



# 1. Desarrollo de software

	Autor	(X) Xerach Casanova
	Clase	Entornos de desarrollo
	Fecha	@Mar 29, 2021 9:05 PM

## 1. Software y programa. Tipos de software

Software de sistema

Software de programación

Aplicaciones informáticas

## Relación hardware - software

Desde el punto de vista del S.O.

Desde el punto de vista de las aplicaciones.

## 3. Fases a seguir en el desarrollo del software

### 3.1. Análisis

Documento especificación de requisitos de software

Documento de diseño de arquitectura

### 3.2. Diseño

### 3.3. Codificación

### 3.4. Compilación

### 3.5. Pruebas

Pruebas unitarias

Pruebas de integración

### 3.6. Explotación

### 3.7. Mantenimiento

### 3.8. Documentación.

Guías técnicas

Guías de uso

Guías de instalación

## 4. Ciclo de vida del software

Modelos de ciclos de vida

## 5. Herramientas de apoyo al desarrollo del software.

## 6. Frameworks

## 7. Lenguajes de programación

### 7.1. Características de los lenguajes de programación.

Lenguaje máquina

Lenguaje ensamblador

Lenguajes de alto nivel basados en código

Lenguajes visuales

7.2. Clasificación de los lenguajes de programación.

Según lo cerca que esté del lenguaje humano

Lenguajes de programación de bajo nivel

Según su forma de ejecución

## **8. Máquinas virtuales**

Características de la máquina virtual

8.1 Entornos de ejecución

Funcionamiento del entorno de ejecución.

8.2 Java runtime environment

# **1. Software y programa. Tipos de software**

Los ordenadores se componen de dos partes bien diferenciadas: software y hardware. El software es el conjunto de programas informáticos que actúan sobre el hardware para ejecutar lo que el usuario desea. Conforme a esta definición, podemos dividirlo en tres:

## **Software de sistema**

Es el software base que ha de estar instalado en el ordenador para que las aplicaciones puedan ejecutarse y funcionar: sistema operativo: Windows, Linux, MacOs, etc...

## **Software de programación**

Conjunto de herramientas para desarrollar programas informáticos, tales como editores de código, IDE, Frameworks...

## **Aplicaciones informáticas**

Conjunto de programas que tienen una finalidad más o menos concretas, tales como procesadores de texto, programas de hojas de cálculo, reproductores de música, etc...

Un programa es un conjunto de instrucciones en algún lenguaje de programación que le indican a la máquina que operaciones realizar sobre determinados datos.

# **Relación hardware - software**

La relación entre el hardware y el software es inseparable. El software necesita del hardware para poder ejecutarse y el hardware necesita del software para poder funcionar.

### **Desde el punto de vista del S.O.**

Es el encargado de coordinar de manera oculta para el usuario y las apps, el hardware durante el funcionamiento del ordenador, haciendo de intermediario entre este y las aplicaciones. Así, se encargará de gestionar los recursos de hardware durante su ejecución: memoria RAM, tiempo de CPU, interrupciones...

### **Desde el punto de vista de las aplicaciones.**

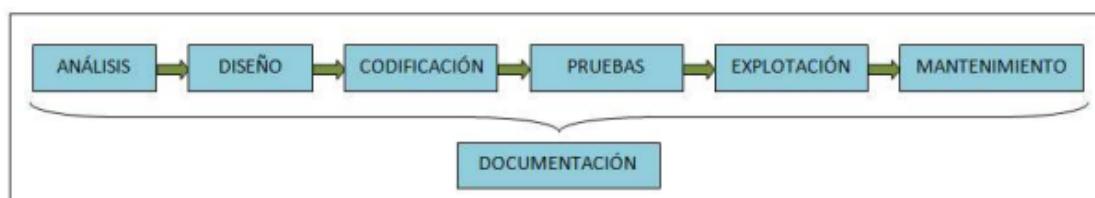
Una aplicación es un conjunto de programas escritos en algún lenguaje de programación que el hardware debe interpretar y ejecutar.

Todos los lenguajes de programación tienen en común que están compuestos de sentencias que el ser humano puede usar y entender fácilmente, mientras que el hardware funciona con código binario.

## **3. Fases a seguir en el desarrollo del software**

Se entiende por desarrollo de software todo el proceso que ocurre desde que es concebida la idea hasta que un programa está implementado en el ordenador funcionando.

Es importante dedicar todos los recursos necesarios a las primeras etapas de su desarrollo para evitar errores que se propaguen durante toda la vida del proyecto.



### **3.1. Análisis**

Es la primera fase y la de mayor importancia, ya que todo depende de lo bien que esta esté detallada.

En esta fase obtendremos dos salidas.

## Documento especificación de requisitos de software

- **Requisitos funcionales.** Definen las funciones del software o sus componentes, pueden ser cálculos, detalles técnicos, manipulación de datos, etc...
- **Requisitos no funcionales.** Complementan a los requisitos funcionales y se refieren a todos los requisitos que no describen información a guardar, ni funciones a realizar, sino características de funcionamiento. Como por ejemplo el cumplimientos de normativas y estándares de calidad, tiempos de respuesta del programa, etc...

## Documento de diseño de arquitectura

Contiene la descripción de la estructura relacional del sistema, la especificación de todas sus partes y la manera en que se combinan todas ellas. También se suele implementar en la primera fase de diseño.

El analista y el cliente deben asegurar buena comunicación entre ellos y...

- Planificar las reuniones
- Relación de objetivos del usuario cliente y del sistema
- Relación de objetivos prioritarios y temporización
- Mecanismos de actuación ante contingencias
- etc...

## 3.2. Diseño

Una vez terminado el análisis y se definen los requisitos se divide el sistema en partes y establecer las relaciones entre ellas, decidiendo que hará exactamente cada parte, generando un módulo funcional - estructural de los requerimientos del sistema global.

En este punto también se deciden las entidades y relaciones de la BBDD, lenguaje de programación que se va a utilizar, el gestor de BBDD, etc...

Se obtendrán dos salidas:

- **Documento de diseño de software**
- **Plan de pruebas** a utilizar en la fase de pruebas pero sin entrar en detalles.

### 3.3. Codificación

Se elige un determinado lenguaje y se codifica todo lo expuesto en el análisis y diseño, obteniendo el código fuente.

Las características del código deben ser:

- **Modularidad** (dividido en trozos más pequeños)
- **Corrección** (que haga lo que se pide realmente)
- **Fácil de leer** (para facilitar su desarrollo y mantenimiento)
- **Eficiencia** (que haga buen uso de los recursos)
- **Portabilidad** (que se pueda implantar en cualquier equipo).

### 3.4. Compilación

Una vez realizado el código fuente, este debe ser traducido a lenguaje máquina, para que las computadoras sean capaces de entenderlo y ejecutarlo. Este proceso se puede realizar de dos formas.

- **Compilación.** La traducción se realiza sobre todo el código fuente en un solo paso a través de un compilador.
- **Interpretación.** La traducción del código se realiza línea a línea y se ejecuta simultáneamente a través del software responsable llamado intérprete.

### 3.5. Pruebas

La realización de pruebas es imprescindible para asegurar la validación y verificación del software construido.

#### Pruebas unitarias

Se prueban todas las partes del software una a una de manera independiente y se genera un documento de procedimiento de pruebas que viene desde la fase de diseño. Los resultados se comparan con los esperados que se habrán determinado de antemano.

#### Pruebas de integración

Consiste en la puesta en común de todos los programas desarrollados una vez pasadas las pruebas unitarias. Se genera un documento de procedimiento de pruebas de integración.

## 3.6. Explotación

La explotación es la fase en la que los usuarios finales conocen la aplicación y comienzan a utilizarla.

A menudo se consideran esta fase y la de mantenimiento como la misma fase.

Esta fase consiste en la instalación y configuración del software en el/los equipo/s del cliente, se asignan los parámetros de configuración y se prueba que la aplicación está operativa.

Puede ocurrir que la configuración la realicen los usuarios finales con la guía de instalación.

## 3.7. Mantenimiento

La etapa de mantenimiento es la más larga de todo el ciclo de vida del software.

Se define como el proceso de control, mejora y optimización del software.

Este debe actualizarse y evolucionar con el tiempo, adaptándose a las mejoras del hardware en el mercado y afrontar situaciones nuevas que no existían cuando se construyó.

Se pacta con el cliente un servicio de mantenimiento de la aplicación, que tendrá un coste temporal y económico.

Según el tipo se dividen en:

- **Perfectivos** (mejoran la funcionalidad del software).
- **Evolutivos** (el cliente propone mejoras para el producto e implica nuevos requisitos).
- **Adaptativos** (modificaciones y actualizaciones para adaptarse a las nuevas tendencias del mercado de hardware, condiciones específicas de organismos reguladores, etc.).
- **Correctivos** (resolver errores detectados).

## 3.8. Documentación.

Todas las etapas de desarrollo de software deben quedar perfectamente documentadas.

Debido a esto no es considerada como una etapa más del desarrollo del proyecto.

La documentación permite pasar de una etapa a otra de una manera clara y definida, además permite reutilización de los programas en otras aplicaciones si la codificación ha sido modular.

La documentación se divide en:

### Guías técnicas

Dirigidos a analistas y programadores.

**Quedan reflejados:** El diseño de la aplicación, codificación de programas y pruebas realizadas.

**Objetivo:** Facilita un correcto desarrollo y realizar correcciones y permitir mantenimiento en el futuro.

### Guías de uso

Dirigido a los usuarios finales o clientes.

**Quedan reflejados:** descripción de funcionalidad de la aplicación, forma de comenzar a ejecutarla, ejemplos de uso, requerimientos de software y solución a problemas.

**Objetivo:** Dar a los usuarios finales toda la información necesaria para utilizar la aplicación.

### Guías de instalación

Dirigido al personal informático responsable de la instalación

**Quedan reflejados:** aspectos de puesta en marcha, explotación y seguridad del sistema.

**Objetivo:** dar toda la información para garantizar la implantación de la aplicación.

## 4. Ciclo de vida del software

Es el conjunto de fases o etapas, procesos y actividades requeridas para ofrecer, desarrollar, probar, integrar, explotar y mantener un software.

El concepto de desarrollo del software aparece por la necesidad de los inconvenientes del proceso individualizado y carente de planificación del mismo.

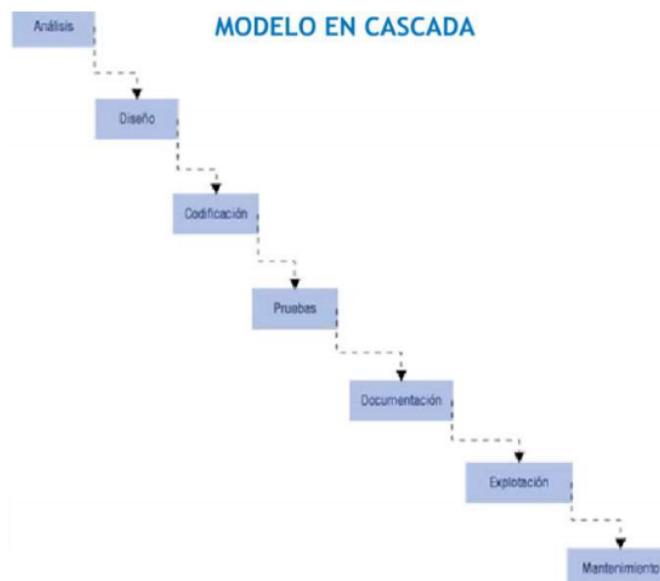
Ventajas:

- El análisis facilitará la codificación en un futuro a pesar de que en este presente en esta fase.
- Asegura un desarrollo progresivo que permite detectar defectos con antelación.
- Control de plazos de entrega y costes excesivos.
- Aumenta la visibilidad y posibilidad de control para la gestión del proyecto.
- Aporta una guía para el personal de desarrollo, marcando las tareas a realizar.
- Minimiza la necesidad de rehacer trabajo y problemas de puesta a punto.

## Modelos de ciclos de vida

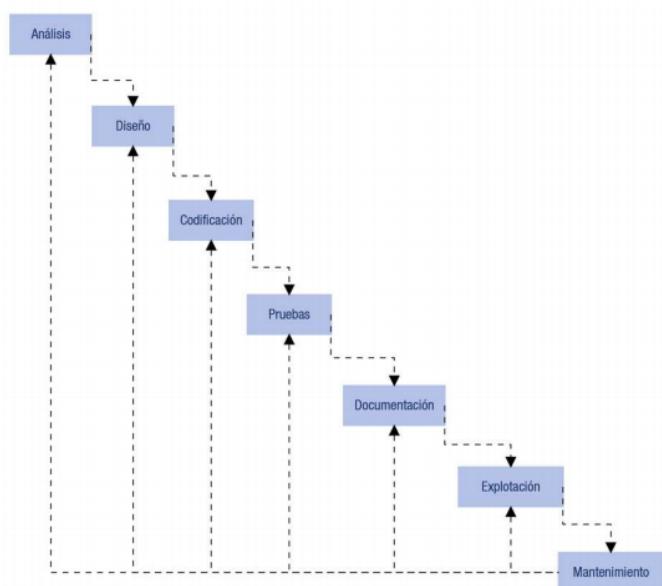
### En cascada

Se pasa de una etapa a otra sin retorno, cualquier error en fases iniciales no es subsanable más adelante. Requiere conocimiento absoluto previo de los requerimientos del sistema y no permite modificaciones ni actualizaciones.



## En cascada con realimentación

Uno de los más utilizado, igual que el modelo en cascada pero se introduce la realimentación entre etapas que sirven para introducir modificaciones o depurar errores. Ideal para proyectos rígidos con pocos cambios y poco evolutivos pero no para proyectos que prevén muchos cambios durante el desarrollo.

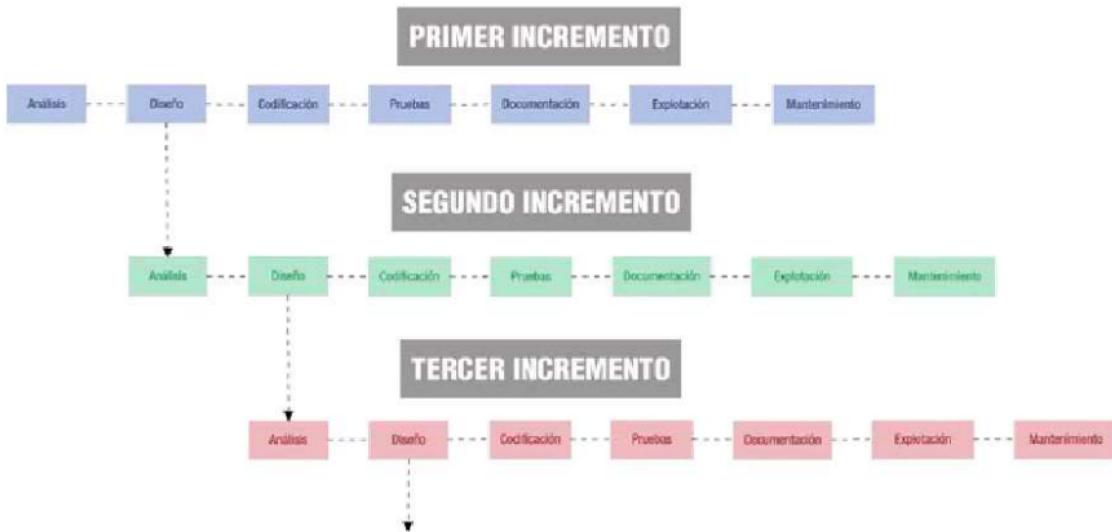


## Evolutivos

Tienen en cuenta la naturaleza cambiante del software.

- **Modelo interactivo incremental.**

Basado en el modelo en cascada con realimentación. Las fases de los ciclos se repiten en cascada y se refinan, de manera que se propagan sus mejoras a las fases siguientes. Estas generan versiones del producto cada vez más completas hasta llegar al producto final.



- **Modelo en espiral**

Es una combinación del modelo anterior con el modelo en cascada.

También se adapta a la evolución del software y reduce riesgos.

El software se construye repetidamente en forma de versiones que son cada vez mejores. Se divide en 6 zonas llamadas regiones de tareas:

- Comunicación con el cliente
- Análisis de riesgos
- Representación de la aplicación
- Codificación y explotación
- Evaluación del cliente.



## Modelos Ágiles

Modelo que ha ganado presencia en el desarrollo de software, centrado en la satisfacción del cliente, gran flexibilidad a la aparición de nuevos requisitos incluso durante el desarrollo de la aplicación. Se trata de desarrollo incremental pero con incrementos cortos y abiertos a que las fases se solapen unas con otras. Un tipo de metodología ágil es Scrum.

## 5. Herramientas de apoyo al desarrollo del software.

Las herramientas CASE (Computer Aided Software Engineering) es el conjunto de aplicaciones que se utiliza en el desarrollo de software con el objetivo de automatizar sus distintas fases.

Estas herramientas permiten:

- Mejorar la planificación del proyecto.
- Darle agilidad al proceso.
- Reutilizar partes del software.
- Hacer aplicaciones que responda a estándares.
- Mejorar la tarea de mantenimiento.
- Visualizar las fases de forma gráfica.

Las herramientas CASE se clasifican en función del ciclo de vida en la que ofrecen ayuda.

- **U-CASE:** fase de planificación y análisis.
- **M-CASE:** fase de análisis y diseño.
- **L-CASE:** ayuda en la programación, detección de errores, depuración, pruebas y documentación.

## 6. Frameworks

Es una estructura de ayuda al programador a desarrollar proyectos sin partir desde cero.

Son plataformas software donde están definidos programas, soporte, bibliotecas, lenguaje interpretado, etc. que ayudan a desarrollar los diferentes módulos o partes de un proyecto.

Ventajas de utilizar un framework:

- **Desarrollo rápido.**
- **Reutilización.**
- **Diseño uniforme.**
- **Portabilidad** (gracias a su ejecución sobre cualquier máquina virtual)

Por contra, uno de los inconvenientes es la dependencia del código al framework utilizado y los recursos del sistema que consume un framework en el equipo.

## 7. Lenguajes de programación

Un lenguaje de programación es aquel que nos permite comunicarnos con el hardware. Se trata de un conjunto de:

- **Alfabeto:** conjunto de símbolos permitidos.
- **Sintaxis:** normas de construcción permitidas.
- **Semántica:** significado de las construcciones para hacer acciones válidas.

### 7.1. Características de los lenguajes de programación.

#### Lenguaje máquina

- Código binario (unos y ceros).
- Es el único que entiende el hardware.
- Fue el primer lenguaje utilizado.
- Es único para cada procesador.
- Nadie programa en lenguaje máquina.

#### Lenguaje ensamblador

- Facilita la labor de programación en comparación al lenguaje máquina.

- Necesita traducción al lenguaje máquina para ejecutarse.
  - Utiliza mnemotécnicos (instrucciones complejas que hacen referencia a la ubicación física de los archivos en el equipo).
- Difícil de utilizar.

## Lenguajes de alto nivel basados en código

- Sustituyen al lenguaje ensamblador
- Utiliza sentencias y órdenes derivadas del inglés
- Necesita traducción al lenguaje máquina para ejecutarse
- Son utilizados hoy en día con tendencia a utilizarse cada vez menos.

## Lenguajes visuales

- Sustituyen a los lenguajes de alto nivel.
- Se programa gráficamente utilizando el ratón.
- El código se genera automáticamente.
- Necesitan traducción al lenguaje máquina para ejecutarse.
- Son portables.

## 7.2. Clasificación de los lenguajes de programación.

La elección del lenguaje de programación para codificar un programa dependerá de las características del problema a resolver.

### Según lo cerca que esté del lenguaje humano

- **Lenguajes de programación de alto nivel.** Tienen el inconveniente de no ser directamente ejecutados por el hardware así que debe traducirse.

### Lenguajes de programación de bajo nivel

- **Lenguaje máquina.** registros con ceros y unos.
- **Lenguaje ensamblador.** Se trata de un primer nivel de abstracción con respecto al lenguaje máquina pero sigue siendo difícil de manejar y solo se puede utilizar en ordenadores de las mismas características.

### Según su forma de ejecución

A la hora de traducir un programa en lenguaje máquina, esta se puede realizar de dos formas:

- **Lenguajes compilados.** Lenguajes que se traducen por el compilador. Se traduce todo el programa y se genera un fichero ejecutable con el código fuente escrito en lenguaje máquina. Para hacer cambios en el programa se debe volver a compilar. No se podrá ejecutar el programa hasta que la codificación no tenga ningún error.
- **Lenguajes interpretados.** Se traduce el programa cada vez que se ejecuta, los programas que realizan esta traducción se llaman intérpretes. Se puede ejecutar el programa aunque haya errores y se parará cuando llegue a ellos.
- **Lenguajes intermedios.** Combinan los dos tipos de lenguaje, se hace una primera traducción a un lenguaje intermedio del cual se obtiene un fichero y en una segunda fase ese fichero se traduce en cada ejecución (interpretado).

- **Código fuente.** Para obtener el código fuente de una aplicación informática se debe partir de las etapas de análisis y diseño, seguidamente se construye un algoritmo que simbolice los pasos a seguir y por último se elegirá el lenguaje de programación de alto nivel para condificar el algoritmo.

Al terminar se obtendrá un documento con todos los módulos, funciones y bibliotecas necesarios.

El código fuente puede ser abierto (disponible para que se pueda estudiar, modificar o reutilizar) o cerrado, el cual no se tiene permisos para editar.

- **Código objeto.** Se trata del código intermedio, no es ejecutable por el hardware directamente, sino que necesita un intérprete. Este código se obtiene a través del compilador solo cuando el código fuente está libre de errores.
- **Código ejecutable.** Es el código máquina resultante de enlazar el código objeto con ciertas rutinas y bibliotecas. El S.O. se encarga de cargarlo en memoria y proceder a su ejecución. El ejecutable es el resultado de todos los archivos de código objeto enlazados a través de un software llamado linker.

## Lenguajes declarativos e imperativos

- En la **programación imperativa** se describen paso a paso el conjunto de instrucciones que se van a ejecutar (Basic, C, Fortran, Pacal, PHP, Java...).
- En la **programación declarativa** se utilizan sentencias que describen el problema que se quiere solucionar pero no las soluciones para hacerlo (SQL, prolog, Haskell...).

## Según la técnica de programación utilizada

- **Lenguajes de programación estructurada.**

Permiten el uso de tres tipos de sentencia o estructuras:

Sentencias secuenciales

Sentencias selectivas (condicionales)

Sentencias repetitivas (iteraciones o bucles),

Su éxito reside en la sencillez a la hora de construir y leer programas, mantenimiento sencillo y estructura clara. Evolucionó hacia la programación modular, que divide el programa en trozos de código llamados módulos.

Su desventaja es que el programa se concentra en un único bloque demasiado grande para manejarlo y la reutilización del código no es eficaz.

- **Lenguajes de programación orientada a objetos.** Se tratan a los programas no como un conjunto ordenado de instrucciones, sino como un conjunto de **objetos** que colaboran entre ellos para realizar acciones.

Se definen **clases** como una colección de objetos con características similares, dichas clases tienen una serie de **atributos** que caracterizan a esos objetos y **métodos** que corresponden a las acciones que pueden realizar.

Mediante llamada a los métodos los objetos se comunican unos con otros cambiando el estado de los mismos.

código reutilizable.

Errores más fáciles de localizar.

- **Lenguajes de programación visuales.** Basados en las técnicas anteriores pero que permiten programar gráficamente: .Net, C#...

## 8. Máquinas virtuales

Una máquina virtual es un tipo especial de software que separa el funcionamiento del ordenador de los componentes hardware instalados. Esta capa desempeña un papel importante en el funcionamiento de lenguajes compilados e interpretados.

Con ellos podemos desarrollar y ejecutar una aplicación sobre cualquier equipo, independientemente de sus características concretas.

- Consiguen que las aplicaciones sean portables.
- Reservan memoria para los objetos y liberan memoria no utilizada.
- Se comunican con el sistema donde se instala la aplicación y controla los dispositivos hardware implicados.
- Cumplimiento de las normas de seguridad de las aplicaciones.

### Características de la máquina virtual

- Cuando se compila el código fuente se obtiene el código intermedio.
- Para ejecutar ese código se requiere tener independencia respecto al hardware.
- La máquina virtual aísla la aplicación de los detalles físicos del equipo.
- Funciona como una capa de software bajo nivel y actúa como puente entre el bytecode (código intermedio) y los dispositivos físicos.
- Verifica el bytecode antes de ejecutarlo.
- Protege direcciones de memoria.

### 8.1 Entornos de ejecución

Un entorno de ejecución es un servicio de la máquina virtual que sirve para ejecutar programas. Puede pertenecer al S.O. pero también se puede instalar como software independiente.

Se denomina runtime al tiempo que tarda en ejecutarse un programa en la computadora.

## **Funcionamiento del entorno de ejecución.**

Está formado por la máquina virtual y las API o bibliotecas de clase estándar necesarias para que la aplicación se ejecute.

El entorno funciona como intermediario entre el lenguaje y el S.O.

## **8.2 Java runtime environment**

Se denomina JRE al conjunto de utilidades que permitirá la ejecución de programas java en cualquier plataforma.

Está formado por la máquina virtual JVM y las bibliotecas de clase estándar (API).



## 2. Instalación y uso de entornos de desarrollo

	Autor	Xerach Casanova
	Clase	Entornos de desarrollo
	Fecha	@Mar 29, 2021 9:05 PM

1. Concepto de entorno de desarrollo. Evolución histórica
  - 1.1. Evolución histórica
  2. Funciones de un entorno de desarrollo
  3. Entornos integrados libres y propietarios
  4. Estructura de entornos de desarrollo.
  5. Instalación de entornos integrados de desarrollo
    - 5.1. Instalación de JDK  
Configuración de las variables de entorno.
  6. Configuración y personalización de entornos de desarrollo
  7. Gestión de módulos (Netbeans)
    - 7.1. Añadir módulos
    - 7.2. Eliminar módulos
    - 7.3. Funcionalidades
    - 7.4. Herramientas concretas
  8. Uso básico de entornos de desarrollo.
    - 8.2. Generación de ejecutables
  9. Actualización y mantenimiento de entornos de desarrollo.

## 1. Concepto de entorno de desarrollo. Evolución histórica

Un entorno integrado de desarrollo (IDE), es un tipo de software compuesto por un conjunto de herramientas de programación. De las cuales podemos destacar:

- Editor de código de programación

- Accesos al compilador desde botones u opciones de menú.
- Acceso a la ejecución del programa desde botones u opciones de menú.
- Depurador.
- Constructor de interfaz gráfico.

El objetivo de los IDE es ganar fiabilidad y tiempo en proyectos de software garantizando comodidad, aumento de eficiencia y reducción de tiempo de codificación.

Suelen estar dedicados a un determinado lenguaje, pero últimamente tienden a ser compatibles varios lenguajes con la instalación de plugins.

## 1.1. Evolución histórica

El primer lenguaje en utilizar un IDE fue BASIC, el primero en abandonar las tarjetas perforadas. Estaba basado solamente en una consola de comandos, pero la gestión de archivos, compilación y depuración es semejante a los IDE actuales.

El primer IDE popular se llama Maestro, nació en los 70 y fue utilizado por unos 22000 programadores en todo el mundo.

Estos son los IDE más relevantes en la actualidad:

Entorno de desarrollo	Lenguajes que soporta	Tipo de licencia
NetBeans.	C/C++, Java, JavaScript, PHP, Python.	De uso público.
Eclipse.	Ada, C/C++, Java, JavaScript, PHP.	De uso público.
Microsoft Visual Studio.	Basic, C/C++, C#.	Propietario.
C++ Builder.	C/C++.	Propietario.
JBuilder.	Java.	Propietario.

La elección del IDE más adecuado depende del lenguaje y el tipo de licencia con la que queramos trabajar.

## 2. Funciones de un entorno de desarrollo

Las funciones de los IDE son:

- Editor de código: coloración de la sintaxis.
- Auto-compleado de código, atributos y métodos.
- Identificación automática de código.
- Herramientas de concepción visual para crear y manipular componentes visuales.
- Asistentes y utilidades de gestión y generación de código.
- Organización de archivos fuente en carpetas y compilados en otras.
- Compilación de proyectos complejos en un solo paso.
- Control de versiones (almacenaje de archivos compartido por todos los colaboradores del proyecto y auto-recuperación).
- Soporta cambios de varios usuarios simultáneamente.
- Generador de documentación integrado.
- Detección de errores de sintaxis en tiempo real.

#### **Otras funciones son:**

- Refactorización de código: cambios menores que facilitan su legibilidad sin alterar funcionalidad (cambiar el nombre a una variable).
- Tabulaciones y espaciados.
- Depuración: seguimiento de variables, puntos de ruptura, mensajes de error del intérprete.
- Aumento de funcionalidades a través de módulos y plugins.
- Administración de interfaz de usuario.
- Administración de configuración de usuario.
- Empaquetar software para su posterior despliegue e instalación en entorno de ejecución.

## **3. Entornos integrados libres y propietarios**

#### **Entornos de desarrollo libres más relevantes:**

Entornos de desarrollo libres más relevantes en la actualidad		
IDE	Algunos lenguajes que soporta	URL
Eclipse	Ada, C/C++, Java, JavaScript, PHP ....	<a href="https://www.eclipse.org/">https://www.eclipse.org/</a>
NetBeans	C/C++, Java, JavaScript, PHP, HTML5 ...	<a href="https://netbeans.org/">https://netbeans.org/</a>
CodeLite	C/C++, PHP, Node.js	<a href="https://codelite.org">https://codelite.org</a>
JDeveloper	Java, HTML, XML, SQL, PL/SQL, Javascript, PHP, UML ...	<a href="http://www.oracle.com/technetwork/developer-tools/jdev/overview/index.html">http://www.oracle.com/technetwork/developer-tools/jdev/overview/index.html</a>
IntelliJ	Java, Groovy, Perl, Scala, XML/XSL, Python ...	<a href="https://www.jetbrains.com/idea/">https://www.jetbrains.com/idea/</a>
Microsoft Studio	C#, Visual Basic, F#, C++, HTML, JavaScript, TypeScript, Python, ...	<a href="https://visualstudio.microsoft.com/es/vs/community/">https://visualstudio.microsoft.com/es/vs/community/</a> <a href="https://code.visualstudio.com/">https://code.visualstudio.com/</a>

### Entornos integrados propietarios:

Entornos de desarrollo propietarios más relevantes en la actualidad		
IDE	Algunos lenguajes que soporta	URL
Microsoft Visual Studio	C++, C#, Visual Basic ...	<a href="https://visualstudio.microsoft.com/es/">https://visualstudio.microsoft.com/es/</a>
C++ Builder	C++	<a href="https://www.embarcadero.com/es/">https://www.embarcadero.com/es/</a>
IntelliJ	Java, Groovy, Perl, Scala, ML/XSL, Python, Ruby, Sql ...	<a href="https://www.jetbrains.com/idea/">https://www.jetbrains.com/idea/</a>
QtCreator	C++ con framework QT	<a href="https://www.qt.io/">https://www.qt.io/</a>

## 4. Estructura de entornos de desarrollo.

**Los entornos de desarrollo están formados por:**

- Editor de textos:
  - Resaltado y coloreado de sintaxis.
  - Funciones de auto-completado de código.
  - Inserción automática de paréntesis, corchetes, tabulaciones y espaciadoos.
  - Ayuda y listado de parámetros de funciones y métodos de clases.
- Compilador/intérprete:
  - Detección de errores de sintaxis en tiempo real.

- Depurador:
  - Ejecución del programa paso a paso, definición de ruptura y seguimiento de variables. Opción depurar en servidores remotos.
- Generador automático de herramientas:
  - Herramientas para la visualización, creación y manipulación de componentes visuales, utilidades de gestión y generación de código.
- Interfaz gráfica:
  - Oportunidad de programar en varios lenguajes en el mismo IDE. Acceso a bibliotecas y plugins.

## 5. Instalación de entornos integrados de desarrollo

### 5.1. Instalación de JDK

JDK (Java Development Kit) es el entorno de desarrollo necesario para poder ejecutar NetBeans. Trae integrado el JRE y un compilador.

Para poder interpretar los archivos bytecodes (obtenidos tras la compilación del código fuente de Java), se necesita tener instalado el JRE (Java Runtime Environment). El JRE está conformado por la máquina virtual de Java (JVM), las bibliotecas Java y otros componentes necesarios para poder ejecutar una aplicación escrita en Java. JRE es un intermediario entre el S.O. y Java.

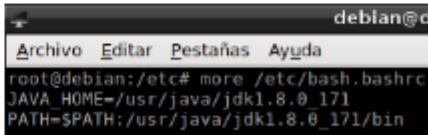
La JVM es el programa que ejecuta los bytecodes y las librerías de clase estándar son las que implementan el API de Java, ambas deben ser consistentes entre sí y se distribuyen en conjunto.

Los programas más importantes que se incluyen en el JDK son:

- **appletviewer**: visor de applets para generar sus vistas previas (carecen de método main y no los puede ejecutar el programa java).
- **javac.exe**: compilador de java
- **java.exe**: intérprete de java.
- **javadoc.exe**: genera la documentación de las clases de un programa en java.

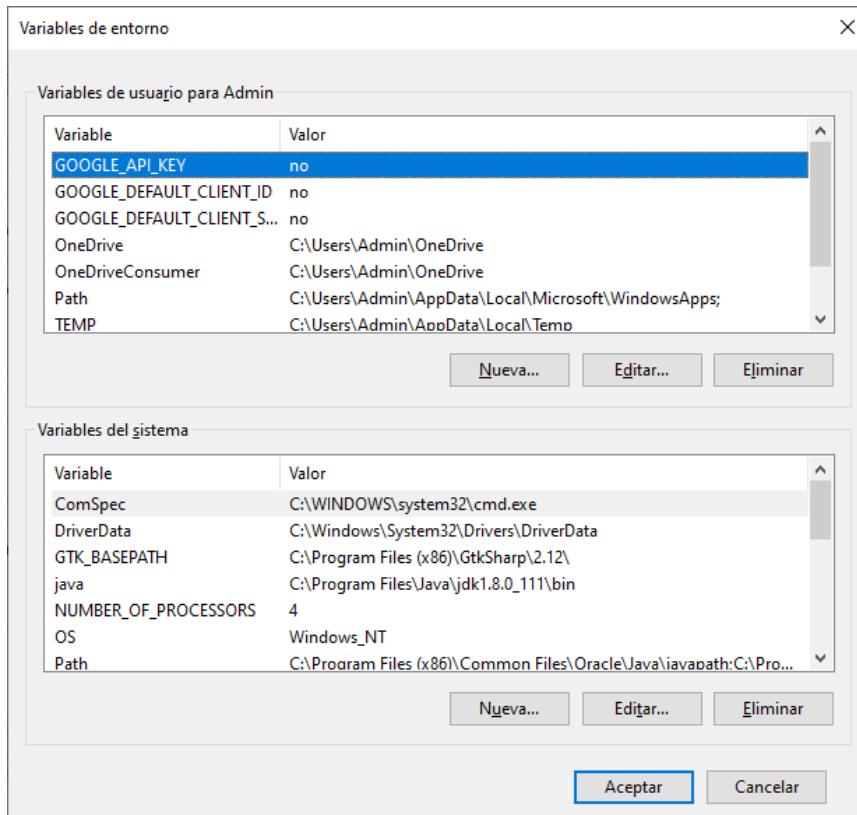
## Configuración de las variables de entorno.

A través de Linux (con bash):

Acción	Orden en consola linux
Abrir un terminal Linux.	
Ganar privilegios de administrador.	sudo su + Contraseña
Con un editor de texto (por ejemplo nano), acceder al fichero /etc/bash.bashrc.	nano /etc/bash.bashrc
Introducir las variables de entorno:  JAVA_HOME=/usr/java/jdk1.8.0_171 PATH=\$PATH:/usr/java/jdk1.8.0_171/bin  En este caso, la ruta del jdk en JAVA_HOME y también añadimos la ruta de los ejecutables en PATH	

A través de Windows 10:

- Ir a variables de entorno en configuración avanzada del sistema.



