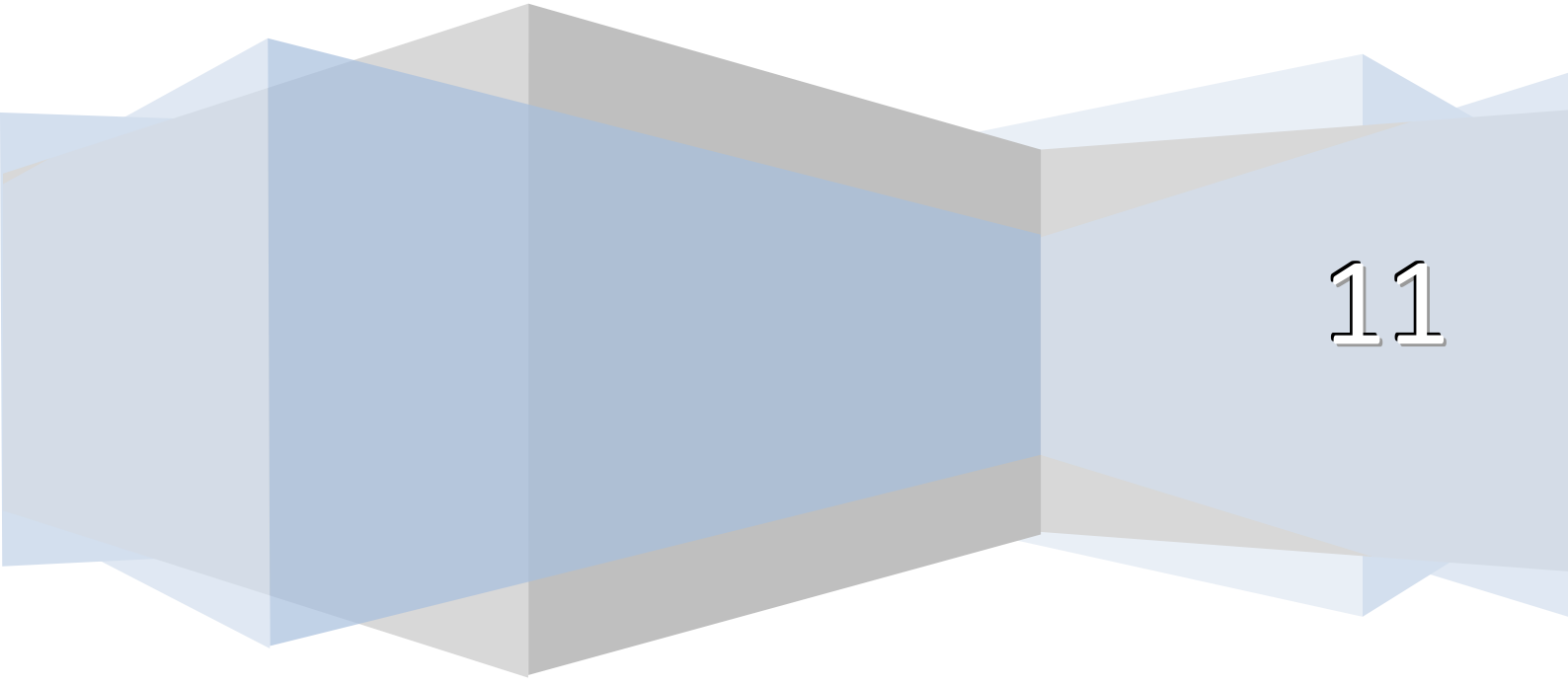
ENTORNOS DE DESARROLLO

Desarrollo de aplicaciones web

José Luis Comesaña



11

Desarrollo de software.

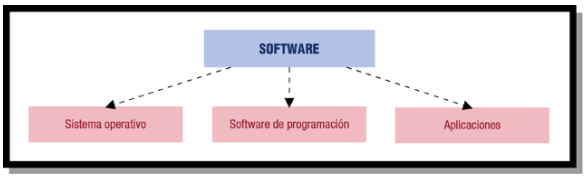
Caso práctico
En BK Programación todos han vuelto ya de sus vacaciones.
Les espera un septiembre agitado, pues acaban de recibir una petición por parte de una cadena hotelera para desarrollar un proyecto software.
Ada, la supervisora de proyectos de BK Programación, se reúne con Juan y María (trabajadores de la empresa) para empezar a planificar el proyecto.
Ana, cuya especialidad es el diseño gráfico de páginas web, acaba de terminar el Ciclo de Grado Medio en Sistemas Microinformáticos y Redes y realizó la FCT en BK Programación. Trabaja en la empresa ayudando en los diseños, y aunque está contenta con su trabajo, le gustaría participar activamente en todas las fases en el proyecto. El problema es que carece de los conocimientos necesarios.
Antonio se ha enterado de la posibilidad de estudiar el nuevo Ciclo de Grado Superior de Diseño de Aplicaciones Multiplataforma a distancia, y está dispuesta a hacerlo. (No tendría que dejar el trabajo).
Le comenta sus planes a su amigo Antonio (que tiene conocimientos básicos de informática), y éste se une a ella.
Después de todo… ¿qué pueden perder?

1.- Software y programa. Tipos de software.

Caso práctico
Todos en la empresa están entusiasmados con el proyecto que tienen entre manos. Saben que lo más importante es planificarlo todo de antemano y elegir el tipo de software más adecuado. Ana les escucha hablar y no llega a entender por qué hablan de "tipos de software". ¿Acaso el software no era la parte lógica del ordenador, sin más? ¿Cuáles son los tipos de software?

Es de sobra conocido que el ordenador se compone de dos partes bien diferenciadas: hardware y software.

El software es el conjunto de programas informáticos que actúan sobre el hardware para ejecutar lo que el usuario desee.



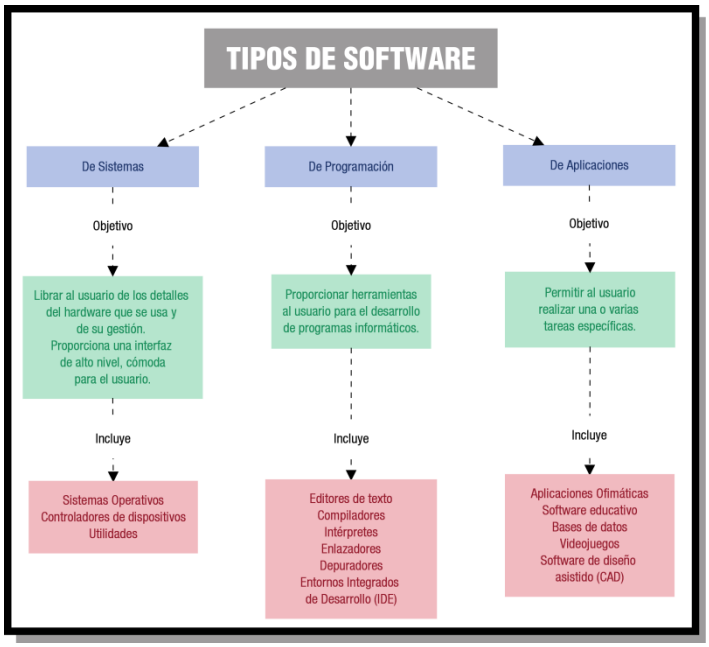
Según su función se distinguen **tres tipos de software**: sistema operativo, software de programación y aplicaciones.

El **sistema operativo** es el software base que ha de estar instalado y configurado en nuestro ordenador para que las aplicaciones puedan ejecutarse y funcionar. Son ejemplos de sistemas operativos: Windows, Linux, Mac OS X ...

