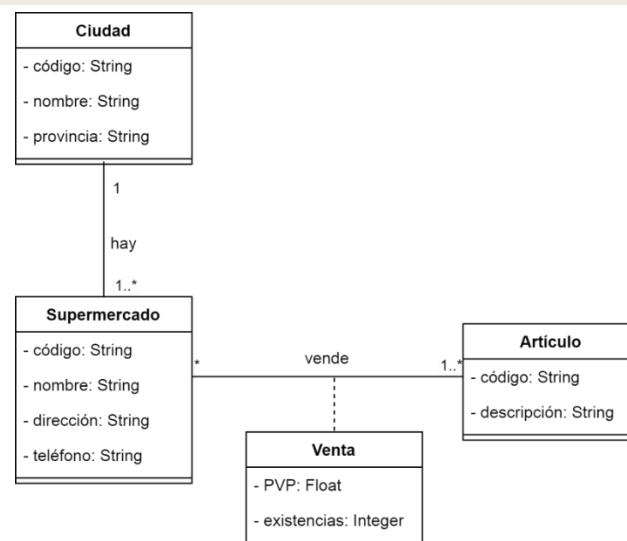


Figura 5.2. Diagramas de clases ampliado en relación con el de la Actividad Propuesta 5.1 que refleja los artículos que se venden en cada supermercado.

Actividad propuesta 5.3

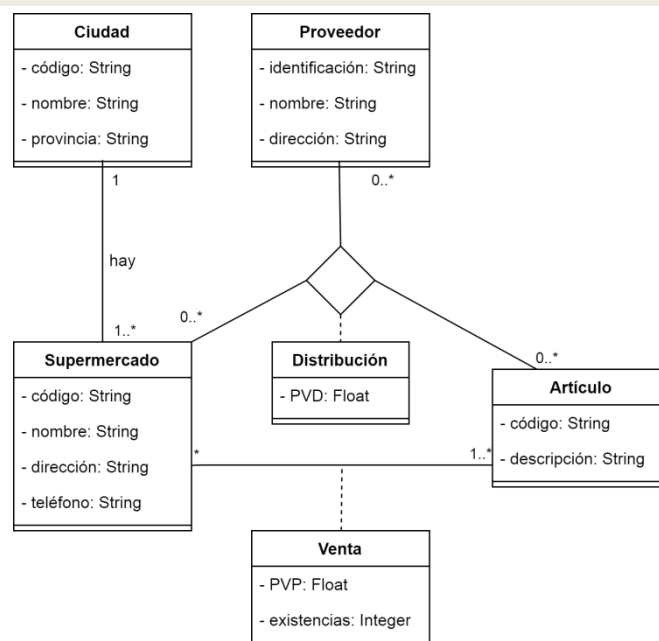


Figura 5.3. Diagramas de clases ampliado en relación con el de la Actividad Propuesta 5.2 que refleja información sobre los proveedores que suministran los artículos que se venden en cada supermercado.

ACTIVIDADES FINALES

Actividades de comprobación

5.1	5.2	5.3	5.4	5.5	5.6	5.7	5.8	5.9	5.10
a)	c)	a)	d)	a)	b)	b)	c)	c)	c)

Actividades de aplicación

Actividad de aplicación 5.11

Mientras que los diagramas de clases muestran un conjunto de clases y las relaciones entre ellas, los diagramas de objetos muestran un conjunto de objetos o instancias de clases y sus relaciones, es decir, estos últimos se puede decir que reflejan la situación del sistema en un instante determinado por medio de los objetos existentes en ese momento y sus relaciones.

Actividad de aplicación 5.12

Los diagramas de casos de uso se usan para mostrar las funciones y las características del sistema desde el punto del usuario. Un diagrama de casos de uso consta de casos de uso, actores y las relaciones entre ellos. Los casos de uso son cada una de las funciones que tiene que realizar el sistema y los actores son conjuntos de roles que desempeñan los usuarios del sistema.

Actividad de aplicación 5.13

La visibilidad se puede definir como el ámbito desde el que es visible el atributo o método de la clase. Los valores que puede tomar la visibilidad de un atributo o un método o, lo que es lo mismo, su modificador de acceso puede tomar los siguientes valores:

- *Private*; quiere decir que ese atributo o método solo es accesible desde métodos de la clase en la que está declarado el atributo o método. Este modificador se simboliza con un guion (-).
- *Public*; quiere decir que ese atributo o método es accesible desde métodos de cualquier clase. Este modificador se simboliza con el signo más (+).
- *Protected*; indica que el atributo o el método que lo tenga asignado es accesible desde métodos de la propia clase y desde métodos de su/s subclase/s. Este modificador se simboliza con el símbolo almohadilla (#).
- *Package*; indica que el atributo o el método es visible en las clases incluidas en el mismo paquete. Este modificador se simboliza con el carácter tilde (~).

Actividad de aplicación 5.14

Las relaciones de grado mayor que 2 se representan mediante un rombo cuyos vértices se unen a las clases relacionadas. Estas relaciones, así como las relaciones binarias y las reflexivas, pueden tener atributos, en cuyo caso estos atributos de la relación se sitúan dentro de una clase asociativa que se representa como una clase normal con dichos atributos, pero unida mediante una línea con trazado discontinuo a la relación. Tenemos un ejemplo de relación ternaria con atributos asociados en la figura 5.27. Se trata de una relación que nos indica las pruebas en las que han participado los atletas de una competición deportiva y las fechas de esa o esas participaciones. Además, dado que por cada participación de un atleta en una prueba un determinado día queremos saber el resultado de dicha participación (el tiempo empleado en la carrera, la altura o longitud saltada, etc.), debemos crear una clase asociativa para contener dicho resultado.

Actividad de aplicación 5.15

Estos conceptos son aplicables a las jerarquías de clases que se establecen cuando se detectan atributos y/o métodos comunes entre varias clases, de manera que los atributos y métodos comunes se asignan a una superclase, dejando los atributos y métodos específicos en las subclases.

Entonces, se dice que la superclase es una generalización de las subclases y que la superclase se especializa en una o varias subclases porque estas tienen, además de las características de la superclase, una serie de atributos y/o métodos específicos.

Actividad de aplicación 5.16

El 1 al lado de *Ciudad* en la relación *esCapitalDe* indica que un país tiene una ciudad que es su capital. El 0..1 al lado de *País* quiere decir que una ciudad no es capital (de ningún país) o lo es de solo un país.

En cuanto a la relación ternaria *imparte*:

- El 0..1 al lado de *Profesor* quiere decir que una asignatura a un grupo de alumnos o bien no la imparte ningún profesor o la imparte solo uno. El cero tiene sentido porque no son posibles todas las combinaciones de asignatura y grupo, esto es, un grupo no recibe clase de todas las asignaturas existentes.
- El 0..* al lado de *Grupo* quiere decir que un profesor puede impartir una asignatura a ningún grupo de alumnos o a varios.
- El 0..* al lado de *Asignatura* quiere decir que un profesor a un grupo de alumnos le puede impartir ninguna o varias asignaturas.

Actividad de aplicación 5.17

La línea que une la relación *vende* con la clase *Venta* es discontinua porque la clase *Venta* no es una clase normal sino una clase asociativa, de forma que esta clase proporciona información sobre la relación entre *Supermercado* y *Artículo*. Los atributos *stock* y *precio* nos indican por cada pareja de supermercado-artículo el stock o las existencias de ese artículo en ese supermercado y el precio al que vende ese artículo ese supermercado. De esto deducimos que el precio de los artículos puede variar de un supermercado a otro. Si no se hubiese creado la clase asociativa *Venta*, habría que haber colocado los atributos *stock* y *precio* en alguna de las otras dos clases (*Supermercado* o *Artículo*) y esto no es posible por lo siguiente:

- No sería correcto colocar los atributos *stock* y *precio* en *Supermercado* porque un supermercado puede vender muchos artículos y para cada uno de dichos artículos habrá que saber su stock y su precio. Un atributo en un objeto (una instancia de *Supermercado*, en este caso) solo puede tomar un valor, no varios.
- No sería correcto colocar los atributos *stock* y *precio* en *Artículo* porque un mismo artículo puede venderse en muchos supermercados y en cada uno de dichos supermercados tendrá un stock y un precio determinados. Habría sido correcto colocar el atributo *precio* en *Artículo* solo si un artículo se vendiese al mismo precio en todos los supermercados.

Actividad de aplicación 5.18

Se pueden identificar tres tipos de clases:

- Clases de interfaz, que se usan para modelar ventanas, formularios, impresos, etc. Se emplean para modelar la interacción entre el sistema y sus actores.
- Clases de entidad, que se usan para modelar información que se debe almacenar.
- Clases de control, que se emplean para coordinar el trabajo de todas las demás clases, de forma que se debe crear una clase de control por cada aplicación y esta se encarga de enviar trabajo a otras clases.