Después elegimos Nueva (en variables de Sistema) e introducimos los datos de la variable de entorno, su nombre: JAVA\_HOME y su valor (la ruta donde está instalado el jdk). En este ejemplo es un jdk 8. Despues Aceptar.

Como la variable PATH ya existe, seleccionamos la variable PATH y elegimos Editar. Añadimos la ruta del directorio bin de nuestro jdk. Ponemos un punto y coma y luego %JAVA\_HOME%\bin. Para finalizar, Aceptar.

## 6. Configuración y personalización de entornos de desarrollo

**Parámetros y configuraciones del entorno de desarrollo:**

- Carpeta o carpetas donde se alojarán los archivos de los proyectos.
- Carpetas de almacenamiento de paquetes fuente y paquetes prueba.
- Administración de la plataforma del entorno de desarrollo.
- Opciones de la compilación de los programas.
- Opciones de empaquetado de la aplicación.
- Opciones de generación de documentación asociada al proyecto.
- Descripción de los proyectos.
- Opciones globales de formato del editor.
- Opciones de combinación de atajos de teclado.

## 7. Gestión de módulos (Netbeans)

Con plataformas como Netbeans podemos hacer uso de módulos o plugins para desarrollar aplicaciones.

En categoría de lenguajes de programación podremos añadir nuevos lenguajes soportados.

Un módulo es un componente software con clases de java que pueden interactuar con las API y el manifest file (archivo especial que lo identifica como módulo). Se pueden construir y desarrollar de forma independiente.

### 7.1. Añadir módulos

Tenemos varias opciones:

- Añadir módulos de los que NetBeans instala por defecto.
- Descargar un módulo de algún sitio web permitido y añadirlo.
- Instalarlo on-line en el entorno.
- Crear el módulo nosotros mismos.

El plugin se descarga en formato .nbm y desde el IDE se carga e instala (adición off-line).

También se pueden instalar on-line, sin salir del IDE. Para ello debemos tener instalado el plugin "Portal Update Center"

## 7.2. Eliminar módulos

Eliminar un módulo requiere seguir los siguientes pasos.

1. Encontrar el módulo en la lista de complementos instalados.
2. Optar por:
  - Desactivarlo.
  - Desinstalarlo.

## 7.3. Funcionalidades

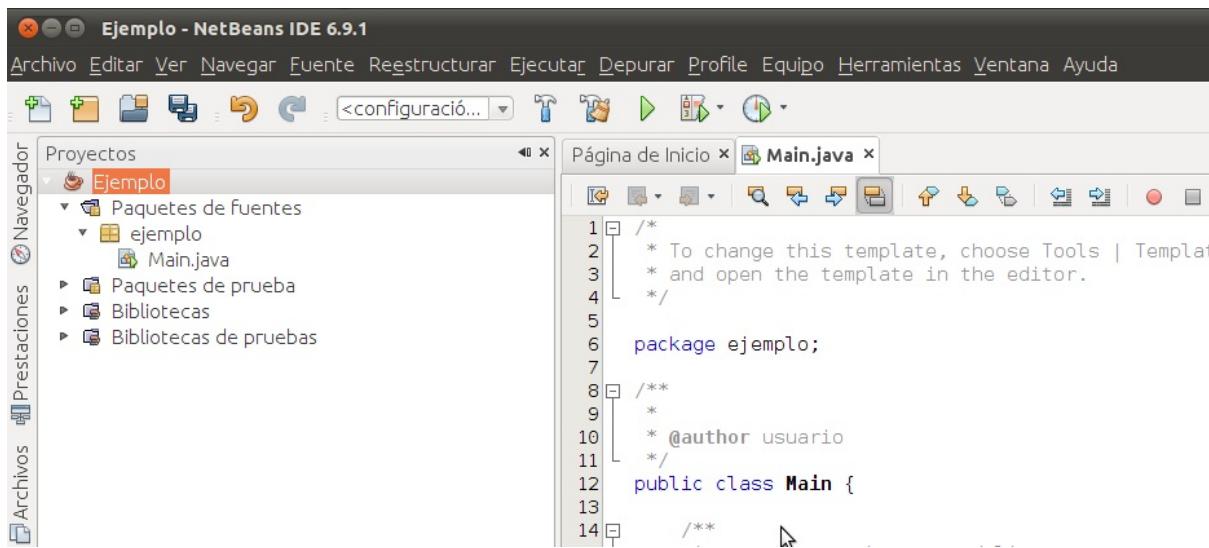
**Los módulos y plugins tienen diversas funciones categorizadas por:**

- Construcción de código.
- Bases de datos.
- Depuradores.
- Aplicaciones: añaden aplicaciones que pueden ser útiles.
- Edición.
- Documentación de aplicaciones.
- Interaz gráfica de usuario.
- Refactorización.
- Aplicaciones web.
- Prueba.

## 7.4. Herramientas concretas

- Importador de proyectos de Netbeans: permite trabajar en lenguajes como JBuilder.
- Servidores de aplicaciones GlassFish: Plataforma completa para aplicaciones de tipo empresarial
- Soporte para JEE: Cumplimiento de estándares, facilidad e uso, mejora de rendimiento.
- NetBeans Swing GUI Builder: simplifica la creación de interfaces gráficos en aplicaciones cliente y permite manejar diferentes aplicaciones sin salir del IDE.
- NetBeans Profiler: permite ver de forma inmediata la eficiencia de un trozo de software.
- Editor WSDL: facilita a los programadores trabajar en servicios Web basados en XML
- XML Schema Editor: permite refinar aspectos de los documentos XML.
- Aseguramiento de la seguridad de datos mediante Sun Java System Access Manager.
- Soporte beta de UML que cubre actividades como clases, comportamiento interacción y secuencias.
- Soporte bidireccional: que permite sincronizar con rapidez modelos de desarrollo con los cambios en el código conforme avanzamos por las etapas del ciclo de vida de la app.

## 8. Uso básico de entornos de desarrollo.



### Ventana izquierda: ventana de proyectos.

Donde aparece la relación de proyectos, archivos, módulos o clases.

Cada proyecto comprende una serie de archivos y bibliotecas. El archivo principal es el Main.java, que es donde empieza la ejecución. No tiene que llamarse Main pero es aconsejable.

### Ventana derecha: espacio de escritura de los códigos de los proyectos.

El esqueleto pripio de un programa escrito en lenguaje java. Tiene las siguientes características:

- Autocompletado de código
- Coloración de comandos.
- Subrayado en rojo cuando hay agún error o posibilidad de deupración y corrección (a través de un pequeño ícono en la parte izquierda de la línea defecutosa).

### Barra de herramientas.

Para acceder a todas las opciones del IDE.

## 8.2. Generación de ejecutables

Una vez tenemos el código terminado y libre de errores de sintaxis los siguientes pasos son: compilación, depuración, ejecución. Para ejecutar solo hay que pulsar shift+F6.

## **9. Actualización y mantenimiento de entornos de desarrollo.**

El mantenimiento y las actualizaciones se hacen de forma On-Line, con el complemento Auto Update Services, el cual permite realizar continuas revisiones del entorno y actualizaciones de plugins.

La gestión de las bases de datos asociadas a nuestros proyectos es importante y hay que realizar copias de seguridad periodicas. Tanto para asegurar su restauración en caso de fallos como para mantenerlas actualizadas para su posible portabilidad a nuevas versiones del entorno que utilicemos.



# 3. Diseño y realización de pruebas

	Autor	(X) Xerach Casanova
	Clase	Entornos de desarrollo
	Fecha	@Dec 19, 2020 10:44 PM

Planificación de las pruebas

2. Tipos de prueba

- 2.1. Funcionales (pruebas de caja negra)
- 2.2. Pruebas estructurales (pruebas de caja blanca)

Cubrimiento

- 2.3. Pruebas de regresión

3. Procedimientos y casos de prueba

4. Herramientas de depuración

- 4.1. Puntos de ruptura
- 4.3. Tipos de ejecución
- 4.3. Examinador de variables

5. Validaciones.

6. Normas de calidad

7. Pruebas unitarias

- 7.1. Herramientas para gráfica
- 7.2. Herramientas para otros lenguajes

8. Documentación de la prueba

## Planificación de las pruebas

Desde la fase de análisis hasta la implantación del software en el cliente, es necesario realizar un conjunto de pruebas que permita comprobar que el producto que estamos creando, es correcto y cumple con las especificaciones solicitadas.

Las pruebas tratan de verificar y validar las aplicaciones:

- **La verificación** es la comprobación de que un sistema o parte de él, cumple con las condiciones impuestas, comprobando así que se está construyendo correctamente. **Es un ejercicio teórico para estar seguro que se cumplen los requerimientos en el diseño.**
- **La validación** es el proceso de evaluación del sistema o uno de sus componentes para determinar si satisface los requisitos especificados. **Es un ejercicio práctico que garantiza que el producto funciona a partir de esos requerimientos.**

El proceso de pruebas se lleva a cabo con estrategias de pruebas. En el **modelo en espiral**, se empiezan las pruebas en cada porción de código y una vez han sido pasadas con éxito, se sigue con la prueba de integración, donde se ponen todas las partes del código en común, comprobando que el ensamblado de todo atiende a lo establecido en la fase de diseño.



El siguiente paso es la prueba de validación y finalmente se alcanza la prueba de sistema, que verifica el funcionamiento total del software y otros elementos del sistema.

El objetivo de las pruebas es conseguir un software libre de errores, además el programador debe evitar probar sus propios programas para evitar que vuelva a pasar inadvertidos aspectos que no tuvo en consideración.

## 2. Tipos de prueba

Existen dos enfoques fundamentales:

- **Prueba de la caja negra (Black Box Testing):** La aplicación se prueba usando interfaz externa, no nos preocupamos de la implementación del código, solo si los resultados son correctos a partir de los datos introducidos. En el siguiente código, mediante pruebas de caja negra se detecta un error mediante caja negra, ya que si se saca un cinco se devuelve que has suspendido.



```
if(nota>=5)
    System.out.println("Has suspendido");
else
    System.out.println("Has aprobado");
```

- **Prueba de la caja blanca (White Box Testing):** se prueba la aplicación desde dentro, usando su lógica de aplicación. Más bien lo que se busca es encontrar estructuras ineficientes o incorrectas. En el siguiente código, mediante prueba de caja blanca detectamos un error en la estructura, ya que el segundo else nunca será ejecutado, aunque el resultado de caja negra haya salido correctamente

```
if(nota>=5)
    System.out.println("Has aprobado");
else
    if(nota<5)
        System.out.println("Has suspendido");
    else
        System.out.println("Esta instrucción nunca se ejecutará.");
```

## 2.1. Funcionales (pruebas de caja negra)

No nos interesa la implementación del software, solo si se realizan las funciones esperadas de él siguiendo el enfoque de las pruebas de Caja Negra. Comprenden actividades para verificar una acción específica o funcional del código de la aplicación.

Se responde a las preguntas: ¿Puede el usuario hacer esto? o ¿Funciona esta utilidad de la aplicación?

Para realizar este tipo de pruebas se analizan las entradas y las salidas de cada componente y se verifica el resultado. Este tipo de prueba no considera en ningún caso el código desarrollado, ni el algoritmo, ni la eficiencia ni las partes de código innecesarias. Existen tres tipos de pruebas:

- **Particiones equivalentes.** Se considera el menor número posible de casos de pruebas abarcando el mayor número posible de entradas distintas.
- **Análisis de valores límite:** Se eligen valores de entrada que se encuentran en el límite de las clases de equivalencias.
- **Pruebas aleatorias.** Generamos entradas aleatorias a la aplicación. Se suelen utilizar generadores de prueba, capaces de crear volúmenes de

casos de prueba al azar (suele utilizarse en aplicaciones no interactivas por su dificultad de implementación en entornos interactivos).

## 2.2. Pruebas estructurales (pruebas de caja blanca)

Las pruebas estructurales son el conjunto de pruebas de la caja blanca, con las cuales verificamos la estructura interna de cada componente de la aplicación, dejando de lado la funcionalidad establecida para el mismo.

No se comprueba la corrección de los resultados producidos, sino que se ejecutan todas las instrucciones del programa, que no hay código no usado, o que los caminos lógicos del programa se van a recorrer. Los criterios de cobertura que se siguen son:

- **Cobertura de sentencias.** Se generan casos de pruebas suficientes para que cada instrucción del programa se ejecute al menos una vez.
- **Cobertura de decisiones.** Se generan casos de prueba para que cada opción resultado de una prueba lógica, se evalúe al menos una vez en cierto o falso.
- **Cobertura de condiciones:** Se generan casos de prueba para que cada elemento de una condición se ejecute al menos una vez a falso y otra a verdadero.
- **Cobertura de condiciones y decisiones:** que se cumplen simultáneamente las dos anteriores.
- **Cobertura de caminos:** es el criterio más importante. Se debe ejecutar al menos una vez cada secuencia de sentencias encadenadas desde la sentencia inicial del programa hasta la sentencia final. Como el número de caminos que puede tener una aplicación es muy grande, se reduce el número a lo que se conoce como camino prueba.
- **Cobertura del camino de prueba:** Se realizan dos variantes. Una indica que cada bucle se debe ejecutar sólo una vez y otra recomienda que se pruebe el bucle tres veces, la primera sin entrar, una ejecutándolo una vez y otra ejecutándolo más veces.

### Cubrimiento

La tarea la realiza el programador y consiste en comprobar los caminos definidos por el código y que se recorren.

Por ejemplo:

Considerando que esta función forma parte de un programa mayor, se considera lo siguiente:

- Si durante la ejecución del programa, la función es llamada, al menos una vez, el cubrimiento de la función es satisfecho.
- El cubrimiento de sentencias para esta función, será satisfecho si es invocada, por ejemplo como prueba(1,1), ya que en este caso, cada línea de la función se ejecuta, incluida `z=x;`
- Si invocamos a la función con prueba(1,1) y prueba(0,1), se satisfará el cubrimiento de decisión. En el primer caso, la if condición va a ser verdadera, se va a ejecutar `z=x`, pero en el segundo caso, no.
- El cubrimiento de condición puede satisfacerse si probamos con prueba(1,1), prueba(1,0) y prueba(0,0). En los dos primeros casos `x<0` se evalúa a verdad mientras que en el tercero, se evalúa a falso. Al mismo tiempo, el primer caso hace (`y>0`) verdad, mientras el tercero lo hace falso.

```
int prueba (int x, int y)
{
    int z=0;
    if ((x>0) && (y>0))
    {
        z=x;
    }
    return z;
}
```

Existen otra serie de criterios, para comprobar el cubrimiento.

- Secuencia lineal de código y salto.
- JJ-Path Cubrimiento.
- Cubrimiento de entrada y salida.

Existen herramientas comerciales y también de software libre, que permiten realizar la pruebas de cubrimiento, entre ellas, para Java, nos encontramos con Clover.

## 2.3. Pruebas de regresión

Las modificaciones en el programa por detección de fallo pueden generar errores colaterales que no existían antes, lo cual nos obliga a repetir pruebas que hemos realizado antes.

Las pruebas de regresión se hacen, en definitiva, cada vez que se hace un cambio en el sistema, ya sea para corregir errores o para realizar mejoras. No solo es suficiente probar componentes modificados o añadidos, sino controlar que esas modificaciones no producen efectos negativos en otros componentes.

Las pruebas de software con éxito, son aquellas que dan como resultado el descubrimiento de errores y, en consecuencia, se produce corrección y modificación de algún componente del software, documentación y datos que lo soportan. La prueba de regresión nos ayuda a asegurar que estos cambios no introducen comportamiento no deseado y errores adicionales.

Las pruebas de regresión contienen tres clases distintas:

- Una muestra representativa de pruebas que ejercite todas las funciones.
- Pruebas adicionales centradas en funciones del software que van a ser afectadas por el cambio.
- Pruebas que se centran en componentes que han cambiado.

Al no ser práctico ejecutar cada una de las pruebas después de cada cambio, se deben diseñar estas pruebas para incluir solo aquellas que traten una o más clases de errores en cada una de las funciones principales del programa.

### 3. Procedimientos y casos de prueba

La prueba consiste en la ejecución de un programa con el objetivo de encontrar errores.

Según el IEEE, un caso de prueba es un conjunto de entradas, condiciones de ejecución y resultados esperados, desarrollados para un objetivo particular (ejercitarse en un camino concreto de un programa o verificar el cumplimiento de un determinado requisito, incluyendo toda la documentación asociada).

Existen varios procedimientos para el diseño de los casos de prueba:

- **Enfoque funcional o de caja negra.** Se centra en probar que el programa o parte de él, recibe una entrada de forma adecuada y se produce una salida correcta. Se centra en funciones, entradas y salidas. Se aplican valores límite y clases de equivalencia.
- **Enfoque estructural o caja blanca.** Nos centramos en la implementación interna del programa, se deben probar todos los caminos que puede seguir la ejecución del programa.
- **Enfoque aleatorio.** A partir de modelos obtenidos estadísticamente, se elaboran casos de prueba de entrada en el programa.

### 4. Herramientas de depuración

Cada IDE y lenguaje de programación, incluye herramientas de depuración como: inclusión de puntos de ruptura, ejecución paso a paso de cada instrucción, ejecución del procedimiento, inspección de variables, etc.

Se pueden producir errores de compilación y errores lógicos.

Cuando cometemos errores de codificación, el entorno nos proporciona información donde se produce y como solucionarlo. Estos errores no permiten compilar hasta que se corrijan.

Los errores lógicos también se llaman bugs, permiten compilar, sin embargo, puede provocar que el programa devuelva resultados erróneos y que el programa no pueda terminar o no termine nunca.

Los entornos de desarrollo incorporan el depurador para solventar estos problemas. El cual solo se puede utilizar cuando el programa puede compilarse. Permite análisis en ejecución, suspender, examinar y establecer valores de las variables, etc.

## 4.1. Puntos de ruptura

Dentro del menú de depuración existe la opción de insertar puntos de ruptura (breakpoints).

Los puntos de ruptura son marcadores que se pueden establecer en cualquier línea de código ejecutable. El programa se ejecuta hasta el punto de ruptura y a partir de ahí se pueden examinar variables y comprobar que los valores asignados son correctos, o se puede iniciar la depuración paso a paso. Una vez realizada la comprobación podemos abortar o continuar la ejecución normal del mismo. Se insertan y se quitan puntos de ruptura con la misma facilidad.

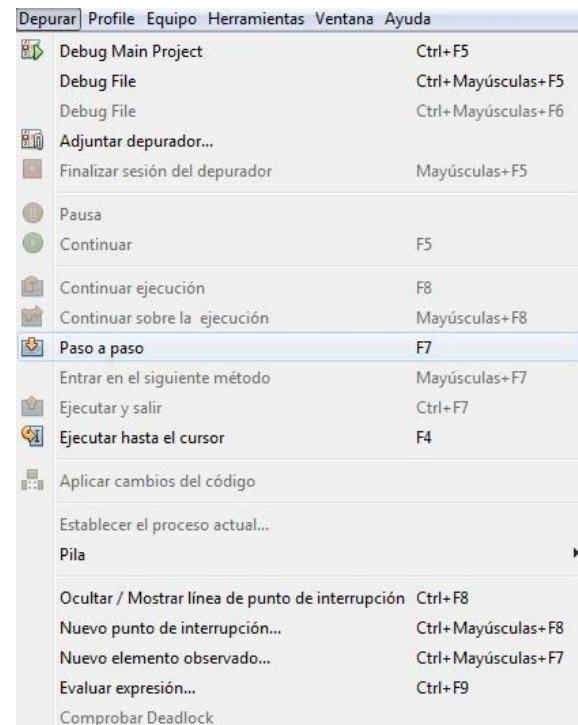
## 4.3. Tipos de ejecución

Se puede ejecutar el programa de diferentes formas en función del problema que queramos solucionar.

Los distintos métodos de ejecución se utilizarán según las necesidades de depuración en cada momento:

- **Paso a paso.** Ayuda a verificar que el código de un método se ejecuta de manera correcta.

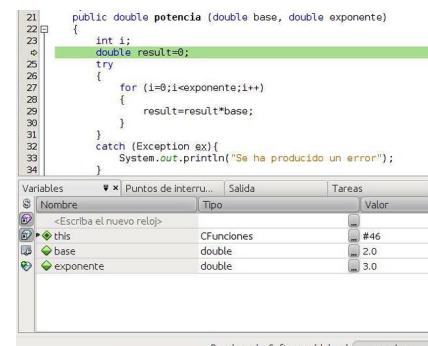
- Paso a paso por procedimientos,** nos permite introducir los parámetros que queramos a un método o función de nuestro programa, pero no ejecuta el método paso a paso, sino devuelve su resultado. Útil si ya hemos comprobado ese procedimiento y solo nos interesa su valor.
- Ejecución hasta una instrucción.** Se ejecuta el programa y se detiene la instrucción en donde se encuentra el cursor y, a partir de ahí podemos depurar paso a paso o por procedimiento.
- Ejecución de un programa hasta el final.** No se detiene en instrucciones intermedias.



## 4.3. Examinador de variables

Una de las maneras más comunes de comprobar que la aplicación funciona adecuadamente es comprobar que las variables van tomando los valores adecuados. Los entornos de desarrollo suelen traer examinadores de variables (En NetBeans es Ventana de inspección).

Los examinadores de variables sirven para comprobar los distintos valores que adquieren las variables, así como su tipo, en el proceso de depuración, por ejemplo en el paso a paso.



## 5. Validaciones.

En el proceso de validación interviene de manera decisiva el cliente. Ellos son quienes deciden si la aplicación se ajusta a sus requerimientos.

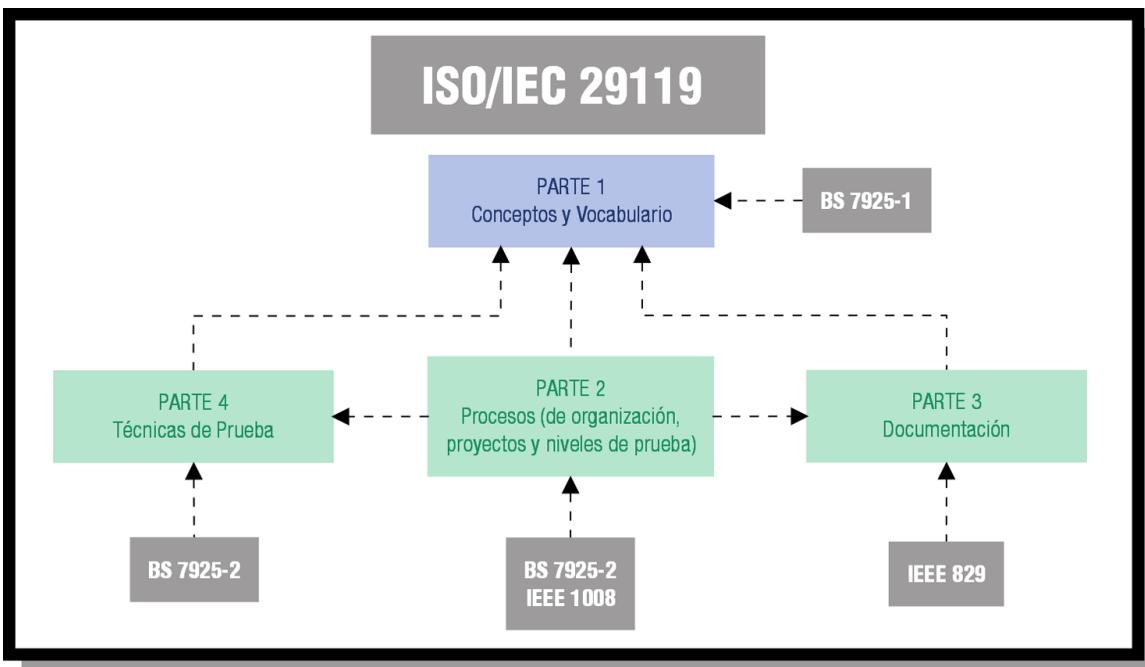
La validación de software se consigue mediante una serie de pruebas de caja negra, que demuestran la conformidad de los requisitos.

Un **plan de prueba** traza la clase de pruebas que se han de llevar a cabo y un **procedimiento de prueba** define los casos de prueba específicos en un intento por descubrir errores de acuerdo con los requisitos. Ambos estarán diseñados para asegurar que se satisfacen los requisitos funcionales, que se alcanzan los requisitos de rendimiento, documentaciones correctas e inteligibles y otros requisitos como portabilidad, recuperación de errores, facilidad de mantenimiento, etc...

## 6. Normas de calidad

Los estándares que se han venido utilizando en la fase de prueba de software son:

- Metodología Métrica V3
- Estándares BSI
- Metodología **Métrica v3.**
- Estándares **BSI**
  - BS 7925-1, Pruebas de software. Parte 1. Vocabulario.
  - BS 7925-2, Pruebas de software. Parte 2. Pruebas de los componentes software.
- Estándares de pruebas de software.:
  - IEEE
    - IEEE estándar 829, Documentación de la prueba de software.
    - IEEE estándar 1008, Pruebas de unidad
    - Otros estándares **ISO / 12207, 15289**
  - IEC
    - Otros estándares sectoriales



Pero estos estándares no cubren determinadas facetas de la fase de pruebas, como la organización, el proceso y la gestión de las pruebas. Ante ello la industria ha desarrollado la norma ISO/IEC 29119, que pretende unificar en una única forma todos los estándares, de forma que proporcione vocabulario, procesos, documentación y técnicas para cubrir todo el ciclo de vida del software (estrategias de prueba para organización y políticas de prueba, prueba de proyecto, análisis de casos de prueba, diseño, ejecución e informe).

## ISO/IEC 29119

se compone de las siguientes partes:

- **Parte 1.** Conceptos y vocabulario.
  - Introducción a la prueba.
  - Pruebas basadas en riesgo.
  - Fases de prueba (unidad, integración, sistema, validación) y tipos de prueba (estática, dinámica, no funcional, ...).
  - Prueba en diferentes ciclos de vida del software.
  - Roles y responsabilidades en la prueba.
  - Métricas y medidas.
- **Parte 2.** Procesos de prueba.
  - Política de la organización.

- Gestión del proyecto de prueba.
- Procesos de prueba estática.
- Procesos de prueba dinámica.
- **Parte 3. Documentación.**
  - Contenido.
  - Plantilla.
- **Parte 4. Técnicas de prueba.**
  - Descripción y ejemplos.
  - Estáticas: revisiones, inspecciones, etc.
  - Dinámicas: caja negra, caja blanca, técnicas de prueba no funcional (seguridad, rendimiento, usabilidad, etc) .

## 7. Pruebas unitarias

Una unidad es la parte de la parte de la aplicación más pequeña para probar.

Una unidad puede ser una función o un procedimiento en programación procedural, en POO, normalmente es un método.



Las pruebas unitarias o de unidad prueban el correcto funcionamiento de un módulo o código. Cada prueba es independiente al resto.

Los entornos de desarrollo integran frameworks que permiten automatizar pruebas.

En el diseño de casos de pruebas unitarias hay que tener en cuenta los siguientes requisitos:

- Automatizable: sin intervención manual.
- Completas: cubrir la mayor cantidad de código.
- Repetibles o reutilizables: que puedan ejecutarse más de una vez.

- Independientes: la ejecución de una prueba no debe afectar a la ejecución de otra
- Profesionales: las pruebas deben ser consideradas con la misma profesionalidad que el código o la documentación.

Las ventajas que proporcionan las pruebas unitarias son:

- Fomentar el cambio. Facilitan que el programador cambie el código para mejorar su estructura.
- Simplifica la integración: permite llegar a la fase de integración con un grado alto de seguridad.
- Documenta el código. Ya que gracias a ellas se puede ver como se utiliza.
- Separación de interfaz e implementación. Dado que la única interacción entre los casos de prueba y las unidades bajo prueba son las interfaces de estas últimas, se puede cambiar cualquiera de los dos sin afectar al otro.
- Errores acotados y fáciles de localizar.

## 7.1. Herramientas para gráfica

Las más destacadas son:

- **Jtiger:**
  - Pruebas unitarias.
  - Código abierto.
  - Exporta informes en HTML, XML o texto plano.
  - Ejecuta casos de prueba de Junit mediante plugin.
  - Gran variedad de aserciones como la comprobación de cumplimiento del contrato de un método.
  - Los metadatos en los casos de prueba son especificados como anotaciones del lenguaje Java.
  - Incluye una tarea de Ant para automatizar las pruebas.
  - Documentación muy completa en JavaDOc y una web con la info necesaria para comprender su uso y utilizarlo en IDE como Eclipse.
  - Incluye pruebas unitarias sobre sí mismo.
- **TestNG**

- Inspirado en JUnit y NUnit.
- Diseñado para cubrir todo tipo de pruebas: unitarias, funcionales, integración
- Anotaciones de Java 1.5.
- Compatible con pruebas Junit.
- Soporte para el paso de parámetros a los métodos de pruebas.
- Permite distribución de pruebas en máquinas esclavas
- Soportado por gran variedad de plugins (Eclipse, NetBeans, IDEA...)
- Las clases de pruebas no necesitan implementar ninguna interfaz ni extender ninguna otra clase.
- Una vez compiladas las pruebas estas se pueden invocar desde la línea de comandos con una tarea de Ant o con un fichero XML.
- Los métodos de prueba se organizan en grupos.
- **Junit.**

Sirve para NetBeans y Eclipse, es una herramienta de automatización de pruebas, para elaborarlas de manera rápida y sencilla. Nos permite diseñar clases de prueba para cada clase diseñada en nuestra aplicación. Una vez creadas, establecemos los métodos que queremos probar y diseñamos casos de prueba. Los criterios de creación de casos de prueba pueden ser muy diversos y dependen de lo que se quiera probar.

Una vez diseñados los casos de prueba, probamos la aplicación, en este caso, nos presenta un informe con los resultados de la prueba y en función de los resultados modificamos o no el código.

- Framework de pruebas unitarias creado por Erich Gamma y Kent Beck.
- Es una herramienta de código abierto.
- Multitud de documentación y ejemplos en la web.
- Se ha convertido en el estándar de hecho para las pruebas unitarias en Java.
- Soportado por la mayoría de los IDE como eclipse o Netbeans.

- Es una implementación de la arquitectura xUnit para los frameworks de pruebas unitarias.
- Posee una comunidad mucho mayor que el resto de los frameworks de pruebas en Java.
- Soporta múltiples tipos de aserciones.
- Desde la versión 4 utiliza las anotaciones del JDK 1.5 de Java.
- Posibilidad de crear informes en HTML.
- Organización de las pruebas en Suites de pruebas.
- Es la herramienta de pruebas más extendida para el lenguaje Java.
- Los entornos de desarrollo para Java, NetBeans y Eclipse, incorporan un plugin para Junit.

## 7.2. Herramientas para otros lenguajes

### **CppUnit**

- Framework de pruebas unitarias para el lenguaje C++.
- Es una herramienta libre.
- Existe diversos entornos gráficos para la ejecución de pruebas como QtTestRunner.
- Es posible integrarlo con múltiples entornos de desarrollo como Eclipse.
- Basado en el diseño de xUnit.

### **Nunit**

- Framework de pruebas unitarias para la plataforma .NET
- Es una herramienta de código abierto.
- También está basado en xUnit.
- Dispone de diversas expansiones como Nunit.Forms o Nunit.ASP

**SimpleTest:** Entorno de pruebas para aplicaciones realizadas en PHP.

**PHPUnit:** framework para realizar pruebas unitarias en PHP.

**FoxUnit:** framework OpenSource de pruebas unitarias para Microsoft Visual FoxPro

**MOQ:** Framework para la creación dinámica de objetos simuladores (mocks).

## 8. Documentación de la prueba

La documentación de las pruebas es un requisito indispensable para su correcta realización. Las metodologías actuales como Métrica V.3 proponen que la documentación se base en estándares ANSI/IEEE sobre verificación y validación de software.

El propósito de estos estándares es describir un conjunto de documentos para las pruebas de software. Este documento puede facilitar la comunicación entre desarrolladores al suministrar un marco de referencia común.

Los documentos a generar son:

- **Plan de Pruebas:** Al principio se desarrollará una planificación general, que quedará reflejada en el "Plan de Pruebas". El plan de pruebas se inicia el proceso de Análisis del Sistema.
- **Especificación del diseño de pruebas.** De la ampliación y detalle del plan de pruebas, surge el documento "Especificación del diseño de pruebas".
- **Especificación de un caso de prueba.** Los casos de prueba se concretan a partir de la especificación del diseño de pruebas.
- **Especificación de procedimiento de prueba.** Una vez especificado un caso de prueba, será preciso detallar el modo en que van a ser ejecutados cada uno de los casos de prueba, siendo recogido en el documento "Especificación del procedimiento de prueba".
- **Registro de pruebas.** En el "Registro de pruebas" se registrarán los sucesos que tengan lugar durante las pruebas.
- **Informe de incidente de pruebas.** Para cada incidente, defecto detectado, solicitud de mejora, etc, se elaborará un "informe de incidente de pruebas".
- **Informe sumario de pruebas.** Finalmente un "Informe sumario de pruebas" resumirá las actividades de prueba vinculadas a uno o más especificaciones de diseño de pruebas.



# 4. Optimización y documentación

	Autor	Xerach Casanova
	Clase	Entornos de desarrollo
	Fecha	@Jan 22, 2021 7:20 AM

## 1. Optimización y documentación

1.1. Concepto

1.2. Limitaciones

1.3. Patrones de refactorizaciones más habituales

1.4. Refactorización en eclipse

1.4.1. Renombrar

1.4.2. Mover

1.4.3. Cambiar firma del método

1.4.4. Extraer variable local

1.4.5. Extraer constante

1.4.6. Convertir variable local en atributo

1.4.7. Extraer método

1.4.8. Incorporar

1.4.9. Autoencapsular atributo

## 2. Control de versiones

2.1. Tipos de herramientas de control de versiones

2.2. Estructura de herramientas de control de versiones

2.2.1. Repositorio

2.2.2. Gestión de versiones y entregas

2.3. Herramientas de control de versiones

2.4. Planificación de la gestión de configuraciones

2.5. Gestión del cambio

## 3. Documentación

3.1. Uso de comentarios

3.2. Documentación de clases

# 1. Optimización y documentación

La refactorización es una técnica que consiste en hacer pequeñas transformaciones en el código del programa para mejorar la estructura sin que cambie el comportamiento o funcionalidad con el objetivo de mejorar estructura, legibilidad o eficiencia.

De esta manera hacemos que el mantenimiento del software sea más sencillo y que el programa sea más rápido. Se basa en el concepto matemático de factorización de polinomios, así que  $(x+1)(x-1)$  se puede expresar con  $x^2-1$  sin que se altere el sentido.

Pistas para refactorizar:

- Código duplicado.
- Métodos muy largos.
- Clases muy grandes o demasiados métodos.
- Métodos más interesados en los datos de otra clase que en los de la propia.
- Grupos de datos que aparecen juntos y parecen más una clase que datos sueltos.
- Clases con pocas llamadas o que se usan poco.
- Exceso de comentarios explicando código.

## 1.1. Concepto

El concepto de refactorización se define como el cambio hecho en la estructura interna del software para hacerlo más fácil de entender y modificar sin alterar su comportamiento.

Se aconseja crear métodos getter y setter para cada campo que se defina en una clase para acceder o modificar su valor.

La refactorización y la optimización son conceptos distintos y la diferencia radica en que cuando se optimiza, se persigue una mejora de rendimiento y velocidad de ejecución, aunque el código resulte más complicado de entender.

## 1.2. Limitaciones

La refactorización presenta problemas en algunos aspectos de su desarrollo al ser una técnica novedosa.