El **software de programación** es el conjunto de herramientas que nos permiten desarrollar programas informáticos, y las **aplicaciones informáticas** son un conjunto de programas que tienen una finalidad más o menos concreta. Son ejemplos de aplicaciones: un procesador de textos, una hoja de cálculo, el software para reproducir música, un videojuego, etc.

A su vez, un programa es un conjunto de instrucciones escritas en un lenguaje de programación.

En definitiva, distinguimos los siguientes tipos de software:



En este tema, nuestro interés se centra en las aplicaciones informáticas: cómo se desarrollan y cuáles son las fases por las que necesariamente han de pasar.

A lo largo de esta primera unidad vas a aprender los conceptos fundamentales de software y las fases del llamado ciclo de vida de una aplicación informática.

También aprenderás a distinguir los diferentes lenguajes de programación y los procesos que ocurren hasta que el programa funciona y realiza la acción deseada.

Para saber más

En el siguiente enlace encontrarás más información de los tipos de software existente, así como ejemplos de cada uno que te ayudarán a profundizar sobre el tema.

http://www.tiposdesoftware.com/

Reflexiona

Hay varios sistemas operativos en el mercado: Linux, Windows, Mac OS X etc. El más conocido es Windows. A pesar de eso, ¿por qué utilizamos cada vez más Linux?

2.- Relación hardware-software.

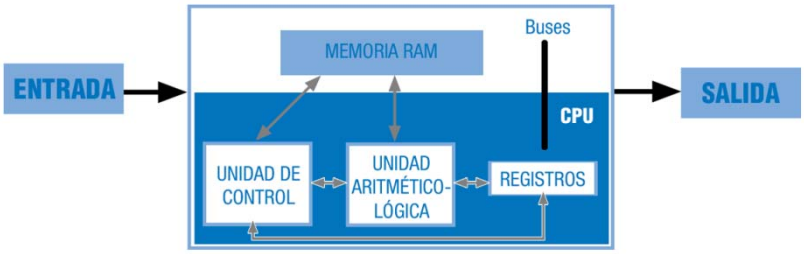
Caso práctico
Después de saber ya diferenciar los distintos tipos de software, Ana se le plantea otra cuestión: El software, sea del tipo que sea, se ejecuta sobre los dispositivos físicos del ordenador. ¿Qué relación hay entre ellos?

Como sabemos, al conjunto de dispositivos físicos que conforman un ordenador se le denomina hardware.

Existe una relación indisoluble entre éste y el software, ya que necesitan estar instalados y configurados correctamente para que el equipo funcione.

El software se ejecutará sobre los dispositivos físicos.

La primera arquitectura hardware con programa almacenado se estableció en 1946 por John Von Neumann:



Esta relación software-hardware la podemos poner de manifiesto desde dos puntos de vista:

a. **Desde el punto de vista del sistema operativo**

   El sistema operativo es el encargado de coordinar al hardware durante el funcionamiento del ordenador, actuando como intermediario entre éste y las aplicaciones que están corriendo en un momento dado.

   Todas las aplicaciones necesitan recursos hardware durante su ejecución (tiempo de CPU, espacio en memoria RAM, tratamiento de interrupciones, gestión de los dispositivos de Entrada/Salida, etc.). Será siempre el sistema operativo el encargado de controlar todos estos aspectos de manera "oculta" para las aplicaciones (y para el usuario).

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

   Ya hemos dicho que una aplicación no es otra cosa que un conjunto de programas, y que éstos están escritos en algún lenguaje de programación que el hardware del equipo debe interpretar y ejecutar.

   Hay multitud de lenguajes de programación diferentes (como ya veremos en su momento). Sin embargo, todos tienen algo en común: estar escritos con sentencias de un idioma que el ser humano puede aprender y usar fácilmente. Por otra parte, el hardware de un ordenador sólo es capaz de interpretar señales eléctricas (ausencias o presencias de tensión) que, en informática, se traducen en secuencias de 0 y 1 (código binario).

Esto nos hace plantearnos una cuestión: ¿Cómo será capaz el ordenador de "entender" algo escrito en un lenguaje que no es el suyo?

Como veremos a lo largo de esta unidad, tendrá que pasar algo (un proceso de traducción de código) para que el ordenador ejecute las instrucciones escritas en un lenguaje de programación.

Autoevaluación

Para fabricar un programa informático que se ejecuta en una computadora:

☐ Hay que escribir las instrucciones en código binario para que las entienda el hardware.

☐ Sólo es necesario escribir el programa en algún lenguaje de programación y se ejecuta directamente.

◉ Hay que escribir el programa en algún Lenguaje de Programación y contar con herramientas software que lo traduzcan a código binario.

☐ Los programas informáticos no se pueden escribir: forman parte de los sistemas operativos.

3.- Desarrollo de software.

Caso práctico
En BK programación ya están manos a la obra. Ada reúne a toda su plantilla para desarrollar el nuevo proyecto.
Ella sabe mejor que nadie que no será sencillo y que habrá que pasar por una serie de etapas. Ana no quiere perderse la reunión, quiere descubrir por qué hay que tomar tantas anotaciones y tantas molestias antes incluso de empezar.

Entendemos por Desarrollo de Software todo el proceso que ocurre desde que se concibe una idea hasta que un programa está implementado en el ordenador y funcionando.

El proceso de desarrollo, que en un principio puede parecer una tarea simple, consta de una serie de pasos de obligado cumplimiento, pues sólo así podremos garantizar que los programas creados son eficientes, fiables, seguros y responden a las necesidades de los usuarios finales (aquellos que van a utilizar el programa).

Como veremos con más detenimiento a lo largo de la unidad, el desarrollo de software es un proceso que conlleva una serie de pasos. Genéricamente, estos pasos son los siguientes:

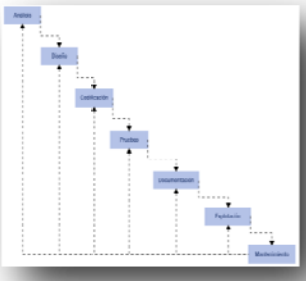
Etapas en el desarrollo de software:

Como vamos a ver en el siguiente punto, según el orden y la forma en que se lleven a cabo las etapas hablaremos de diferentes ciclos de vida del software.

La construcción de software es un proceso que puede llegar a ser muy complejo y que exige gran coordinación y disciplina del grupo de trabajo que lo desarrolle.

Reflexiona
Según estimaciones, el 26% de los grandes proyectos de software fracasan, el 48% deben modificarse drásticamente y sólo el 26% tienen rotundo éxito. La principal causa del fracaso de un proyecto es la falta de una buena planificación de las etapas y mala gestión de los pasos a seguir. ¿Por qué el porcentaje de fracaso es tan grande? ¿Por qué piensas que estas causas son tan determinantes?

3.1.- Ciclos de vida del software.



Ya hemos visto que la serie de pasos a seguir para desarrollar un programa es lo que se conoce como Ciclo de Vida del Software.

Cada etapa vendrá explicada con más detalle en el punto de la presente unidad dedicado a las fases del desarrollo y ejecución del software.

Diversos autores han planteado distintos modelos de ciclos de vida, pero los más conocidos y utilizados son los que aparecen a continuación:

Siempre se debe aplicar un modelo de ciclo de vida al desarrollo de cualquier proyecto software.

1. **Modelo en Cascada**

   Es el modelo de vida clásico del software.

   Es prácticamente imposible que se pueda utilizar, ya que requiere conocer de antemano todos los requisitos del sistema. Sólo es aplicable a pequeños desarrollos, ya que las etapas pasan de una a otra sin retorno posible. (se presupone que no habrá errores ni variaciones del software).

2. **Modelo en Cascada con Realimentación**

   Es uno de los modelos más utilizados. Proviene del modelo anterior, pero se introduce una realimentación entre etapas, de forma que podamos volver atrás en cualquier momento para corregir, modificar o depurar algún aspecto. No obstante, si se prevén muchos cambios durante el desarrollo no es el modelo más idóneo.