Actividad de aplicación 5.19

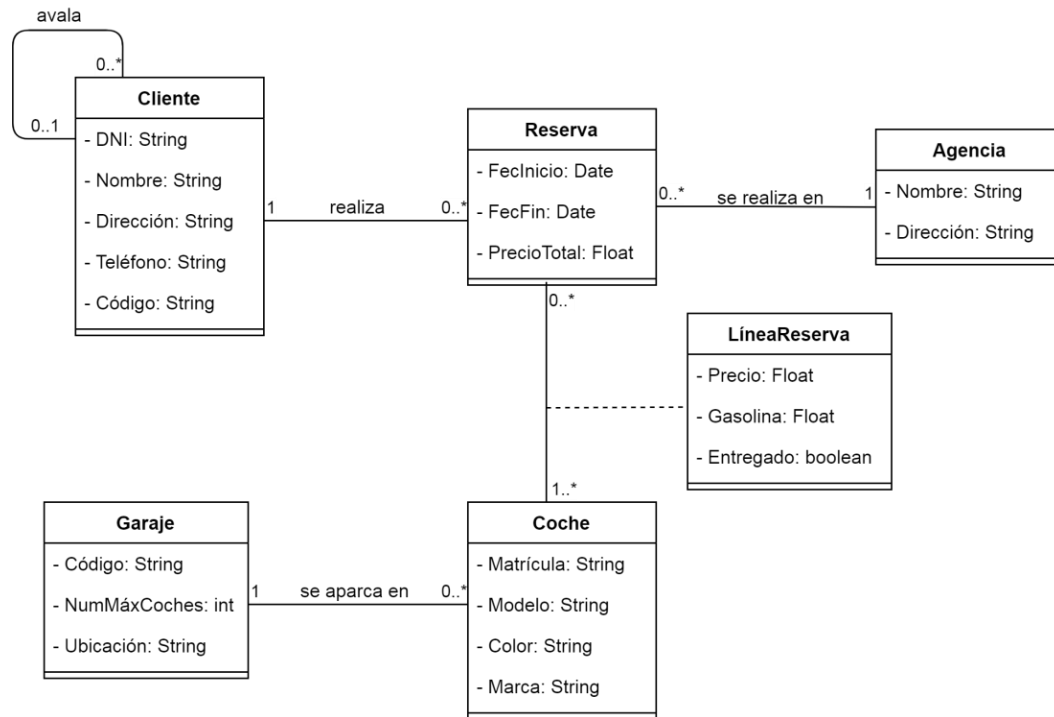


Figura 5.4. Diagrama de clases para una empresa dedicada al alquiler de vehículos.

Actividad de aplicación 2.20

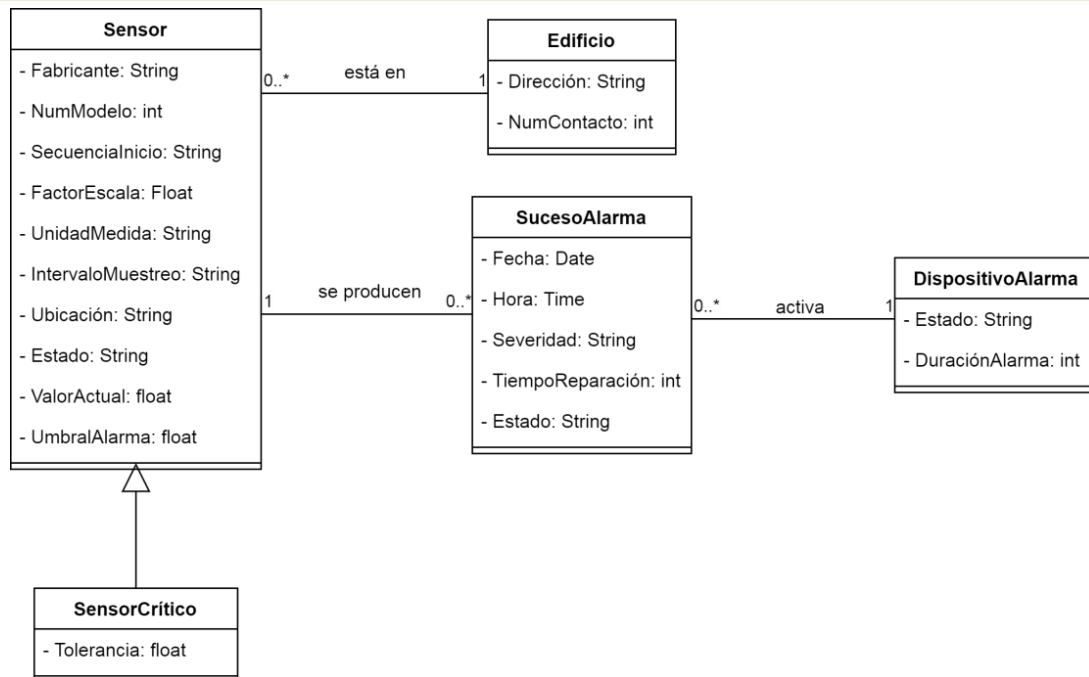


Figura 5.5. Diagrama de clases para un sistema de monitorización de sensores.

Actividad de aplicación 5.21

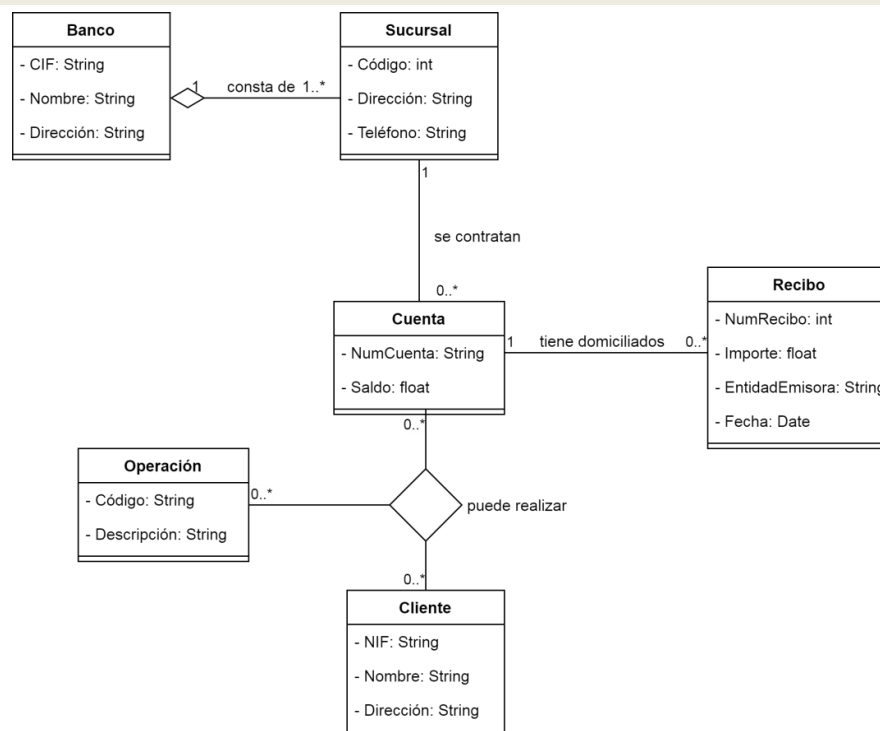


Figura 5.6. Diagrama de clases para una entidad bancaria.

Actividad de aplicación 5.22

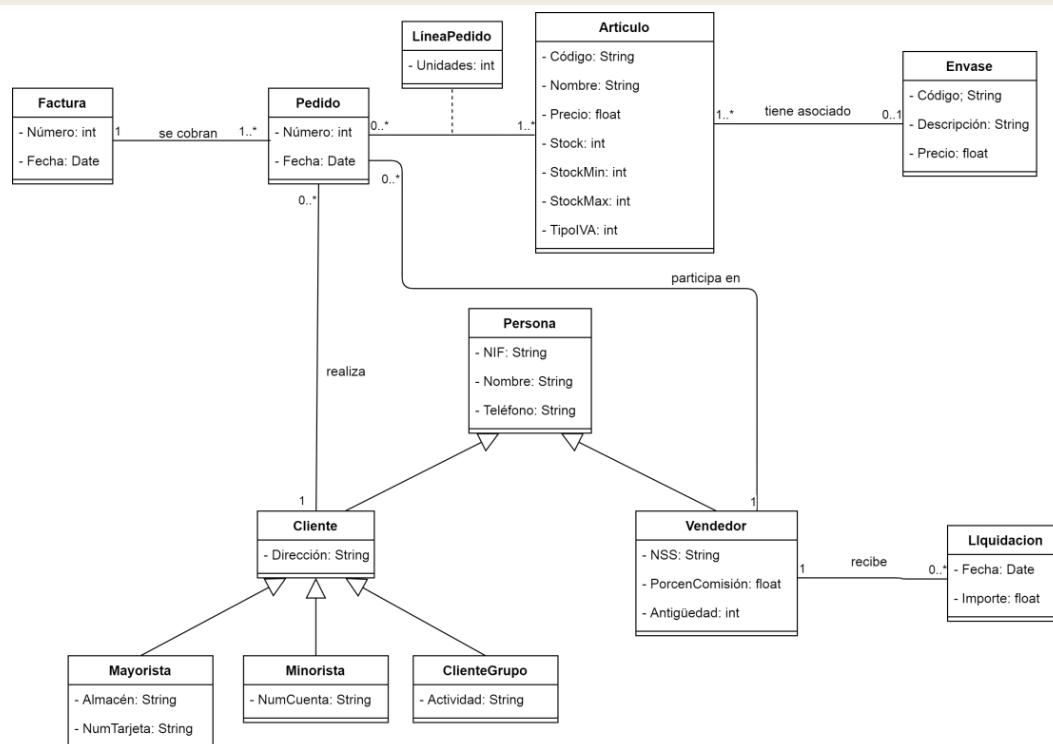


Figura 5.7. Diagrama de clases para una empresa de venta al por mayor.