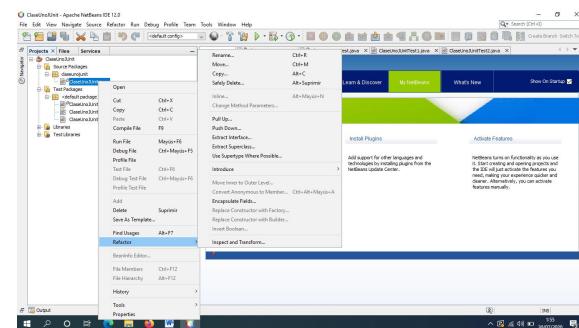
- Uno de estos problemas son las bases de datos, las cuales son difíciles de modificar al ser demasiado interdependientes. Cualquier modificación a las bases de datos (esquema, migración), suele ser tarea costosa, por ello, la refactorización en aplicaciones con bases de datos está limitada al diseño de la base de datos.
- Otro problema es el cambio de interfaces. Cuando modificamos la estructura interna de un programa o un método, no afecta al comportamiento de la interfaz, sin embargo, si renombramos un método genera un problema si la interfaz es pública y otros programas llaman a ese método. La solución pasa por mantener la interfaz vieja junta a la nueva.
- Es difícil refactorizar cuando existen errores de diseño o cuando la estructura a modificar es de vital importancia en el diseño de la aplicación.
- En los casos en los que no se debe refactorizar en absoluto es cuando es más fácil reescribirlo desde el principio que modificarlo.

### 1.3. Patrones de refactorizaciones más habituales

Los patrones más habituales de refactorizaciones ya vienen integrados en la mayoría de IDEs.

Estos son:

- **Renombrar:** Cambia el nombre de un paquete, clase, método o campo.
- **Encapsular campos.** Crear métodos getter y setter para los campos de clase.
- **Sustituir bloques de código por un método.** En ocasiones se observa que, al aparecer repetido en distintos sitios, podemos sustituir el bloque de código por un método, de manera que, cada vez que usemos ese bloque de código invoquemos al método.
- **Modificar la extensión del código.** Hacer un código más extenso si se gana en claridad o menos extenso solo si se gana eficiencia.
- **Crear código común** en una clase o método para evitar el código repetido.



- **Mover la clase de** un paquete a otro o de un proyecto a otro. Implica actualización en todo el código fuente de las referencias a la clase en su nueva ubicación.
- **Borrado seguro.** Garantizar que un elemento del código ya no es necesario, se borran todas las referencias a él en cualquier parte del proyecto.
- **Cambiar parámetros del método,** permite añadir, modificar o eliminar los parámetros de un método y cambiar modificadores de acceso.
- **Extraer la interfaz.** Crea una nueva interfaz de los métodos public non-static seleccionados en una clase o interfaz.

## 1.4. Refactorización en eclipse

Las herramientas de refactorización en los entorno de desarrollo ayudan a automatizar la refactorización, ahorrando tiempo y evitando redundancias o errores en el código que modificamos.

En eclipse podemos refactorizar haciendo click secundario en un fragmento de código y escogiendo la opción refactorizar. Según el código seleccionado, se mostrarán unas opciones u otras.

### 1.4.1. Renombrar

Modificar el nombre de cualquier elemento: variable, atributo, método o clase. Los cambios se realizan también sobre las referencias que haya a dicho elemento en todo el proyecto. Por ejemplo:

```

1 package principal;
2 import clases.profesor;
3 // Clase color, el profesor pregunta
4 public class clasecolor {
5     public static void main(String args) {
6         profesor teacher = new profesor();
7         String color = teacher.getColor();
8         System.out.println("La respuesta es: " + color);
9     }
}

```

```

1 package principal;
2 import clases.profesor;
3 // Clase color, el profesor pregunta
4 public class clasecolor {
5     public static void main(String args) {
6         profesor tch = new profesor();
7         String color = tch.getColor();
8         System.out.println("La respuesta es: " + color);
9     }
}

```

### 1.4.2. Mover

Cambia una clase de un paquete a otro, afectando a su declaración "package" y a su localización en el disco. por ejemplo:

The screenshot shows three Java code editors side-by-side. The left editor shows the 'ordenador.java' file in the 'clases' package. The middle editor shows the same file moved to the 'principal' package. A green arrow points from the left editor to the middle one. The right editor shows the 'ordenador.java' file in the 'principal' package, with other files like 'estudiante.java', 'persona.java', 'profesor.java', 'clasecolor.java', and 'ordenador.java' also present.

```

1 package clases;
2
3 import java.util.
4 public class ordenador {
5     public String color;
6     Random random = new Random();
7     int randomInt = random.nextInt(3);
8     if(randomInt == 0) {
9         return "rojo";
10    } else if(randomInt == 1) {
11        return "amarillo";
12    } else {
13        return "verde";
14    }
15 }

```

```

1 package principal;
2
3 import java.util.Random;
4 public class ordenador {
5     public String color;
6     Random random = new Random();
7     int randomInt = random.nextInt(3);
8     if(randomInt == 0) {
9         return "rojo";
10    } else if(randomInt == 1) {
11        return "amarillo";
12    } else {
13        return "verde";
14    }
15 }

```

### 1.4.3. Cambiar firma del método

Modificar la firma o cabecera del método. Si este ya ha sido usado, cambiar el número o tipo de parámetros, o el tipo devuelto provoca fallos de compilación.

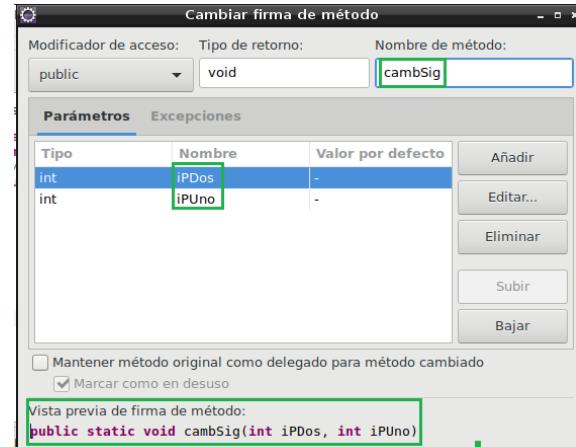
Es útil para cambiar el nombre de los parámetros o su orden (se modificará también el orden en todas las llamadas al método).

Ejemplo: Cambiar el nombre del método y los parámetros:

```

1 package principal;
2 import clases.profesor;
3 // Clase color, el profesor pregunta a un alumno por un color en
4 public class clasecolor {
5     public static void main(String[] args) {
6         profesor tch = new profesor();
7         String color = tch.preguntacolor();
8         System.out.println("La respuesta recibida es:" + color);
9     }
10 }
11
12 public static void CambSignatura(int iParamUno, int iParamDos) {
13     System.out.println("Primer Parametro" + iParamUno);
14     System.out.println("Segundo Parametro" + iParamDos);
15 }
16
17
18 }

```



```

1 package principal;
2 import clases.profesor;
3 // Clase color, el profesor pregunta a un alumno por un color en
4 public class clasecolor {
5     public static void main(String[] args) {
6         profesor tch = new profesor();
7         String color = tch.preguntacolor();
8         System.out.println("La respuesta recibida es:" + color);
9     }
10 }
11
12 public static void cambSig(int iParamUno, int iParamDos) {
13     System.out.println("Primer Parametro" + iParamUno);
14     System.out.println("Segundo Parametro" + iParamDos);
15 }
16
17
18 }

```

### 1.4.4. Extraer variable local

Crear una variable local inicializada con el valor de un literal. Las referencias a esa expresión se modifican por una referencia a la variable.

The diagram illustrates the extraction of a local variable. On the left, a code editor shows a Java method main with a line of code: `System.out.println("La respuesta recibida es:" + color);`. A green box highlights the string "La respuesta recibida es:". A blue arrow points down to a window titled "Extraer variable local". In this window, the variable name `sResp` is entered into the "Nombre de variable:" field. A checked checkbox indicates "Sustituir todas las apariciones de la expresión seleccionada por referencias a la variable local". Below the window, the modified code is shown: `String sResp = "La respuesta recibida es:"; System.out.println(sResp + color);`.

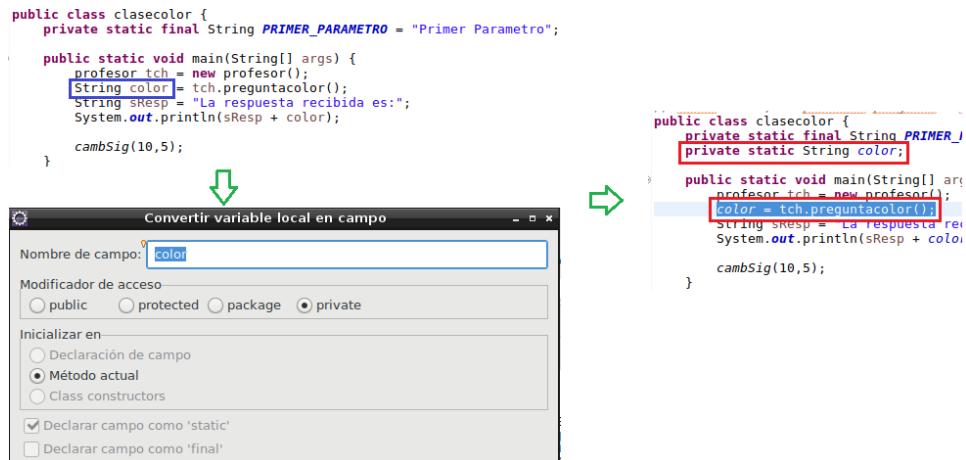
### 1.4.3. Extraer constante

Genera una constante a partir de la expresión seleccionada.

The diagram illustrates the extraction of a constant. On the left, a code editor shows a Java method `cambSig` with two `System.out.println` statements. Both statements contain the string "Primer Parametro" followed by a plus sign and a variable. A green box highlights the first "Primer Parametro". A blue arrow points down to a window titled "Extraer constante". In this window, the constant name `PRIMER PARAMETRO` is entered into the "Nombre de constante:" field. A checked checkbox indicates "Sustituir todas las apariciones de la expresión seleccionada por referencias a la constante". Below the window, the modified code is shown: `private static final String PRIMER_PARAMETRO = "Primer Parametro";` and the original code with the constant reference: `System.out.println(PRIMER_PARAMETRO + iPUno);`

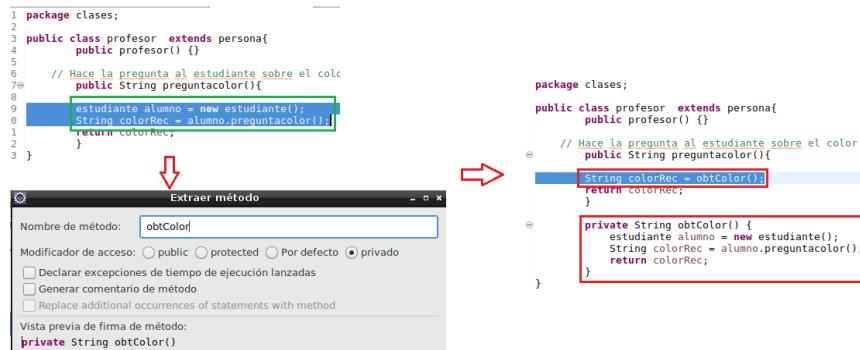
### 1.4.6. Convertir variable local en atributo

A veces se definen variables locales dentro de un método y luego decidimos usarlas como variables de clase.



## 1.4.7. Extraer método

Convierte el código seleccionado en un método, útil para reutilizarlo en varios sitios del programa.



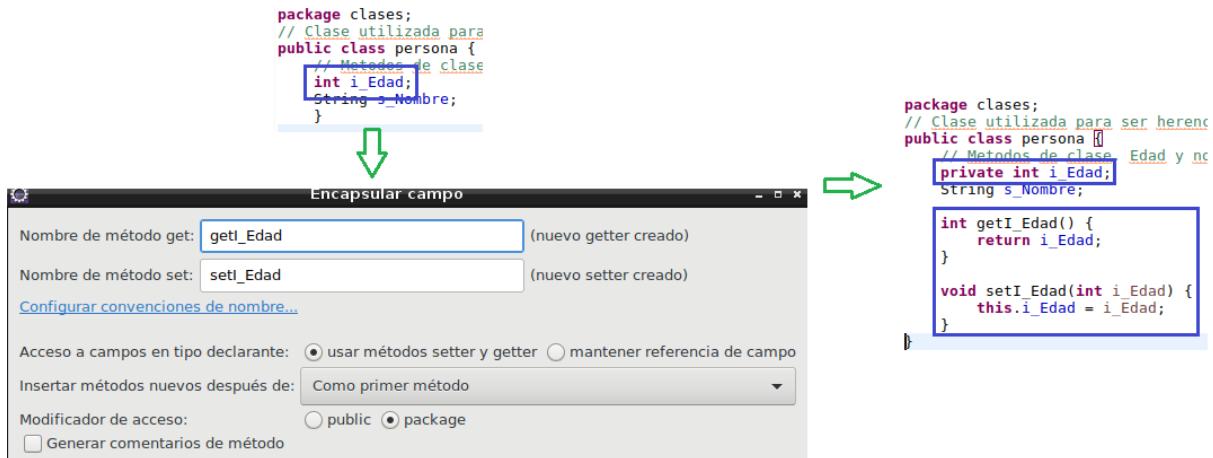
## 1.4.8. Incorporar

Hace lo contrario que los extract. Elimina una declaración de variable, método o constante y coloca su valor (en el caso de variables y constantes) o su código (en el caso de métodos), en los lugares donde se referenciaban.

Útil si la variable, constante solo se utiliza una vez y no merece la pena almacenar su valor, o cuando el método solo se utiliza una o dos veces y no merece la pena aislarlo.

## 1.4.9. Autoencapsular atributo

Convierte una variable de clase en privada y genera los métodos Get y Set públicos para acceder a la misma.



## 2. Control de versiones

Cuando se desarrolla un software, el código fuente cambia constantemente, ya sea por el propio desarrollo o por el mantenimiento. Además en ocasiones se desarrollan por fases o entregas a cliente. Se hace necesario por tanto un sistema de control de versiones.

Un sistema de control de versiones bien diseñado facilita el desarrollo, permitiendo que varios programadores trabajen en el mismo proyecto e incluso sobre los mismos archivos de manera simultánea, gestionando los conflictos que puedan surgir de actualizaciones simultáneas sobre el mismo código. Las herramientas de control de versiones proveen de un sitio central donde se almacena el código de la aplicación y un historial de cambios realizado a lo largo de la vida del proyecto. También se permite volver a versiones estables previas si es necesario.

Una versión se define como la forma particular de un objeto en un instante o contexto dado. Se denomina revisión cuando se refiere a la evolución en el tiempo.

Pueden coexistir varias versiones alternativas en un instante dado y para ello se debe disponer de un método para designar las diferentes versiones de manera organizada.

Las alternativas de código abierto son: GIT, CVS, Subversion, Mercurial...

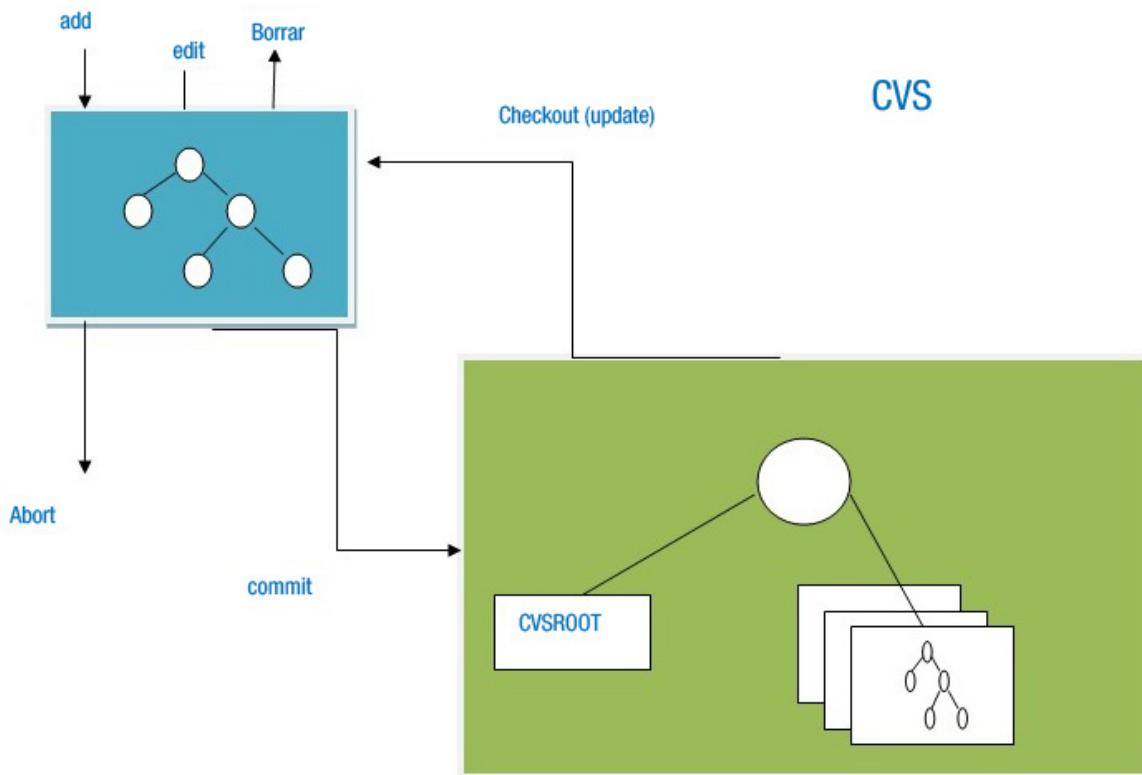
### 2.1. Tipos de herramientas de control de versiones

Se pueden clasificar las herramientas de control de versiones como:

- **Sistemas locales.** Control de versiones que se guarda en diferentes directorios en función de sus versiones. La gestión recae sobre el responsable del proyecto y no dispone de herramientas de automatización. Viable para pequeños proyectos de un único programador.
- **Sistemas centralizados:** arquitectura cliente - servidor. Un equipo contiene todos los archivos y sus diferentes versiones. Los clientes replican la información en sus entornos de trabajo locales.
- **Sistemas distribuidos.** Cada sistema hace una copia completa de los ficheros de trabajo y de todas sus versiones. Todos los equipos tienen un rol de igual a igual y los cambios se pueden sincronizar entre cada par de copias disponibles. Habitualmente funcionan siendo uno el repositorio principal y el resto asumiendo papel de clientes sincronizando sus cambios a este.

## 2.2. Estructura de herramientas de control de versiones

Suelen estar formadas por un conjunto de elementos sobre los que se pueden ejecutar órdenes e intercambiar datos entre ellos. (Estudiamos la herramienta CVS).



Una herramienta de control de versiones es un sistema de mantenimiento de código fuente extraordinariamente útil para grupos de desarrolladores que trabajan cooperativamente usando alguna clase de red.

CVS permite trabajar y modificar ficheros organizados en proyectos. Dos personas pueden modificar un fichero sin perder el trabajo de ninguna.

CVS utiliza arquitectura cliente-servidor. El servidor guarda la versión actual y su historia. Los clientes conectan al servidor y sacan una copia completa del proyecto, trabajan en ella e ingresan sus cambios. El servidor utilizado es similar a UNIX, pero los clientes pueden funcionar en cualquier S.O.

Los clientes pueden comparar versiones diferentes de ficheros, solicitar historia completa de cambios o sacar foto histórica del proyecto en una fecha determinada o en un número de revisión determinado.

Muchos proyectos de código abierto permiten acceso de lectura anónima, de manera que se pueden descargar una copia del proyecto y solo necesitan contraseña para ingresar cambios.

También se utiliza el comando de actualización para tener sus copias al día de la última versión en servidor.

El sistema de control de versiones está formado por:

- **Repositorio:** Lugar de almacenamiento de los datos de los proyectos. Suele ser un directorio en algún ordenador.
- **Módulo:** directorio específico del repositorio, puede identificar una parte del proyecto o el proyecto completo.
- **Revisión:** Cada una de las versiones parciales o cambios en los archivos o repositorio completo. La evolución del sistema se mide en revisiones y cada cambio se considera incremental.
- **Etiqueta:** información textual que se añade a un conjunto de archivos o a un módulo completo para aportar información importante.
- **Rama:** revisiones paralelas de un módulo para efectuar cambios sin tocar la evolución principal. Se emplea para pruebas o para obtener cambios en versiones antiguas.

Algunos servicios que se proporcionan son:

- **Creación de repositorios.** Esqueleto de un repositorio sin información inicial del proyecto.

- **Clonación de repositorios.** Crea un nuevo repositorio y vuelca la información de algún otro repositorio existente (réplica).
- **Descarga de la información del repositorio principal al local.** Sincroniza la copia local con la información disponible en el repositorio principal.
- **Carga de información al repositorio principal desde local.** Actualiza los cambios realizados en la copia local en el repositorio principal.
- **Gestión de conflictos.** Si los cambios que se desean realizar en el repositorio principal entran en conflicto con otros cambios que se han subido por otro desarrollador, se trata de combinar automáticamente todo los cambios y, si no es posible por pérdida de información, se muestran al programador los conflictos para que se tome una decisión de como combinarlos.
- **Gestión de ramas.** Creación, eliminación, integración de diferencias entre ramas y selección de ramas de trabajo.
- **Información sobre registro de actualizaciones.**
- **Comparativa de versiones.** Genera información sobre las diferencias entre versiones del proyecto.

Las órdenes a ejecutar son:

- **checkout:** obtiene una copia del trabajo para trabajar con ella.
- **Update:** actualiza la copia con cambios recientes en el repositorio
- **Commit:** almacena la copia modificada en el repositorio
- **Abort:** abandona los cambios en la copia de trabajo.

### 2.2.1. Repositorio

El repositorio es un almacén general de versiones, suele ser un directorio en la mayoría de herramientas.

El repositorio centraliza todos los componentes de un mismo sistema, incluyendo las distintas versiones de cada componente. Se ahorra espacio de almacenamiento ya que evitamos guardar por duplicado elementos comunes a varias versiones. Nos facilita el almacenaje de la info y la evolución del sistema.

Netbeans utiliza el control de versiones CVS, el cual tiene un componente principal: el repositorio, en el cual se deben almacenar todos los ficheros de los proyectos que puedan ser accedidos por varios desarrolladores.

Cuando usamos un sistema de control de versiones se trabaja de forma local y sincronizamos con el repositorio haciendo los cambios en nuestra copia local. Realizando el cambios se realiza también en el repositorio. En Netbeans se realiza de varias formas:

- Abrir un proyecto CVS en el IDE.
- Comprobando los archivos de un repositorio.
- Importando los archivos hacia un repositorio.

Teniendo un proyecto CVS versionado, podemos abrirlo en el IDE y acceder a las características del versionado. El IDE escanea nuestros proyectos abiertos y si contienen directorios CVS, se activan automáticamente el estado del archivo y la ayuda contextual para proyectos de versiones CVS.

### 2.2.2. Gestión de versiones y entregas

Las versiones hacen referencia a la evolución de un único elemento dentro de un sistema software. Puede representarse en forma de grafo, donde los nodos son las versiones y los arcos son la creación de una versión a otra parte ya existente.



**Grafo de evolución simple:** las revisiones sucesivas de un componente dan lugar a una simple secuencia lineal. No presenta problemas en la organización del repositorio y las versiones se designan con números correlativos.

**Variantes:** existen varias versiones del componente, el grafo no es secuencia lineal y adopta forma de árbol. La numeración requiere varios niveles. El primer número designa la variante o línea de evolución, el segundo la revisión a lo largo de dicha variante.

La terminología para referirse a los elementos del grafo son:

- Tronco (trunk). Variante principal
- Cabeza (head). Última versión del tronco

- **Ramas (branches):** son las variantes secundarias
- **Delta:** es el cambio de una revisión respecto a la anterior.

**Propagación de cambios:** cuando se tienen variantes que se desarrollan en paralelo es necesario aplicar un mismo cambio a varias variantes.

**Fusión de variantes:** Se puede fundir una rama con otra (Merge)

**Técnicas de almacenamiento:** las distintas versiones tienen en común parte del contenido. Para aprovechar el espacio se utilizan distintas técnicas:

- **Deltas directos:** se almacena la primera versión completa y luego los cambios mínimos para reconstruir cada nueva versión a partir de la anterior
- **Deltas inversos:** se almacena completa la última versión del tronco y los cambios necesarios para reconstruir cada versión anterior a partir de la siguiente. En las ramas se mantiene el uso de deltas directos.
- **Marcado selectivo:** se almacena el texto refundido de todas las versiones como secuencia lineal, marcando cada sección del conjunto con los números de versiones correspondiente.

Una entrega es una instancia de un sistema que se distribuye a los usuarios externos al equipo de desarrollador.

La planificación de la entrega se ocupa de cuando emitir una versión del sistema como una entrega. La entrega es un conjunto de programas ejecutables, archivos de configuración, que definen como se configura la entrega para la instalación particular.

## 2.3. Herramientas de control de versiones

Además de CVS y subversion (sucesor natural de CVS), existen otras herramientas de amplia difusión:

- **SourceSafe:** forma parte de Microsoft Visual Studio.
- **Visual Studio Team Foundation Server:** es el sustituto de Source Safe, ofrece control de código fuente, recolección de datos, informes, seguimiento de proyectos... destinado a proyectos de colaboración.
- **Darcs:** Sistema de gestión de versiones distribuido: posibilidad de hacer commits locales (sin conexión), cada repositorio es una rama en sí misma, independencia de servidor central, posibilidad de renombrar ficheros, varios métodos de acceso como local: ssh, http y ftp, etc.

- **Git:** herramienta de control de versiones, diseñada por Linus Torvalds
- **Mercurial:** funciona en linux, windows y mac os x, programa en línea de comandos, permite desarrollo distribuido, capacidades avanzadas de ramificación e integración.

## 2.4. Planificación de la gestión de configuraciones

La gestión de configuraciones es un conjunto de actividades para gestionar los cambios a lo largo del ciclo de vida y está regulado por el estandar IEEE 828.

Una configuración es una combinación de versiones particulares de los componentes que forman un sistema consciente. Desde el punto de vista de la evolución en el tiempo, es el conjunto de las versiones de los objetos componentes en un instante dado.

Una configuración cambia porque se añaden o modifican elementos, o debido a la reorganización de los componentes sin que estos cambien.

La gestión de configuraciones de software compone cuatro tareas básicas:

- **Identificación:** se trata de establecer estándares de documentación y esquema de identificación de documentos.
- **Control de cambios:** evaluación y registro de todos los cambios que se hagan a la configuración software.
- **Auditoría de configuraciones:** garantizan que el cambio se haga correctamente.

## 2.5. Gestión del cambio

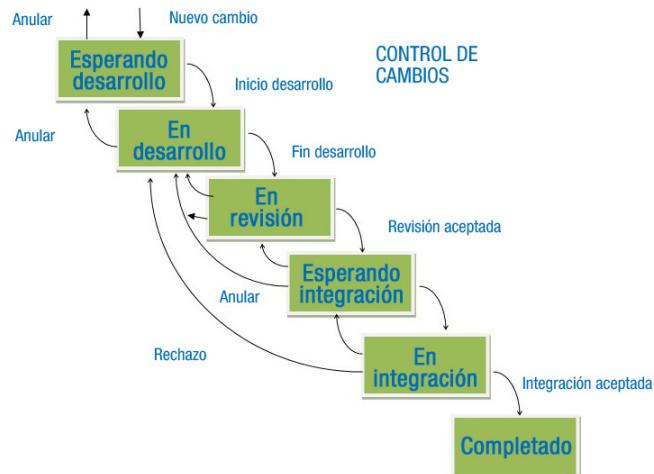
Las herramientas de control de versiones no garantizan desarrollos razonables. Para garantizar que siempre disponemos de una línea base para continuar el desarrollo hay que aplicar controles al desarrollo e integración de los cambios.

Se pueden establecer:

- **Control individual:** antes de aprobarse un nuevo elemento. El programador cambia la documentación cuando se requiere, el cambio se registra de manera individual pero no genera un documento formal.
- **Control de gestión organizado,** conduce a la aprobación de un nuevo elemento e implica procedimiento de revisión y aprobación para cada cambio propuesto en la configuración. Ocurre también durante el proceso

de desarrollo pero es usado después de haber sido aprobado en configuración de software.

- **Control formal**, durante el mantenimiento, el impacto de cada tarea se evalúa por un comité de control de cambios, que aprueba las modificaciones de configuración de software.



## 3. Documentación

El proceso de documentación de código es uno de los aspectos más importantes de un programador. El desarrollo rápido de aplicaciones va en detrimento de una buena documentación.

```
/* Método para ingresar cantidades en la cuenta. Modifica el saldo.
 * Este método va a ser probado con Junit
 */
/** Comentarios estilo Javadoc
 *
 * @param cantidad
 * @throws Exception
 */
public void ingresar(double cantidad) throws Exception {
    // el parámetro cantidad debe ser positivo. Comentario de una línea
    if (cantidad<0)
        throw new Exception("No se puede ingresar una cantidad negativa");
    saldo = saldo + cantidad;
    /* Este método realiza el ingreso de una cantidad de dinero
     * la cuenta Comentario multilínea
    */
}
```

Es fundamental para la detección de errores y para el mantenimiento posterior. Explica lo que no resulta evidente, cual es la finalidad de una clase, un paquete, un método o una variable, que se espera del uso de una variable, que algoritmo se usa, etc...

### 3.1. Uso de comentarios

Los comentarios tienen dos propósitos distintos:

- Explicar el objetivo de las sentencias, de manera que se sepa en todo momento la función de esa sentencia.
- Explicar qué realiza un método o clase, no cómo lo realiza. Se trata de explicar los valores que devolverá un método.

En el caso de comentarios de una sola línea, en Java, C# y C se utilizan los caracteres // y en el caso de comentarios multilínea se utiliza /\* \*/.

Es conveniente poner comentarios al principio de un fragmento de código, a lo largo de bucles o si hay alguna línea que no resulta evidente y puede llevar a confusión.

Cuando se modifica código, también se deben modificar los comentarios.

## 3.2. Documentación de clases

Existen distintas herramientas para automatizar, completar y enriquecer la documentación: JavaDoc, SchemeSpy, Doxygen, estos dos últimos incluyen modelos de bases de datos y diagramas.

Las clases que implementan una aplicación deben incluir comentarios, Debemos seguir una serie de pautas cuando implementamos una clase, para documentarla.

En los IDE de java, se incluye una herramienta que genera HTML de documentación a partir de los comentarios del código. Este es JavaDoc. Para ello se siguen una serie de normas:

- Los comentarios se hacen al principio de cada clase, cada método y cada variable de clase. Se escriben empezando por /\*\* y terminando con \*/
- Los comentarios son a nivel de clase, nivel variable y nivel método.
- Se genera para métodos public y protected.
- Se pueden usar tags para documentar diferentes aspectos del código como los parámetros.

Las tags se añaden de forma automática, como la de @author y @version también se suele añadir @see para referenciar a otras clases y métodos.

Estas son las más utilizadas:

Etiqueta y parámetros	Uso	Asociada a
@author <i>nombre</i>	Nombre del autor (programador)	Clase, interfaz
@version <i>numero-version</i>	Comentario con datos indicativos del número de versión.	Clase, interfaz
@since <i>numero-version</i>	Fecha desde la que está presente la clase.	Clase, interfaz, campo, método.
@see <i>referencia</i>	Permite crear una referencia a la documentación de otra clase o método.	Clase, interfaz, campo, método.
@param seguido del nombre del parámetro	Describe un parámetro de un método.	Método.
@return descripción	Describe el valor devuelto de un método.	Método.
@exception <i>clase descripción</i> @throws <i>clase descripción</i>	Comentario sobre las excepciones que lanza.	Método.
@deprecated descripción	Describe un método obsoleto.	Clase, interfaz, campo, método.

Los campos de una clase, también pueden incluir comentarios, aunque no existen etiquetas obligatorias en JavaDoc.



# 5. Diseño orientado a objetos. Elaboración de diagramas estructurales

	Autor	Xerach Casanova
	Clase	Entornos de desarrollo
	Fecha	@Feb 27, 2021 4:23 PM

## 1. Programación orientada a objetos

- 1.1. Conceptos de orientación a objetos
- 1.2. Ventajas de la orientación a objetos
- 1.3. Clases, atributos y métodos
- 1.4. Visibilidad
- 1.5. Objetos. Instanciación

## 2. UML

### 2.1 Algunos conceptos UML

- 2.1.1. Notación
- 2.1.2. Modelos y herramientas
- 2.1.3. Métodos

### 2.2. Tipos de diagrama UML

### 2.3. Herramientas para la elaboración de diagramas UML

- 2.3.1. Generación de la documentación
- 2.3.2. - UMLet

### 2.4. Ingeniería inversa

## 3. Diagrama de clases

### 3.1. Creación de clases

### 3.2. Atributos

### 3.3. Métodos

### 3.4. Relaciones entre clases

- 3.4.1. Cardinalidad o multiplicidad de la relación
- 3.4.2. Relación de herencia (generalización)
- 3.4.3. Agregación y composición
- 3.4.4. Atributos de enlace
- 3.4.5. Restricciones

### 3.5. Pautas para crear diagramas de clase

#### 3.5.1. Obtención de atributos y operaciones

#### 3.6. Generación de código a partir del diagrama de clases

##### 3.6.1. Elección del lenguaje de programación. Orientaciones para el lenguaje java.

#### Mapa conceptual

# 1. Programación orientada a objetos

En la construcción de software, si queremos obtener un producto de calidad, es preciso realizar un proceso previo de análisis y especificación del proceso que vamos a seguir.

## **El enfoque estructurado**

Los problemas se someten a un proceso de división en subproblemas reiteradamente hasta llegar a problemas elementales resueltos con una función. Todas las funciones resultantes se hilan y entrelazan hasta formar una solución global. Este proceso está centrado en procedimientos que se codifican mediante funciones que actúan sobre estructuras de datos.

## **Enfoque orientado a objetos**

Con este paradigma el proceso se centra en simular elementos de la realidad asociada al problema, de la forma más cercana posible. La abstracción que representa estos elementos se denomina objeto:

- Está formado por un conjunto de atributos, que son los datos que lo caracterizan.
- Realizan un conjunto de operaciones que definen su comportamiento. Estas actúan sobre sus atributos para modificar su estado. Cuando le decimos a un objeto que ejecute una operación determinada, se dice que se le pasa un mensaje.

Las aplicaciones orientadas a objetos se forman por un conjunto de objetos que interaccionan enviándose mensajes para producir resultados. Los objetos similares se abstraen en clases. Un objeto es una instancia de una clase.

Cuando se ejecuta un programa ocurren tres sucesos:

- Primero, los objetos se crean a medida que se necesitan.
- Segundo. Los mensajes se mueven de un objeto a otro, o del usuario a un objeto, a medida que el programa procesa información o responde a la entrada del usuario.

- Tercero, cuando los objetos ya no se necesitan se borran y liberan memoria.

## 1.1. Conceptos de orientación a objetos

- **Abstracción.** Permite capturar las características y comportamiento de un conjunto de objetos y darles una descripción formal. Es clave en el proceso de análisis y diseño orientado a objetos. Mediante ella, se puede armar el conjunto de clases para modelar la realidad.
- **Encapsulación.** Organiza los datos y métodos de una clase, evitando el acceso a datos por cualquier otro medio distinto a los definidos. El estado de los objetos solo debe ser modificado desde métodos de la propia clase. Este concepto y el principio de ocultación van de la mano.
- **Modularidad.** Permite subdividir una aplicación en partes más pequeñas, llamadas módulos. Cada una de ellas debe ser lo más independiente posible, tanto de la aplicación en sí como de las partes restantes.
- **Principio de ocultación.** La implementación de la clase solo la conocen los responsables de su desarrollo, esta puede ser modificada para mejorar su algoritmo sin tener repercusión en el resto del programa. Principalmente se ocultan las propiedades del objeto y se reduce la propagación de efectos colaterales cuando se producen cambios.
- **Polimorfismo.** Se reúnen bajo el mismo nombre, comportamientos distintos. La selección de uno u otro depende del objeto que lo ejecute.
- **Herencia:** Relación entre objetos en los que unos utilizan las propiedades y comportamientos de otros, formando una jerarquía. Los objetos heredan propiedades y comportamiento de las clases a las que pertenecen.
- **Recolección de basura.** El entorno de objetos se encarga de destruir automáticamente objetos y desvincularlos de la memoria asociada.