   Es el modelo perfecto si el proyecto es rígido (pocos cambios, poco evolutivo) y los requisitos están claros.

3. **Modelos Evolutivos**

   Son más modernos que los anteriores. Tienen en cuenta la naturaleza cambiante y evolutiva del software.

   Distinguimos dos variantes:

   1. **Modelo Iterativo Incremental**

      Está basado en el modelo en cascada con realimentación, donde las fases se repiten y refinan, y van propagando su mejora a las fases siguientes.

   2. **Modelo en Espiral**

      Es una combinación del modelo anterior con el modelo en cascada. En él, el software se va construyendo repetidamente en forma de versiones que son cada vez mejores, debido a que incrementan la funcionalidad en cada versión. Es un modelo bastante complejo.

Autoevaluación
Si queremos construir una aplicación pequeña, y se prevé que no sufrirá grandes cambios durante su vida, ¿sería el modelo de ciclo de vida en espiral el más recomendable?

- ☐ Sí.
- ◉ No.

3.2.- Herramientas de apoyo al desarrollo del software.

En la práctica, para llevar a cabo varias de las etapas vistas en el punto anterior contamos con herramientas informáticas, cuya finalidad principal es automatizar las tareas y ganar fiabilidad y tiempo.

Esto nos va a permitir centrarnos en los requerimientos del sistema y el análisis del mismo, que son las causas principales de los fallos del software.

Las herramientas **CASE** son un conjunto de aplicaciones que se utilizan en el desarrollo de software con el objetivo de reducir costes y tiempo del proceso, mejorando por tanto la productividad del proceso.

¿En qué fases del proceso nos pueden ayudar?

En el diseño del proyecto, en la codificación de nuestro diseño a partir de su apariencia visual, detección de errores…

El desarrollo rápido de aplicaciones o **RAD** es un proceso de desarrollo de software que comprende el desarrollo iterativo, la construcción de prototipos y el uso de utilidades CASE. Hoy en día se suele utilizar para referirnos al desarrollo rápido de interfaces gráficas de usuario o entornos de desarrollo integrado completos.

La tecnología CASE trata de automatizar las fases del desarrollo de software para que mejore la calidad del proceso y del resultado final.

En concreto, estas herramientas permiten:

- Mejorar la planificación del proyecto.
- Darle agilidad al proceso.
- Poder reutilizar partes del software en proyectos futuros.
- Hacer que las aplicaciones respondan a estándares.
- Mejorar la tarea del mantenimiento de los programas.
- Mejorar el proceso de desarrollo, al permitir visualizar las fases de forma gráfica.

CLASIFICACIÓN

Normalmente, las herramientas CASE se clasifican en función de las fases del ciclo de vida del software en la que ofrecen ayuda:

- **U-CASE**: ofrece ayuda en las fases de planificación y análisis de requisitos.
- **M-CASE**: ofrece ayuda en análisis y diseño.
- **L-CASE**: ayuda en la programación del software, detección de errores del código, depuración de programas y pruebas y en la generación de la documentación del proyecto.

Ejemplos de herramientas CASE libres son: ArgoUML, Use Case Maker, ObjectBuilder…

Para saber más

En el siguiente enlace se presenta una ampliación de los tipos y ayudas concretas de la herramientas CASE.

http://temariotic.wikidot.com/tema-58-boe-13-02-1996

4.- Lenguajes de programación.

Caso práctico
Una de los aspectos del proyecto que más preocupa a Ana es la elección del lenguaje de programación a utilizar.
Necesita tener muy claros los requerimientos del cliente para enfocar correctamente la elección, pues según sean éstos unos lenguajes serán más efectivos que otros.

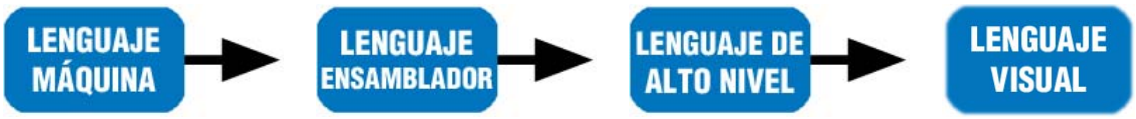
Ya dijimos anteriormente que los programas informáticos están escritos usando algún lenguaje de programación. Por tanto, podemos definir un Lenguaje de Programación como un idioma creado de forma artificial, formado por un conjunto de símbolos y normas que se aplican sobre un alfabeto para obtener un código, que el hardware de la computadora pueda entender y ejecutar.

Los lenguajes de programación son los que nos permiten comunicarnos con el hardware del ordenador.

En otras palabras, es muy importante tener muy clara la función de los lenguajes de programación. Son los instrumentos que tenemos para que el ordenador realice las tareas que necesitamos.

Hay multitud de lenguajes de programación, cada uno con unos símbolos y unas estructuras diferentes. Además, cada lenguaje está enfocado a la programación de tareas o áreas determinadas. Por ello, la elección del lenguaje a utilizar en un proyecto es una cuestión de extrema importancia.

Los lenguajes de programación han sufrido su propia evolución, como se puede apreciar en la figura siguiente:



Características de los Lenguajes de Programación

- **Lenguaje máquina**:
  - Sus instrucciones son combinaciones de unos y ceros.
  - Es el único lenguaje que entiende directamente el ordenador. (No necesita traducción).
  - Fue el primer lenguaje utilizado.
  - Es único para cada procesador (no es portable de un equipo a otro).
  - Hoy día nadie programa en este lenguaje.
- **Lenguaje ensamblador**:
  - Sustituyó al lenguaje máquina para facilitar la labor de programación.
  - En lugar de unos y ceros se programa usando mnemotécnicos (instrucciones complejas).
  - Necesita traducción al lenguaje máquina para poder ejecutarse.
  - Sus instrucciones son sentencias que hacen referencia ala ubicación física de los archivos en el equipo.
  - Es difícil de utilizar.

- **Lenguaje de alto nivel basados en código**:
  - Sustituyeron al lenguaje ensamblador para facilitar más la labor de programación.
  - En lugar de mnemotécnicos, se utilizan sentencias y órdenes derivadas del idioma inglés. (Necesita traducción al lenguaje máquina).
  - Son más cercanos al razonamiento humano.
  - Son utilizados hoy día, aunque la tendencia es que cada vez menos.
- **Lenguajes visuales**:
  - Están sustituyendo a los lenguajes de alto nivel basados en código.
  - En lugar de sentencias escritas, se programa gráficamente usando el ratón y diseñando directamente la apariencia del software.
  - Su correspondiente código se genera automáticamente.
  - Necesitan traducción al lenguaje máquina.
  - Son completamente portables de un equipo a otro.

Para saber más

En el siguiente enlace, verás la evolución entre los distintos tipos de Lenguajes de Programación en la historia.

http://www.monografias.com/trabajos38/tipos-lenguajes-programacion/tipos-lenguajes-programacion.shtml

4.1.- Concepto y características.

Ya sabemos que los lenguajes de programación han evolucionado, y siguen haciéndolo, siempre hacia la mayor usabilidad de los mismos (que el mayor número posible de usuarios lo utilicen y exploten).

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

CONCEPTO

Un lenguaje de programación es el conjunto de:

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

CARACTERÍSTICAS

Podemos clasificar los distintos tipos de Lenguajes de Programación en base a distintas características:

- **Según lo cerca que esté del lenguaje humano**

  - Lenguajes de Programación De alto nivel: por su esencia, están más próximos al razonamiento humano.
  - Lenguajes de Programación De bajo nivel: están más próximos al funcionamiento interno de la computadora:
    - Lenguaje Ensamblador.
    - Lenguaje Máquina.
- **Según la técnica de programación utilizad**a:
  - Lenguajes de Programación Estructurados: Usan la técnica de programación estructurada. Ejemplos: Pascal, C, etc.
  - Lenguajes de Programación Orientados a Objetos: Usan la técnica de programación orientada a objetos. Ejemplos: C++, Java, Ada, Delphi, etc.
  - Lenguajes de Programación Visuales: Basados en las técnicas anteriores, permiten programar gráficamente, siendo el código correspondiente generado de forma automática. Ejemplos: Visual Basic.Net, Borland Delphi, etc.