Actividades de ampliación

Actividad de ampliación 5.23

Se puede encontrar información sobre roles en los diagramas de clases en la web https://www.ctr.unican.es/asignaturas/mc_oo/doc/m_estructural.pdf (página 20), donde se indica que un rol o papel describe el significado de la clase en términos de la asociación, es decir, el papel que desempeña esa clase en esa asociación. Así, por ejemplo, en una relación binaria entre *Persona* y *Coche* con el nombre *conduce*, la clase *Persona* juega el papel de *conductor* y la clase *Coche* desempeña el rol de *coche de la empresa*. Los roles se suelen especificar sobre todo para las relaciones reflexivas (relaciones entre varios objetos de la misma clase). Por ejemplo, en la relación reflexiva *es padre de* de la clase *Persona*, uno de los objetos juega el papel de *padre* o *madre* y el otro, el de *hijo*.

Actividad de ampliación 5.24

Se puede encontrar información sobre las relaciones de dependencia entre clases en la web <http://ingesis22.blogspot.com/2016/01/diagramas-de-clases.html>, donde se indica que en una relación de dependencia una clase es instanciada desde la otra, esto es, desde una clase se usa la otra, es decir, se crean objetos de la otra para su funcionamiento. Por ejemplo, la clase *Aplicación* (que representa una aplicación gráfica) hace uso o depende de la clase *Ventana*, ya que crea objetos de la clase *Ventana* para funcionar. Un cambio en la clase empleada (*Ventana*, en este caso) puede afectar a la clase que la usa (*Aplicación*, en este caso).

Otro ejemplo típico de relación de dependencia es el que se establece entre la clase *Ecuación* y la clase *Math*, debido a que para obtener el resultado de una ecuación de 2º grado es necesario hacer uso de las funciones *pow* (potencia) y *sqrt* (raíz cuadrada de la clase *Math*).

Estas relaciones se representan mediante una línea con trazado discontinuo y con punta de flecha en la clase utilizada, como se puede ver en la siguiente figura para los ejemplos indicados anteriormente:

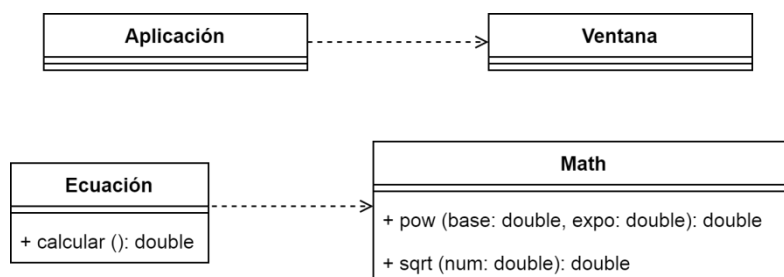


Figura 5.8. Representación de dos relaciones de dependencia entre clases.

Actividad de ampliación 5.25

Es posible encontrar información sobre la restricción *xor* en diagramas de clases en <https://stackoverflow.com/questions/40866828/how-works-xor-constraint-in-uml>. Esta restricción sirve para indicar, cuando hay dos relaciones entre dos clases, que un objeto de una de las clases solo puede participar en una de las relaciones, pero no en las dos. Así, en el siguiente diagrama, la restricción *xor* se debe interpretar como que un coche solo puede pertenecer a una persona o a una

empresa, pero no a una persona y a una empresa a la vez. Como se puede observar, esta restricción se representa uniendo las líneas correspondientes a las dos relaciones mediante una línea con trazado discontinuo y escribiendo al lado de esta línea {xor}.

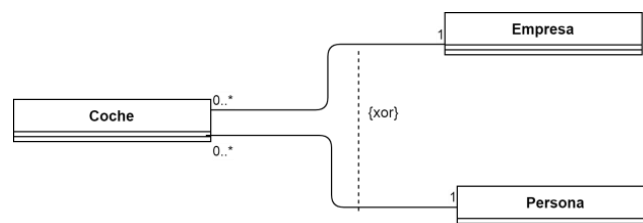


Figura 5.9. Representación de una restricción *xor* entre dos relaciones.

Actividad de ampliación 5.26

Podemos encontrar información sobre la restricción *subset* en diagramas de clases en la web <https://stackoverflow.com/questions/20426052/what-subset-constraint-means-in-the-uml-class-diagram>. Esta restricción sirve para indicar que una relación es un subconjunto de otra relación. En la siguiente figura, existen dos relaciones entre las clases *Departamento* y *Empleado*, una de las cuales indica todos los empleados que trabajan en un departamento y la otra cuál de esos empleados es su director. Pues bien, la línea punteada con la leyenda {subset} indica que el director del departamento también trabaja en el mismo departamento que dirige:

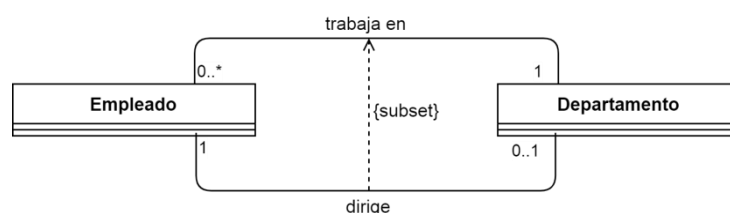


Figura 5.10. Representación de una restricción *subset* entre dos relaciones.

Actividad de ampliación 5.27

Podemos encontrar información sobre la restricción *ordered* en la web <https://users.monash.edu/~jonmc/CSE2305/Topics/09.17.OODesign3/html/text.html>. Esta restricción sirve para indicar, que el orden es relevante en los elementos de una relación. Así, la leyenda {ordered} en la relación que se muestra en la siguiente figura indica que un polígono consta de una serie de puntos (3 o más) en un orden determinado, que podría ser, por ejemplo, comenzando por el extremo superior izquierdo y siguiendo el orden de las agujas del reloj:

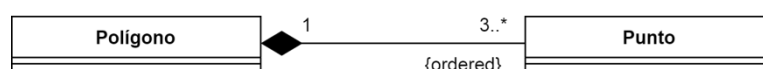


Figura 5.11. Representación de una restricción *ordered* en una relación.

Actividad de ampliación 5.28

Podemos encontrar información sobre este tema en la web http://ayudasydemascosas.blogspot.com/2016/05/modelado-de-datos-con-uml_12.html.

Una generalización es disjunta (*disjoint*) si un objeto de la superclase solo puede corresponderse con un objeto de una de las subclases, esto es, si las subclases son excluyentes entre sí. La generalización será solapada (*overlapping*) si un objeto de la superclase se puede corresponder con objetos de varias subclases, lo que quiere decir que las subclases son compatibles. Dada la relación de generalización/especialización de la figura 5.20, esta relación debe ser disjunta, ya que una cuenta bancaria no puede ser a la vez cuenta corriente y cuenta de ahorro.

Una generalización es total o completa (*complete*) si un objeto de la superclase se corresponde siempre con un objeto de una subclase o varios objetos de varias subclases, esto es, si las subclases representan todas las opciones posibles. Aplicado al ejemplo, la generalización sería total si solo puede haber dos tipos de cuentas (cuentas corrientes y cuentas de ahorro), de forma que no puede haber objetos de la clase *Cuenta* que no tengan ningún objeto de las subclases asociado. La generalización sería parcial si, además de cuentas corrientes y cuentas de ahorro, pudiese haber otro tipo de cuentas, de forma que habría objetos de la clase *Cuenta* sin ningún objeto de ninguna subclase asociado.

Se pueden indicar, como se muestra en la siguiente figura, entre llaves las características de la relación de generalización/especialización.

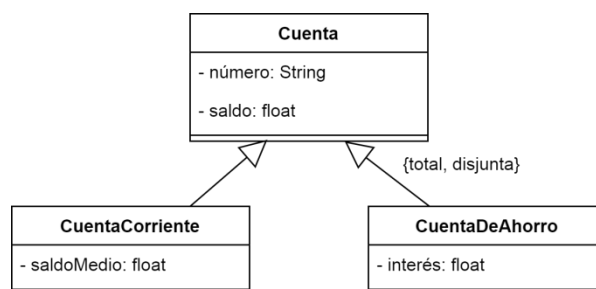


Figura 5.12. Representación de una relación de generalización/especialización con indicación de sus características (total y disjunta).

Entornos de desarrollo

Solucionario Unidad 6

Elaboración de diagramas de comportamiento

Actividades propuestas

Actividad propuesta 6.1

Caso de uso “Consultar cena”

- Actores involucrados: usuario.
- Pre-condición: la agenda tiene que estar creada y deben existir cenas.
- Flujo básico:
 1. El usuario solicita consultar una cena.
 2. Se pide al usuario la introducción del año de la cena.
 3. El sistema comprueba que haya una cena ese año y, si es así, se muestran los siguientes datos de la cena: año, lugar y nombre del organizador.
 4. Se vuelve a solicitar al usuario la selección de una operación.
- Caminos alternativos:
 - Si en el paso 3 se detecta que no existe ninguna cena ese año, se muestra un mensaje que el usuario debe aceptar y se pasa al paso 4.
- Post-condición: el estado del sistema no cambia.

Caso de uso “Eliminar asistente”

- Actores involucrados: usuario.
- Pre-condición: la agenda tiene que estar creada y deben existir cenas.
- Flujo básico:
 1. El usuario solicita eliminar un asistente a una cena.
 2. Se pide al usuario la introducción del año de la cena.
 3. El sistema comprueba que haya una cena ese año.
 4. Se solicita al usuario la introducción del nombre de un excompañero.
 5. El sistema comprueba que el excompañero figure entre los asistentes a la cena.
 6. El sistema elimina la relación entre la cena y el excompañero indicado y muestra un mensaje de confirmación.
 7. Se vuelve a solicitar al usuario la selección de una operación.
- Caminos alternativos:
 - Si en el paso 3 se detecta que no existe ninguna cena ese año, se muestra un mensaje que el usuario debe aceptar y se pasa al paso 7.