## 1.2. Ventajas de la orientación a objetos

1. Permite desarrollar software en menos tiempo y con menos coste y de mayor calidad gracias a la reutilización de los módulos.
2. Consigue aumentar la calidad de los sistemas, haciéndolos más extensibles ya que es sencillo aumentar o modificar la funcionalidad de la aplicación modificando operaciones.

3. Facilidad para modificar y mantener gracias a la modularidad y encapsulación.
4. Facilita la adaptación al entorno y el cambio haciendo aplicaciones escalables. Se puede modificar la estructura y el comportamiento de los objetos sin cambiar la aplicación.

## 1.3. Clases, atributos y métodos

Una clase está formada por un conjunto de procedimientos y datos que resumen características similares a un conjunto de objetos. La clase tiene dos propósitos: definir abstracciones y favorecer modularidad.

Los elementos de una clase se denominan miembros y son:

- **Atributos.** Conjunto de características asociadas a una clase. Cuando se toman valores concretos dentro de su dominio, se define el estado del objeto. Se definen por su nombre y su tipo, pueden ser simples o compuestos como otra clase.
- **Protocolo:** Operaciones (métodos y mensajes) que manipulan el estado. Un método es el procedimiento o función que se invoca para actuar sobre un objeto. Un mensaje es el resultado de cierta acción efectuada sobre un objeto. Los métodos determinan como actúan los objetos cuando reciben un mensaje. El conjunto de mensajes a los cuales puede responder un objeto se le conoce como protocolo del objeto.

Los valores asignados a los atributos de un objeto concreto hacen a ese objeto ser único. La clase a la que pertenece define sus características generales y su comportamiento.

## 1.4. Visibilidad

El aislamiento protege los datos de que sean modificados por alguien que no tenga derecho a acceder a ellos. Se eliminan así efectos secundarios e interacciones.

1. **Interfaz:** captura la visión externa de una clase, abarcando la abstracción del comportamiento común a los ejemplos de esa clase.
2. **Implementación:** comprende como se representa la abstracción y los mecanismos que conducen al comportamiento deseado.

Existen distintos niveles de ocultación que se implementan en lo que se denomina visibilidad. Es una característica que define el tipo de acceso a atributos y métodos:

- **Público:** se puede acceder desde cualquier clase y cualquier parte del programa.
- **Privado:** solo pueden acceder desde operaciones de la clase.
- **Protegidos:** solo se puede acceder desde operaciones de la clase o derivadas en cualquier nivel.

A la hora de definir la visibilidad se tiene en cuenta que:

- El estado debe ser privado en los atributos de una clase. Se deben modificar mediante métodos de la clase creados a tal efecto.
- Las operaciones que definen la funcionalidad de la clase deben ser públicas.
- Las operaciones que ayudan a implementar parte de la funcionalidad deben ser privadas siempre que no se utilicen desde clases derivadas o privadas, si se utilizan desde clases derivadas.

## 1.5. Objetos. Instanciación

Cada vez que construimos un objeto en un programa a partir de una clase, se crea una instancia de ella. Cada instancia en el sistema sirve como modelo de un objeto del contexto del problema relevante para su solución. Puede realizar un trabajo, informar y cambiar su estado, y comunicarse con otros objetos del sistema, sin revelar como se implementan estas características.

Un objeto se define por:

- Su estado: cada atributo definido tiene un valor concreto.
- Su comportamiento: definido por los métodos públicos de su clase.
- Tiempo de vida: intervalo de tiempo en el programa, que el objeto existe. Comienza su creación en la instanciación y finaliza cuando se destruye.

La encapsulación y ocultamiento aseguran que los datos del objeto están ocultos y que no se pueden modificar accidentalmente por funciones externas.

Hay un caso particular, llamada clase abstracta, que no puede ser instanciada. Se suele usar para definir métodos genéricos relacionados con el sistema que

no son traducidos a objetos concretos, o para definir interfaces de métodos, cuya implementación se realiza en clases derivadas.

Ejemplos de objetos:

1. Objetos físicos: aviones en un sistema de control de tráfico aéreo, casas, parques...
2. Elementos de interfaces gráficas de usuario: ventanas, menús, teclado, cuadros de diálogo...
3. Animales: vertebrados, invertebrados...
4. Tipos de datos definidos por el usuario: Datos complejos, puntos de un sistema de coordenadas...
5. Alimentos: carnes, frutas, verduras...

## 2. UML

Unified Modeling Language o Lenguaje Unificado de Modelado. Es un conjunto de herramientas que permite modelar, construir y documentar los elementos que forman un sistema de software orientado a objetos. Es el estándar de facto de la industria, concebido por los autores de los tres métodos más usados de la orientación a objetos: Grady Booch, Ivar Jacobson y Jim Rumbaugh

- OMT - Object Modeling Technique (Rumbaugh et al.).
- Método - Booch (G. Booch)
- OOSE - Object-Oriented Software Engineering (I. Jacobson)

### ¿Por qué es útil modelar?

- Se permite utilizar un lenguaje que facilita la comunicación entre el equipo de desarrollo.
- Con UML podemos documentar todos los artefactos de un proceso de desarrollo: requisitos, arquitectura, pruebas, versiones...). Por lo que se dispone documentación que trasciende al proyecto.
- Hay estructuras que trascienden lo representable en un lenguaje de programación, como las que hacen referencia a la arquitectura del sistema, utilizando estas tecnologías podemos incluso indicar que módulos de

software vamos a desarrollar y sus relaciones, o en qué nodos hardware se ejecutan en sistemas distribuidos.

- Permite especificar todas las decisiones de análisis, diseño e implementación, construyéndose modelos precisos, no ambiguos y completos.

## 2.1 Algunos conceptos UML

### 2.1.1. Notación

Es un conjunto de símbolos y técnicas para combinarlos, que en el contexto UML permiten crear diagramas normalizados. Estas representaciones posibilitan al analista o desarrollador describir el comportamiento del sistema (análisis) y detalles de una arquitectura (diseño) de forma ambigua.

Que una notación sea detallada no significa que todos sus aspectos deban ser utilizados en todas las ocasiones. Utilizar UML debe facilitar el desarrollo y entendimiento de todos los participantes en el proyecto, pero no complicarlo. Utilizar todos los diagramas al máximo nivel puede liar más que aclarar.

Las notaciones UML deben ser independientes del lenguaje de programación.

Un diagrama es una representación gráfica de una colección de elementos de modelado (modelo), a menudo dibujada en un grafo con vértices conectados por arcos (notación).

- **Notación musical -1.** La notación musical formal considera términos como pentagrama, nota redonda, blanca, negra, corchea, semicorchea, fusa, semifusa, clave de sol, fa, do...
- **Notación musical -2.** Otra alternativa útil para inexpertos es escribir la secuencia de notas que forman la melodía: DO, RE, MI, FA, SOL, LA SI.

### 2.1.2. Modelos y herramientas

Un modelo captura una vista de un sistema del mundo real. Es una abstracción de dicho sistema considerando un cierto propósito. El modelo describe completamente aquellos aspectos del sistema que son relevantes al propósito del modelo y a un apropiado nivel de detalle.

Volviendo al ejemplo musical, la melodía se puede ver desde diferentes vistas:

- **Vista - 1.** Utilizando la primera notación del apartado anterior.



- **Vista -2** Usando la segunda notación del apartado anterior:

DO - SI - DO - FA - LA - FA# - MIB

- **Vista -3** La interpretación de la melodía puede ser una vista distinta, se pueden considerar vistas distintas en función del compás, instrumento...

En UML existen una serie de modelos o diagramas que nos proporcionan vistas en las fases de análisis y diseño.

Cada modelo es completo desde su punto de vista del sistema, pero existen relaciones de trazabilidad entre diferentes modelos.

Una herramienta de soporte automático de una notación, en nuestro ejemplo hablaríamos de un lápiz, un cuaderno musical, un programa de ordenador, un órgano...

Para el desarrollo de diagramas UML se usará la herramienta UMLet.

### 2.1.3. Métodos

Un método es un proceso disciplinado para generar uno o varios modelos que describen aspectos del sistema de software en desarrollo, utilizando alguna notación bien definida.

## 2.2. Tipos de diagrama UML

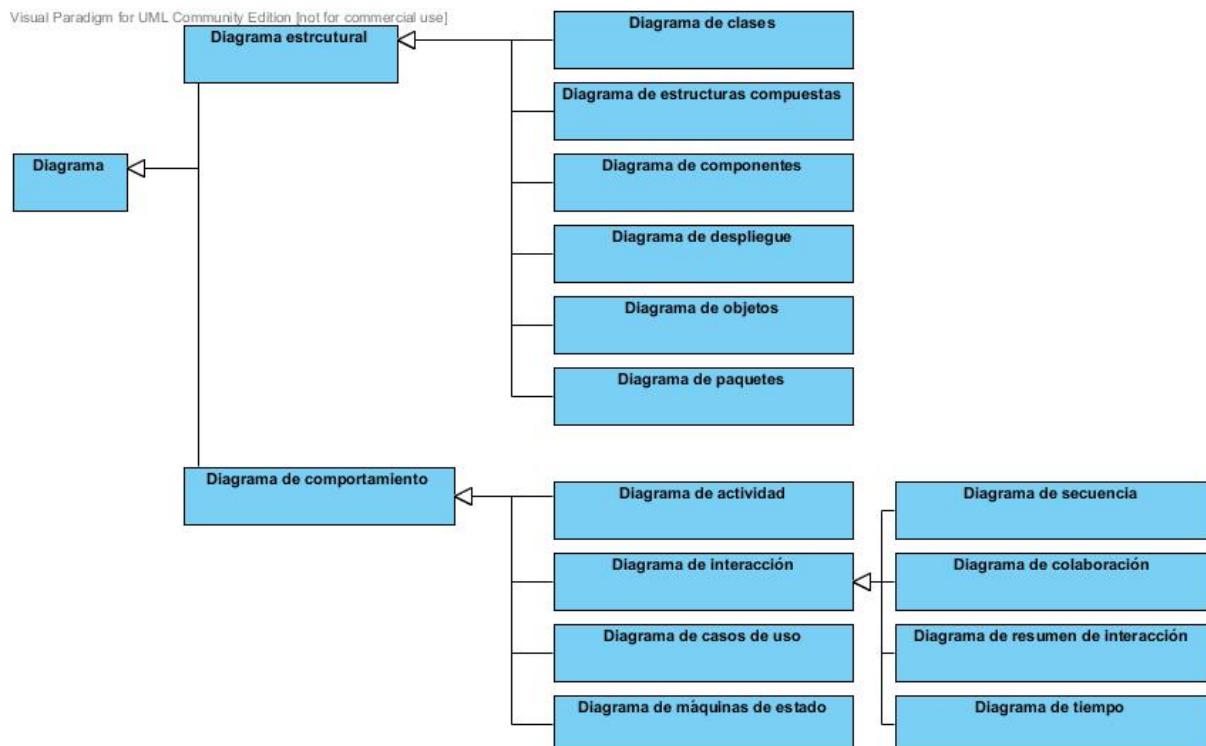
UML define un sistema como una colección de modelos que describen sus diferentes perspectivas. Los modelos se implementan en una serie de diagramas que contienen una colección de elementos de modelado, dibujando como un grafo conexo de arcos (relaciones y vértices (otros elementos del modelo)).

Se clasifican en

- **Diagramas estructurales.** Representan la visión estática del sistema. Especifican clases y objetos y como se distribuyen en el sistema físicamente.

- **Diagramas de clases.** Muestra los elementos del modelo estático abstracto, formado por un conjunto de clases y sus relaciones.
- **Diagramas de objetos.** Muestra los elementos del modelo estático en un momento concreto, habitualmente en casos especiales de un diagrama de clases o de comunicaciones. Está formado por el conjunto de objetos y sus relaciones.
- **Diagrama de componentes.** Especifica la configuración lógica de la implementación de una aplicación, indicando sus componentes, sus interrelaciones, interacciones y sus interfaces públicas y las dependencias entre ellas.
- **Diagrama de despliegue.** Representa la configuración del sistema en tiempo de ejecución, aparecen los nodos de procesamiento y sus componentes. Exhibe la ejecución de la arquitectura del sistema. Incluye nodos, ambientes operativos de hardware y software interfaces que las conectan.... Se utiliza en sistemas distribuidos.
- **Diagrama de estructuras compuestas.** Muestra la estructura interna de una clase, e incluye los puntos de interacción de esta con otras partes del sistema.
- **Diagrama de paquetes.** Exhibe cómo los elementos del modelo se organizan en paquetes, así como las dependencias entre ellos. Son útiles en sistemas de mediano o gran tamaño.
- **Diagramas de comportamiento.** muestran la conducta en tiempo de ejecución del sistema, tanto desde el punto de vista del sistema completo como de las instancias u objetos que lo integran.
  - **Diagrama de caso de uso.** Representa las acciones a realizar en el sistema desde el punto de vista de los usuarios. Se representan las acciones, los usuarios y relaciones entre ellos. Especifican funcionalidad y comportamiento del sistema.
  - **Diagrama de estado de la máquina.** Describe el comportamiento de un sistema dirigido por eventos. Parecen los estados que puede tener un objeto o interacción.
  - **Diagrama de actividades.** Muestra el orden en el que se van realizando las tareas dentro del sistema. En él aparecen dos procesos de alto nivel de la organización. Incluye flujo de datos, o un modelo de la lógica compleja dentro del sistema.

- **Diagrama de secuencia.** Representa la ordenación temporal en el paso de mensajes. Modela la secuencia lógica a través del tiempo, de los mensajes de las instancias.
- **Diagrama de comunicación/colaboración.** Resalta la organización estructural de los objetos que se pasan mensajes. Ofrece las instancias de las clases, sus interrelaciones y el flujo de mensajes entre ellas. Comúnmente enfoca la organización estructural de los objetos que reciben y envían mensajes.
- **Diagrama de tiempos.** Muestra el cambio en un estado o una condición de una instancia o un rol a través del tiempo. Se usa normalmente para exhibir el cambio en el estado de un objeto en el tiempo, en respuesta a eventos externos.



## 2.3. Herramientas para la elaboración de diagramas UML

Las herramientas CASE facilitan en gran medida el desarrollo de diagramas UML, suelen contar con entorno de ventanas tipo wysiwyg, que permiten documentar los diagramas e integrarse con otros entornos de desarrollo, incluyendo la generación de código y procedimientos de ingeniería inversa.

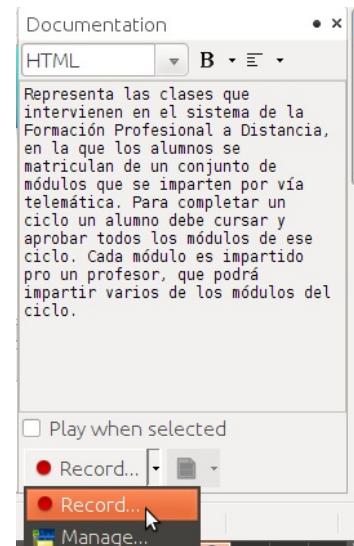
Entre otras podemos encontrar

- **Rational Systems Developer de IBM**, herramienta propietaria, permite desarrollo de proyectos software basados en UML. Creada en su origen por los creadores de UML y absorbida por IBM.
- **Visual Paradigm for UML (VP-UML)**. Incluye versión para uso no comercial que se distribuye libremente al registrarse para obtener archivo de licencia (LGPL).
  - Incluye diferentes módulos para UML, diseñar bases de datos, realizar actividades de ingeniería inversa y diseñar con Agile.
  - Compatible con UML 2.0.
  - Admite generar informes PDF, HTML y otros.
  - Compatible con IDEs como Eclipse, Netbeans, Visual Studio ,net, IntelliJDEA...
  - Multiplataforma
  - Disponible para Windows y Linux

### 2.3.1. Generación de la documentación

Se pueden hacer las anotaciones necesarias abriendo la especificación de cualquiera de los elementos, clases o relaciones, o bien del diagrama en sí mismo en la pestaña "Specification"

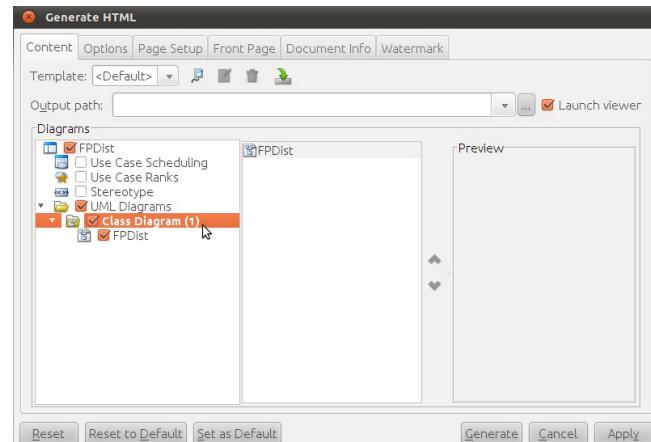
La ventana del editor cuenta con herramientas para formatear texto y añadir elementos como imágenes o hiperenlaces. También se puede grabar voz.



#### Generar informes.

Cuando los modelos están completos se puede generar un informe en varios formatos con la documentación escrita.

Desde VP-UML accedemos a tools/Reports/Report writer y seleccionamos el tipo de informe.



Desde SDE para NetBeans seleccionamos Modelin/reports/report writer.

En ambos, una vez elegido el tipo de informe se obtiene la siguiente ventana en la que se selecciona:

- Qué diagramas queremos que intervengan y donde se almacena el informe.
- La pestaña opciones permite configurar los elementos a añadir al informe: tablas de contenidos, títulos, etc.
- Propiedades de la página.
- Añadir marca de agua.

El resultado es un archivo html, pdf o doc en el directorio donde hayamos indicado con la documentación de los diagramas seleccionados.

### 2.3.2. - UMLet

Para los diagramas UML utilizaremos UMLet:

- Software gratuito
- Multiplataforma
- Incluye módulo de integración con eclipse
- Dispone de versión web que no requiere instalación:  
<http://www.umletino.com/umletino.html>

## 2.4. Ingeniería inversa

Se define como el proceso de analizar código, documentación y comportamiento de una aplicación para identificar sus componentes actuales y sus dependencias para extraer y crear una abstracción del sistema e

información del diseño. El sistema en estudio no es alterado, sino que se produce un conocimiento adicional del mismo.

Tiene como caso particular la reingeniería que es el proceso de extraer el código fuente de un archivo ejecutable.

Puede ser de varios tipos:

- **Ingeniería inversa de datos:** se aplica sobre algún código de bases de datos para obtener modelos relacionales o sobre el modelo relacional para obtener el diagrama entidad - relación.
- **Ingeniería inversa de lógica o de proceso.** Cuando la ingeniería inversa se aplica sobre el código de un programa para averiguar su lógica, o cualquier documento de diseño para obtener sus documentos de análisis o requisitos.
- **Ingeniería inversa de interfaces de usuario.** Consiste en estudiar la lógica interna de las interfaces para obtener los modelos y especificaciones que sirvieron para su construcción, con objeto de tomarlas como punto de partida en procesos de ingeniería directa que permitan su actualización.

### 3. Diagrama de clases

El diagrama de clases se considera el más importante de los existentes en UML. Representa los elementos estáticos del sistema, sus atributos y comportamiento, así como su relación entre ellos. Contiene las clases del dominio del problema y a partir de este se obtienen las clases que formarán el programa informático.

En él encontramos los siguientes elementos:

- **Clases:** agrupan conjuntos de objetos con características comunes, que llamamos atributos y su comportamiento que serán métodos. Atributos y métodos tienen una visibilidad que determina quien puede acceder a ellos.
- **Relaciones:** en el diagrama se representan las relaciones reales entre los distintos elementos a los que hacen referencia las clases. Pueden ser de asociación, agregación, composición y generalización.
- **Notas:** se representan como un cuadro donde podemos escribir comentarios que ayuden al entendimiento del diagrama.

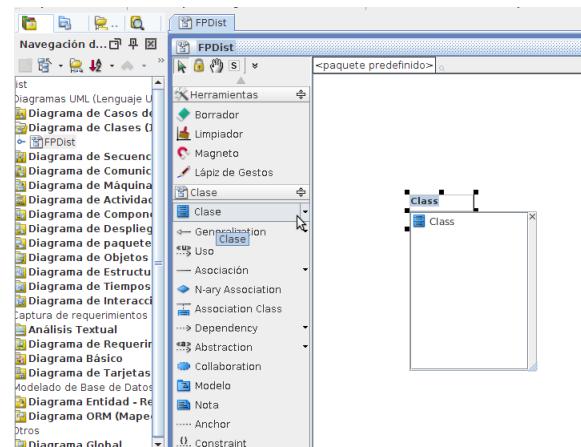
- **Elementos de agrupación:** se utilizan cuando hay que modelar un sistema grande. Las clases y sus relaciones se agrupan en paquetes que se relacionan entre sí.

### 3.1. Creación de clases

Se representa en el diagrama como un cuadro dividido en tres filas: nombre de la clase, atributos con su visibilidad y por último los métodos y su visibilidad.

Nombre Clase
-lista de atributos
+lista de métodos()

Cuando generamos un diagrama nuevo aparece un panel con los elementos que podemos añadir al diagrama. Si hacemos clic sobre el ícono que genera una clase nueva y a continuación sobre el lienzo aparecerá un cuadro para definir el nombre de la nueva clase.



Posteriormente, cuando la clase esté creada si hacemos clic con el botón secundario sobre la clase y seleccionamos "Abrir Especificación" podremos añadir atributos y métodos.

### 3.2. Atributos

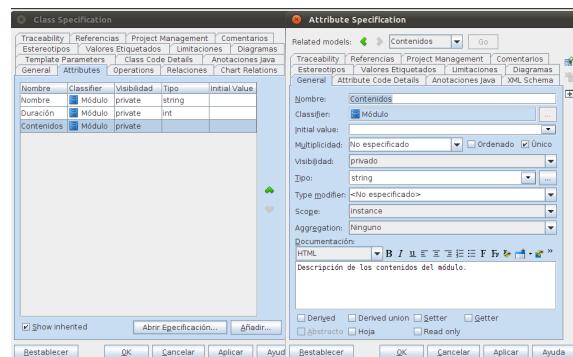
Forman la parte estática de la clase. Son un conjunto de variables para la que es preciso definir su nombre y su tipo, que puede ser simple o compuesto.

Se puede indicar su valor inicial o su visibilidad, que puede ser:

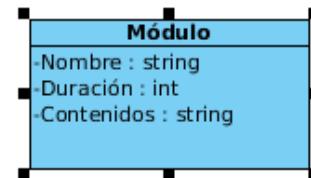
- **Público (+).** Se accede desde cualquier clase y cualquier parte del programa
- **Privado (-).** Solo se accede desde operaciones de la clase.
- **Protegido(#).** Se accede desde clases derivadas en cualquier nivel.

- **Paquete(~).** Se accede desde operaciones de las clases que pertenecen al mismo paquete.

Crear la clase como hemos visto en el punto anterior y modificar su nombre a "Módulo". Para añadir un atributo a una clase basta con seleccionar **Añadir atributo** del menú contextual y escribir su nombre.



Si queremos añadir más información podemos hacerlo desde la especificación de la clase en la pestaña Atributos, en la imagen vemos la especificación de una clase llamada Módulo, y de su atributo Contenidos para el que se ha establecido su tipo (string) y su descripción.



Por defecto la visibilidad de los atributos es privado y no se cambia a menos que sea necesario. Así queda la representación de la clase, los guiones al lado del atributo significan visibilidad privada.

Tenemos la posibilidad de añadir, desde el menú contextual de la clase, con el atributo seleccionado dos métodos llamados getter y setter que se utilizan para leer y establecer el valor del atributo cuando el atributo no es calculado, con la creación de estos métodos se contribuye al encapsulamiento y la ocultación de los atributos.

### 3.3. Métodos

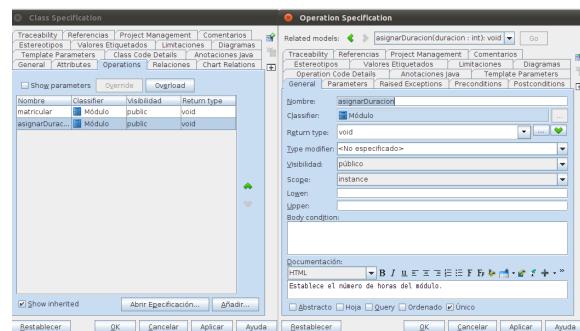
Representan la funcionalidad de la clase. Lo que puede hacer. Para definir un método se indica como mínimo su nombre, parámetros, tipo que devuelve y visibilidad. También se incluye descripción del método que aparece en la documentación que se genere del proyecto. Existen dos métodos particulares:

- **Constructor:** no devuelve ningún valor, tiene el mismo nombre de la clase y ejecuta acciones necesarias cuando se instancia un nuevo objeto.
- **Destructor:** Cuando no se va a utilizar más el objeto se puede utilizar el destructor que libere recursos del sistema que tenía asignado. Se utiliza de

forma diferente según el lenguaje de programación.

El método más directo para crear un método es en el menú contextual seleccionar "Añadir operación" y escribir la firma del método:

+nombre(<lista\_parámetros>) :  
tipo\_devuelto



También se puede añadir desde la especificación de la clase en la pestaña *Operations*.



En la imagen vemos la especificación de la clase y del método "asignarDuración" que asigna el número de horas del módulo.

El signo + en la firma del método indica que es público. Así queda la clase en el diagrama:

### 3.4. Relaciones entre clases

Una relación es una conexión entre dos clases que incluimos en el diagrama.

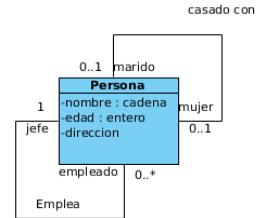
Se representan con una línea continua. Los mensajes navegan por las relaciones entre clases y objetos relacionados, normalmente en ambas direcciones, aunque a veces es necesario navegar en una sola dirección (punta de flecha).

Las relaciones se caracterizan por su cardinalidad, que representa cuantos objetos de una clase se pueden involucrar en la relación.

Creamos la clase como hemos visto en puntos anteriores. Para crear una relación utilizamos el elemento asociación de la paleta o bien el ícono Association >> Class del menú contextual de la clase. Otra forma consiste en hacer clic sobre la clase Alumno, seleccionar Association >> Class y estirar la linea hasta la clase Módulo, aparecerá un recuadro para nombrar la relación.

Es posible establecer relaciones unarias de una clase consigo misma. En el ejemplo se ha llenado

en la especificación de la relación los roles y la multiplicidad.

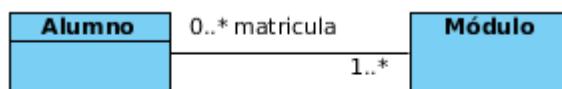


### 3.4.1. Cardinalidad o multiplicidad de la relación

Representa cuantos objetos de una clase se relacionan con objetos de otra clase. En una relación hay una cardinalidad para cada extremo de la relación:

Cardinalidad	Significado
1	Uno y sólo uno
0..1	Cero o uno
N..M	Desde N hasta M
*	Cero o varios
0..*	Cero o varios
1..*	Uno o varios (al menos uno)

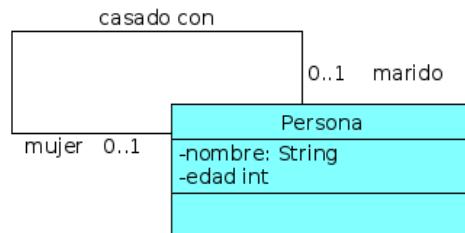
Por ejemplo, en la siguiente relación un alumno se relaciona con la clase módulo. En un módulo pueden matricularse varios alumnos pero puede no tener alumnos matriculados y un alumno puede estar matriculado en varios módulos pero mínimo debe estar matriculado en uno:



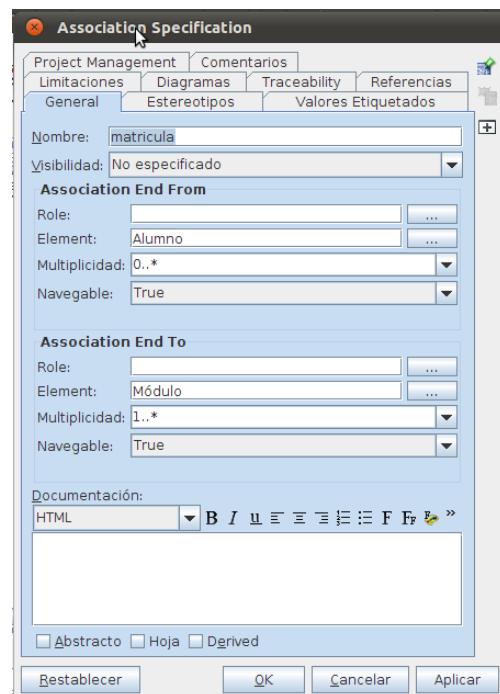
En esta otra quiere decir que la clase Profesor relaciona con la clase Módulo debido a que los profesores imparten diferentes módulos y un módulo es impartido por un profesor. La cardinalidad indicada quiere decir que todo profesor imparte al menos un módulo pudiendo impartir varios y, todo módulo es impartido por un profesor y sólo uno:



En este otro: Indica que una persona se relaciona con otra persona por la relación "casado con". La cardinalidad indica que una mujer puede estar soltera o casada con una persona y los maridos igual:



Si queremos establecer la cardinalidad abrimos la especificación de la relación y establecemos el apartado Multiplicidad a alguno de los valores que indica, si necesitamos utilizar algún valor concreto también podemos escribirlo nosotros mismos. En el caso que nos ocupa seleccionaremos la cardinalidad 0..\* para los alumnos y 1..\* para los módulos.



### 3.4.2. Relación de herencia (generalización)

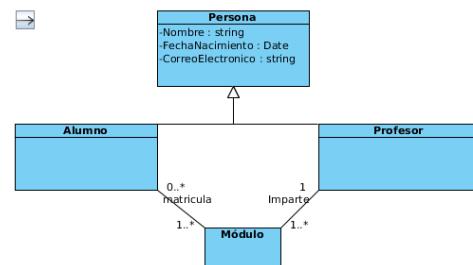
La herencia o generalización es una propiedad que permite a los objetos ser construidos a partir de otros objetos, utilizando estructuras de datos y métodos presentes en sus antepasados. Gracias a ello se puede reutilizar código realizado con anterioridad.

Consiste en una clase base y una jerarquía de clases que contiene las clases derivadas, que heredan código y datos de su clase base, además de añadir su propio código especial y datos, o incluso, cambiando elementos de la clase base que necesita que sean distintos.

- **Herencia simple:** Una clase solo tiene un ascendente.
- **Herencia múltiple:** se tienen más de un ascendente inmediato y se adquieren datos y métodos de más de una clase.

En el diagrama de clases se representa como una asociación en la que el extremo de la clase base tiene un triángulo.

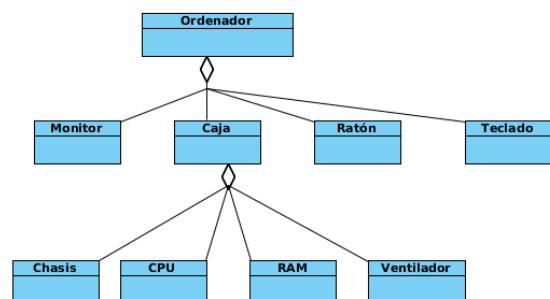
Podemos utilizar la relación de herencia para crear una clase nueva que se llame Persona y que recoja las características comunes de profesor y alumno. Persona será la **clase base** y Profesor y Alumno las **clases derivadas**.



Como los atributos Nombre, FechaNacimiento y correoElectronico se heredan de la clase base no hace falta que aparezcan en las clases derivadas, por lo que las hemos eliminado. Después podemos añadir atributos o métodos propios a las clases derivadas. La relación se añade de igual manera que una relación de asociación, pero seleccionando la opción Generalization.

### 3.4.3. Agregación y composición

**Agregación** es una asociación binaria que representa una relación todo-parte (pertenece a, tiene un, es parte de). Los elementos parte pueden existir sin el elemento contenedor y no son propiedad suya.



Por ejemplo. Un centro comercial tiene clientes o un equipo tiene unos miembros. El tiempo de vida de los objetos no tiene por qué coincidir.

En el siguiente caso, tenemos un ordenador que se compone de piezas sueltas que pueden ser sustituidas y que tienen entidad por si mismas, por lo que se

representa mediante relaciones de agregación. Utilizamos la agregación porque es posible que una caja, ratón o teclado o una memoria RAM existan con independencia de que pertenezcan a un ordenador o no.

**Composición** es una agregación fuerte en la que una instancia "parte" está relacionada como máximo con una instancia "todo" en un momento dado, cuando el objeto "todo" es eliminado, también son eliminados sus objetos "parte". Por ejemplo, un rectángulo tiene 4 vértices, un centro comercial está organizado mediante un conjunto de secciones de venta.

Para modelar la estructura de un ciclo formativo vamos a usar las clases Módulo, Competencia y Ciclo que representan lo que se puede estudiar en Formación Profesional y su estructura lógica. Un ciclo formativo se compone de una serie de competencias que se le acreditan cuando supera uno o varios módulos formativos.

Dado que si eliminamos el ciclo, las competencias no tienen sentido, y lo mismo ocurre con los módulos hemos usado relaciones de composición. Si los módulos o competencias pudieran seguir existiendo sin su contenedor habríamos utilizado relaciones de agregación.

Estas relaciones se representan con un rombo en el extremo de la entidad contenedora. En el caso de la agregación es de color blanco y para la composición negro. Como en toda relación hay que indicar la cardinalidad.



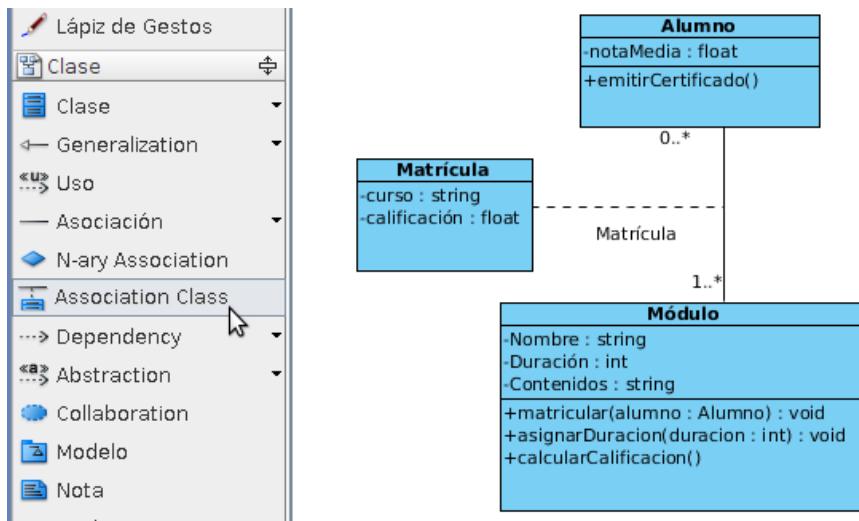
### 3.4.4. Atributos de enlace

Se pueden añadir atributos a relaciones para añadir algún tipo de información que la complete de alguna manera.

Cuando un alumno se matricula de un módulo es preciso especificar el curso al que pertenece la matrícula, las notas obtenidas en el examen y la tarea y la calificación final obtenida. Estas características no pertenecen totalmente al alumno ni al módulo sino a la relación específica que se crea entre ellos, que además será diferente si cambia el alumno o el módulo. Añade estos atributos al enlace entre Alumno y Módulo.

Para modelar esto en Visual Paradigm creamos una clase nueva (Matrícula) junto a Alumno y Módulo, y la unimos a la relación utilizando el icono de la

paleta "Association class", el diagrama queda así:



### 3.4.5. Restricciones

La relación entre dos clases puede estar condicionada al cumplimiento de algún requisito, o un parámetro de una clase tiene un valor constante.

Las restricciones se incluyen mediante una descripción textual encerrada entre llaves.

## 3.5. Pautas para crear diagramas de clase

La clave es hacer una buena elección de las clases que sugiere el programa.

Para identificar las clases candidatas a formar parte del diagrama es recomendable subrayar cada nombre o sintagma nominal que aparece en el enunciado.