A pesar de la inmensa cantidad de lenguajes de programación existentes, Java, C, C++, PHP y Visual Basic concentran alrededor del 60% del interés de la comunidad informática mundial.

Para saber más
En la página web siguiente encontrarás un resumen de las características de los Lenguajes de Programación más utilizados en la actualidad.
http://www.larevistainformatica.com/LENGUAJES-DE-PROGRAMACION-listado.html

4.2.- Lenguajes de programación estructurados.

Aunque los requerimientos actuales de software son bastante más complejos de lo que la técnica de programación estructurada es capaz, es necesario por lo menos conocer las bases de los Lenguajes de Programación estructurados, ya que a partir de ellos se evolucionó hasta otros lenguajes y técnicas más completas (orientada a eventos u objetos) que son las que se usan actualmente.

La programación estructurada se define como una técnica para escribir lenguajes de programación que permite sólo el uso de tres tipos de sentencias o estructuras de control:

- Sentencias secuenciales.
- Sentencias selectivas (condicionales).
- Sentencias repetitivas (iteraciones o bucles).

Los lenguajes de programación que se basan en la programación estructurada reciben el nombre de lenguajes de programación estructurados.

Para saber más
En el siguiente enlace encontrarás un breve documento donde se explica para qué sirve cada sentencia de control con unos sencillos ejemplos escritos usando el lenguaje C.
http://www.juntadeandalucia.es/educacion/adistancia/cursos/file.php/420/ED01/ED01_Web/index.html#anexo_i_sentencias_de_control_de_la_programacin_estructurada.html

La programación estructurada fue de gran éxito por su sencillez a la hora de construir y leer programas. Fue sustituida por la programación modular, que permitía dividir los programas grandes en trozos más pequeños (siguiendo la conocida técnica "divide y vencerás"). A su vez, luego triunfaron los lenguajes orientados a objetos y de ahí a la programación visual (siempre es más sencillo programar gráficamente que en código, ¿no crees?).

VENTAJAS DE LA PROGRAMACIÓN ESTRUCTURADA

- Los programas son fáciles de leer, sencillos y rápidos.
- El mantenimiento de los programas es sencillo.
- La estructura del programa es sencilla y clara.

INCONVENIENTES

- Todo el programa se concentra en un único bloque (si se hace demasiado grande es difícil manejarlo).
- No permite reutilización eficaz de código, ya que todo va "en uno". Es por esto que a la programación estructurada le sustituyó la programación modular, donde los programas se codifican por módulos y bloques, permitiendo mayor funcionalidad.

Ejemplos de lenguajes estructurados: *Pascal, C, Fortran*.

La Programación estructurada evolucionó hacia la Programación modular, que divide el programa en trozos de código llamados módulos con una funcionalidad concreta, que podrán ser reutilizables.

4.3.- Lenguajes de programación orientados a objetos.

Después de comprender que la programación estructurada no es útil cuando los programas se hacen muy largos, es necesaria otra técnica de programación que solucione este inconveniente. Nace así la Programación Orientada a Objetos (en adelante, P.O.O.).

Los lenguajes de programación orientados a objetos tratan a los programas no como un conjunto ordenado de instrucciones (tal como sucedía en la programación estructurada) sino como un conjunto de objetos que colaboran entre ellos para realizar acciones.

En la P.O.O. los programas se componen de objetos independientes entre sí que colaboran para realizar acciones.

Los objetos son reutilizables para proyectos futuros.

Su primera desventaja es clara: no es una programación tan intuitiva como la estructurada.

A pesar de eso, alrededor del 55% del software que producen las empresas se hace usando esta técnica.

Razones:

- El código es reutilizable.
- Si hay algún error, es más fácil de localizar y depurar en un objeto que en un programa entero.

Características:

- Los objetos del programa tendrán una serie de atributos que los diferencian unos de otros.
- Se define clase como una colección de objetos con características similares.
- Mediante los llamados métodos, los objetos se comunican con otros produciéndose un cambio de estado de los mismos.
- Los objetos son, pues, como unidades individuales e indivisibles que forman la base de este tipo de programación.

Principales lenguajes orientados a objetos: *Ada, C++, VB.NET, Delphi, Java, PowerBuilder, etc.*

Para saber más

En el siguiente enlace hay un documento muy interesante de introducción a la programación orientada a objetos, en concreto, del lenguaje C++.

http://mat21.etsii.upm.es/ayudainf/aprendainf/Cpp/manualcpp.pdf

5.- Fases en el desarrollo y ejecución del software.

Caso práctico
En la reunión de BK acerca del nuevo proyecto Ada, la supervisora, dejó bien claro que lo primero y más importante es tener claro qué queremos que haga el software y con qué herramientas contamos: lo demás vendría después, ya que si esto no está bien planteado, ese error se propagará a todas las fases del proyecto.
—¿Por dónde empezamos? —pregunta Juan.
—ANÁLISIS de REQUISITOS —contesta Ada.

Ya hemos visto en puntos anteriores que debemos elegir un modelo de ciclo de vida para el desarrollo de nuestro software.

Independientemente del modelo elegido, siempre hay una serie de etapas que debemos seguir para construir software fiable y de calidad.

Estas etapas son:

1. **ANÁLISIS DE REQUISITOS.**

   Se especifican los requisitos funcionales y no funcionales del sistema.

2. **DISEÑO.**

   Se divide el sistema en partes y se determina la función de cada una.

3. **CODIFICACIÓN.**

   Se elige un Lenguajes de Programación y se codifican los programas.

4. **PRUEBAS.**

   Se prueban los programas para detectar errores y se depuran.

5. **DOCUMENTACIÓN.**

   De todas las etapas, se documenta y guarda toda la información.

6. **EXPLOTACIÓN.**

   Instalamos, configuramos y probamos la aplicación en los equipos del cliente.

7. **MANTENIMIENTO.**

   Se mantiene el contacto con el cliente para actualizar y modificar la aplicación el futuro.

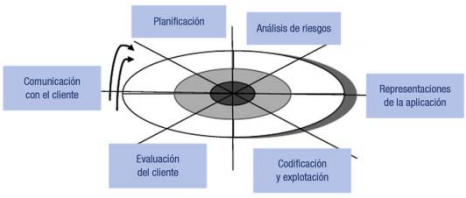
Autoevaluación

¿Crees que debemos esperar a tener completamente cerrada una etapa para pasar a la siguiente?

Sí.

No.

5.1.- Análisis.



Esta es la primera fase del proyecto. Una vez finalizada, pasamos a la siguiente (diseño).

Es la fase de mayor importancia en el desarrollo del proyecto y todo lo demás dependerá de lo bien detallada que esté. También es la más complicada, ya que no está automatizada y depende en gran medida del analista que la realice.

Es la primera etapa del proyecto, la más complicada y la que más depende de la capacidad del analista.

¿Qué se hace en esta fase?

Se especifican y analizan los requisitos funcionales y no funcionales del sistema.

Requisitos:

- **Funcionales**: Qué funciones tendrá que realizar la aplicación. Qué respuesta dará la aplicación ante todas las entradas. Cómo se comportará la aplicación en situaciones inesperadas.
- **No funcionales**: Tiempos de respuesta del programa, legislación aplicable, tratamiento ante la simultaneidad de peticiones, etc.