- Si en el paso 5 se detecta que no figura el excompañero entre los asistentes a la cena, se muestra un mensaje que el usuario debe aceptar y se pasa al paso 7.
- Post-condición: en la cena indicada figura un asistente menos o los mismos en caso de que no se haya podido eliminar el asistente a la cena.

Caso de uso “Consultar asistentes”

- Actores involucrados: usuario.
- Pre-condición: la agenda tiene que estar creada y deben existir cenas.
- Flujo básico:
 1. El usuario solicita consultar los asistentes a una cena.
 2. Se pide al usuario la introducción del año de la cena.
 3. El sistema comprueba que haya una cena ese año.
 4. El sistema muestra los nombres de los asistentes a la cena.
 5. Se vuelve a solicitar al usuario la selección de una operación.
- Caminos alternativos:
 - Si en el paso 3 se detecta que no existe ninguna cena ese año, se muestra un mensaje que el usuario debe aceptar y se pasa al paso 5.

Post-condición: el estado del sistema no cambia.

Actividad propuesta 6.2

El caso de uso “Consultar cena” se inicia solicitando el usuario consultar una cena, la ventana obtiene el año de la cena y pide a la clase de control que consulte los datos de la cena celebrada ese año. A continuación, se comprueba si existe alguna cena para ese año comparando el año de la cena que se desea consultar con los años de las cenas existentes. En caso de que no haya ninguna cena para ese año, finaliza el caso de uso; en caso contrario, se muestran los datos de la cena, obteniendo el nombre del organizador de la cena para ello.

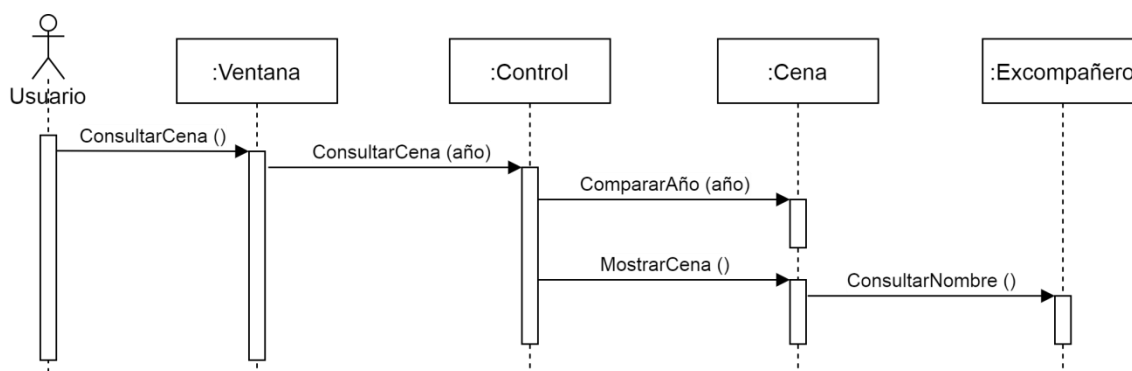


Figura 6.1. Diagrama de secuencia correspondiente al caso de uso “Consultar cena”.

El caso de uso “Eliminar asistente” se inicia solicitando el usuario eliminar un asistente a una cena, la ventana obtiene el año de la cena y pide a la clase de control que consulte si hay cena ese año. La clase de control comprueba si existe alguna cena para ese año comparando el año de la cena que se desea consultar con los años de las cenas existentes. En caso de que no haya ninguna cena para ese

año, finaliza el caso de uso; en caso contrario, la ventana obtiene el nombre del asistente a la cena que se desea eliminar y pide a la clase de control eliminar dicho asistente. Esta clase de control solicita buscar dicho asistente entre los asistentes a la cena comparando el nombre del asistente buscado con el nombre de cada asistente a la cena. En caso de que coincida, se elimina al asistente de la lista de asistentes a la cena.

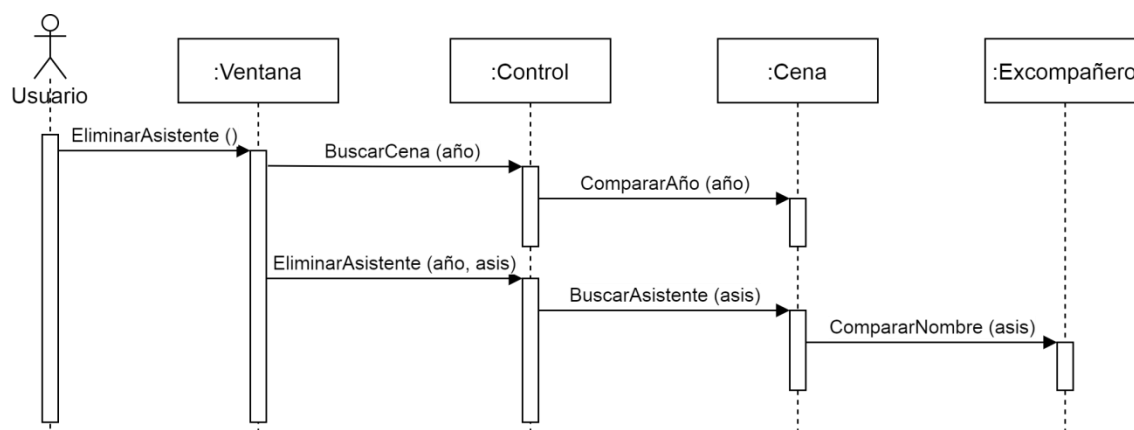


Figura 6.2. Diagrama de secuencia correspondiente al caso de uso “Eliminar asistente”.

El caso de uso “Consultar asistentes” se inicia solicitando el usuario consultar los asistentes a una cena, la ventana obtiene el año de la cena y pide a la clase de control que consulte los asistentes a la cena celebrada ese año. La clase de control busca una cena para ese año, para lo que compara el año de la cena que se desea consultar con los años de las cenas existentes. En caso de que no haya ninguna cena para ese año, finaliza el caso de uso. En caso contrario, se pide mostrar los asistentes a la cena mediante el envío de un mensaje a la cena correspondiente. Esta, por cada asistente a la cena, solicita mostrar su nombre a la clase *Excompañero*.

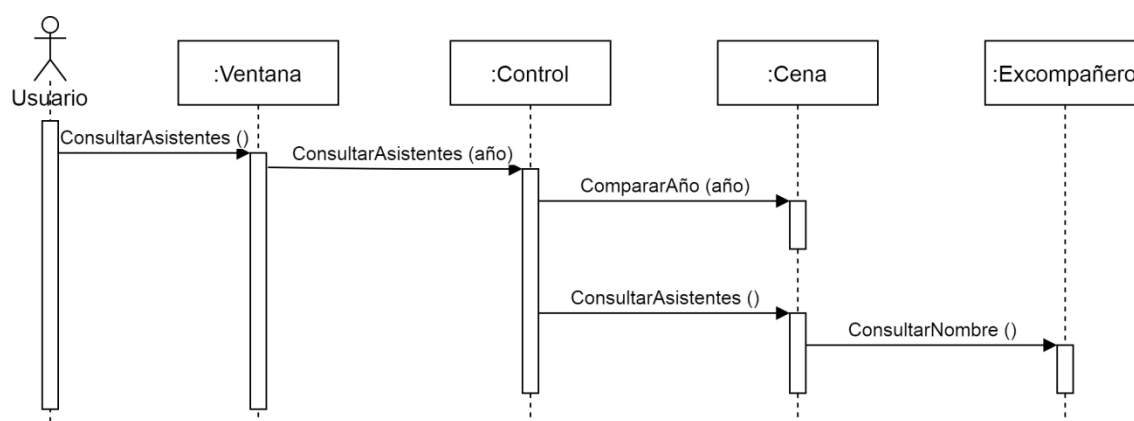


Figura 6.3. Diagrama de secuencia correspondiente al caso de uso “Consultar asistentes”.

A raíz de estos diagramas de secuencia podemos ampliar el diagrama de clases de la Figura 6.14 del libro de texto con los métodos que recibe cada clase, quedándonos el siguiente diagrama:

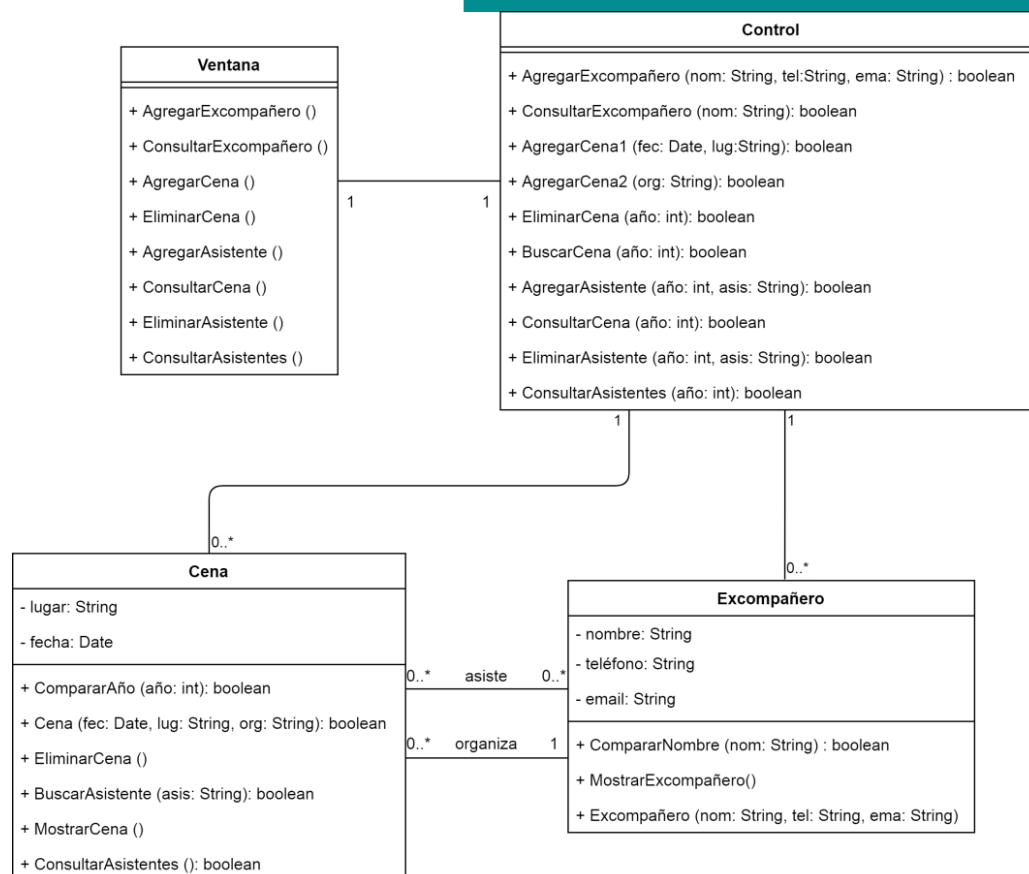


Figura 6.4. Diagrama de clases que incluye los métodos correspondientes a los diagramas de secuencia para los casos de uso “Consultar cena”, “Eliminar asistente” y “Consultar asistentes”.

Actividad propuesta 6.3

La ventana parte de un estado llamado *En espera* hasta que el usuario selecciona una de las opciones del menú.

En el momento en que se selecciona la opción de menú “Consultar excompañero”, pasa al estado *Comprobando excompañero* con el fin de determinar si el excompañero que se pretende consultar está en la agenda. Si el excompañero no existe, se muestra un mensaje de error y se vuelve al estado de espera. En caso de que el excompañero esté en la agenda, se muestran sus datos y se vuelve al estado de espera.

Si se selecciona la opción de menú “Eliminar cena”, pasa al estado *Comprobando cena* con el fin de determinar si ya hay una cena ese año. Si no la hay, se muestra un mensaje de error y se vuelve al estado de espera. En caso de que haya cena ese año, se procede a su eliminación y se vuelve al estado de espera.

Si en estado de espera se selecciona la opción de menú “Salir”, se pasa al estado final.

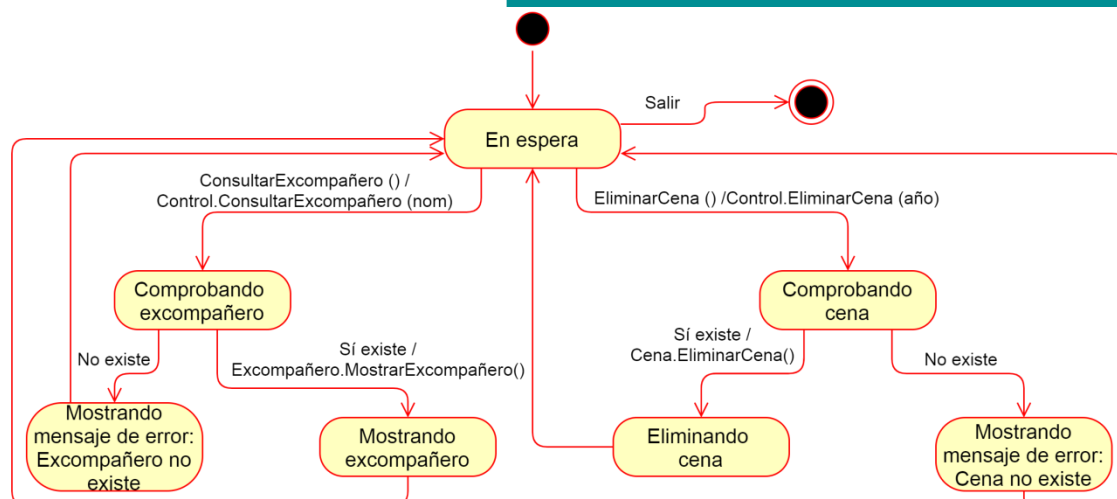


Figura 6.5. Diagrama de estados que representa los estados por los que pasa la clase *Ventana* en la agenda de excompañeros en relación con los casos de uso “Consultar excompañero” y “Eliminar cena”.

ACTIVIDADES FINALES

Actividades de comprobación

6.1	6.2	6.3	6.4	6.5	6.6	6.7	6.8	6.9	6.10	6.11	6.12	6.13
d)	a)	d)	a)	a)	c)	b)	d)	c)	b)	d)	a)	c)

Actividades de aplicación

Actividad de aplicación 6.14

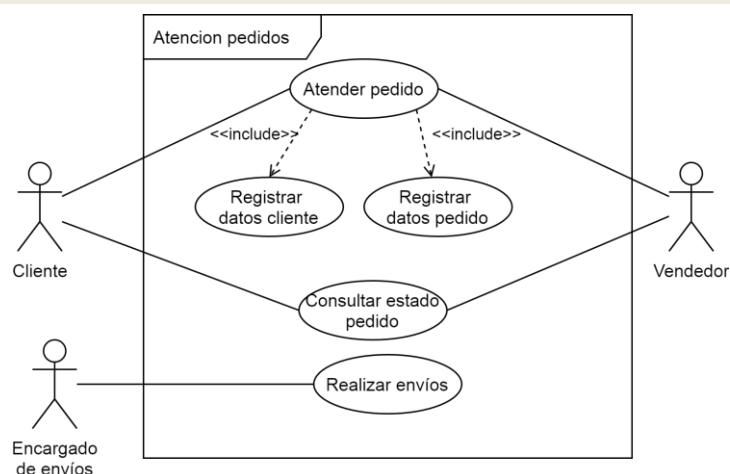
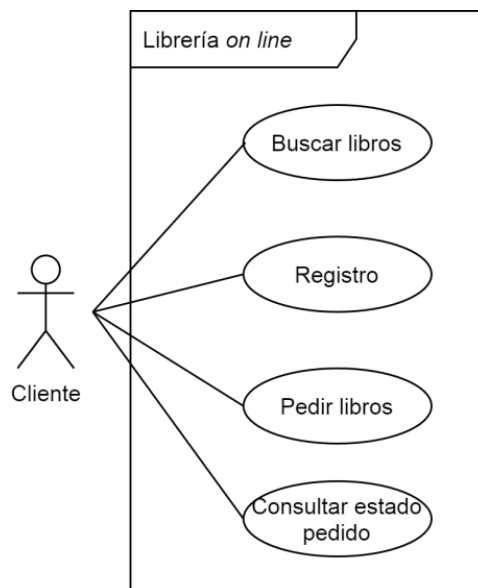


Figura 6.6. Diagrama de casos de uso para la gestión de pedidos de una empresa.

Actividad de aplicación 6.15

Figura 6.7. Diagrama de casos de uso para una librería *on line*.

Actividad de aplicación 6.16

- Actores implicados: cliente.
- Pre-condición: el cliente no debe estar registrado ya.
- Flujo básico:
 1. El cliente solicita el tipo de operación (registro).
 2. El cliente introduce sus datos personales (NIF, nombre y apellidos, domicilio completo, correo electrónico), nombre de usuario, contraseña y preferencias literarias.
 3. Si se han introducido todos los datos correctamente, el sistema comprueba que el cliente no esté dado de alta. Si es así, se almacenan los datos del cliente en el sistema.
 4. Se envía a la dirección de correo electrónico del cliente un *email* confirmándole el registro.
- Caminos alternativos:
 - a) Si el usuario deja en blanco alguno de los datos solicitados en el paso 2 o introduce un NIF incorrecto o un correo electrónico incorrecto, se muestra un mensaje de error en pantalla que tiene que ser aceptado por el usuario y se permite modificar los datos introducidos.
 - b) En el paso 3, si el usuario ya está dado de alta, se muestra un mensaje de error que tiene que aceptar el usuario y no se deja continuar.
- Post-condición: hay un usuario más en el sistema o los mismos usuarios si el usuario ya estaba dado de alta o se produjo algún error.

Actividad de aplicación 6.17

- Actores implicados: cliente.
- Pre-condición: el cliente se tiene que haber registrado en el sistema y debe haber incluido en el carrito de la compra uno o varios libros.
- Flujo básico:
 1. El cliente solicita el tipo de operación (pedir libro).
 2. Se solicita la introducción de un nombre de usuario y contraseña.
 3. Si el usuario y contraseña son correctos, se muestran en pantalla los datos de los libros solicitados y la dirección de envío del pedido, pudiendo esta ser modificada.
 4. Se solicita al usuario para proceder al pago la introducción de los siguientes datos: tipo de tarjeta, número de tarjeta, fecha de caducidad y CVV.
 5. Si los datos de cobro introducidos son correctos, se procede al cobro del pedido y se muestra un número de pedido en pantalla para posteriores consultas.
 6. Se envía una factura al e-mail del cliente.
- Caminos alternativos:
 - a) En el paso 1, si el carrito está vacío, se muestra un mensaje de error y no se deja continuar.
 - b) En el paso 2, si el usuario no introduce un nombre de usuario y/o contraseña correctos, se muestra un mensaje de error y se permiten modificar los datos. Se proporcionan enlaces para recuperar la contraseña y/o el nombre de usuario.
 - c) En el paso 4, si alguno de los datos de la tarjeta no son correctos, se muestra un mensaje de error y se permite volver a introducirlos.
- Post-condición: hay un pedido más en el sistema si los datos introducidos por el usuario fueron correctos, o hay los mismos pedidos, en caso contrario.