Una vez teniendo la lista completa habrá que estudiar cada clase potencial para ver si se incluye en el diagrama. Para ello se utilizan los siguientes criterios y deben cumplirse todos o casi todos estos criterios:

1. La información de la clase es necesaria para que el sistema funcione.
2. La clase posee un conjunto de atributos que podemos encontrar en cualquier ocurrencia de sus objetos. Si solo aparece un atributo se suele rechazar y será añadido como atributo de otra clase.
3. La clase tiene un conjunto de operaciones identificables que pueden cambiar el valor de sus atributos y son comunes en sus objetos.

4. Es una entidad externa que consume o produce información esencial para la producción de cualquier solución en el sistema.

### 3.5.1. Obtención de atributos y operaciones

#### Atributos

Definen al objeto en el contexto del sistema, es decir, el mismo objeto en sistema diferentes tendría diferentes atributos. Debemos contestar a la pregunta: ¿Qué elementos compuesto o simples definen completamente en el contexto del problema actual.

#### Operaciones

Describen el comportamiento del objeto y modifican sus características:

- Manipulan los datos.
- Realizan algún cálculo.
- Monitorizan un objeto frente a la ocurrencia de un suceso de control.

Se obtienen analizando verbos en el enunciado del problema.

#### Relaciones

Se debe estudiar de nuevo el enunciado para obtener como los objetos descritos se relacionan entre sí. Para facilitar el trabajo podemos buscar mensajes que se pasen entre objetos y relaciones de composición y agregación. Las relaciones de herencia se suelen encontrar al comprar objetos semejantes entre sí y se contrasta que tengan atributos y métodos comunes.

## 3.6. Generación de código a partir del diagrama de clases

La generación automática de código consiste en la creación del código fuente de manera automatizada a través de herramientas CASE.

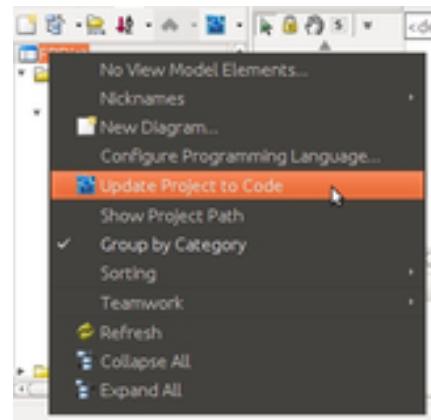
El proceso pasa por establecer correspondencia entre los elementos formales de los diagramas y la estructura de un lenguaje en concreto.

#### Utilizando el SDE integrado de VP-UML en NetBeans:

Se abre el modelo desde NetBeans, usando el SDE, se crea un proyecto nuevo y se importa el proyecto VP-UML que hemos creado.

Se hace de dos formas:

- **Sincronizar con el código:** el código fuente eliminado no se recupera, solo se actualiza el existente.



- **Forzar sincronizado a código.** Se actualiza todo el código, incluido el de código eliminado del proyecto NetBeans

Para generar todas las clases y paquetes, abrimos el proyecto CE-NB y desplegamos el menú contextual y seleccionamos Update Project to Code.

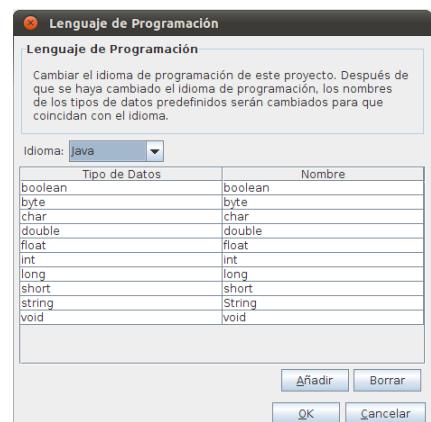
Si se produce algún problema se muestra en la ventana de mensaje. Este procedimiento produce archivos .java necesarios para implementar las clases del diagrama.

#### Desde VP-UML

Para generar el código java de un diagrama de clases utilizamos el menú herramientas/generación instantánea/java. Se muestra la ventana en la que configuramos el idioma, las clases y otras características básicas con la nomenclatura de atributos y métodos.

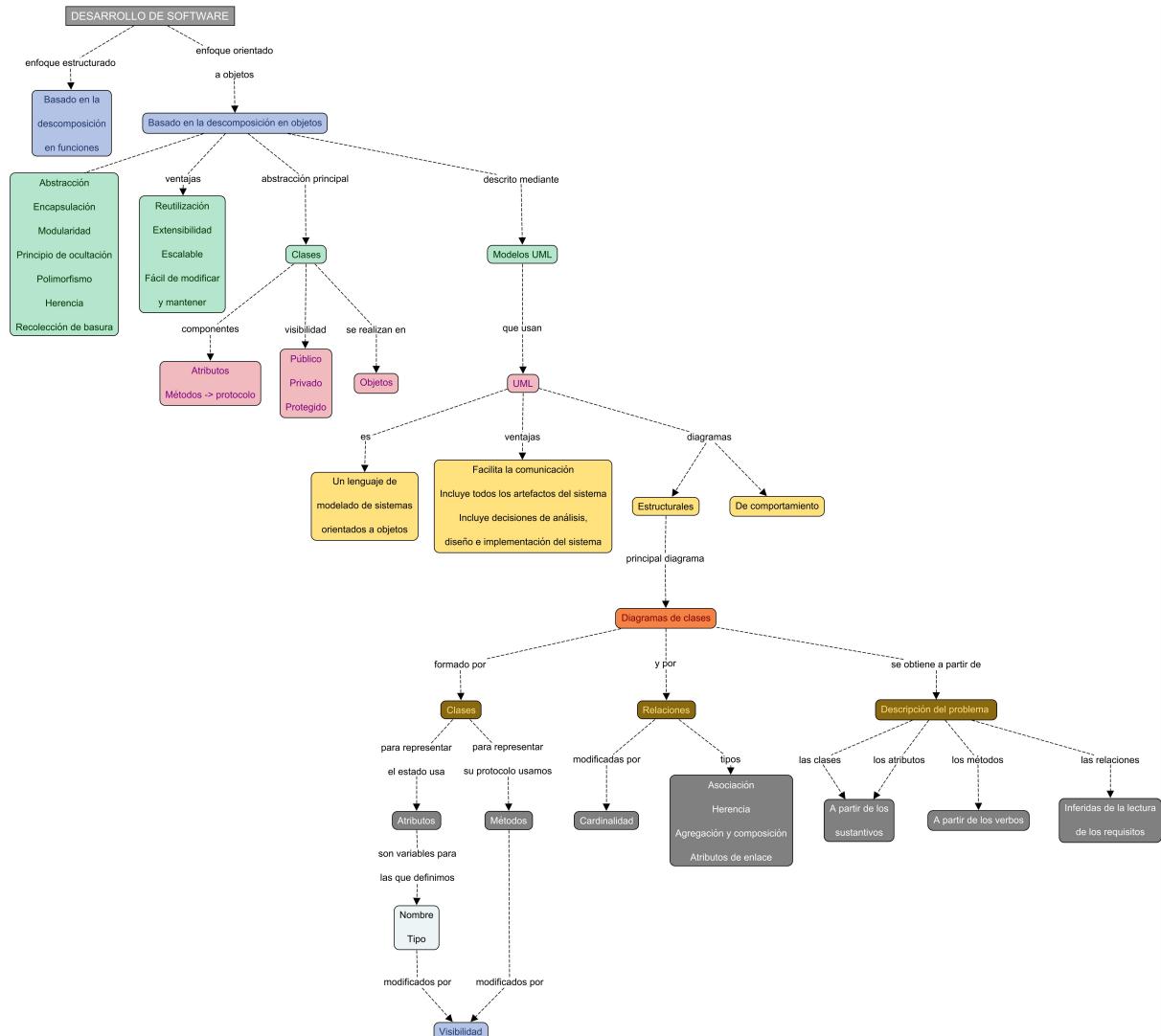
#### 3.6.1. Elección del lenguaje de programación. Orientaciones para el lenguaje java.

El lenguaje final de implementación de la aplicación influyen en algunas decisiones a tomar cuando estamos creando el diagrama ya que el proceso de traducción es inmediato. Por ejemplo, si queremos utilizar la herramienta de generación de código tendremos que asegurarnos de utilizar tipos de datos simples apropiados, es decir, si usamos Java el tipo de dato para las cadenas de caracteres será String en lugar de string o char\*.



Podemos definir el lenguaje de programación final desde el menú Herramientas  
 >> Configurar lenguaje de programación. Si seleccionamos Java automáticamente cambiará los nombres de los tipos de datos al lenguaje escogido.

## Mapa conceptual





# 6. Diseño orientado a objetos. Elaboración de diagramas de comportamiento.

	Autor	Xerach Casanova
	Clase	Entornos de desarrollo
	Fecha	@Mar 18, 2021 9:46 PM

## 1. Introducción

2. Diagramas de casos de uso.

2.1. Elementos del diagrama de casos de uso

2.2.1. Actores

2.1.2. Casos de uso

2.1.3. Relaciones

2.2. Elaboración de casos de uso

2.3. Escenarios

## 3. Diagramas de interacción

3.1. Diagramas de secuencia

3.1.1. Representación de objetos, línea de vida y paso de mensajes

3.2. Diagramas de colaboración

3.2.1. Representación de objetos

3.2.2. Paso de mensajes

## 4. Diagramas de estados

4.1. Estados y eventos

4.2. Transiciones

## 5. Diagramas de actividad

5.1. Elementos del diagrama de actividad

## Mapa conceptual

## Anexos

Ejercicio resuelto 1 ("ZAPATERÍA TACÓN DE ORO"). Elaboración de un diagrama de casos de uso

Ejercicio resuelto 2 ("QUIJOTE"). Elaboración de un diagrama de casos de uso.

Ejercicio resuelto 3 ("ALQUILER DE PISOS Y LOCALES"). Elaboración de un diagrama de casos de usos.

Ejercicio resuelto 1 ("Generar pedido"). Elaboración de un diagrama de secuencias

Ejercicio resuelto 2 ("Estadio"). Elaboración de un diagrama de secuencia

Ejercicio resuelto 3 ("ROPERO"). Elaboración de un diagrama de secuencia.

Ejemplo de un diagrama de colaboración

Ejercicio resuelto 1 ("Generar pedido"). Elaboración de un diagrama de estados.

Ejercicio resuelto 2 ("RELOJ"). Elaboración de un diagrama de estados.

Ejercicio resuelto 3 ("VIDA LABORAL"). Elaboración de un diagrama de estados

Ejemplo de un diagrama de actividad

## 1. Introducción

Un diagrama de clases nos da información estática, pero no nos dice nada acerca del comportamiento dinámico de los objetos que lo forman. Para este tipo de información se utilizan diagramas de comportamiento, los cuales incluyen:

- Diagramas de casos de uso.
- Diagramas de actividad.
- Diagramas de estados.
- Diagramas de interacción.
  - Diagramas de secuencia.
  - Diagramas de comunicación/colaboración.
  - Diagramas de interacción.
  - Diagramas de tiempo

## 2. Diagramas de casos de uso.

Cuando se construye un software es necesario conocer los requerimientos del mismo. Se precisa alguna herramienta que ayude a especificarlos de manera clara, sistemática y que el cliente pueda entender.

No basta una lista de requerimientos descritos, ya que se puede inducir a errores de interpretación y se suelen dejar cabos sueltos.

La solución la da los diagramas de casos de uso. Son un elemento fundamental en la etapa de análisis de un sistema y resuelven el problema de la falta de comunicación entre el equipo de desarrollo y el equipo que necesita una solución de software.

Los diagramas de casos de uso nos ayudan a determinar qué puede hacer cada tipo diferente de usuario con el sistema.

Los diagramas de casos de uso documentan el comportamiento de un sistema desde el punto de vista de usuario. Determinan los requisitos funcionales del sistema, es decir, las funciones del sistema que se pueden ejecutar.

Es una visualización gráfica de los requisitos funcionales del sistema que está formado por casos de uso, representados como elipses. Los actores que interactúan con ellos son monigotes. Su principal función es dirigir el proceso de creación del software definiendo lo que se espera de él. Son fáciles de interpretar y útiles para comunicarse con el cliente.

## 2.1. Elementos del diagrama de casos de uso

- Actores.
- Casos de uso.
- Relaciones.

/code

### 2.2.1. Actores

Representan un tipo de usuario del sistema y se entiende por usuario cualquier cosa externa que interactúa con el sistema.

Puede ser humano, otro sistema informático o unidades organizativas.



La diferencia entre actores y usuarios es que un usuario puede interpretar diferentes roles según la operación, estos roles representará un actor diferente. Por tanto, cada actor puede ser interpretado por diferentes usuarios.

Tipos de actores:

- **Primarios:** interaccionan con el sistema para explotar su funcionalidad, de forma directa y frecuentemente con el software.
- **Secundarios:** soporte del sistema para que los primarios puedan trabajar. Precisos para alcanzar algún objetivo.
- **Iniciadores:** es posible que haya casos de uso que no sean iniciados por ningún usuario, se considera un actor tiempo o sistema que asume el

arranque del caso.

### 2.1.2. Casos de uso

Se representa mediante un óvalo o elipse y su descripción.

Especifican una secuencia de acciones, incluyendo variantes, que el sistema puede llevar a cabo y que producen un resultado observable de valor para un actor concreto.

Casos de Uso

El conjunto de casos de uso forma el comportamiento requerido de un sistema. El objetivo principal de elaborar el diagrama de casos de uso no es crear el diagrama en sí, sino la descripción que de cada caso se debe realizar. Esto ayuda al equipo de desarrollo a crear el sistema. Junto al diagrama se crea una tabla con descripción textual, en la que se incluyen al menos los siguientes datos (contrato).

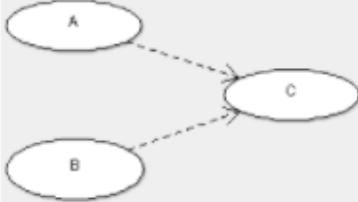
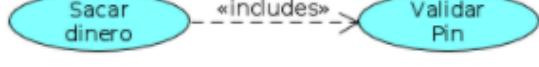
- **Nombre:** nombre del caso de uso.
- **Actores:** aquellos que interactúan con el sistema a través del caso de uso.
- **Propósito:** breve descripción de lo que se espera que haga.
- **Precondiciones:** aquellas que deben cumplirse para que pueda llevarse a cabo el caso de uso.
- **Flujo normal:** flujo normal de eventos que deben cumplirse para ejecutar el caso de uso exitosamente, desde el punto de vista del actor que participa y del sistema.
- **Flujo alternativo:** flujo de eventos que se llevan a cabo cuando se producen casos inesperados o poco frecuentes. No se deben incluir aquí errores como escribir un tipo de dato incorrecto o la omisión de un parámetro necesario.
- **Postcondiciones:** las que se cumplen una vez que se ha realizado el caso de uso.

<b>Super Use Case</b>							
<b>Author</b>	usuario						
<b>Date</b>	26-agosto-2011 13:56:56						
<b>Brief Description</b>							
<b>Preconditions</b>							
<b>Post-conditions</b>							
<b>Flow of Events</b>	<table border="1"> <thead> <tr> <th></th> <th><b>Actor Input</b></th> <th><b>System Response</b></th> </tr> </thead> <tbody> <tr> <td>1</td> <td></td> <td></td> </tr> </tbody> </table>		<b>Actor Input</b>	<b>System Response</b>	1		
	<b>Actor Input</b>	<b>System Response</b>					
1							

## 2.1.3. Relaciones

Los diagramas de casos de uso son grafos no conexos. Los nodos son actores y casos de uso y las aristas son las relaciones entre ellos. Estas relaciones representan qué actores realizan las tareas descritas en los casos de uso. Además existen otros tipos de relaciones que se utilizan para especificar relaciones más complejas, como uso o herencia entre casos de uso o actores.

Relación	Descripción/ Ejemplo
Asociación	
	<p>Representa la relación entre el actor y un caso de uso en el que participa.</p> <p>Ejemplo: Relación entre el caso de uso <i>sacar dinero</i> y el <i>cliente</i> de un banco.</p> <pre> graph LR     Cliente --&gt; SacarDinero     Cliente     SacarDinero     </pre> <p>The diagram shows a UML association line connecting a stick figure representing the 'Cliente' actor to an oval representing the 'Sacar dinero' use case. The actor is labeled 'Cliente' below it, and the use case is labeled 'Sacar dinero' inside its oval.</p>

	<p>Esta relación es muy útil cuando se desea especificar algún comportamiento común en dos o más casos de uso, aunque es frecuente cometer el error de utilizar esta técnica para hacer subdivisión de funciones, por lo que se debe tener mucho cuidado cuando se utilice.</p>
	<p><b>Ejemplo 1:</b> Al ejecutar el caso de uso <i>sacar dinero</i>, obligatoriamente se ejecuta el caso de uso <i>validar pin</i> de la tarjeta de crédito.</p> <p></p> <p><b>Ejemplo 2:</b> Por ejemplo, a la hora de hacer un pedido se debe buscar la información de los artículos para obtener el precio, es un proceso que necesariamente forma parte del caso de uso, sin embargo también forma parte de otros, como son el que visualiza el catálogo de productos y la búsqueda de un artículo concreto, y dado que tiene entidad por sí solo se separa del resto de casos de uso y se incluye en los otros tres.</p>

Extensión (extend)	
 <b>extend</b>	<p>Se trata de una relación entre casos de uso. La ejecución de un caso de uso puede provocar la ejecución del segundo</p>
	<p>Se utiliza una relación entre dos casos de uso de tipo "extends" cuando se desea especificar que el comportamiento de un caso de uso es diferente dependiendo de ciertas circunstancias.</p> <p>La principal función de esta relación es simplificar el flujo de casos de uso complejos. Se utiliza cuando existe una parte del caso de uso que se ejecuta sólo en determinadas ocasiones, pero no es imprescindible para su completa ejecución. Cuando un caso de uso extendido se ejecuta, se indica en la especificación del caso de uso como un punto de extensión. Los puntos de extensión se pueden mostrar en el diagrama de casos de uso.</p>

	<p><b>Ejemplo 1:</b></p> <p><i>Imprimir ticket</i> es consecuencia del caso de uso <i>sacar dinero</i>, pero su ejecución es opcional a que sea requerida por el cliente.</p> <pre> graph LR     SD([Sacar dinero]) -- "&lt;--extends--&gt;" --&gt; IT([Imprimir ticket])   </pre>
	<p><b>Ejemplo 2:</b></p> <p>Cuando un usuario hace un pedido si no es socio se le ofrece la posibilidad de darse de alta en el sistema en ese momento, pero puede realizar el pedido aunque no lo sea.</p> <pre> sequenceDiagram     participant Profesor     participant NuevoSocio     participant Alumno     Profesor-&gt;&gt;NuevoSocio:      NuevoSocio--&gt;&gt;Alumno:    </pre>

Generalización	
→	<p>Se utiliza para representar relaciones de herencia entre casos de uso o actores. No se contemplan generalizaciones combinadas entre actores y casos de uso.</p> <p>Se utiliza cuando se tiene uno o más casos de uso que son especificaciones de un caso más general.</p>
	<p>Por ejemplo, entre actores: tanto <i>profesor</i> como <i>alumno</i> son casos particulares del actor <i>persona</i>.</p> <pre> classDiagram     Persona &lt; -- Profesor     Persona &lt; -- Alumno   </pre> <p><b>Ejemplos, entre casos de uso:</b></p> <p>Un ejemplo de generalización de casos de uso sería la compra de artículos en un comercio, pudiendo considerarse la compra de alimentos o de bebidas. Ambos tipos de compras tendrán las características heredadas del caso de uso compra general, más las particulares definidas para cada caso.</p>

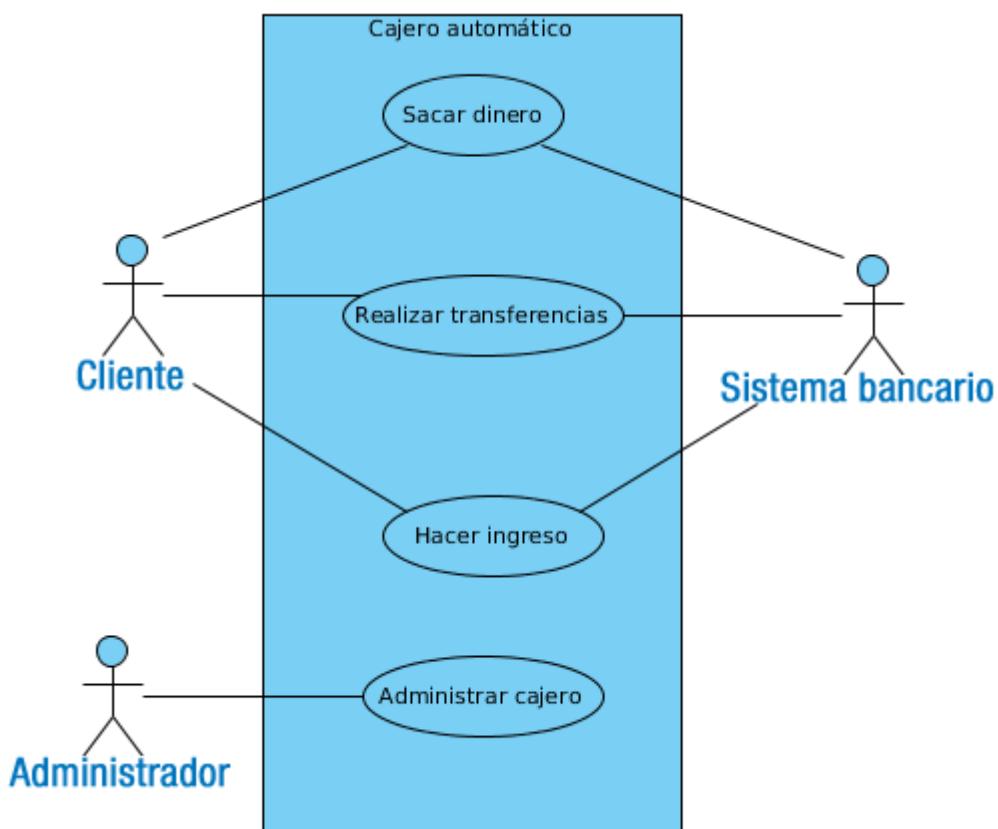
## 2.2. Elaboración de casos de uso

En los diagramas de casos de uso se hace una abstracción de la realidad en la que representamos qué cosas se pueden hacer en un sistema y quién las va a hacer.

Necesitamos implementar diagramas cuya información permita al equipo de desarrollo tomar decisiones adecuadas en la fase de análisis y diseño, y que sean útiles en la fase de implementación.

Partimos de una descripción detallada del posible problema a resolver y tratamos de detectar aspectos como:

- Usuarios que interactúan para obtener a los actores.
- Tareas que realizan estos actores para determinar los casos de uso más genéricos.
- Refinar el diagrama analizando los casos de uso más generales, para detectar casos relacionados por inclusión, extensión y generalización.



## 2.3. Escenarios

Un caso de uso especifica un comportamiento deseado, pero no impone como se llevará a cabo.

Un escenario es una ejecución particular de un caso de uso que se describe como una secuencia de eventos. Un caso de uso es una generalización de un escenario.

### Ejemplo

Para el caso de uso hacer pedido se establecen diferentes escenarios, uno podría ser:

1. El usuario inicia el pedido.
2. Se crea el pedido en estado "en construcción".
3. Se selecciona un par de zapatos "Lucía" de piel negros, del número 39.
4. Se selecciona la cantidad 1.
5. Se recupera la información de los zapatos y se modifica la cantidad a pagar sumándole 45 €.
6. Se selecciona un par de botas "Aymara" de ante marrón del número 40.
7. Se selecciona la cantidad 1.
8. Se recupera la información de las botas y se modifica la cantidad a pagar sumándole 135 €.
9. El usuario acepta el pedido.
10. Se comprueba que el usuario es, efectivamente socio.
11. Se comprueban los datos bancarios, que son correctos.
12. Se calcula el total a pagar añadiendo los gastos de envío.
13. Se realiza el pago a través de una entidad externa.
14. Se genera un pedido para el usuario con los dos zapatos que ha comprado, con el estado "pendiente".

Los escenarios pueden y deben posteriormente documentarse mediante diagramas de secuencia.

## 3. Diagramas de interacción

También se hace necesario buscar la forma de representar como circula la información, los objetos que participan en los casos de uso, los mensajes que

se envían, el momento en que se producen...

Los diagramas de interacción son vistas del sistema que muestran como grupos de objetos interactúan para un comportamiento. Captan la ejecución de los casos de uso, representando a los actores que participan y los mensajes que se pasan.

Existen dos: diagramas de secuencia y diagramas de colaboración, en el que la anotación es distinta en ambas aunque la información es la misma.

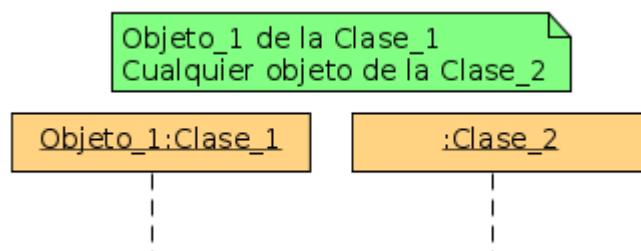
### 3.1. Diagramas de secuencia

En estos diagramas los objetos/actores que forman parte del escenario de un caso de uso, se representan mediante rectángulos distribuidos horizontalmente en la zona superior del diagrama a los que se asocia una línea temporal vertical para cada actor de las que salen los diferentes mensajes que se pasan entre ellos.

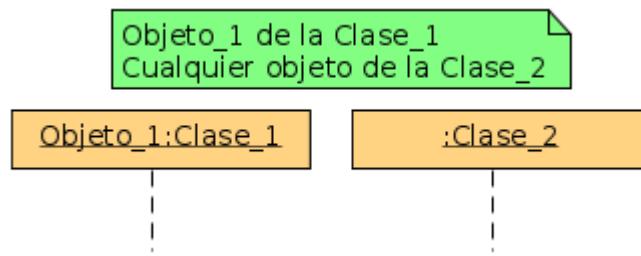
El equipo de desarrollo se puede hacer una idea de las distintas operaciones al ejecutar determinada tarea y su orden.

#### 3.1.1. Representación de objetos, línea de vida y paso de mensajes

##### Representación de objetos y línea de vida



Los objetos son rectángulos y se distribuyen horizontalmente en la parte superior, por cada objeto se identifica su nombre seguido del símbolo de dos puntos y después el nombre de su clase. Si no se indica nombre del objeto, se considera que para el propósito del diagrama es válido cualquier objeto de la clase.



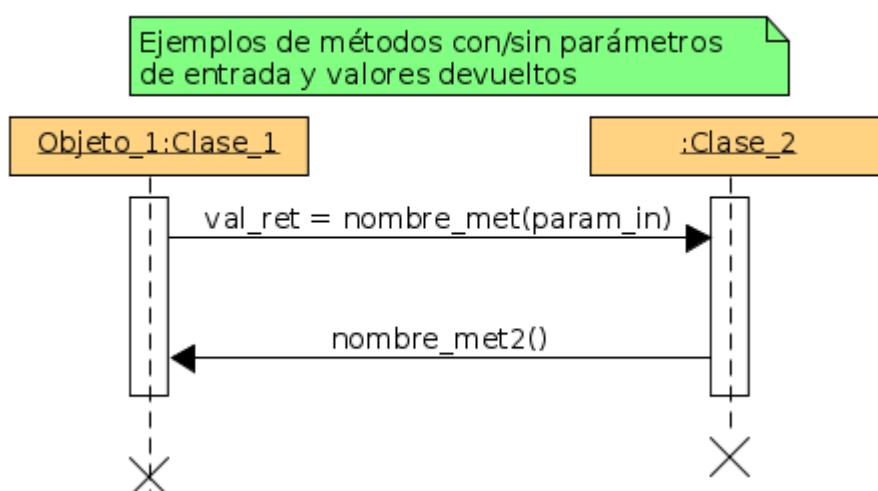
De cada rectángulo sale una línea discontinua que representa el paso del tiempo (línea de vida), esta línea representa la vida del objeto mientras es relevante en el diagrama, cuando deja de serlo se marca una cruz.

Una línea de vida puede ser encabezada por otro tipo de instancias como el sistema o un actor que aparecerán con su propio nombre. Usaremos el sistema para representar solicitudes al mismo, como por ejemplo pulsar una ventana o una llamada a una subrutina.

### Paso de mensajes (invocación de métodos)

Los mensajes significan invocación de métodos y se representan con flechas horizontales que van de una línea de vida a otra. Los mensajes se dibujan desde el objeto que envía el mensaje hasta el que lo recibe, pudiendo ser el mismo objeto emisor y receptor.

El orden en el tiempo va determinado por su posición vertical, un mensaje que se dibuja debajo de otro indica que se envía después. Los mensajes tienen un nombre y pueden incluir argumentos de entrada, valores devueltos e información de control (condición o iteración).



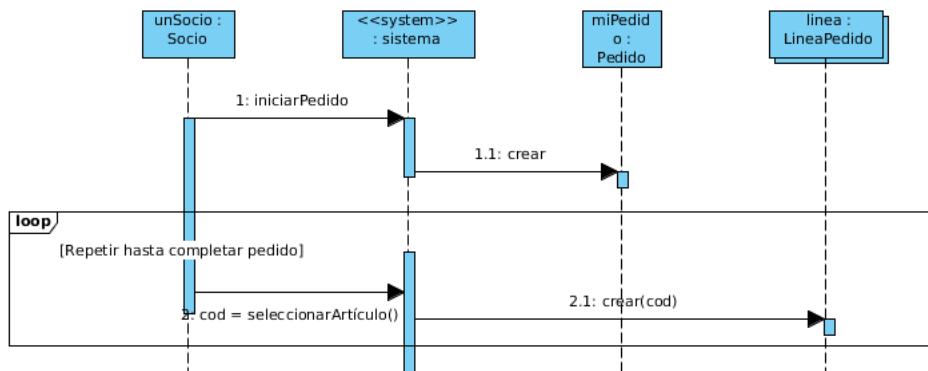
Una notación alternativa para recoger valores devueltos por los métodos es dibujar una línea de puntos finalizada en flecha, desde el objeto destinatario

del mensaje al que lo ha generado, acompañado del valor devuelto.

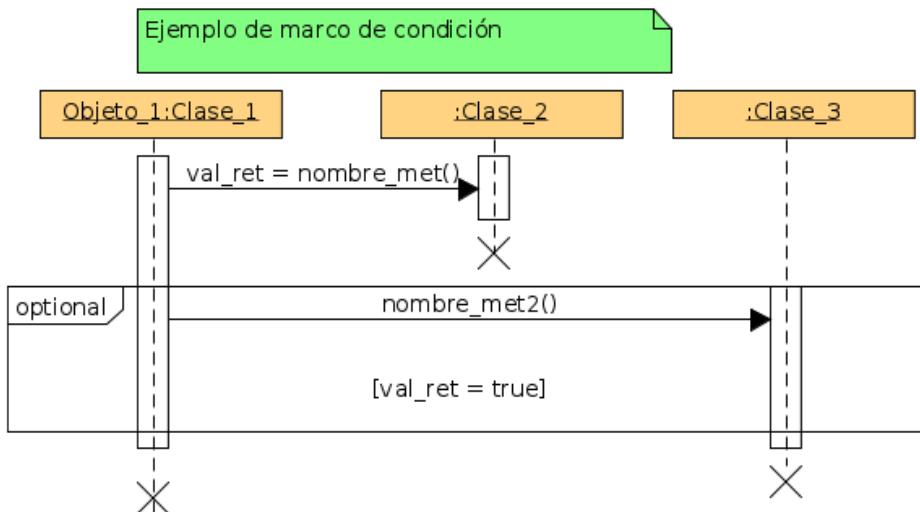
## Condiciones e iteraciones

Las secuencias de control se pueden representar usando marcos, normalmente se nombra el marco con el tipo de bucle a ejecutar y la condición de parada.

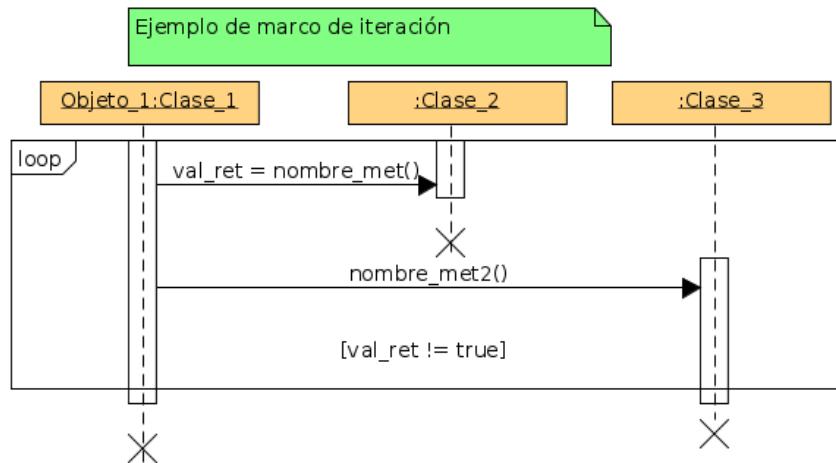
También se pueden representar flujos de mensajes condicionales en función de un valor determinado.



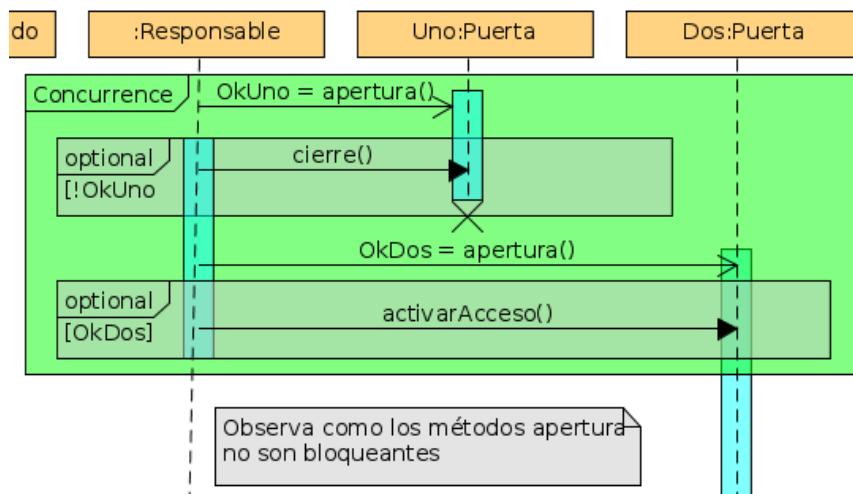
La expresión a evaluar para la condición o iteración se representa entre corchetes:



Combinando varios marcos opcionales es posible representar diferentes alternativas en la ejecución de un diagrama de secuencia. Para el caso de una iteración, tenemos el siguiente ejemplo.



Por defecto los métodos son bloqueantes, se entiende que el proceso del diagrama de secuencia completa cada método antes de continuar con el siguiente, es una secuencia de métodos en el tiempo. Pero en ocasiones se producen situaciones en las que se desea mostrar varios procesos en paralelo (conurrencia), se puede reflejar mediante el uso de marcos con la etiqueta `concurrence`.



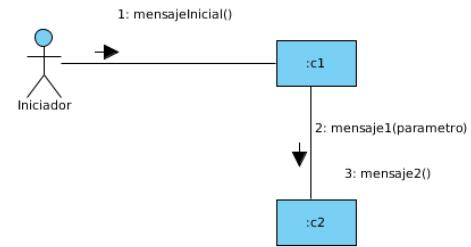
Se puede completar el diagrama añadiendo etiquetas y notas en el margen izquierdo que aclare la operación que se está realizando.

## 3.2. Diagramas de colaboración

Muestran una secuencia de ejecución de uno o varios casos de uso, al igual que los diagramas de secuencia.

La notación es muy similar y la principal diferencia radica en el modo de mostrar el orden de mensajes intercambiados entre objetos.

En el diagrama de colaboración se hace mediante el etiquetado de mensajes. Las interacciones entre objetos se describen en forma de grafo, en el que los nodos son objetos y las aristas son enlaces entre objetos a través de los cuales se envían mensajes entre ellos.