Lo fundamental es la buena comunicación entre el analista y el cliente para que la aplicación que se va a desarrollar cumpla con sus expectativas.

La culminación de esta fase es el documento ERS (Especificación de Requisitos Software).

En este documento quedan especificados:

- La planificación de las reuniones que van a tener lugar.
- Relación de los objetivos del usuario cliente y del sistema.
- Relación de los requisitos funcionales y no funcionales del sistema.
- Relación de objetivos prioritarios y temporización.
- Reconocimiento de requisitos mal planteados o que conllevan contradicciones, etc.

Citas para pensar

Todo aquello que no se detecte, o resulte mal entendido en la etapa inicial provocará un fuerte impacto negativo en los requisitos, propagando esta corriente degradante a lo largo de todo el proceso de desarrollo e **incrementando su perjuicio cuanto más tardía sea su detección***(Bell y Thayer 1976)(Davis 1993).*

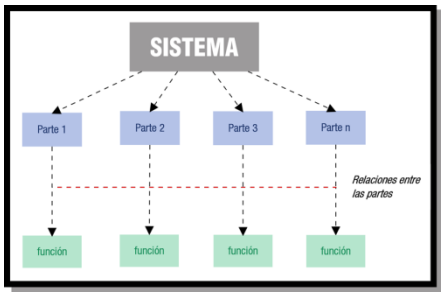
Como ejemplo de requisitos funcionales, en la aplicación para nuestros clientes de las tiendas de cosmética, habría que considerar:

- Si desean que la lectura de los productos se realice mediante códigos de barras.
- Si van a detallar las facturas de compra y de qué manera la desean.
- Si los trabajadores de las tiendas trabajan a comisión, tener información de las ventas de cada uno.
- Si van a operar con tarjetas de crédito.
- Si desean un control del stock en almacén.
- Etc.

5.2.- Diseño.

Caso práctico
Juan está agobiado por el proyecto. Ya han mantenido comunicaciones con el cliente y saben perfectamente qué debe hacer la aplicación. También tiene una lista de las características hardware de los equipos de su cliente y todos los requisitos. Tiene tanta información que no sabe por dónde empezar.
Decide hablar con Ada. Su supervisora, amable como siempre, le sugiere que empiece a dividir el problema en las partes implicadas.
—Vale, Ada, pero, ¿cómo lo divido?

Durante esta fase, donde ya sabemos lo que hay que hacer, el siguiente paso es ¿Cómo hacerlo?



Se debe dividir el sistema en partes y establecer qué relaciones habrá entre ellas.

Decidir qué hará exactamente cada parte.

En definitiva, debemos crear un modelo funcional-estructural de los requerimientos del sistema global, para poder dividirlo y afrontar las partes por separado.

En este punto, se deben tomar decisiones importantes, tales como:

- Entidades y relaciones de las bases de datos.
- Selección del lenguaje de programación que se va a utilizar.
- Selección del Sistema Gestor de Base de Datos.
- Etc.

Citas para pensar
Design is not just what it looks like and feels like. Designishowitworks.Steve Jobs ("*El diseño no es sólo lo que parece y cómo parece. Diseño es cómo se trabaja*").

Reflexiona
Según estimaciones, las organizaciones y empresas que crecen más son las que más dinero invierten en sus diseños.

5.3.- Codificación. Tipos de código.

Caso práctico
En BK, ya tienen el proyecto dividido en partes.
Ahora llega una parte clave: codificar los pasos y acciones a seguir para que el ordenador los ejecute. En otras palabras, programar la aplicación. Saben que no será fácil, pero afortunadamente cuentan con herramientas CASE que les van a ser de gran ayuda. A Ana el gustaría participar, pero cuando se habla de "código fuente", "ejecutable", etc. sabe que no tiene ni idea y que no tendrá más remedio que estudiarlo si quiere colaborar en esta fase del proyecto.

Durante la fase de codificación se realiza el proceso de programación.

Consiste en elegir un determinado lenguaje de programación, codificar toda la información anterior y llevarlo a código fuente.

Esta tarea la realiza el programador y tiene que cumplir exhaustivamente con todos los datos impuestos en el análisis y en el diseño de la aplicación.

Las características deseables de todo código son:

1. Modularidad: que esté dividido en trozos más pequeños.
2. Corrección: que haga lo que se le pide realmente.
3. Fácil de leer: para facilitar su desarrollo y mantenimiento futuro.
4. Eficiencia: que haga un buen uso de los recursos.
5. Portabilidad: que se pueda implementar en cualquier equipo.

Durante esta fase, el código pasa por diferentes estados:

- **Código Fuente**: es el escrito por los programadores en algún editor de texto. Se escribe usando algún lenguaje de programación de alto nivel y contiene el conjunto de instrucciones necesarias.
- **Código Objeto**: es el código binario resultado de compilar el código fuente.

  La compilación es la traducción de una sola vez del programa, y se realiza utilizando un compilador. La interpretación es la traducción y ejecución simultánea del programa línea a línea.

  El código objeto no es directamente inteligible por el ser humano, pero tampoco por la computadora. Es un código intermedio entre el código fuente y el ejecutable y sólo existe si el programa se compila, ya que si se interpreta (traducción línea a línea del código) se traduce y se ejecuta en un solo paso.

- **Código Ejecutable**: Es el código binario resultante de enlazar los archivos de código objeto con ciertas rutinas y bibliotecas necesarias. El sistema operativo será el encargado de cargar el código ejecutable en memoria RAM y proceder a ejecutarlo. También es conocido como código máquina y ya sí es directamente inteligible por la computadora.

Los programas interpretados no producen código objeto. El paso de fuente a ejecutable es directo.

5.4.- Fases en la obtención de código.

Caso práctico
Juan y María ya han decidido el Lenguajes de Programación que van a utilizar.
Saben que el programa que realicen pasará por varias fases antes de ser implementado en los equipos del cliente. Todas esas fases van a producir transformaciones en el código. ¿Qué características irá adoptando el código a medida que avanza por el proceso de codificación?

5.4.1.- Fuente.

El código fuente es el conjunto de instrucciones que la computadora deberá realizar, escritas por los programadores en algún lenguaje de alto nivel.

Este conjunto de instrucciones no es directamente ejecutable por la máquina, sino que deberá ser traducido al lenguaje máquina, que la computadora será capaz de entender y ejecutar.

Un aspecto muy importante en esta fase es la elaboración previa de un algoritmo, que lo definimos como un conjunto de pasos a seguir para obtener la solución del problema. El algoritmo lo diseñamos en pseudocódigo y con él, la codificación posterior a algún Lenguaje de Programación determinado será más rápida y directa.

Para obtener el código fuente de una aplicación informática:

1. Se debe partir de las etapas anteriores de análisis y diseño.
2. Se diseñará un algoritmo que simbolice los pasos a seguir para la resolución del problema.
3. Se elegirá una Lenguajes de Programación de alto nivel apropiado para las características del software que se quiere codificar.
4. Se procederá a la codificación del algoritmo antes diseñado.

La culminación de la obtención de código fuente es un documento con la codificación de todos los módulos (*Cada parte, con una funcionalidad concreta, en que se divide una aplicación*), funciones (*Parte de código muy pequeña con una finalidad muy concreta.*), bibliotecas y procedimientos (*Igual que las función, pero al ejecutarse no devuelven ningún valor.*) necesarios para codificar la aplicación.

Puesto que, como hemos dicho antes, este código no es inteligible por la máquina, habrá que TRADUCIRLO, obteniendo así un código equivalente pero ya traducido a código binario que se llama código objeto. Que no será directamente ejecutable por la computadora si éste ha sido compilado.