Actividad de aplicación 6.18

Al permitir registrarse el usuario cuando va a pedir libros, se puede considerar el caso de uso “Registro” como una extensión del caso de uso “Pedir libros”.

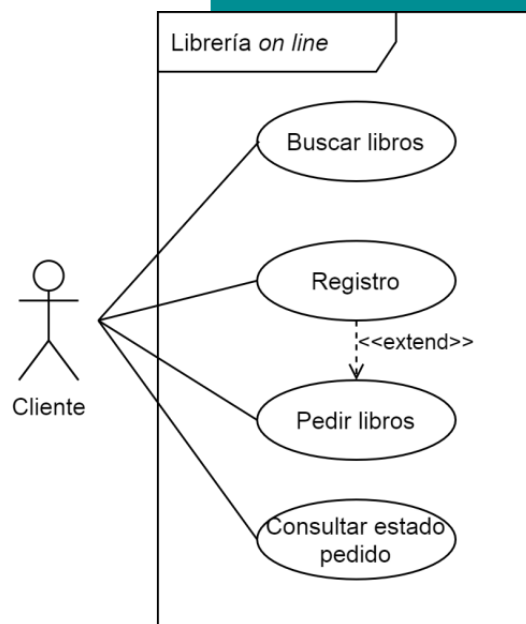


Figura 6.8. Diagrama de casos de uso para una librería *on line* considerando el caso de uso “Registro” una extensión del caso de uso “Pedir libros”.

Actividad de aplicación 6.19

Si tenemos en cuenta que tanto para pedir libros como para consultar el estado de un pedido el usuario se debe identificar suministrando su nombre de usuario y contraseña, podemos crear un caso de uso “Identificación” que es común a los casos de uso “Pedir libros” y “Consultar estado pedido”, por lo que podemos indicar que estos dos casos de uso incluyen o usan el caso de uso “Identificación”, como se muestra en el siguiente diagrama de casos de uso:

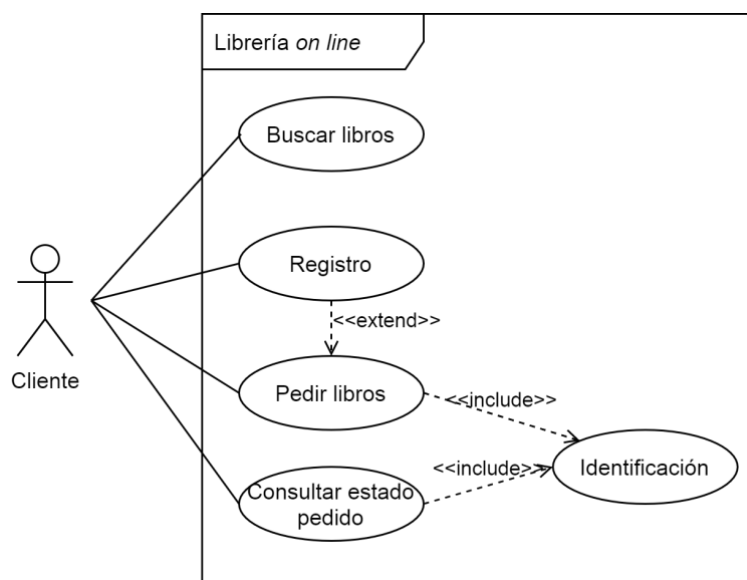


Figura 6.9. Diagrama de casos de uso para una librería *on line* considerando que el caso de uso “Identificación” está incluido dentro de los casos de uso “Pedir libros” y “Consultar estado pedido”.

Actividad de aplicación 6.20

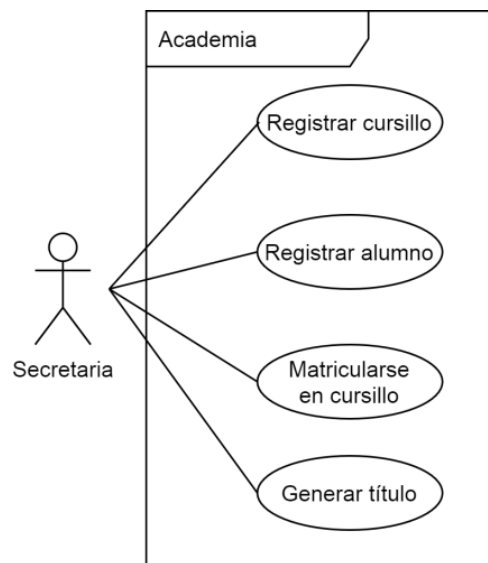


Figura 6.10. Diagrama de casos de uso para una academia de baile.

Actividad de aplicación 6.21

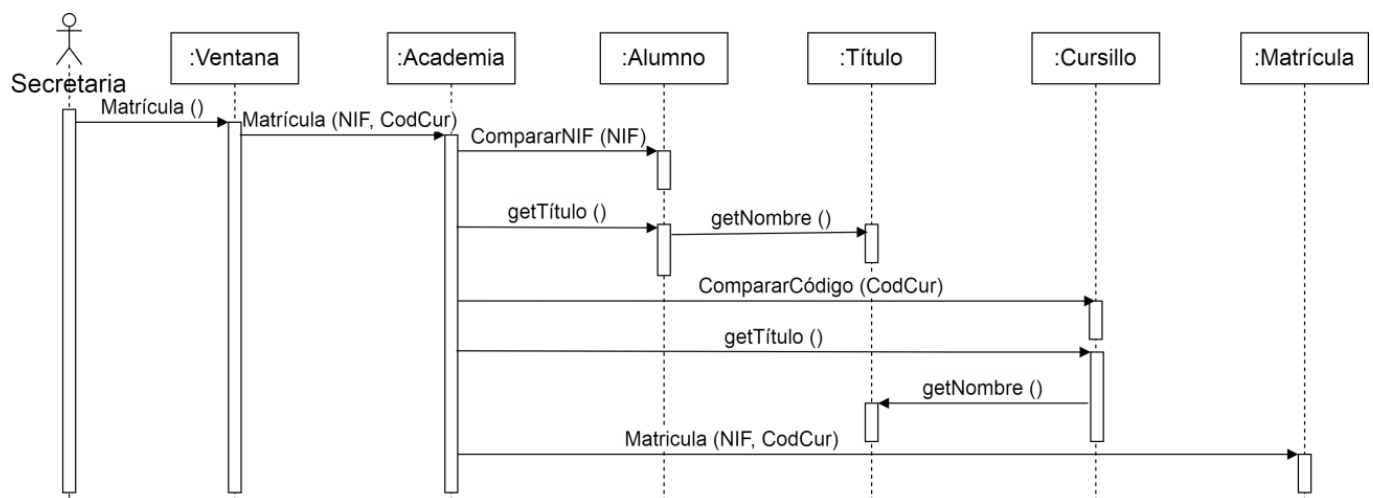


Figura 6.11. Diagrama de secuencia para el caso de uso “Matricularse en cursillo”.

Actividad de aplicación 6.22

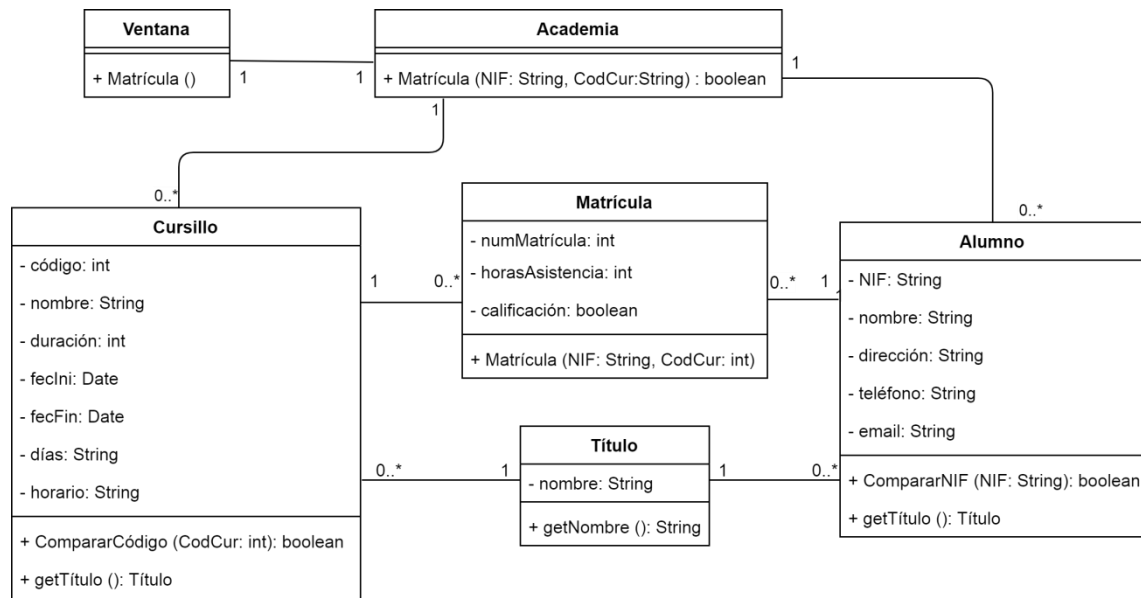


Figura 6.12. Diagrama de clases para la academia incluyendo los métodos correspondientes al caso de uso “Matricularse en cursillo”.