Permiten una mejor organización visual que los diagramas de secuencia, además son más fáciles de seguir.

UMLet no dispone de herramientas para la elaboración de diagramas de colaboración directamente. No obstante, no resulta complicado generarlos a partir de los símbolos disponibles para otros diagramas: representación de objetos mediante cajas, paso de mensajes mediante líneas, información de los métodos mediante descripciones textuales; todos ellos disponibles en los diagramas de secuencia de UMLet.

### 3.2.1. Representación de objetos

Un objeto puede ser cualquier instancia de las clases definidas en el sistema, pero también pueden incluirse objetos como la interfaz del sistema o el propio sistema, ayudándonos a modelar las operaciones que se llevan a cabo.

Los objetos se representan mediante rectángulos con sus nombres en el interior.

- **NombreClase:** se puede utilizar el nombre de la clase al que pertenece el objeto que participa en la interacción.
- **NombreObjeto:** se usa el nombre del objeto que participa en la interacción, normalmente subrayado.
- **:nombreClase:** cuando se colocan dos puntos delante del objeto significa que es un objeto genérico de esa clase.

Clase

:objeto

:Clase

objeto:clase

- **:nombreObjeto:nombreClase:** hace referencia al objeto concreto que se nombre añadiendo la clase a la que pertenece.

### 3.2.2. Paso de mensajes

Para hacer posible el paso de mensajes se necesita una asociación entre los objetos, que se hará mediante una línea que los une y una flecha que indique la dirección.

También se pueden incluir parámetros en mensajes, valores devueltos, condicionales y bucles.

La sintaxis de un mensaje es la siguiente:

- [Secuencia] [\*] [Condición] {valorDevuelto} : mensaje (argumentos de entrada)
- [Secuencia] [\*] [Condición] mensaje (argumentos de entrada) : {valorDevuelto}

**Secuencia:** representa el nivel de anidamiento del envío del mensaje dentro de la iteración. Los mensajes se numeran para indicar el orden en el que se envían y si es necesario se puede indicar anidamiento incluyendo subrangos.

\*: indica que el mensaje es iterativo.

**Condición de guarda:** debe cumplirse para que el mensaje pueda ser enviado.

**Valor devuelto:** lista de valores devueltos por el mensaje. Estos valores se pueden utilizar como parámetros de otros mensajes. los corchetes indican que es opcional.

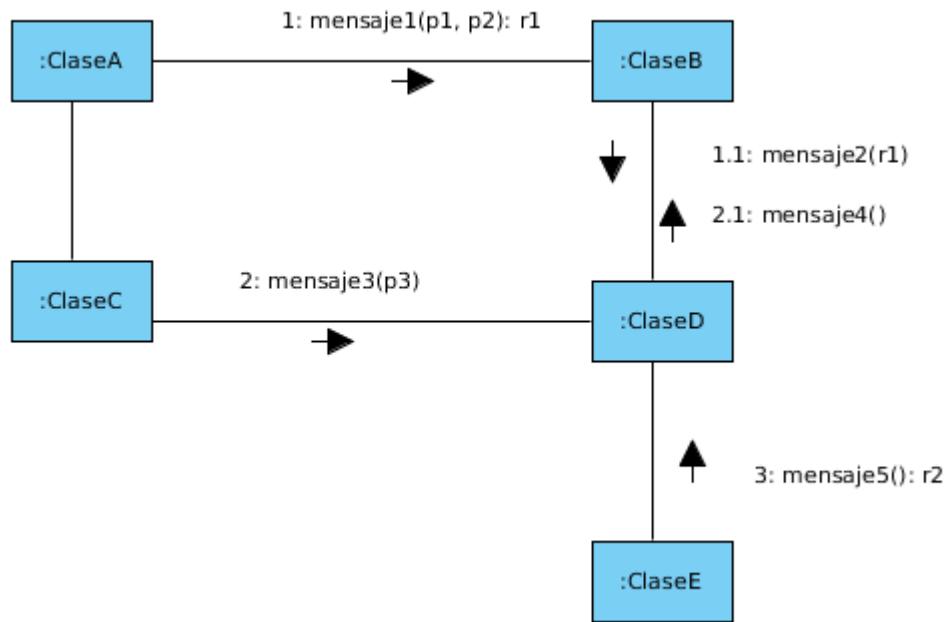
**Mensaje:** nombre del mensaje.

**Argumentos:** parámetros que se pasan al mensaje.

La enumeración de los mensajes se puede hacer de dos maneras:

- **Numeración simple:** Empieza en 1 y se va incrementando en 1 y no hay ningún nivel de anidamiento.
- **Numeración decimal:** se muestran varios niveles de subíndices para indicar anidamiento en operaciones. Por ejemplo, 1 es el primer mensaje, 1.1 es el primer mensaje anidado en el mensaje 1, 1.2 el segundo mensaje...

Como se ve en el ejemplo, se puede usar la misma asociación para enviar varios mensajes. Vemos que hay dos mensajes anidados, el 1.1 y el 2.1, se ha usado el nombre de los mensajes para indicar el orden real en el que se envían.



Los mensajes 1, 1.1 y 2 tienen parámetros y los mensajes 1 y 3 devuelven un resultado.

Se contempla la bifurcación en la secuencia añadiendo una condición en la sintaxis del mensaje:

[Secuencia][\*][CondiciónGuarda]{valorDevuelto} : mensaje (argumentos)

Cuando tenemos una condición se repite el número de secuencia y se añaden las condiciones necesarias, como vemos en la imagen según la condición se enviará el mensaje 1 o el 2, pero no ambos, por lo que coinciden en número de secuencia.

La iteración se representa mediante un \* al lado del número de secuencia, pudiendo indicarse entre corchetes la condición de parada del bucle.

## 4. Diagramas de estados

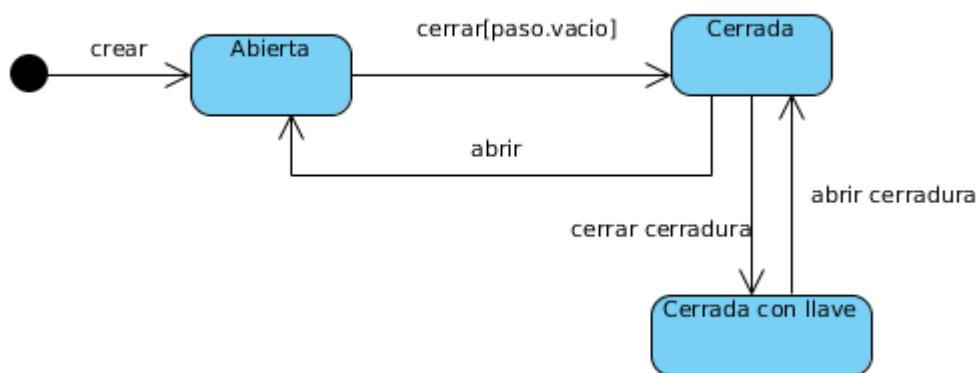
Los diagramas de estados permiten analizar como evoluciona el estado de un objeto a lo largo del tiempo, es decir, su comportamiento transitado por una serie de estados.

Modelan el comportamiento dinámico de los objetos en respuesta a los eventos.

En relación con el diagrama de estados se cumple que

- Un objeto está en un estado concreto en un cierto momento, que principalmente viene determinado por los valores de sus atributos.
- La transición de un estado a otro es momentánea y se produce cuando ocurre un evento.

Ejemplo de estados de una puerta:



## 4.1. Estados y eventos

Un estado es una situación en la vida del objeto en la que satisface cierta condición, realiza alguna actividad o espera algún evento.

Existen tres tipos de estados:

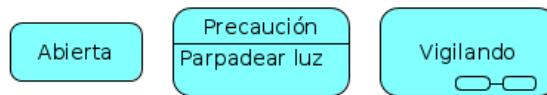
- **Estado inicial.** Punto de partida del diagrama de estados, corresponde a los valores de los atributos de la clase en el momento de instanciar.
- **Estado final.** Estado en el que se encuentra el objeto una vez finalizada la secuencia de eventos.
- **Estado intermedio.** Cualquiera de los estados intermedios entre los dos anteriores.



Los estados se representan mediante una caja y admite variantes, la información mostrada en los estados suele ser:

- Nombre de estado.

- Nombre de estado y acción/actividad asociada al objeto. En un semáforo en estado precaución se produce parpadeo de luz.
- Estado con subestados. En el ejemplo se indica que el estado vigilando tiene asociados, si se trata de un vigilante de seguridad, el estado vigilando podría tener relacionados "de ruta a pie" o "visionado de cámaras".



Un evento es un acontecimiento que dispara una transición entre dos estados del objeto. Existen eventos externos e internos, según el agente que los produzca:

- **Señales (excepciones).** La recepción de una señal producida por una situación excepcional en el sistema.
- **Llamadas:** la recepción de una petición para invocar una operación, normalmente un evento de llamada se maneja por el método de un objeto.
- **Paso de tiempo:** se genera como consecuencia del cumplimiento de un temporizador.
- **Cambio de estado:** se genera por el cambio en el estado o el cumplimiento de una condición.

## 4.2. Transiciones

Una transición de un estado A a un estado B se produce cuando se origina el evento asociado y se satisface cierta condición especificada, lo cual hace que se ejecute la acción de salida de A, la acción de entrada a B y la acción asociada a la transición.

La notación de una transición tiene tres partes, todas optativas:

Evento(argumentos)[condición]/acción

Elementos de una transición:

- **Evento:** cuando se produce un evento afecta a todas las transiciones que lo contienen en su etiqueta.
- **Condición:** expresión evaluable como verdadera o falsa, si es falsa no se dispara.

- **Acción.** Conjunto de actuaciones que lleva asociada la transición. Pueden incluir llamadas a operaciones de objetos, creación o destrucción de objetos.

Por ejemplo en el diagrama de estados de un semáforo:



## 5. Diagramas de actividad

Los diagramas de actividad son una especialización del diagrama de estados, organizado en torno a las acciones en lugar de los objetos, se compone de una serie de actividades y representa como se pasa de unas a otras.

Las actividades se enlazan por transiciones automáticas. Es decir, cuando una actividad termina se desencadena el paso a la siguiente.

Resultan útiles cuando se quieren representar solo las acciones que tienen lugar, prescindiendo de quien las genera.

Se usan para modelar el flujo de control entre actividades, en el que se puede distinguir cuales ocurren secuencialmente a lo largo del tiempo y cuales se pueden usar concurrentemente. Permite visualizar la dinámica del sistema desde otro punto de vista complementando al resto de diagramas.

Es un grafo conexo en el que los nodos son estados que pueden ser de actividad o acción y los arcos son transiciones.

### 5.1. Elementos del diagrama de actividad

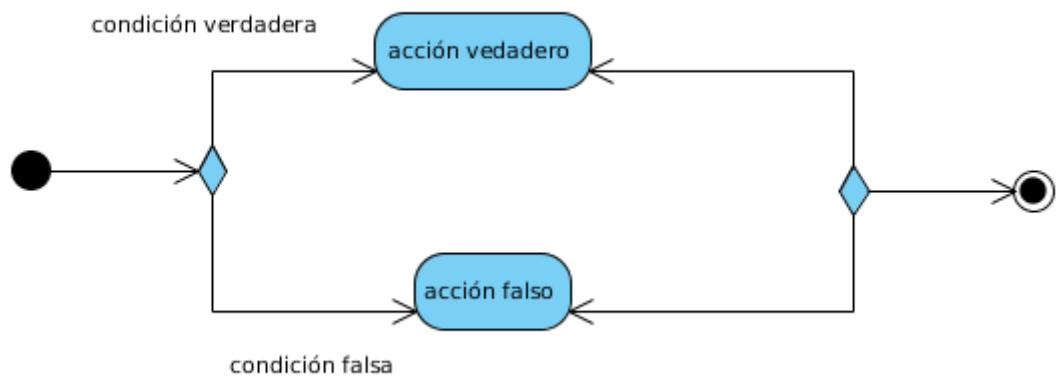
- **Estados:** de actividad y de acción
  - **De actividad:** elemento compuesto cuyo flujo de control se compone de otros estados de actividad y acción.
  - **De acción:** representa la ejecución de una acción atómica, que no se puede descomponer ni interrumpir, normalmente la invocación de una

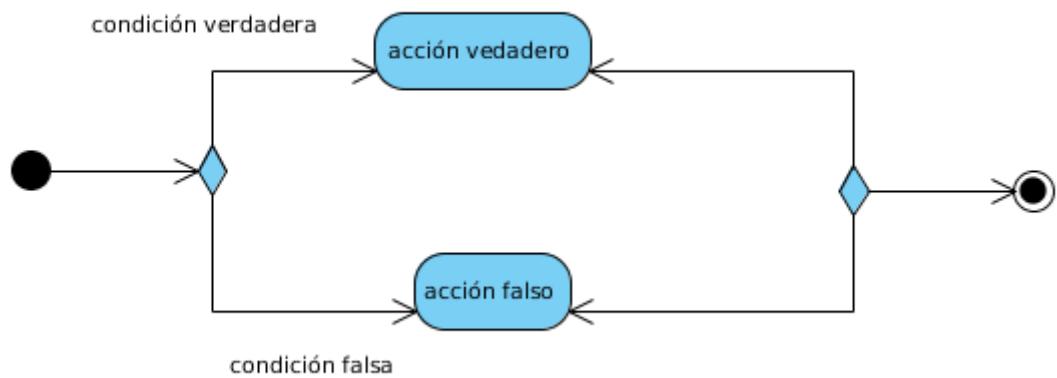
operación. El tiempo de ejecución normalmente es un tiempo insignificante.

- **Otros tipos de estado:** Inicial y Final.

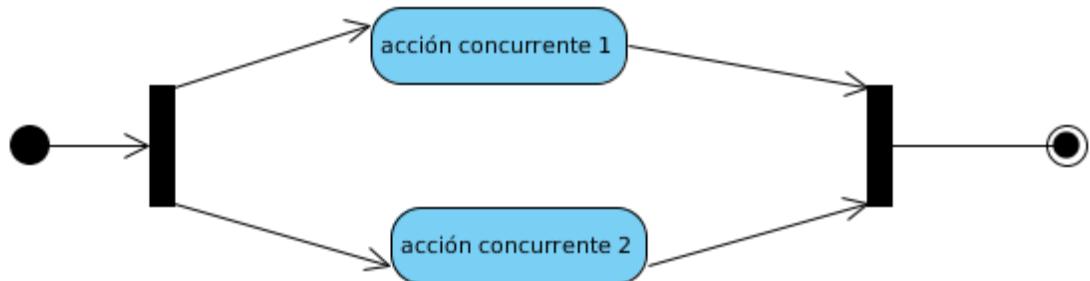


- **Transiciones:** relación entre dos estados que indica que un objeto en el primer estado realiza ciertas acciones y pasará al segundo estado cuando ocurra un evento específico y satisfaga ciertas condiciones. Se representa mediante una línea dirigida del estado inicial al siguiente.
  - **Secuencial o sin disparadores.** Al completar la acción del estado de origen se ejecuta la acción de salida, sin ningún retraso, el control sigue por la transición y pasa al siguiente estado.
  - **Bifurcación (Decision node).** Especifica caminos alternativos, elegidos según el valor de una expresión booleana. Las condiciones deben cubrir todas las posibilidades, puede usarse la palabra else y pueden utilizarse para lograr el efecto de iteraciones.

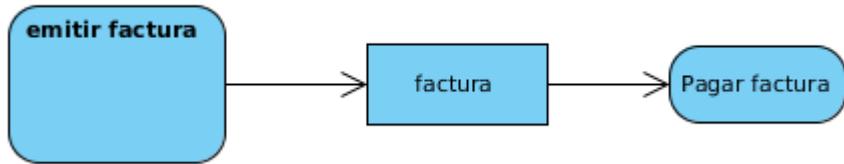




- **Fusion (Merge node):** Redirigen varios flujos de entrada en un único flujo de salida. No requiere tiempo de espera ni sincronización.
- **Division (Fork node).** Expresa la sincronización o ejecución paralela de actividades. Las actividades invocadas después de una división son concurrentes.



- **Union (Join node).** En la unión de flujos entrantes se sincronizan, es decir, cada uno espera hasta que todos los flujos de entrada han alcanzado la unión.
- **Objetos:** Manifestación concreta de una abstracción o instancia de una clase, cuando interviene un objeto no se utilizan flujos de eventos habituales, sino flujos de objetos, que se representan con flecha igualmente y permiten mostrar objetos que participan dentro del flujo de control asociado a un diagrama de actividades. Se puede indicar como cambian los valores de sus atributos, su estado o sus roles.



Utilizamos carriles o calles para ver quienes son los responsables en realizar distintas actividades y especificar que parte de la organización es responsable de una actividad.

- Cada calle tiene un nombre único dentro del diagrama.
- Puede ser implementada por una o varias clases.
- Las actividades de cada calle se consideran independientes y se ejecutan concurrentemente a las otras calles.

## Mapa conceptual

## Anexos

### Ejercicio resuelto 1 ("ZAPATERÍA TACÓN DE ORO"). Elaboración de un diagrama de casos de uso

**Descripción del problema: "El tacón de oro".**

La zapatería Tacón de oro ha decidido crear un espacio web para ampliar su línea de negocio, así sus usuarios podrán adquirir los artículos: zapatos, bolsos y complementos que se venden en la tienda.

Los usuarios del sistema navegarán por la web para ver los artículos, zapatos, bolsos y complementos que se venden en la tienda. De los artículos nos interesa su nombre, descripción, material, color, precio y stock. De los zapatos nos interesa su número y el tipo. De los bolsos nos interesa su tipo (bandolera, mochila, fiesta). De los complementos (cinturones y guantes) su talla.

Los artículos se organizan por campañas para cada temporada (primavera/verano y otoño/invierno) de cada año.

Los artículos son de fabricación propia, pero, opcionalmente, pueden venderse artículos de otras firmas. De las firmas nos interesa saber su nombre, CIF y domicilio fiscal. La venta de artículos de firma se realiza a través de

proveedores, de forma que un proveedor puede llevar varios artículos de diferentes firmas, y una firma puede ser suministrada por más de un proveedor. Los artículos pertenecen a una firma solamente. De los proveedores debemos conocer su nombre, CIF, y domicilio fiscal.

Los usuarios pueden registrarse en el sitio web para hacerse socios. Cuando un usuario se hace socio debe proporcionar los siguientes datos: nombre completo, correo electrónico y dirección.

Los socios pueden hacer pedidos de los artículos. Los usuarios pueden consultar todos los productos que tienen a su disposición, pero para realizar compras han de registrarse como socios.

Para comprar productos, se generan pedidos. Un pedido está formado por un conjunto de detalles de pedido que son parejas formadas por artículo y la cantidad. De los pedidos interesa saber la fecha en la que se realizó y cuánto debe pagar el socio en total. El pago se hace a través tarjeta bancaria, cuando se va a pagar una entidad bancaria comprueba la validez de la tarjeta. De la tarjeta interesa conocer el número.

Las campañas son gestionadas por el administrativo de la tienda que se encargará de dar de baja la campaña anterior y dar de alta la nueva siempre que no haya ningún pedido pendiente de cumplimentar.

Existe un empleado de almacén que revisa los pedidos a diario y los cumplimenta. Esto consiste en recopilar los artículos que aparecen en el pedido y empaquetarlos. Cuando el paquete está listo se pasa al almacén a la espera de ser repartido. Del reparto se encarga una empresa de transportes que tiene varias rutas preestablecidas. Según el destino del paquete (la dirección del socio) se asigna a una u otra ruta. De la empresa de transportes se debe conocer su nombre, CIF y domicilio fiscal. Las rutas tienen un área de influencia que determina los destinos, y unos días de reparto asignados. Se debe conocer la fecha en la que se reparte el pedido. Si se produce alguna incidencia durante el reparto de algún pedido se almacena la fecha en la que se ha producido y una descripción.

Los socios pueden visualizar sus pedidos y una vez comprobados, puede cancelarlos (siempre y cuando no hayan sido cumplimentados por el empleado de almacén) o confirmar la compra. Las compras deberán ser abonadas a través de una entidad bancaria. Así mismo los socios pueden modificar sus datos personales.

**Se pide:**

- Diagrama de casos de uso. Identifica los actores y casos de uso, incluye relaciones de asociación, identifica generalizaciones (de actores y de casos de uso). Si consideras alguna relación tipo include o extend, justifica su uso.
- Del diagrama que hayas obtenido en el apartado anterior, agrupa todos los casos de uso que hayas considerado en el proceso de gestión de pedidos en uno sólo. Para este apartado, se entiende que el pedido ya ha sido dado de alta y que el proceso normal de un pedido es que termine siendo comprado. Desarrolla su notación escrita (tabla del caso de uso).

### **Solución:**

Los diagramas de casos de uso quedan encuadrados en la fase de análisis de los proyectos, se entienden como una puesta en común entre el cliente y el analista sobre como entender requisitos funcionales.

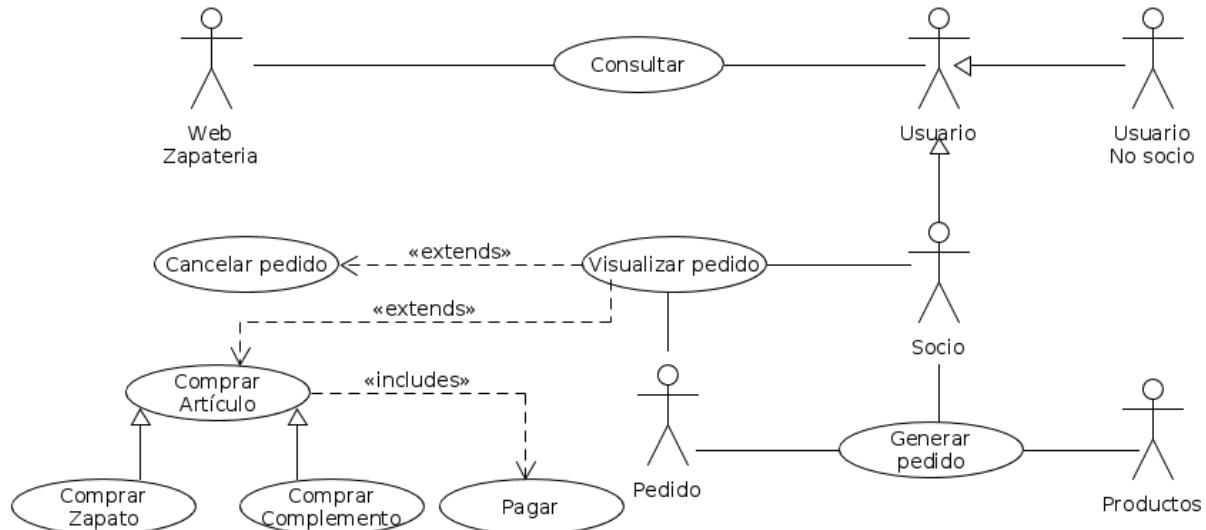
Al utilizarse UML, se entiende que ambos interlocutores conocen la notación propia de estos diagramas y las reglas sobre como combinar los diferentes símbolos, de forma que no hay ambigüedades que sí podrían darse mediante una descripción escrita.

Completar un diagrama de casos de uso supone versionar el diagrama tantas veces como sean necesarias hasta que la vista del proyecto que se presenta al cliente garantiza que el requisito funcional ha sido entendido.

En nuestros ejercicios, sólo hay una descripción que no permite ir refinando el diagrama mediante sucesivas consultas con el cliente, por lo tanto, las soluciones propuestas podrán ser diversas, todas ellas válidas siempre que reflejen razonablemente lo propuesto en el enunciado.

A la hora de valorar el ejercicio, lo que no es interpretable, es que el uso de la notación sea el correcto y que se muestre una variedad de los símbolos que conocemos para estos diagramas.

Con estas consideraciones, se propone la siguiente solución:



### **Actores que participan en el problema.**

Zapatería (web), usuario (que podrá ser socio o no socio), productos/artículos (que podrán ser zapatos o complementos), y pedidos.

### **Generalizaciones.**

Se observan dos posibles generalizaciones, que en el desarrollo del diagrama podrán ser expresadas en relación a actores o en relación a los casos de uso en los que participan. En la solución propuesta se ha decidido hacer una generalización de actores -- usuario socio o no socio -- y de casos de uso -- comprar zapatos o complementos (que podría haber sido una generalización de productos) --.

### **Relaciones extends.**

De la ejecución del caso de uso visualizar pedido se pueden derivar los casos de uso cancelar pedido y comprar artículo. Se trata de casos de uso que no tienen porque siempre llevarse a cabo.

### **Relaciones includes/use.**

Cuando se realiza la compra de un artículo, siempre habrá que pagar.

### **Notas:**

- Puesto que todos los usuarios pueden consultar la web de la zapatería, la relación consultar web la hace el actor usuario (padre de la relación de generalización), en cambio, generar pedido sólo puede llevarse a cabo por los socios, así que es éste el que participa en este caso de uso.

- También podría haberse considerado el actor entidad bancaria, partícipe del caso de uso pagar.

Como se indicaba en los contenidos del apartado 2.2, lo más importante en la elaboración de un diagrama de casos de uso, no es el diagrama en sí, sino la documentación de los casos de uso que es lo que permitirá desarrollar otros diagramas que ayuden en la codificación del sistema, y la elaboración de los casos de prueba de caja negra.

A modo de ejemplo vamos a desarrollar la documentación del caso de uso **Generar Pedido**, ya que, por su complejidad abarca todos los apartados que hemos visto. El ejemplo se hará con la herramienta Visual Paradigm for UML, aunque tu puedes usar la herramienta que consideres más oportuna

Los datos que debemos incluir para elaborar la documentación del caso de uso eran:

- **Nombre:** nombre del caso de uso.
- **Actores:** aquellos que interactúan con el sistema a través del caso de uso.
- **Propósito:** breve descripción de lo que se espera que haga.
- **Precondiciones:** aquellas que deben cumplirse para que pueda llevarse a cabo el caso de uso.
- **Flujo normal:** flujo normal de eventos que deben cumplirse para ejecutar el caso de uso exitosamente.
- **Flujo alternativo:** flujo de eventos que se llevan a cabo cuando se producen casos inesperados o poco frecuentes. No se deben incluir aquí errores como escribir un tipo de dato incorrecto o la omisión de un parámetro necesario.
- **Postcondiciones:** las que se cumplen una vez que se ha realizado el caso de uso.

Para incluir el nombre, actores, propósito, precondiciones y postcondiciones abrimos la especificación del caso de uso. Esto da lugar a la aparición de una ventana con un conjunto de pestañas que podemos llenar:

- En la pestaña "**General**" rellenamos el nombre "**Hacer pedido**", y tenemos un espacio para escribir una breve descripción del caso de uso, por ejemplo:

*"El cliente visualiza los productos que están a la venta, que se pueden seleccionar para añadirlos al pedido. Puede añadir tantos artículos como desee, cada artículo añadido modifica el total a pagar según su precio y la cantidad seleccionada.*

*Cuando el cliente ha llenado todos los productos que quiere comprar debe formalizar el pedido.*

*En caso de que el cliente no sea socio de la empresa antes de formalizar la compra se le indica que puede hacerse socio, si el cliente acepta se abre el formulario de alta, en caso contrario se cancela el pedido.*

*En caso de que se produzca algún problema con los datos bancarios se ofrecerá la posibilidad de volver a introducirlos.*

*Al finalizar un pedido se añade al sistema con el estado pendiente."*

- En la pestaña "**Valores etiquetados**" encontramos un conjunto de campos predefinidos, entre los que se encuentran el autor, precondiciones y postcondiciones, que podemos llenar de la siguiente manera:

**Autor:** usuario.

**Precondiciones:** Existe una campaña abierta con productos de la temporada actual a la venta.

**Postcondiciones:** Se ha añadido un pedido con un conjunto de productos para servir con el estado "pendiente" que deberá ser revisado.

También se pueden incluir otros datos como la complejidad, el estado de realización del caso de uso la complejidad que no hemos visto en esta unidad.

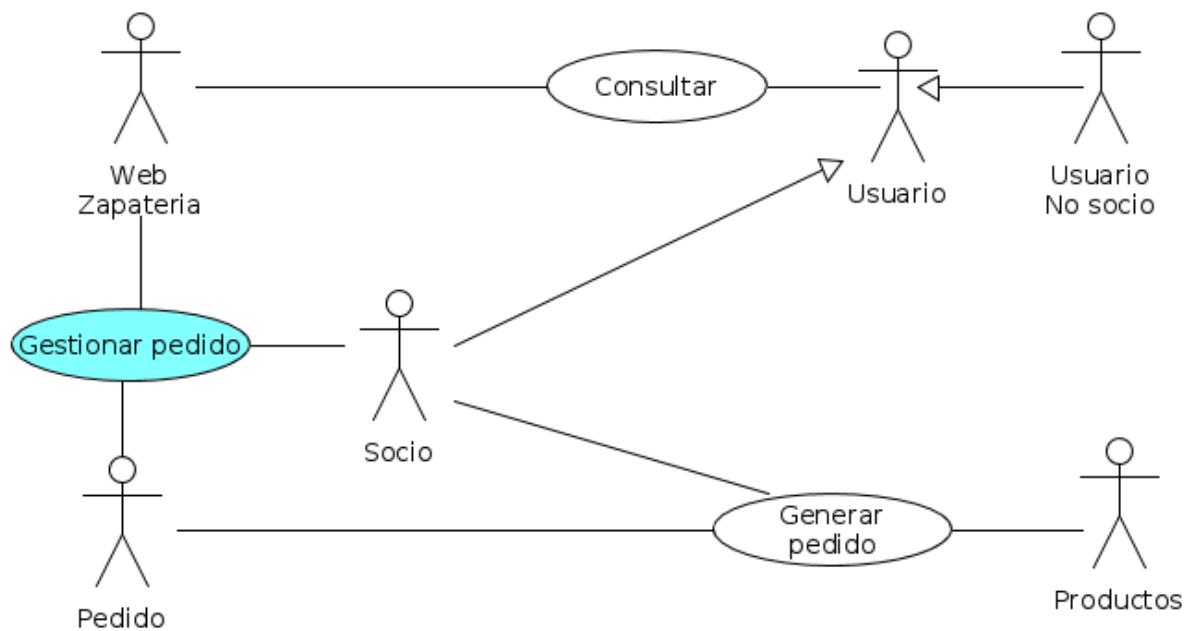
Para incluir el resto de los datos en el caso de uso hacemos clic en la opción "**Open Use Case Details...**" del menú contextual, lo que da lugar a la aparición de una ventana con una serie de pestañas. En ellas se recupera la información de la especificación que hemos llenado antes, además, podemos llenar el flujo de eventos del caso de uso, en condiciones normales usaríamos la pestaña "Flow of events", sin embargo como esta opción solo está disponible en la versión profesional, utilizaremos la pestaña "**Descripción**", que está disponible en la versión community. Para activarla pulsamos el botón "**Create/Open Description**".

Podemos añadir varias descripciones de diferentes tipos, pulsando el botón "**Nuevo**". Para añadir filas al flujo de eventos pinchamos en el botón. En

principio añadimos la descripción principal, luego añadiremos otras alternativas.

### **Tabla gestionar pedido.**

Una vez adaptado el diagrama para obtener el caso de uso gestionar pedidos, podría quedar como sigue:



La tabla resultante sería:

<b>Caso de uso</b>	Gestionar pedidos.		
<b>Actores</b>	Principal: usuario-socio. Secundarios: pedido, zapatería.		
<b>Propósito</b>	Gestión de los pedidos ya dados de alta en el sistema.		
<b>Pre-condiciones</b>	El pedido ha sido generado y está en modo activo. El usuario debe ser socio.		
<b>Flujo normal</b>	El flujo normal finaliza con la compra del pedido.	Visualizado del pedido.	
		Compra del pedido.	
		Pago del pedido.	
		Fin de flujo.	
<b>Flujos alternativos</b>	Cancelación del pedido.	Visualizado del pedido.	
		Cancelación del pedido. El pedido desaparece de la lista de pedidos activos.	
		Fin de flujo.	
	Imposibilidad de efectuar el pago.	Visualizado del pedido.	
		Compra del pedido.	
		Error en la operación de pago. La compra del pedido no se completa.	
		Fin de flujo.	

<b>Post-condiciones</b>	Las post-condiciones son estados en los que ha de quedar el sistema siempre, sea cual sea el flujo ejecutado (normal o alternativo). Para este enunciado no se identifica ninguna post-condición en particular.		
<b>Requerimientos trazados</b>	Los que correspondan en el documento de requisitos. Recuerda que el tipo de requisitos que se consideran en los diagramas de casos de uso son siempre de tipo funcional.		
<b>Puntos de inclusión</b>	Ninguno. Este diagrama no dispone de otros casos de uso con los que establecer estas relaciones.		
<b>Puntos de extensión</b>	Ninguno. Este diagrama no dispone de otros casos de uso con los que establecer estas relaciones.		

Esta es la descripción principal, en ella se describe el flujo normal de eventos que se producen cuando se ejecuta el caso de uso sin ningún problema.

### **Flujo de eventos normal para el caso de uso Hacer Pedido.**

<b>Use Case</b>	Hacer pedido
<b>Author</b>	usuario
<b>Date</b>	26-agosto-2011 13:56:56
<b>Brief Description</b>	El usuario selecciona un conjunto de artículos, junto con la cantidad de los mismos, para crear el pedido. Cuando se formaliza se comprueba que el usuario sea socio. A continuación se comprueban los datos bancarios, se realiza el cobro y se crea el pedido.
<b>Preconditions</b>	Existe un catálogo de productos disponibles para pedir. El usuario está registrado. Los datos bancarios son correctos.
<b>Post-conditions</b>	Se crea un pedido con los datos del usuario que lo realiza y los artículos solicitados.

Flow of Events		Actor Input	System Response
	1	Inicia el pedido.	
	2		Se crea un pedido en estado "en construcción".
	3	Selecciona un artículo.	
	4	Selecciona una cantidad.	
	5		Recupera la información del artículo para obtener el precio y modifica el precio total del pedido.
	6	El proceso se repite hasta completar la lista de artículos.	
	7	Se acepta el pedido.	
	8		Se comprueba si el usuario es socio, si no lo es se le muestra un aviso para que se registre en el sitio.
	9		Se comprueban los datos bancarios con una entidad externa.
	10		Se genera: calcula el total, sumando los gastos de envío.
	11		Se realiza el pago a través de la entidad externa.
	12		Se almacena la información del pedido con el estado "pendiente".

Añadimos un par de descripciones alternativas para indicar que hacer cuando el usuario no es socio y cuando los datos bancarios no son correctos:

**Flujo alternativo para el caso de uso Hacer Pedido cuando el usuario no está registrado.**

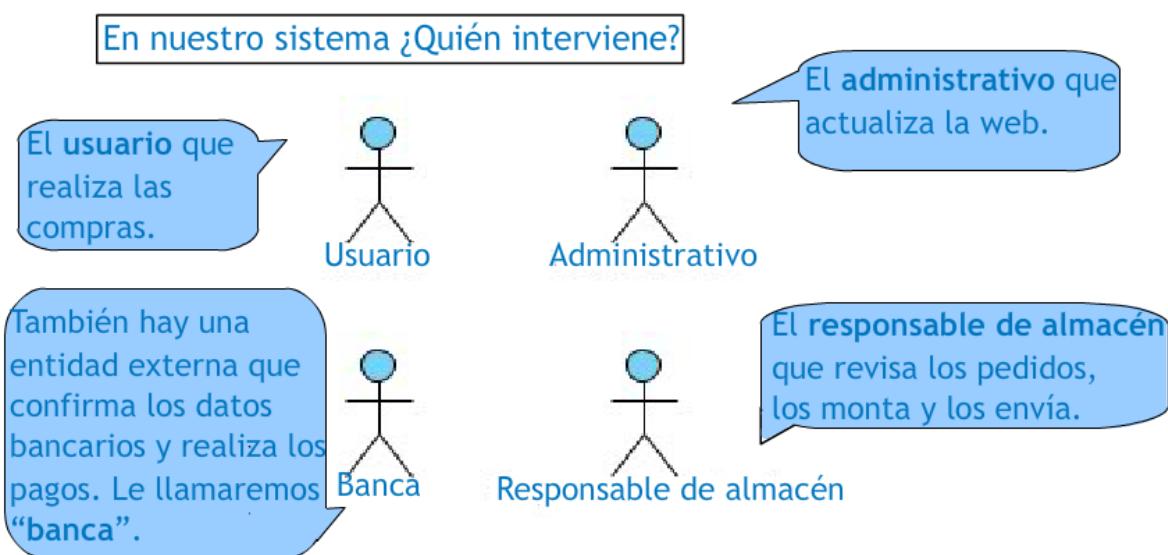
<b>Author</b>	usuario
<b>Date</b>	26-agosto-2011 18:14:35
<b>Brief Description</b>	Una vez que se han seleccionado los artículos y se han introducido los datos del socio, al hacer la comprobación de los datos bancarios con la entidad externa se produce algún error, se da la posibilidad al usuario de modificar los datos o de cancelar el pedido.
<b>Preconditions</b>	Existe un catálogo de productos disponibles para pedir. El usuario está registrado. Los datos bancarios no son correctos.
<b>Post-conditions</b>	Se crea un pedido con los datos del usuario que lo realiza y los artículos solicitados.