Un aspecto importante a tener en cuenta es su licencia. Así, en base a ella, podemos distinguir dos tipos de código fuente:

- Código fuente abierto. Es aquel que está disponible para que cualquier usuario pueda estudiarlo, modificarlo o reutilizarlo.
- Código fuente cerrado. Es aquel que no tenemos permiso para editarlo.

Autoevaluación
Para obtener código fuente a partir de toda la información necesaria del problema:

☐ Se elige el Lenguaje de Programación más adecuado y se codifica directamente.

☐ Se codifica y después se elige el Lenguaje de Programación más adecuado.

Se elige el Lenguaje de Programación más adecuado, se diseña un algoritmo y se codifica.
Muy bien. El diseño del algoritmo (los pasos a seguir) nos ayudará a que la codificación posterior se realice más rápidamente y tenga menos errores.

5.4.2.- Objeto.

El código objeto es un código intermedio.

Es el resultado de traducir código fuente a un código equivalente formado por unos y ceros que aún no puede ser ejecutado directamente por la computadora.

Es decir, es el código resultante de la compilación del código fuente.

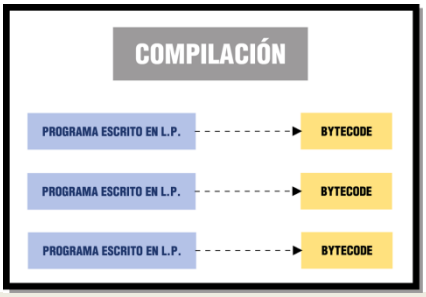
Consiste en un bytecode (*Código binario resultante de la traducción de código de alto nivel que aún no puede ser ejecutado.*) que está distribuido en varios archivos, cada uno de los cuales corresponde a cada programa fuente compilado.

Sólo se genera código objeto una vez que el código fuente está libre de errores sintácticos y semánticos.

El proceso de traducción de código fuente a código objeto puede realizarse de dos formas:

a. **Compilación**: El proceso de traducción se realiza sobre todo el código fuente, en un solo paso. Se crea código objeto que habrá que enlazar. El software responsable se llama compilador (*Software que traduce, de una sola vez, un programa escrito en un lenguaje de programación de alto nivel en su equivalente en lenguaje máquina*.).
b. **Interpretación**: El proceso de traducción del código fuente se realiza línea a línea y se ejecuta simultáneamente. No existe código objeto intermedio. El software responsable se llama intérprete (*Software que traduce, instrucción a instrucción, un programa escrito en un lenguaje de alto nivel en su equivalente en lenguaje máquina*). El proceso de traducción es más lento que en el caso de la compilación, pero es recomendable cuando el programador es inexperto, ya que da la detección de errores es más detallada.

El código objeto es código binario, pero no puede ser ejecutado por la computadora



Para saber más
En el siguiente enlace podrás visitar una página web, que te permitirá aprender más acerca de la generación de códigos objeto:
http://www.monografias.com/trabajos11/compil/compil2.shtml#co

5.4.3.- Ejecutable.

El código ejecutable, resultado de enlazar los archivos de código objeto, consta de un único archivo que puede ser directamente ejecutado por la computadora. No necesita ninguna aplicación externa. Este archivo es ejecutado y controlado por el sistema operativo.

Para obtener un sólo archivo ejecutable, habrá que enlazar todos los archivos de código objeto, a través de un software llamado linker (*Enlazador. Pequeño software encargado de unir archivos para generar un programa ejecutable.*) y obtener así un único archivo que ya sí es ejecutable directamente por la computadora.

Para saber más

En el siguiente enlace podrás visitar una página web, que te permitirá aprender más acerca de la generación de ejecutables:

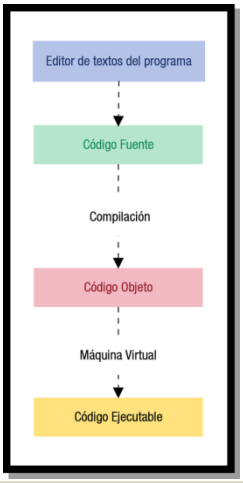
http://www.palomatica.info/juckar/sistemas/software/generacion_de_ejecutable.html

En el esquema de generación de código ejecutable, vemos el proceso completo para la generación de ejecutables.

A partir de un editor, escribimos el lenguaje fuente con algún Lenguaje de programación. (En el ejemplo, se usa Java).

A continuación, el código fuente se compila obteniendo código objeto o bytecode.

Ese bytecode, a través de la máquina virtual (se verá en el siguiente punto), pasa a código máquina, ya directamente ejecutable por la computadora.



Autoevaluación

Relaciona los tipos de código con su característica más relevante, escribiendo el número asociado a la característica en el hueco correspondiente.

Ejercicio de relacionar

Tipo de código.	Relación.	Características.
Código Fuente	2	1. Escrito en Lenguaje Máquina pero no ejecutable.
Código Objeto	1	2. Escrito en algún Lenguaje de Programación de alto nivel, pero no ejecutable.
Código Ejecutable	3	3. Escrito en Lenguaje Máquina y directamente ejecutable.

El código fuente escrito en algún lenguaje de programación de alto nivel, el objeto escrito en lenguaje máquina sin ser ejecutable y el código ejecutable, escrito también en lenguaje máquina y ya sí ejecutable por el ordenador, son las distintas fases por donde pasan nuestros programas.

5.5.- Máquinas virtuales.

Una máquina virtual es un tipo especial de software cuya misión es separar el funcionamiento del ordenador de los componentes hardware instalados.

Esta capa de software desempeña un papel muy importante en el funcionamiento de los lenguajes de programación, tanto compilado como interpretado.

Con el uso de máquinas virtuales podremos desarrollar y ejecutar una aplicación sobre cualquier equipo, independientemente de las características concretas de los componentes físicos instalados. Esto garantiza la portabilidad (*Capacidad de un programa para ser ejecutado en cualquier arquitectura física de un equipo.*) de las aplicaciones.

Las funciones principales de una máquina virtual son las siguientes:

- Conseguir que las aplicaciones sean portables.
- Reservar memoria para los objetos que se crean y liberar la memoria no utilizada.
- Comunicarse con el sistema donde se instala la aplicación (huésped), para el control de los dispositivos hardware implicados en los procesos.
- Cumplimiento de las normas de seguridad de las aplicaciones.

CARACTERÍSTICAS DE LA MÁQUINA VIRTUAL

Cuando el código fuente se compila se obtiene código objeto (bytecode, código intermedio).

Para ejecutarlo en cualquier máquina se requiere tener independencia respecto al hardware concreto que se vaya a utilizar.

Para ello, la máquina virtual aísla la aplicación de los detalles físicos del equipo en cuestión.

Funciona como una capa de software de bajo nivel y actúa como puente entre el bytecode de la aplicación y los dispositivos físicos del sistema.

La Máquina Virtual verifica todo el bytecode antes de ejecutarlo.

La Máquina Virtual protege direcciones de memoria.

La máquina virtual actúa de puente entre la aplicación y el hardware concreto del equipo donde se instale.

Para saber más
En el siguiente enlace te presentamos el proceso de instalación de la JVM (Máquina Virtual de Java) y su apariencia.
http://www.unabvirtual.edu.co/ayuda/manuales_pdf/maquinavirtualjava.pdf

5.5.1.- Frameworks.

Un framework (*Plataforma, entorno, marco de trabajo del desarrollo rápido de aplicaciones*) es una estructura de ayuda al programador, en base a la cual podemos desarrollar proyectos sin partir desde cero.

Se trata de una plataforma software donde están definidos programas soporte, bibliotecas, lenguaje interpretado, etc., que ayuda a desarrollar y unir los diferentes módulos o partes de un proyecto.

Con el uso de framework podemos pasar más tiempo analizando los requerimientos del sistema y las especificaciones técnicas de nuestra aplicación, ya que la tarea laboriosa de los detalles de programación queda resuelta.