Actividad de aplicación 6.23

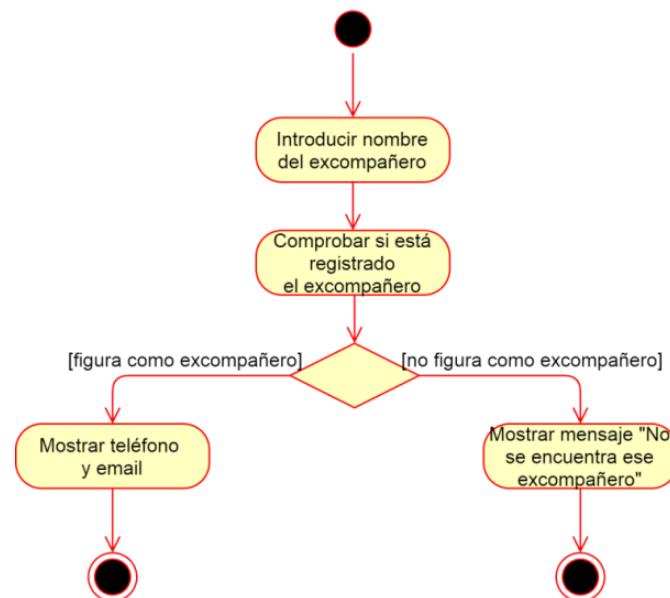


Figura 6.13. Diagrama de actividades que muestra el flujo de control para el caso de uso “Consultar excompañero”.

Actividad de aplicación 6.24

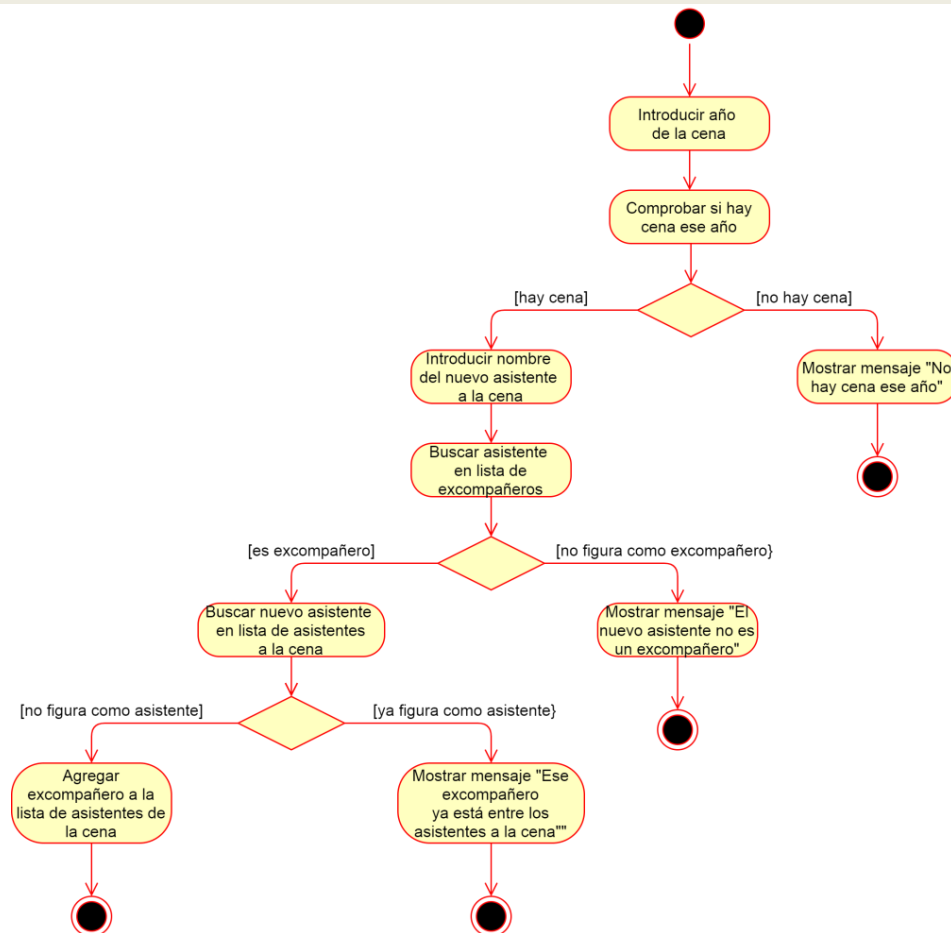


Figura 6.14. Diagrama de actividades que muestra el flujo de control para el caso de uso “Agregar asistente”.

Actividad de aplicación 6.25

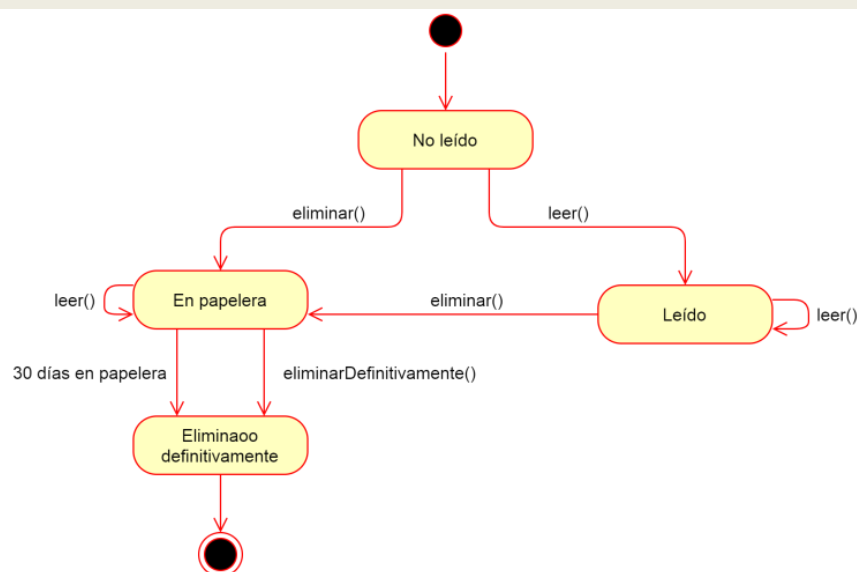


Figura 6.15. Diagrama de clases que refleja los estados por los que puede pasar un mensaje de correo electrónico.

Actividad de aplicación 6.26

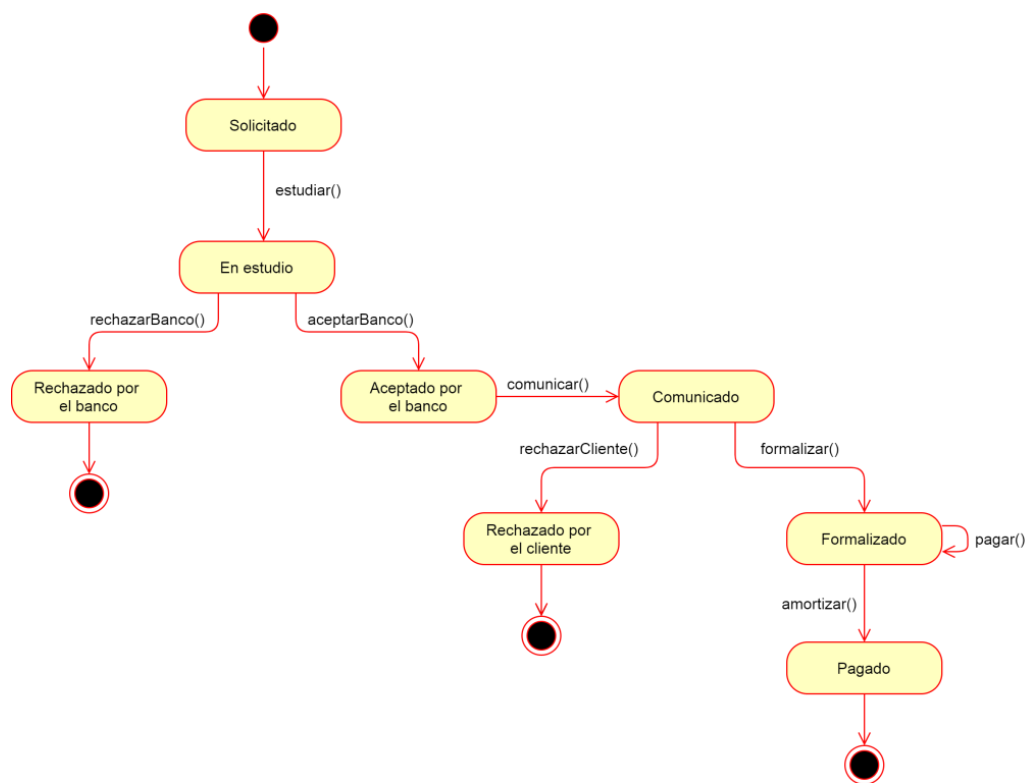


Figura 6.16. Diagrama de estados que refleja los estados por los que puede pasar un préstamo solicitado a una entidad bancaria.

Actividades de ampliación

Actividad de ampliación 6.27

En la web <https://creately.com/blog/es/diagramas/tutorial-del-diagrama-de-secuencia/#Best> se indica que la forma de representar una estructura alternativa simple es englobando dentro de un marco rectangular con la etiqueta *opt* el mensaje o los mensajes que solo se han de enviar en caso de que se cumpla la condición especificada entre corchetes.

Actividad de ampliación 6.28

En la web <https://creately.com/blog/es/diagramas/tutorial-del-diagrama-de-secuencia/#Best> se indica que la forma de representar una estructura alternativa doble o múltiple es englobando dentro de un marco rectangular con la etiqueta *alt* toda la estructura y dividiéndola en varias partes con una línea con trazado discontinuo para cada una de las ramas, esto es, en dos si se trata de una estructura alternativa doble (*if ... else ...*) o en más si es múltiple. Para cada rama se debe indicar entre corchetes la condición bajo la cual se deben enviar los mensajes, o bien *else* en caso de que sea la rama final.