	Actor Input	System Response
Flow of Events	1 Inicia el pedido.	
	2	Se crea un pedido en estado "en construcción".
	3 Selecciona un artículo.	
	4 Selecciona una cantidad.	
	5	Recupera la información del artículo para obtener el precio y modifica el precio total del pedido.
	6 El proceso se repite hasta completar la lista de artículos.	
	7 Se acepta el pedido.	
	8 Acepta el pedido.	
	9	Se comprueban los datos bancarios con una entidad externa, fallando la comprobación.
	10	Se solicitan los datos de nuevo.
	11 Introduce los datos de nuevo.	
	12	Se repite el proceso hasta que se acepten los datos bancarios o se cancele la operación.
	13	Se genera: calcula el total, sumando los gastos de envío.
	14	Se realiza el pago a través de la entidad externa.
	15	Se almacena la información del pedido con el estado "pendiente".

### Debes conocer:

## Primeros pasos

- Antes de elaborar el diagrama tienes que leer con detenimiento el documento con la especificación del problema a resolver y asegurarte de que entiendes la idea central del problema, crear una tienda virtual en la que se puedan realizar pedidos de los productos a la venta (zapatos). El proceso se centra en el pedido, desde poner a disposición del cliente los artículos en venta, pasando por la selección de artículos a pedir, la cumplimentación de toda la información necesaria para el pedido, pago, confección del pedido, envío y reajuste del stock en almacén, todo ello, a través de la web.



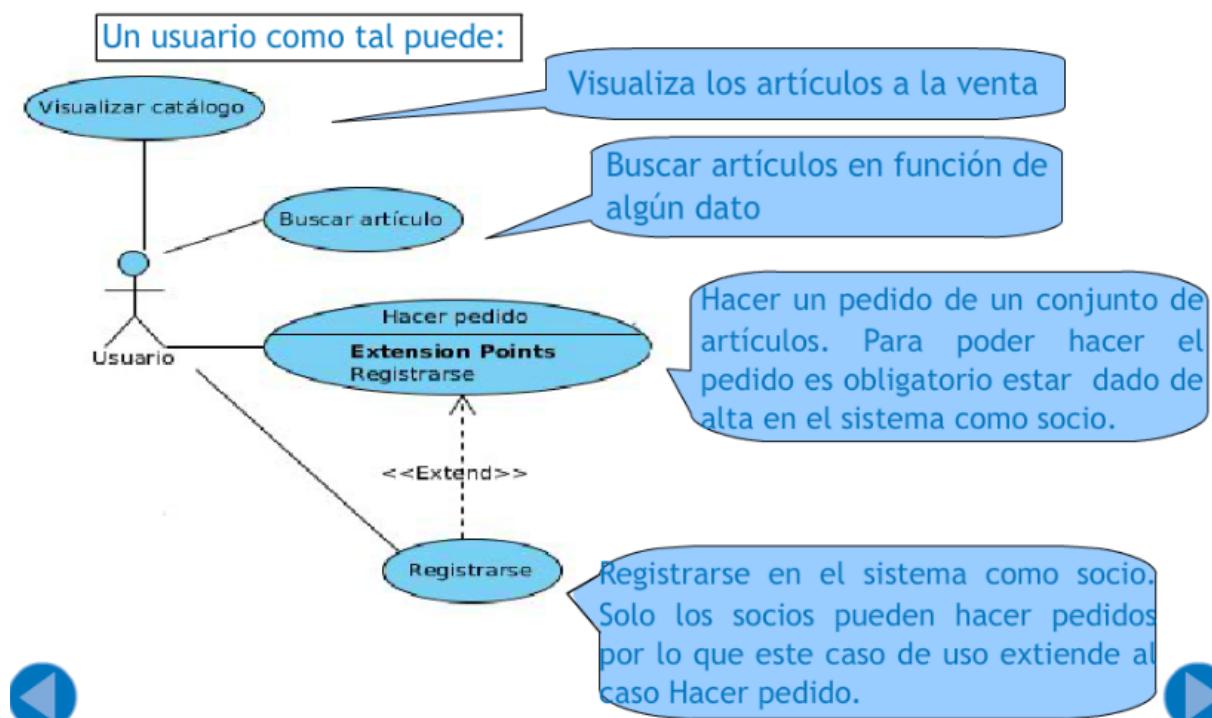
## Identificar funcionalidades

- Para facilitar la creación del diagrama vamos a ir sacando funcionalidades para casa usuario.
- Debemos recordar que una caso de uso representa una interacción de un actor con el sistema, que está relacionado con los requisitos funcionales de la aplicación final y que, en definitiva representa tareas que llevará a cabo el sistema.

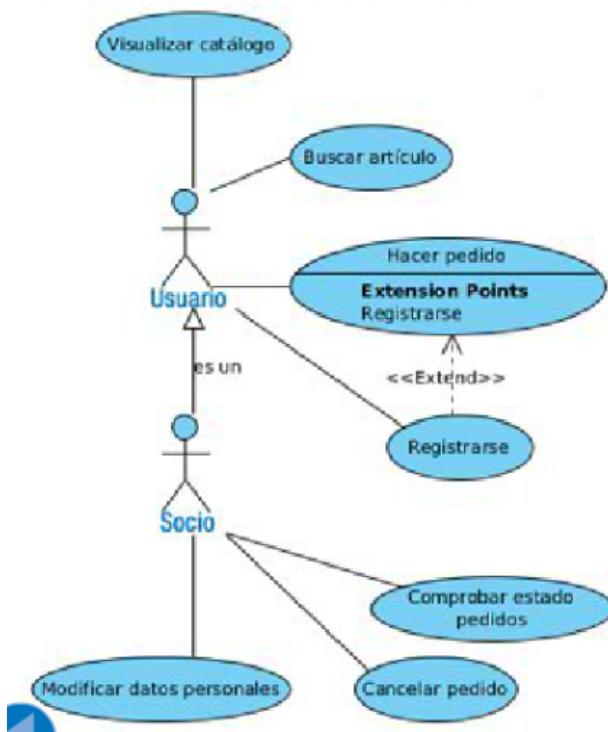
## Funcionalidades del usuario

- Cuando una persona se conecta al sistema lo primero que podrá hacer será visualizar el catálogo de la temporada.
- También puede hacer un **pedido** con uno o varios artículos del catálogo, para ello visualizará los artículos de forma que pueda seleccionar algunos de ellos e indicar la cantidad que quiere comprar.
- También puede hacer **búsquedas** por datos concretos de artículos.
- Cualquier persona que acceda al sistema puede **darse de alta** para ser socio.
- Así mismo, si es socio, podrá **comprobar el estado** de sus pedidos y cancelarlos.

## Funcionalidades del usuario



## Funcionalidades del usuario



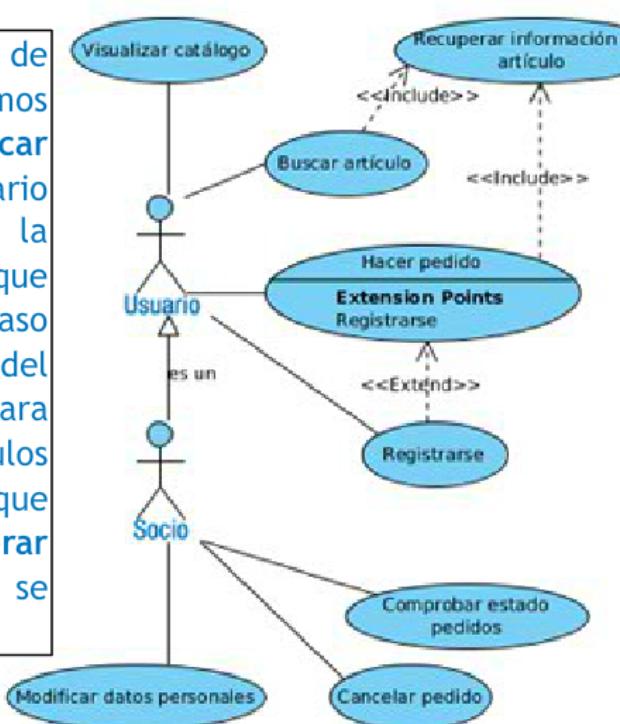
Al hacerse socio la identidad del usuario con respecto del sistema cambia, por lo que surge un nuevo tipo de actor que hereda de usuario:

Un socio es un usuario que se ha registrado en el sistema. Los socios almacena cierta información, como sus datos personales y bancarios email, etc.

El socio, además, reúne una serie de funcionalidades propias (el usuario no puede hacerlas), como comprobar el estado de sus pedidos, o cancelarlos, así como modificar sus datos en el sistema

## Funcionalidades del usuario

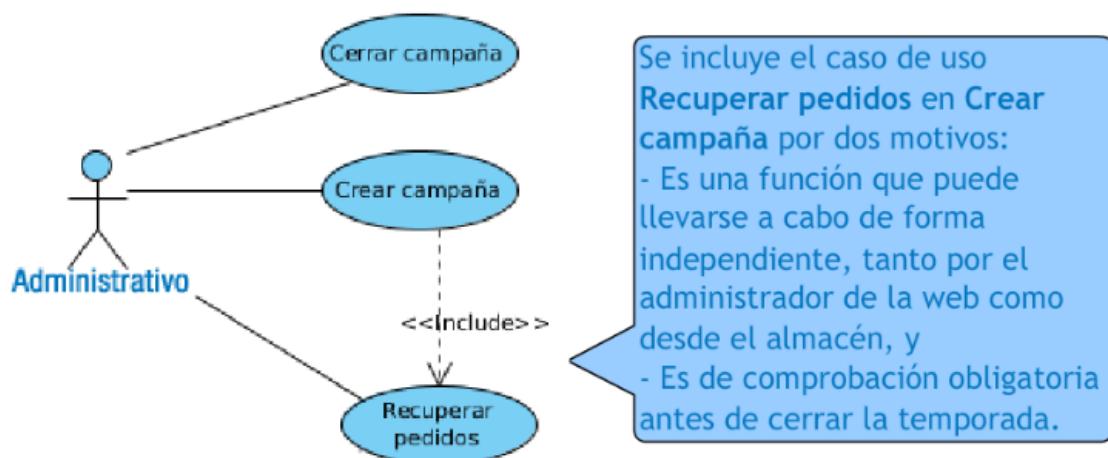
Al revisar un poco la funcionalidad de los casos de uso del usuario podemos comprobar que en los casos **Buscar artículo** y **Hacer pedido** es necesario buscar en el sistema y recuperar la información de un artículo del que tenemos algún dato, en el primer caso para obtener todos los datos del artículo buscado y en el segundo para recuperar el precio de los artículos que se añaden al pedido, por lo que extraemos el caso uso “**Recuperar información de artículo**” que se incluye en los otros dos.



## Funcionalidades del administrador

- El objetivo del administrador web es gestionar los contenidos de la web, en concreto de las diferentes campañas, ya que cada temporada se debe cerrar la campaña antigua, retirando los artículos de la temporada anterior y abrir la temporada nuevo, añadiendo sus artículos. Para que se pueda cerrar una temporada es necesario que en el almacén se hayan gestionado todos sus pedidos, por lo que es obligatorio comprobarlo, antes de cerrar.

## Funcionalidades del administrador

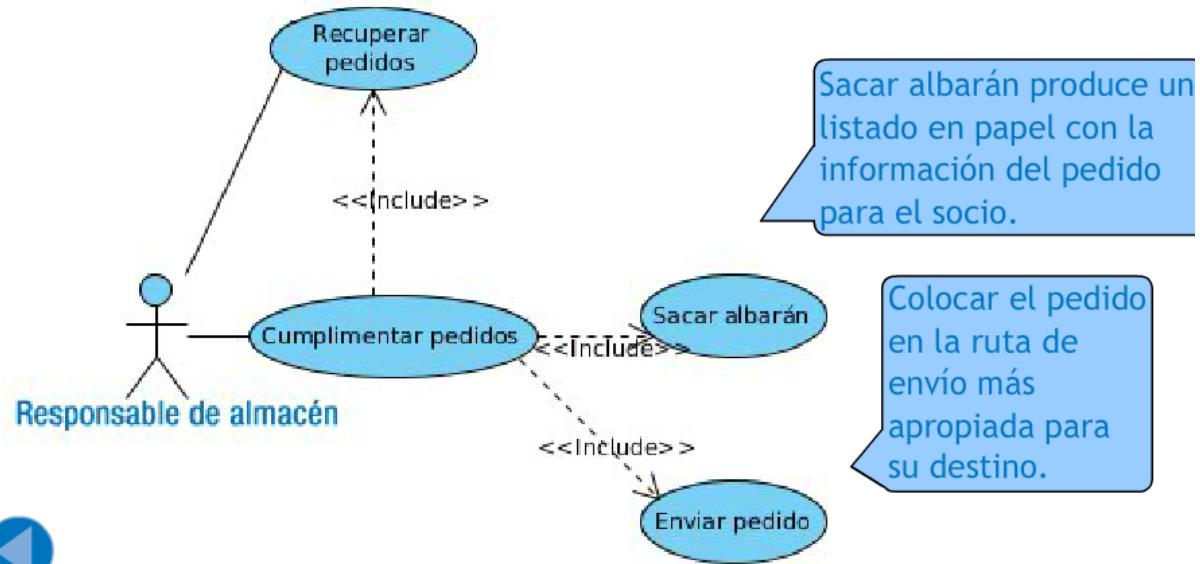


## Funcionalidades del responsable de almacén

- Es el encargado de leer los pedidos de los usuarios y cumplimentarlos, está será su única función, si bien, es una función complicada, ya que implica realizar una serie de tareas:
  - Seleccionar el pedido más antiguo.
  - Buscar los artículos a servir.
  - Empaquetarlos junto con un albarán para el socio.
  - Colocarlos en su ruta de envío.

## Funcionalidades del responsable de almacén

El primer paso es recuperar la lista de pedidos sin procesar. Esta tarea recupera el pedido más antiguo para ser procesado.



## Ejercicio resuelto 2 ("QUIJOTE"). Elaboración de un diagrama de casos de uso.

La empresa Quijote se dedica a la venta de material informático puerta a puerta, ofrece sus productos a los clientes en sus propios domicilios. Sus empleados se organizan en dos grandes grupos: vendedores y publicitarios. Los publicitarios tratan de facilitar el acceso de los vendedores a los clientes para que éstos les hagan llegar los catálogos de productos y realicen las operaciones de ventas.

Los publicitarios anualmente encargan a la consultora Sancho un estudio de sus resultados, y en función de los datos que desprenda pueden realizar un análisis de mercado. Gracias a la información obtenida en el análisis se hacen campañas publicitarias en radio y televisión.

La política de la empresa Quijote se basa en tener grandes profesionales en sus filas, por lo que todos sus empleados reciben formación periódicamente.

### Se pide:

- Diagrama de casos de uso. Identifica los actores y casos de uso, incluye relaciones de asociación, identifica generalizaciones. Si consideras alguna relación tipo include o extend, justifica su uso.

- Si agrupamos todos los casos de uso que hayas considerado en el proceso relacionado con los empleados de marketing/publicitarios en uno sólo.  
Desarrolla la notación escrita del caso de uso.

### **Resolución:**

Los diagramas de casos quedan encuadrados en la fase de análisis de los proyectos, se entienden como una puesta en común entre el cliente y el analista sobre como entender requisitos funcionales.

Al utilizarse UML, se entiende que ambos interlocutores conocen la notación propia de estos diagramas y las reglas sobre como combinar los diferentes símbolos, de forma que no hay ambigüedades que sí podrían darse mediante una descripción escrita.

Completar un diagrama de casos de uso supone versionar el diagrama tantas veces como sean necesarias hasta que la vista del proyecto que se presenta al cliente garantiza que el requisito funcional ha sido entendido.

En nuestros ejercicios, sólo hay una descripción que no permite ir refinando el diagrama mediante sucesivas consultas con el cliente, por lo tanto, las soluciones propuestas podrán ser diversas, todas ellas válidas siempre que reflejen razonablemente lo propuesto en el enunciado.

A la hora de valorar el ejercicio, lo que no es interpretable, es que el uso de la notación sea el correcto y que se muestre una variedad de los símbolos que conocemos para estos diagramas.

Con estas consideraciones, se propone la siguiente solución:

### **Actores que participan en el problema.**

Empleado (que podrá ser ventas o publicitario), cliente y consultora.

### **Generalizaciones.**

Se observan dos posibles generalizaciones, que en el desarrollo del diagrama podrán ser expresadas en relación a actores o en relación a los casos de uso en los que participan. En la solución propuesta se ha decidido hacer una generalización de actores -- empleado de ventas y empleado publicitario -- y de casos de uso -- lanzar campaña por radio o por televisión --.

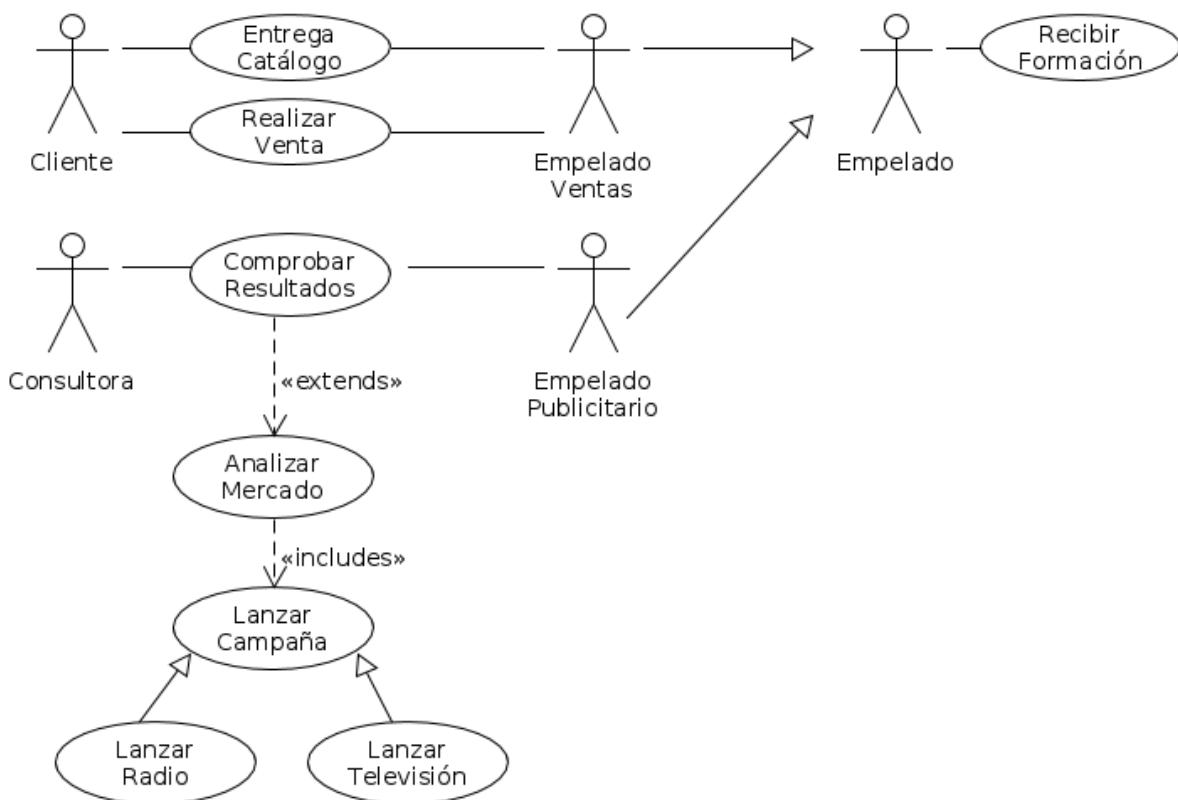
### **Relaciones extends.**

De la ejecución del caso de uso comprobar resultados se puede derivar el caso de uso analizar mercado.

### **Relaciones includes/use.**

Cuando se realiza un análisis de mercado, siempre se lanza una campaña publicitaria.

Quedando el siguiente diagrama.



## Tabla promocionar Quijote.

Una vez adaptado el diagrama para obtener el caso de uso promocionar Quijote, podría quedar como sigue:



La tabla resultante sería:

<b>Caso de uso</b>	Promocionar Quijote.
<b>Actores</b>	Empleado marketing (principal), consultor (secundario).
<b>Propósito</b>	Promocionar la empresa para ganar nuevos clientes.
<b>Pre-condiciones</b>	Seleccionar consultora con experiencia en el sector.
<b>Flujo normal</b>	<p>Como flujo normal vamos a suponer que del estudio se genera un análisis de mercado, sería igualmente válida la decisión contraria.</p> <p>El empleado solicita comprobar los resultados del año en curso a la consultora.</p> <p>La consultora concluye que se precisa un análisis de mercado de acuerdo al estado actual de la empresa.</p> <p>Se lanza una campaña publicitaria en radio.</p> <p>Se lanza una campaña publicitaria en televisión.</p> <p>Fin de flujo.</p>

<b>Flujos alternativos</b>	El análisis de mercado no muestra potenciales clientes.	El empleado solicita comprobar los resultados del año en curso a la consultora.
		La consultora concluye que se precisa un análisis de mercado de acuerdo al estado actual de la empresa, pero no se detectan clientes futuros.
	El presupuesto para las campañas publicitarias disponible no es suficiente	Fin de flujo.
		El empleado solicita comprobar los resultados del año en curso a la consultora.
		La consultora concluye que se precisa un análisis de mercado de acuerdo al estado actual de la empresa.
		Se lanza una campaña publicitaria en radio.
		Se suspende la campaña publicitaria en televisión por falta de fondos.
		Fin de flujo.
<b>Post-condiciones</b>	No se identifican en este ejercicio. Serían aquellas condiciones que se deben cumplir <u>siempre</u> al finalizar el caso de uso sea cual sea el flujo ejecutado.	
<b>Requisitos trazados</b>	Los que correspondan en el documento de requisitos. Recuerda que el tipo de requisitos que se consideran en los diagramas de casos de uso son siempre de tipo funcional.	
<b>Puntos de inclusión</b>	Ninguno. Este diagrama no dispone de otros casos de uso con los que establecer estas relaciones.	
<b>Puntos extensión</b>	Ninguno. Este diagrama no dispone de otros casos de uso con los que establecer estas relaciones	

## Ejercicio resuelto 3 ("ALQUILER DE PISOS Y LOCALES"). Elaboración de un diagrama de casos de usos.

Una empresa de alquiler de pisos y locales desea diseñar un sistema que cumpla los siguientes requisitos:

- Los propietarios previa identificación en el sistema, podrán dar de alta o baja un piso o un local. También podrán modificar los datos de ese piso o local.
- Los futuros inquilinos también deben identificarse antes de poder usar el sistema. Al acceder se les presenta un menú donde pueden elegir la acción a realizar:

1. listar pisos y locales disponibles.
2. solicitar el alquiler de un piso o local.

Para alquilar un local se le pedirá su email y para alquilar un piso su número de teléfono

### **Diagrama de casos de uso**

1. Identificar los actores.
2. Identificar los casos de uso.
3. Implementar con UMLet el diagrama de casos de uso.

#### **Resolución:**

##### **1.-Identificar los actores.**

Los actores identificados son : Propietario e Inquilino.

PropietarioEl propietario será el responsable de la gestión de los pisos o locales mediante el alta, la baja y las modificaciones de los mismos. Una vez identificado en el sistema podrá gestionar pisos o locales.

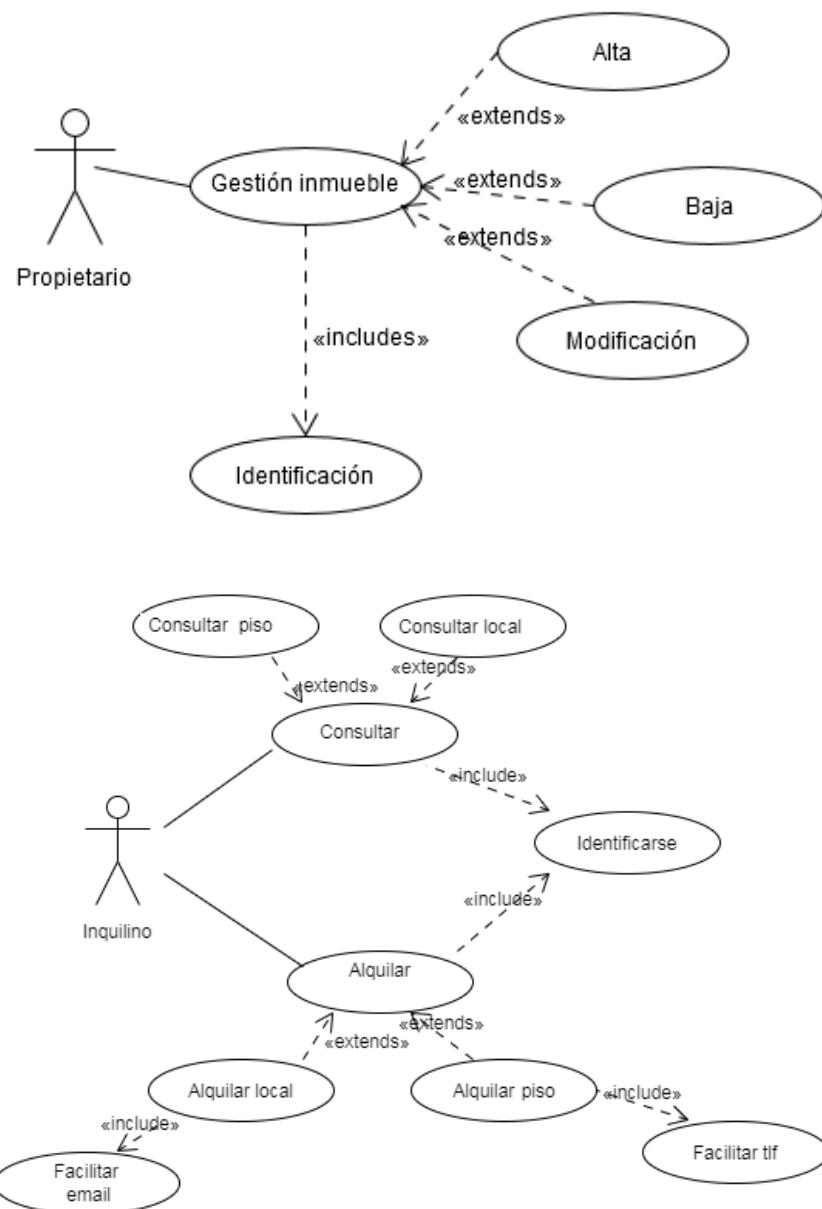
InquilinoEl inquilino consultará los pisos o locales y también podrá solicitar el alquiler. Debe identificarse en el sistema.

##### **2.-Identificar los casos de uso.**

El propietario podrá dar de alta, baja dar de baja o modificar un inmueble. Si se ofrece un menú podemos usar extends con estos casos de uso y uno llamado gestión inmueble. El inquilino siempre que quiera gestionar un inmueble se identifica.

El inquilino podrá consultar o alquilar. En base a lo que elija podrá ser local o piso. Cuando alquile deberá facilitar teléfono o email según proceda. También debe identificarse para realizar las acciones.

##### **3.-Implementar con UMLet el diagrama de casos de uso.**

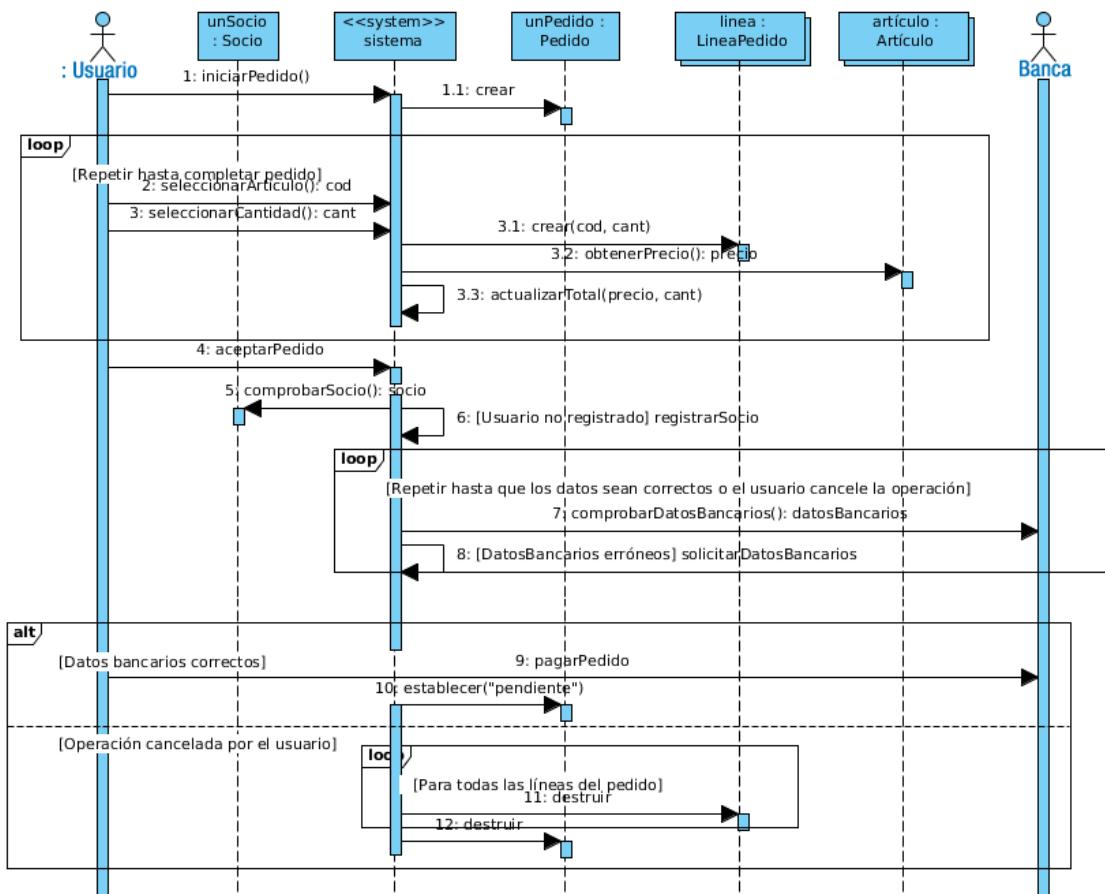


## Ejercicio resuelto 1 ("Generar pedido"). Elaboración de un diagrama de secuencias

Vamos a generar el diagrama de secuencia que representa el caso de uso "**Generar pedido**" del diagrama de casos de uso del ejercicio resuelto 1 "ZAPATERÍA TACÓN DE ORO" (en el punto 2 de "Los diagramas de casos de uso"). En dicho diagrama se establece la secuencia de operaciones que se llevarán a cabo entre los diferentes objetos que intervienen en el caso de uso.

Este es el diagrama ya terminado, en el se han incluido todas las entidades (actores, objetos y sistema) que participan en el diagrama, y se han descrito todas las operaciones, incluidos los casos especiales, como es el registro de usuarios o la gestión de los datos bancarios. También incluye el modelado de

acciones en bucle, como es la selección de artículos y de acciones regidas por condición, como es la posibilidad de cancelar el pedido si hay problemas con la tarjeta de crédito.

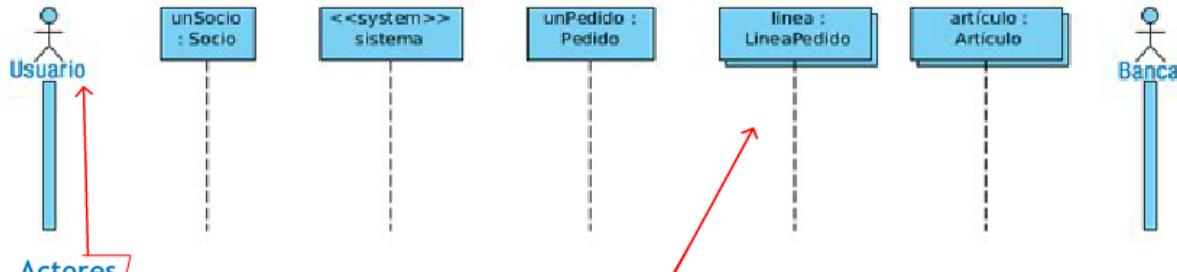


**Debes conocer:**

## Crear el diagrama de secuencia

- El diagrama de secuencia que vamos a tratar es el del caso de uso Hacer pedido, que hemos descrito en el apartado anterior. Si no recuerdas bien cual era la descripción del caso te invito a que la repases en el punto 2.4 de los contenidos de la unidad.
- Este es un diagrama muy completo que incluye bastantes elementos de este diagrama, como bucles o condicionantes, veamos como incluirlos con la herramienta Visual Paradigm, como siempre, debes saber que puedes investigar y utilizar otras herramientas que sirvan para este mismo propósito.

## Incluir entidades



### Actores.

El primer paso es incluir las entidades participantes. Tenemos dos actores, el usuario que inicia el caso de uso y la banca que es un actor secundario que realiza una operación por nosotros como es comprobar que los datos de la tarjeta sean correctos.

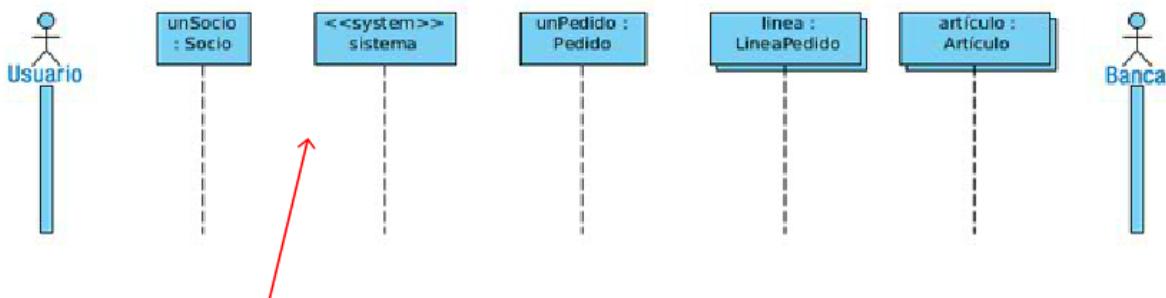
### Objetos.

También necesitamos que intervengan cuatro objetos de nuestro sistema: un socio, un pedido con sus líneas de pedido, y los artículos a pedir.

Para nombrar estas entidades en la especificación indicamos su nombre y seleccionamos en el clasificador de base, que muestra la lista de clases del sistema la clase a la que pertenece, así aparece automáticamente su nombre en el formato adecuado.

Indicaremos que las líneas de pedido y los artículos son un conjunto de objetos marcando en la especificación la casilla “Multi-object”, en la zona inferior del cuadro de diálogo.

## Incluir entidades



### Sistema.

Por último también interviene el sistema en si mismo, que utilizaremos para las operaciones relacionadas con la interfaz gráfica y las que se lanzan directamente para crear objetos, recuperar información del sistema como los datos de un artículo, o comunicarnos con entidades externas como la banca.