- Ventajas de utilizar un framework:
  - **Desarrollo rápido** de software.
  - **Reutilización** de partes de código para otras aplicaciones.
  - **Diseño** uniforme del software.
  - **Portabilidad** de aplicaciones de un computador a otro, ya que los bytecodes que se generan a partir del lenguaje fuente podrán ser ejecutados sobre cualquier máquina virtual.
- Inconvenientes:
  - Gran dependencia del código respecto al framework utilizado (sin cambiamos de framework, habrá que reescribir gran parte de la aplicación).
  - La instalación e implementación del framework en nuestro equipo consume bastantes recursos del sistema.

Para saber más

El uso creciente de frameworks hace que tengamos que estar reciclándonos constantemente. En el siguiente enlace, hay un documento muy interesante de sus principales características, ventajas y formas de uso:

http://www.maestrosdelweb.com/editorial/los-frameworks-de-php-agilizan-tu-trabajo/

Ejemplos de Frameworks:

- **.NET** es un framework para desarrollar aplicaciones sobre Windows. Ofrece el "Visual Studio .net" que nos da facilidades para construir aplicaciones y su motor es el ".Net framework" que permite ejecutar dichas aplicaciones. Es un componente que se instala sobre el sistema operativo.
- Spring de Java. Son conjuntos de bibliotecas (API's) para el desarrollo y ejecución de aplicaciones.

Debes conocer

El proceso de instalación y configuración del framework Spring de Java, así como varios ejemplos de uso. En el siguiente enlace encontrarás una guía muy útil detallando los pasos a seguir:

http://pablolg.wikispaces.com/file/view/spring_tutorial_v0.271.pdf

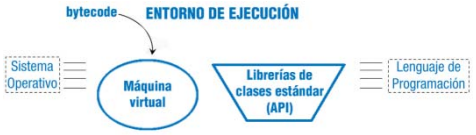
5.5.2.- Entornos de ejecución.

Un entorno de ejecución es un servicio de máquina virtual que sirve como base software para la ejecución de programas. En ocasiones pertenece al propio sistema operativo, pero también se puede instalar como software independiente que funcionará por debajo de la aplicación.

Es decir, es un conjunto de utilidades que permiten la ejecución de programas.

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

Durante la ejecución, los entornos se encargarán de:



- Configurar la memoria principal disponible en el sistema.
- Enlazar los archivos del programa con las bibliotecas existentes y con los subprogramas creados. Considerando que las bibliotecas son el conjunto de subprogramas que sirven para desarrollar o comunicar componentes software pero que ya existen previamente y los subprogramas serán aquellos que hemos creado a propósito para el programa.
- Depurar los programas: comprobar la existencia (o no existencia) de errores semánticos del lenguaje (los sintácticos ya se detectaron en la compilación).

Funcionamiento del entorno de ejecución:

El Entorno de Ejecución está formado por la máquina virtual y los API's (bibliotecas de clases estándar, necesarias para que la aplicación, escrita en algún Lenguaje de Programación pueda ser ejecutada). Estos dos componentes se suelen distribuir conjuntamente, porque necesitan ser compatibles entre sí.

El entorno funciona como intermediario entre el lenguaje fuente y el sistema operativo, y consigue ejecutar aplicaciones.

Sin embargo, si lo que queremos es desarrollar nuevas aplicaciones, no es suficiente con el entorno de ejecución.

Adelantándonos a lo que veremos en la próxima unidad, para desarrollar aplicaciones necesitamos algo más. Ese "algo más" se llama entorno de desarrollo.

Autoevaluación
Señala la afirmación falsa respecto de los entornos de ejecución:

- (•) Su principal utilidad es la de permitir el desarrollo rápido de aplicaciones.
- ( ) Actúa como mediador entre el sistema operativo y el código fuente.
- ( ) Es el conjunto de la máquina virtual y bibliotecas necesarias para la ejecución.

5.5.3.- Java runtimeenvironment.

En esta sección de va a explicar el funcionamiento, instalación, configuración y primeros pasos del RuntimeEnvironment del lenguaje Java (se hace extensible a los demás lenguajes de programación).

Concepto.

Se denomina JRE al Java RuntimeEnvironment (entorno en tiempo de ejecución Java).

El JRE se compone de un conjunto de utilidades que permitirá la ejecución de programas java sobre cualquier tipo de plataforma.

Componentes.

JRE está formado por:

- Una Máquina virtual Java (JMV o JVM si consideramos las siglas en inglés), que es el programa que interpreta el código de la aplicación escrito en Java.
- Bibliotecas de clase estándar que implementan el API de Java.
- Las dos: JMV y API de Java son consistentes entre sí, por ello son distribuidas conjuntamente.

Lo primero es descargarnos el programa JRE. (Java2 RuntimeEnvironment JRE 1.6.0.21). Java es software libre, por lo que podemos descargarnos la aplicación libremente.

Una vez descargado, comienza el proceso de instalación, siguiendo los pasos del asistente.

Debes conocer
El proceso de descarga, instalación y configuración del entorno de ejecución de programas. En el siguiente enlace, se explican los pasos para hacerlo bajo el sistema operativo Linux.
http://www.java.com/es/download/help/linux_install.xml

Para saber más
En el siguiente enlace encontrarás un tutorial del lenguaje Java, con sus principales características y órdenes y comandos principales.
http://www.tecnun.es/asignaturas/Informat1/AyudaInf/aprendainf/Java/Java2.pdf

5.6.- Pruebas.

Caso práctico
María reúne todos los códigos diseñados y los prepara para implementarlos en el equipo del cliente. Juan se percata de ello, y le recuerda a su amiga que aún no los han sometido a pruebas. Juan se acuerda bien de la vez que le pasó aquello: hace dos años, cuando fue a presentar una aplicación a sus clientes, no paraba de dar errores de todo tipo… los clientes, por supuesto, no la aceptaron y Juan perdió un mes de duro trabajo y estuvo a punto de perder su empleo…
"—No tan deprisa María, tenemos que PROBAR la aplicación".

Una vez obtenido el software, la siguiente fase del ciclo de vida es la realización de pruebas.

Normalmente, éstas se realizan sobre un conjunto de datos de prueba, que consisten en un conjunto seleccionado y predefinido de datos límite a los que la aplicación es sometida.

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

Entre todas las pruebas que se efectúan sobre el software podemos distinguir básicamente:

PRUEBAS UNITARIAS

Consisten en probar, una a una, las diferentes partes de software y comprobar su funcionamiento (por separado, de manera independiente). JUnit es el entorno de pruebas para Java.

PRUEBAS DE INTEGRACIÓN

Se realizan una vez que se han realizado con éxito las pruebas unitarias y consistirán en comprobar el funcionamiento del sistema completo: con todas sus partes interrelacionadas.

La prueba final se denomina comúnmente Beta Test, ésta se realiza sobre el entorno de producción donde el software va a ser utilizado por el cliente (a ser posible, en los equipos del cliente y bajo un funcionamiento normal de su empresa).

El período de prueba será normalmente el pactado con el cliente.

Autoevaluación
Si las pruebas unitarias se realizan con éxito, ¿es obligatorio realizar las de integración?

- Sí, si la aplicación está formada por más de cinco módulos diferentes.
- Sí, en cualquier caso.

Para saber más
Puedes visitar la siguiente página web, donde se detallan los tipos de pruebas que suelen hacer al software y la función de cada una.
http://www.sistedes.es/TJISBD/Vol-1/No-4/articles/pris-07-raja-ctps.pdf

5.7.- Documentación.