Actividad de ampliación 6.29

En la web <https://creately.com/blog/es/diagramas/tutorial-del-diagrama-de-secuencia/#Best> se indica que la forma de representar un bucle es englobando dentro de un marco rectangular con la etiqueta *loop* el mensaje o los mensajes que se han de enviar varias veces mientras que se cumpla una condición, la cual debe ser especificada entre corchetes.

Actividad de ampliación 6.30

Hemos de tener en cuenta que el mensaje *ComparaNombre ()* enviado desde la clase *Control* a *Excompañero* tiene como objetivo encontrar a un excompañero con el mismo nombre del que se desea añadir, ya que en ese caso no se debe permitir agregarlo, motivo por el cual este mensaje solo se ha de enviar mientras no se encuentre un excompañero con ese nombre. Dado que se trata de una estructura repetitiva, englobaremos este mensaje dentro de un marco con la etiqueta *loop* y la condición de que no se haya encontrado a un excompañero con ese nombre.

Por otro lado, solo se ha de crear un nuevo excompañero en caso de que no se haya encontrado uno con el mismo nombre del que se desea agregar, motivo por el cual la creación del excompañero nuevo se ha de colocar dentro de un marco con la etiqueta *opt* y la condición de que no se haya encontrado ningún excompañero con ese nombre. El diagrama de secuencia nos quedaría como se muestra a continuación:

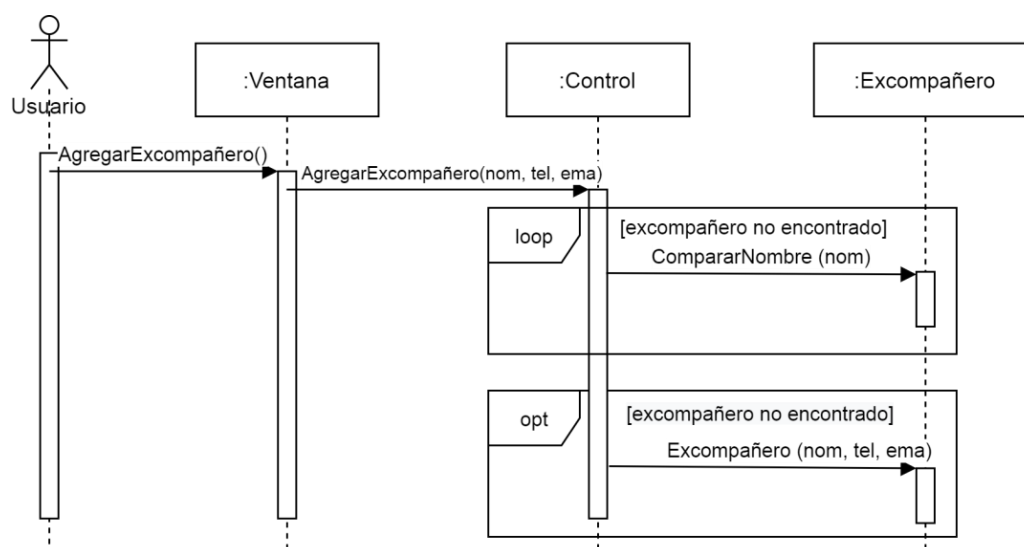


Figura 6.17. Diagrama de secuencia correspondiente al caso de uso “Agregar excompañero” con extensiones para el uso de una estructura alternativa simple y una estructura repetitiva.

Actividad de ampliación 6.31

Hemos de tener en cuenta que el mensaje *CompararAño ()* enviado desde la clase *Control* a *Cena* tiene como objetivo encontrar una cena para el mismo año de la que se desea añadir, ya que en ese caso no se debe permitir agregarla, motivo por el cual este mensaje solo se ha de enviar mientras no se encuentre una cena para ese año. Dado que se trata de una estructura repetitiva, englobaremos este mensaje dentro de un marco con la etiqueta *loop* y la condición de que no se haya encontrado una cena para ese año.

Por otro lado, solo se ha de continuar con la ejecución del caso de uso en caso de que no se haya encontrado una cena para el año recibido como parámetro, motivo por el cual el mensaje *AgregarCena2 ()* se ha de colocar dentro de un marco con la etiqueta *opt* y la condición de que no se haya encontrado ninguna cena para el año recibido como parámetro,

Por otro lado, a la hora de agregar una cena hay que realizar otra comprobación, consistente en que el organizador de la cena figure como excompañero, por lo que el mensaje *CompararNombre ()* enviado desde la clase *Control* a *Excompañero* tiene como objetivo encontrar a un excompañero con el nombre del organizador de la cena, ya que si no se encuentra, no se debe permitir agregar la cena. Dado que se trata de una estructura repetitiva, englobaremos este mensaje dentro de un marco con la etiqueta *loop* y la condición de que no se haya encontrado a un excompañero con el nombre del organizador.

Solo en caso de que se encuentre a un excompañero con el nombre del organizador de la cena, se debe crear la cena, por lo que el mensaje de creación de la cena se debe colocar dentro de un marco con la etiqueta *opt* y la condición de que el organizador de la cena figure como excompañero.

El diagrama de secuencia nos quedaría como se muestra a continuación:

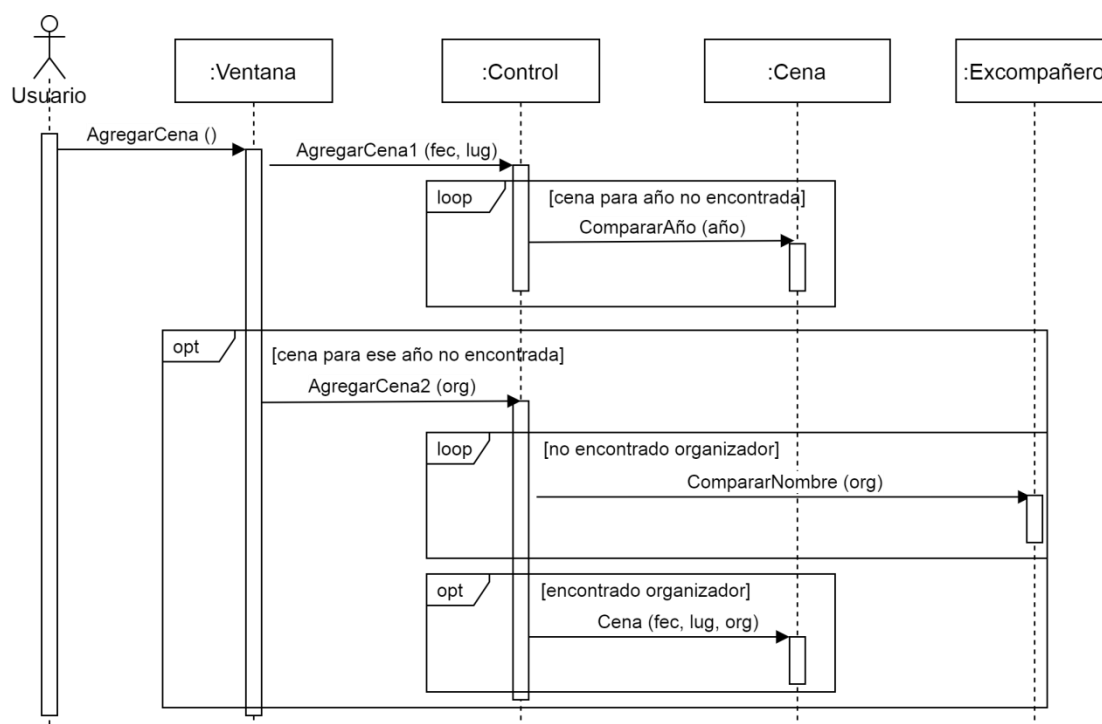


Figura 6.18. Diagrama de secuencia correspondiente al caso de uso “Agregar cena” con extensiones para el uso de estructuras alternativas y bucles.

Actividad de ampliación 6.32

En la web <https://www.uml-diagrams.org/state-machine-diagrams.html> se especifica en qué consisten los elementos *entry*, *exit* y *do*:

- *Entry* especifica la acción que se lleva a cabo nada más entrar en el estado.
- *Exit* especifica la acción que se lleva a cabo cuando se produce la salida del estado.

- *Do* especifica la acción o acciones que se llevan a cabo mientras se está en el estado, esto es, después de entrar en el estado y hasta que se sale de él.

Los estados detallados de esta forma se representan mediante un rectángulo con bordes redondeados y dos secciones: en la superior se escribe el nombre del estado y en la inferior se especifican los elementos *entry*, *exit* y *do* escribiendo su nombre, una barra y la acción que se lleva a cabo en cada caso. Se muestra a continuación esta representación genérica y un ejemplo de estado en el que se está esperando la introducción de datos por parte de un usuario. Al llegar a este estado se muestra un mensaje de bienvenida al usuario y al salir de él, un mensaje de agradecimiento.

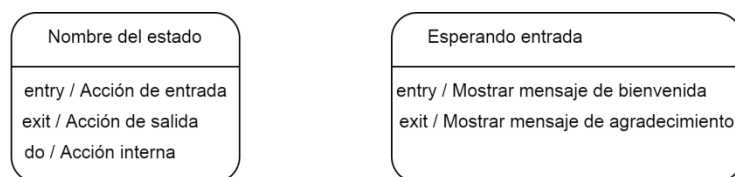


Figura 6.19. Representación detallada de un estado genérico y de un ejemplo de estado en el que se encuentra una aplicación mientras se espera la introducción de datos por parte de un usuario.