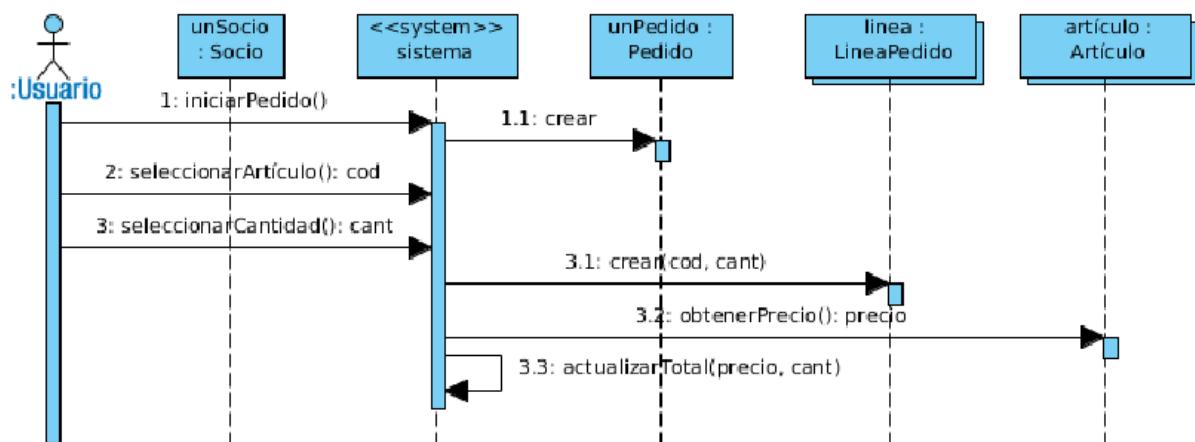
Se añade como una línea de vida más, a la que en su especificación indicamos que su nombre es sistema, y en la pestaña **Estereotipos** seleccionamos **System**.

## Mensajes

Los mensajes pueden devolver un valor, que podemos escribir en la especificación del mismo en la opción return value.

## Mensajes

En esta imagen vemos los mensajes correspondientes al apartado de añadir los datos del pedido. Se selecciona el artículo y la cantidad, se crea una nueva línea de pedido y se recuperan los datos del artículo para obtener su precio y actualizar el total. Puesto que este proceso lo podemos repetir en más de una ocasión lo meteremos en un bucle.

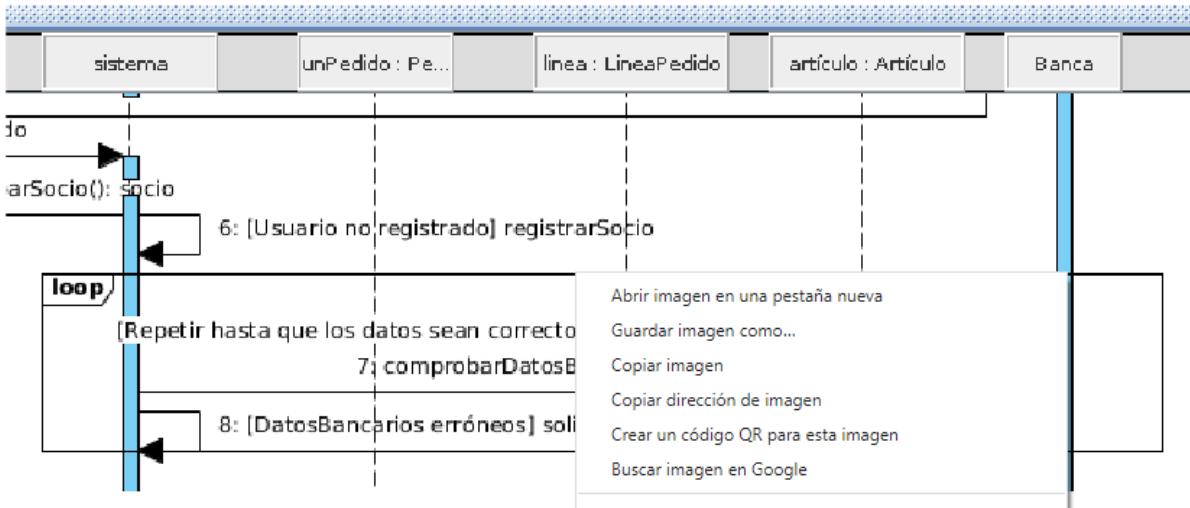


## Mensajes

Para añadir un bucle seleccionamos la opción Loop Combined Fragment.

## Mensajes

Añadiremos condiciones de guarda cuando queramos indicar que un mensaje se enviará sólo si se cumple cierta condición, por ejemplo, sólo registraremos a un usuario si no es socio ya.



## Mensajes

También se pueden incluir condiciones más elaboradas que implique una bifurcación en el flujo de eventos, como es el caso de la comprobación de la tarjeta, si todo marcha bien se finalizará la creación del pedido, si no, el usuario puede cancelar la operación y terminar sin guardar nada.

## Ejercicio resuelto 2 ("Estadio"). Elaboración de un diagrama de secuencia

Se pretende desarrollar un programa que dé respuesta a las necesidades de un estadio de fútbol en uno de sus partidos. Tras varias entrevistas con el responsable, se ha llegado al acuerdo de que los requisitos funcionales se pueden recoger en el caso de uso Gestión del estadio donde hay que considerar actividades como:

- **Control de puntos de acceso.** Se dispone de 2 puertas, a las que el responsable del estadio dará orden de apertura. Cada puerta hace un test interno y devuelve al responsable Ok o Ko. Si una puerta no abre, queda inactiva para el resto del partido. El proceso de apertura se realiza de forma simultánea en ambas puertas. Al finalizar el partido, el responsable del estadio dará orden de cierre de las puertas que se han usado durante el partido.

- **Control de acceso de aficionados.** Si la puerta está disponible, el responsable da orden de iniciar el acceso al estadio. Mientras haya aficionados, el operario de la puerta valida la entrada, si es correcta le da acceso al campo, en caso contrario avisa al responsable del estadio que ha habido un intento de acceso con entrada falsa.

**Nota:**

- Considera en este diagrama que la puerta uno está averiada y la puerta dos operativa.

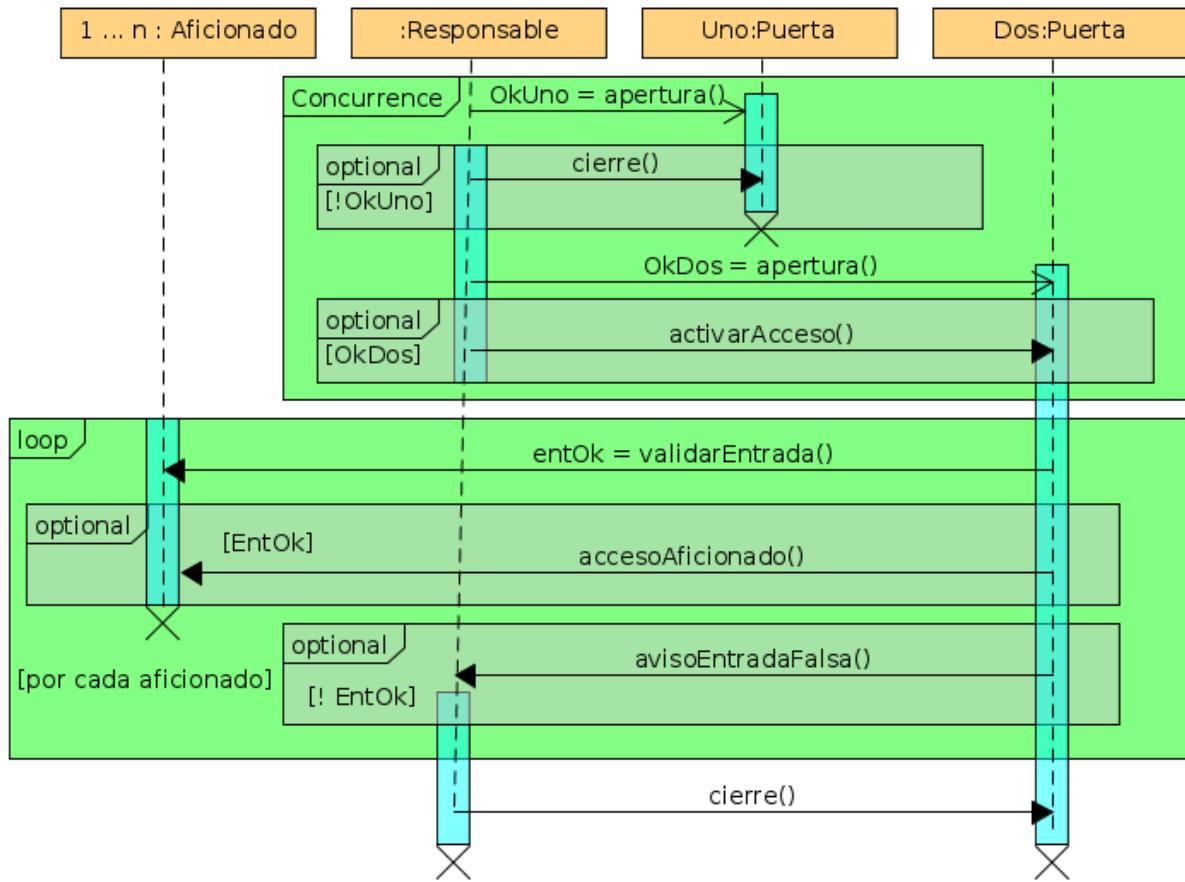
**Se pide:**

Desarrollar el diagrama de secuencia resultante del caso de uso planteado.

**Atiende a aspectos como:**

- Identificación de los objetos involucrados en el diagrama de secuencia.
- Que el diagrama recoja la secuencia de mensajes intercambiados, tomando en consideración las funcionalidades descritas en el enunciado.
- Uso correcto de la notación vista para este tipo de diagramas.
- Que los diagramas generados sean visualmente útiles. Un diagrama de secuencia debe dar una idea clara/rápida de la secuencia de acciones que se derivan de la ejecución del caso de uso que representa.

**Resolución:**



## Ejercicio resuelto 3 ("ROPERO"). Elaboración de un diagrama de secuencia.

Se pretende crear un programa para la gestión de ciertas funciones de una discoteca. En particular, el depósito de los abrigos en el ropero.

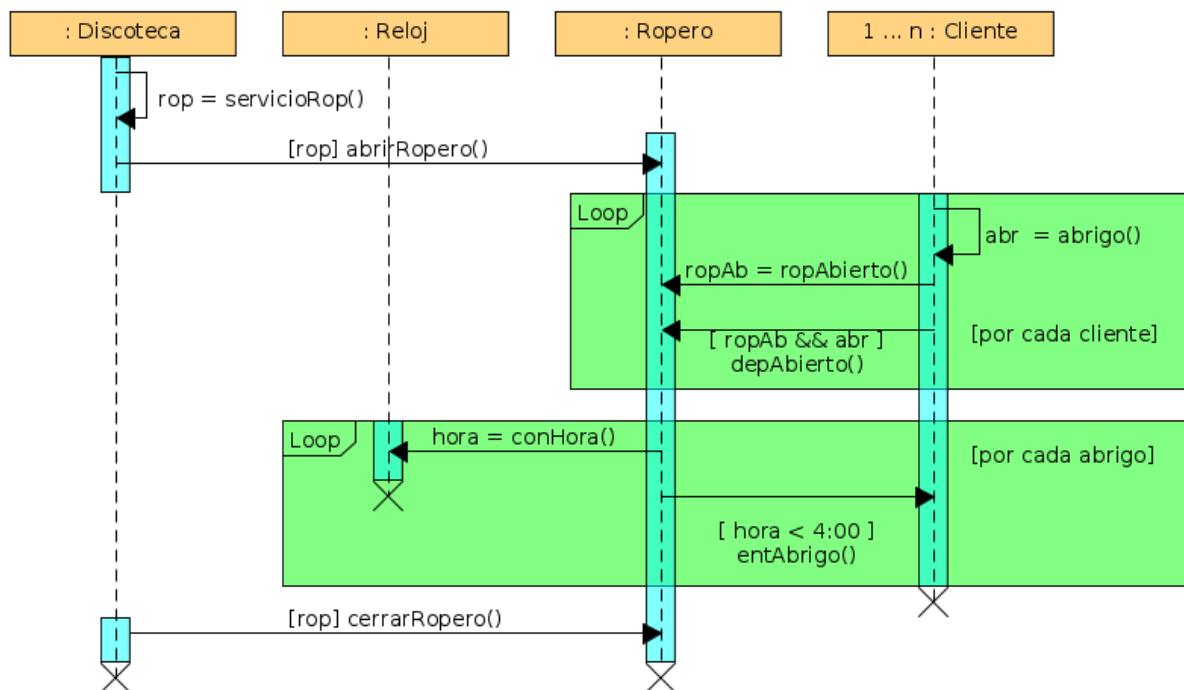
Algunas consideraciones:

- El responsable de la discoteca cada día decide si abre el vestíbulo, y es el encargado de su apertura y cierre.
- El propio ropero informa a los clientes si proporciona o no servicio de guardarropa.
- Si está abierto, a primera hora recoge los abrigos de los clientes que tengan abrigo.
- Al finalizar la jornada, mientras haya abrigos en el ropero se devuelven siempre que sea antes de las 4 de la mañana.

- Se pide desarrollar el diagrama de secuencia resultante del caso de uso planteado.

Considera aspectos como:

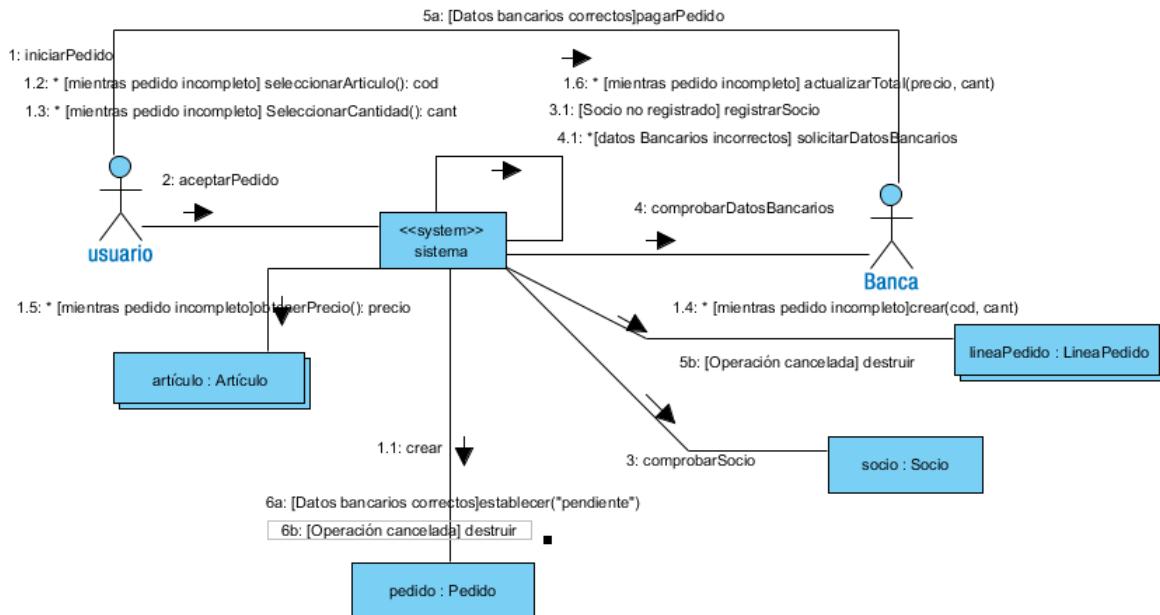
- Identificación de los objetos involucrados.
- Que el diagrama recoja la secuencia de mensajes intercambiados, tomando en consideración las funcionalidades descritas en el enunciado.
- Uso correcto de la notación vista para este tipo de diagramas.



## Ejemplo de un diagrama de colaboración

A continuación se muestra un diagrama de colaboración de ejemplo.

Este es el diagrama de colaboración que representa el caso de uso "**Generar pedido**" del diagrama de casos de uso del ejercicio resuelto 1 "ZAPATERÍA TACÓN DE ORO" (en el punto 2 de "Los diagramas de casos de uso"). Se ha creado siguiendo el diagrama de secuencia, por lo que no te debe ser muy difícil seguirlo, de hecho algunas aplicaciones para la creación de estos diagramas permiten la obtención de uno a partir de otro. Debes tener en cuenta que la aplicación modifica un poco la firma de los mensajes, el valor devuelto se representa al final precedido de dos puntos.



Los aspectos más destacados son los siguientes:

- Las actividades que se repiten o pueden repetirse se marcan con un asterisco y su condición.
- Las condiciones de guarda se escriben en el mismo nombre del mensaje.
- El flujo alternativo de eventos según si el usuario cancela el pedido o no, obliga a modificar los números de secuencia de los mensajes 5 y 6, pasando a tener los mensajes 5a y 6a y 5b y 6b, según la condición. Puedes modificar el número de secuencia de los mensajes abriendo la especificación del diagrama, y seleccionando la pestaña Mensajes, donde puedes editar los números de secuencia haciendo doble clic sobre ellos.
- Al objeto "sistema" se le ha asignado el estereotipo system.

## Ejercicio resuelto 1 ("Generar pedido"). Elaboración de un diagrama de estados.

Para exemplificar la creación de un diagrama de estados vamos a ver el que representa el caso de uso "**Generar pedido**" del diagrama de casos de uso del ejercicio resuelto 1 "ZAPATERÍA TACÓN DE ORO" (en el punto 2 de "Los diagramas de casos de uso") que cumple con las condiciones que hemos visto al principio, tiene un comportamiento significativo en tiempo real, ya que su

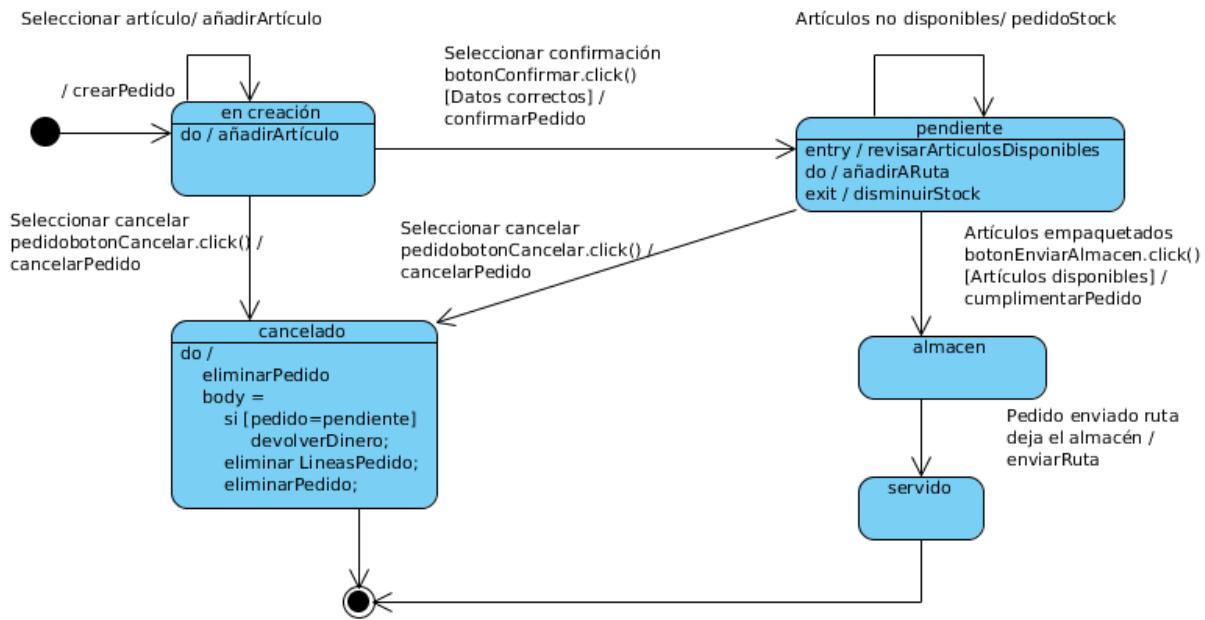
situación tanto física, como el sistema, va evolucionando conforme pasa el tiempo, y participa en varios casos de uso (como Hacer pedido y Cumplimentar pedido).

Los diferentes estados en los que puede estar un pedido son:

- **En creación:** es cuando se están seleccionando los productos que formará el pedido.
- **Pendiente:** está en este estado desde que se confirma el pedido hasta que se selecciona para preparar su envío.
- **En almacén:** está en este estado cuando es elaborado el paquete y se ha asignado a una ruta, hasta que se envía a través de la ruta que le corresponde.
- **Servido:** Cuando el pedido es enviado. En este caso se envía una señal física desde el almacén cuando el transporte abandona el almacén.
- **Cancelado:** puede llegar a esta situación por dos motivos, o bien se cancela mientras se está haciendo por problemas con la tarjeta de crédito, o bien porque, una vez pendiente de su gestión el usuario decide cancelarlo, la diferencia fundamental entre ambos es que en el segundo caso hay que devolver el importe pagado por el pedido al socio que lo ha comprado.

Las transiciones entre estados se producen por llamadas a procedimientos en todos los casos, no intervienen cambios de estado o el tiempo, ni señales.

El diagrama quedaría de la siguiente manera:



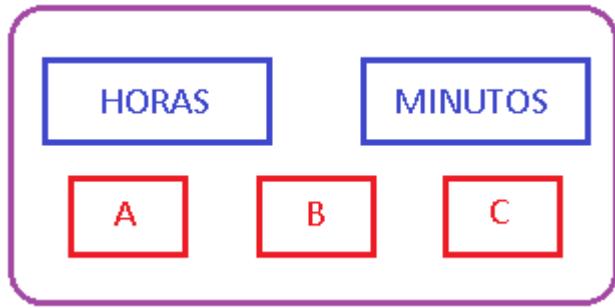
En las **transiciones** se ha incluido el nombre de la transición, el evento que la dispara (normalmente hacer clic en algún botón de la interfaz), si existe condición de guarda se pone entre corchetes y la acción a realizar para llegar al siguiente estado junto al símbolo /. En todos los casos el evento de disparo es de tipo llamada (incluye la llamada a una función o pulsar un botón de la interfaz), salvo en el caso de pedido enviado que se controla por una señal que se envía cuando el transporte abandona el almacén.

A los **estados** se les ha añadido la acción a realizar, apartado do/ y en algunos casos la acción de entrada, por ejemplo en el caso del estado pendiente, se debe revisar que los artículos a enviar tengan disponibilidad y la de salida, en el ejemplo disminuir el stock.

**Nota:** para incluir las condiciones de guarda en el diagrama debes rellenar el apartado "Guard" de la especificación, si necesitas añadir alguna acción puedes hacerlo rellenando el apartado "Effect". Los eventos de disparo.

## Ejercicio resuelto 2 ("RELOJ"). Elaboración de un diagrama de estados.

La siguiente figura muestra un **reloj digital** cuyo comportamiento se describe a continuación:



El reloj se enciende y está visualizando las horas y minutos.

Funciones de reloj:

Pulsado de A durante tres segundos: parpadea la hora. Para evitar cambios de hora involuntarios, si el tiempo de pulsado es inferior a tres segundos no se activa la función.

El botón B no funciona, si no se ha pulsado antes el botón A durante 3 segundos.

De tal forma que si el reloj está en el estado en el que la hora está parpadeando:

1. Si se pulsa el botón B incrementa la hora en una unidad.
2. Si se pulsa el botón A, pasará al estado de poder cambiar los minutos. Los minutos parpadearán. No se precisa mantener pulsado el botón porque se entiende que se está modificando la hora de forma voluntaria.

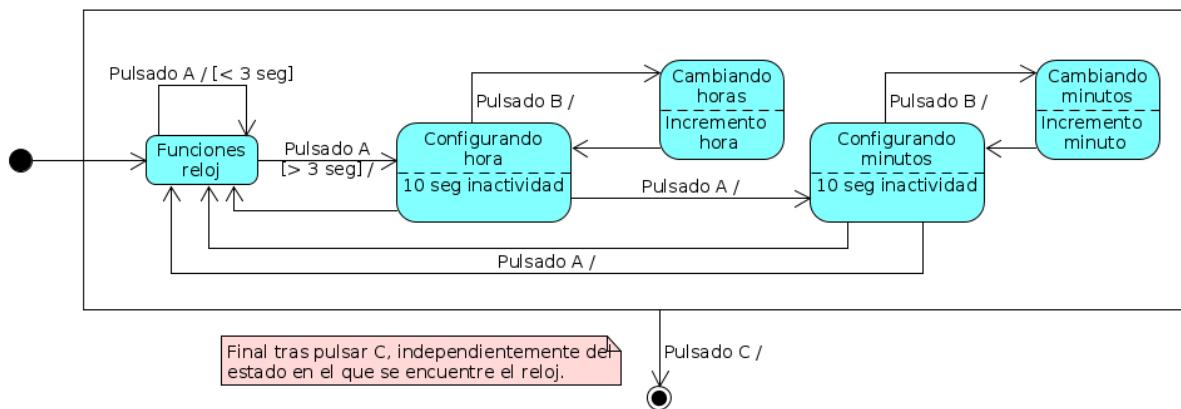
Si está el reloj en el estado de poder cambiar los minutos:

1. Pulsado del botón B: cada pulsación del botón B incrementa los minutos en una unidad
2. Si pulsamos el botón A, finaliza el modo configuración y vuelve a mostrar la hora.

Pulsado de C: apaga del reloj sin tener en consideración el estado en el que se encuentre.

Cuando el reloj está en modo configuración de horas o minutos, tras 10 segundos de inactividad abandona la configuración y pasa a modo funciones de reloj.

**Resolución:**



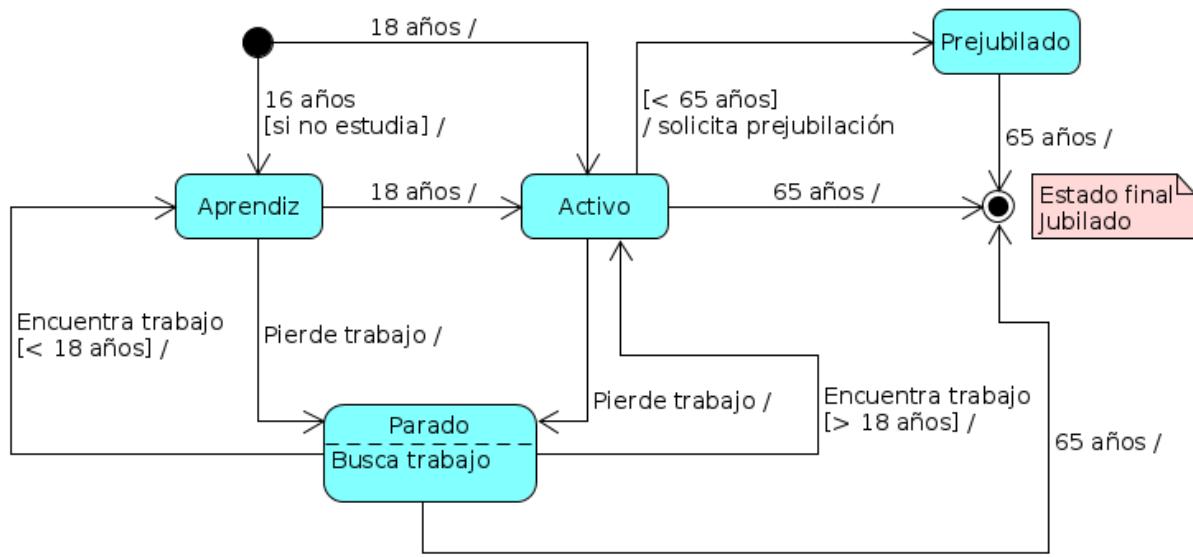
- Las transiciones sin eventos, aunque pueden tener condiciones, se producen al finalizar la actividad del estado.
  - Es posible salir de un estado tanto por un evento externo, como por el fin de la actividad del estado.
  - El evento pulsado C genera el apagado del reloj con independencia del estado en el que se encuentre.

## Ejercicio resuelto 3 ("VIDA LABORAL"). Elaboración de un diagrama de estados

Crea un diagrama de estados que muestre la **evolución de un empleado** a lo largo de su vida laboral.

Se considera que el proceso transcurre entre los 16 y 65 años de edad y se plantean los siguientes **estados**:

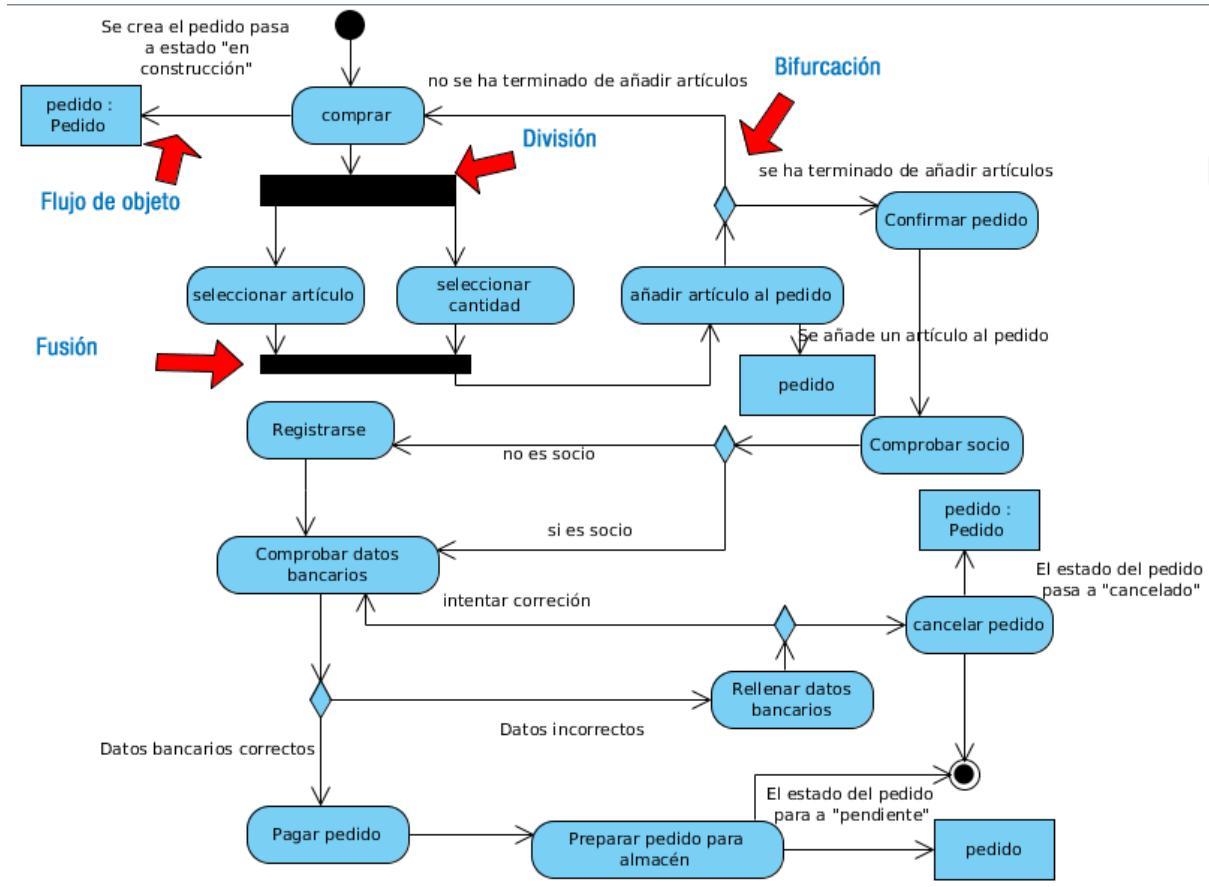
- **Preempleado.** Anterior a los 16 años. Se considera el estado inicial.
  - **Aprendiz.** Es el periodo comprendido entre los 16 y 18 años para aquellas personas que han decidido no continuar sus estudios.
  - **Activo.** El trabajador se encuentra en activo y con contrato en vigor.
  - **Parado.** El trabajador ha perdido el empleo, su tarea principal es la búsqueda de un nuevo trabajo.
  - **Prejubilado.** El trabajador solicita dejar de estar activo, pero no ha alcanzado la edad de 65 años. Desde el estado de parado no se considera la opción de solicitar la prejubilación.
  - **Jubilado.** El trabajador ha cumplido los 65 años y pasa a disfrutar de un merecido descanso. Se considera el estafo final.



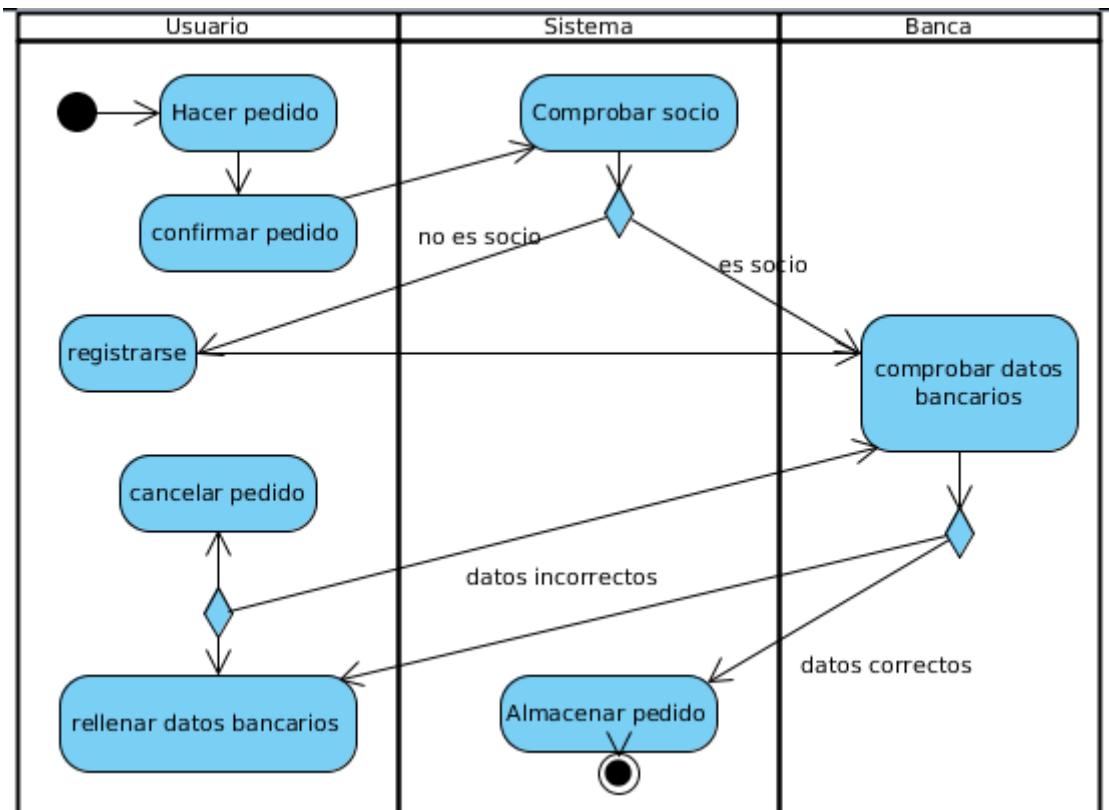
## Ejemplo de un diagrama de actividad

El siguiente diagrama de actividad representa el caso de uso "Generar pedido" del diagrama de casos de uso del ejercicio resuelto 1 "ZAPATERÍA TACÓN DE ORO" (en el punto 2 de "Los diagramas de casos de uso"), en el aparecen los elementos que hemos visto en las secciones anteriores.

- En las bifurcaciones se ha añadido la condición que indica si se pasa a una acción o a otra.
- Las acciones Seleccionar artículo y Seleccionar cantidad se han considerado concurrentes.



En este otro diagrama se simplifican las acciones a realizar y se eliminan los objetos para facilitar la inclusión de calles que indican quien realiza cada acción:



Nota: Para añadir las calles en Visual Paradigm se utiliza la herramienta del panel "Vertical Swimlane".