Caso práctico
Ada ha quedado dentro de dos días con su cliente. Pregunta a María por todos los dossiers de documentación. La pálida expresión de la joven hace que Ada arda en desesperación: "—¿No habéis documentado las etapas? ¿Cómo voy a explicarle al cliente y sus empleados el funcionamiento del software? ¿Cómo vamos a realizar su mantenimiento?".

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

¿Por qué hay que documentar todas las fases del proyecto?

Para dar toda la información a los usuarios de nuestro software y poder acometer futuras revisiones del proyecto.

Tenemos que ir documentando el proyecto en todas las fases del mismo, para pasar de una a otra de forma clara y definida. Una correcta documentación permitirá la reutilización de parte de los programas en otras aplicaciones, siempre y cuando se desarrollen con diseño modular.

Distinguimos tres grandes documentos en el desarrollo de software:

Documentos a elaborar en el proceso de desarrollo de software			
	GUÍA TÉCNICA	**GUÍA DE USO**	**GUÍA DE INSTALACIÓN**

Quedan reflejados:	• El diseño de la aplicación. • La codifica-ción de los programas. • Las pruebas realizadas.	• Descripción de la funciona-lidad de la aplicación. • Forma de comenzar a ejecu-tar la aplicación. • Ejemplos de uso del progra-ma. • Requerimientos software de la aplicación. • Solución de los posibles problemas que se pueden presentar.	Toda la información necesaria para: • Puesta en marcha. • Explotación. • Seguridad del sistema.
¿A quién va dirigido?	Al personal técni-co en informática (analistas y pro-gramadores).	A los usuarios que van a usar la aplicación (clientes).	Al personal informático res-ponsable de la instalación, en colaboración con los usuarios que van a usar la aplicación (clientes).
¿Cuál es su objetivo?	Facilitar un co-rrecto desarrollo, realizar correc-ciones en los pro-gramas y permitir un mantenimiento futuro.	Dar a los usuarios finales toda la información necesaria para utilizar la aplicación.	Dar toda la información nece-saria para garantizar que la implantación de la aplicación se realice de forma segura, confiable y precisa.

5.8.- Explotación.

Caso práctico
Llega el día de la cita con la cadena hotelera. Ada y Juan se dirigen al hotel sonde se va a instalar y configurar la aplicación. Si todo va bien, se irá implementando en los demás hoteles de la cadena. Ada no quiere que se le pase ni un detalle: lleva consigo la guía de uso y la guía de instalación.

Después de todas las fases anteriores, una vez que las pruebas nos demuestran que el software es fiable, carece de errores y hemos documentado todas las fases, el siguiente paso es la explotación.

Aunque diversos autores consideran la explotación y el mantenimiento como la misma etapa, nosotros vamos a diferenciarlas en base al momento en que se realizan.

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

La explotación es la instalación, puesta a punto y funcionamiento de la aplicación en el equipo final del cliente.

En el proceso de instalación, los programas son transferidos al computador del usuario cliente y posteriormente configurados y verificados.

Es recomendable que los futuros clientes estén presentes en este momento e irles comentando cómo se va planteando la instalación.

En este momento, se suelen llevan a cabo las Beta Test, que son las últimas pruebas que se realizan en los propios equipos del cliente y bajo cargas normales de trabajo.

Una vez instalada, pasamos a la fase de configuración.

En ella, asignamos los parámetros de funcionamiento normal de la empresa y probamos que la aplicación es operativa. También puede ocurrir que la configuración la realicen los propios usuarios finales, siempre y cuando les hayamos dado previamente la guía de instalación. Y también, si la aplicación es más sencilla, podemos programar la configuración de manera que se realice automáticamente tras instalarla. (Si el software es "a medida", lo más aconsejable es que la hagan aquellos que la han fabricado).

Una vez se ha configurado, el siguiente y último paso es la fase de producción normal. La aplicación pasa a manos de los usuarios finales y se da comienzo a la explotación del software.

Es muy importante tenerlo todo preparado antes de presentarle el producto al cliente: será el momento crítico del proyecto.

Reflexiona

Realizas un proyecto software por vez primera y no te das cuenta de documentarlo. Consigues venderlo a buen precio a una empresa. Al cabo de un par de meses te piden que actualices algunas de las funciones, para tener mayor funcionalidad. Estás contento o contenta porque eso significa un ingreso extra. Te paras un momento...¿Dónde están los códigos? ¿Qué hacía exactamente la aplicación? ¿Cómo se diseñó? No lo recuerdas... Probablemente hayas perdido un ingreso extra y unos buenos clientes.

5.9.- Mantenimiento.

Caso práctico

Ada reúne por última vez durante estas semanas a su equipo. Todos celebran que el proyecto se ha implementado con éxito y que sus clientes han quedado satisfechos.
—Esto aún no ha terminado —comenta Ada—, nos quedan muchas cosas por hacer. Esta tarde me reúno con los clientes. ¿Cómo vamos a gestionar el mantenimiento de la aplicación?

Sería lógico pensar que con la entrega de nuestra aplicación (la instalación y configuración de nuestro proyecto en los equipos del cliente) hemos terminado nuestro trabajo.

En cualquier otro sector laboral esto es así, pero el caso de la construcción de software es muy diferente.

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

Por su naturaleza, el software es cambiante y deberá actualizarse y evolucionar con el tiempo. Deberá ir adaptándose de forma paralela a las mejoras del hardware en el mercado y afrontar situaciones nuevas que no existían cuando el software se construyó.

Además, siempre surgen errores que habrá que ir corrigiendo y nuevas versiones del producto mejores que las anteriores.

Por todo ello, se pacta con el cliente un servicio de mantenimiento de la aplicación (que también tendrá un coste temporal y económico).

El mantenimiento se define como el proceso de control, mejora y optimización del software.

Su duración es la mayor en todo el ciclo de vida del software, ya que también comprende las actualizaciones y evoluciones futuras del mismo.

Los tipos de cambios que hacen necesario el mantenimiento del software son los siguientes:

- **Perfectivos**: Para mejorar la funcionalidad del software.
- **Evolutivos**: El cliente tendrá en el futuro nuevas necesidades. Por tanto, serán necesarias modificaciones, expansiones o eliminaciones de código.
- **Adaptativos**: Modificaciones, actualizaciones... para adaptarse a las nuevas tendencias del mercado, a nuevos componentes hardware, etc.
- **Correctivos**: La aplicación tendrá errores en el futuro (sería utópico pensar lo contrario).

Autoevaluación

¿Cuál es, en tu opinión, la etapa más importante del desarrollo de software?

- (•) El análisis de requisitos.
- ( ) La codificación.
- ( ) Las pruebas y documentación.
- ( ) La explotación y el mantenimiento.

Anexo I.- Sentencias de control de la programación estructurada.

SENTENCIAS SECUENCIALES

Las sentencias secuenciales son aquellas que se ejecutan una detrás de la otra, según el orden en que hayan sido escritas.

Ejemplo en lenguaje C:

```
printf ("declaración de variables");
intnumero_entero;
espacio=espacio_inicio + veloc*tiempo;
```

SENTENCIAS SELECTIVAS (CONDICIONALES)

Son aquellas en las que se evalúa una condición. Si el resultado de la condición es verdad es ejecutan una serie de acción o acciones y si es falso se ejecutan otras.

`if` → señala la condición que se va a evaluar

`then`→ Todas las acciones que se encuentren tras esta palabra reservada se ejecutarán si la condición del **`if`** es cierta (en C, se omite esta palabra).

`else`→ Todas las acciones que se encuentren tras esta otra palabra reservada se ejecutarán si la condición de **`if`**es falsa.

Ejemplo en lenguaje C:

```
if (a >= b)
        c= a-b;
else
        c=a+b;
```

SENTENCIAS REPETITIVAS (ITERACIONES O BUCLES)

Un bucle iterativo de una serie de acciones harán que éstas se repitan mientras o hasta que una determinada condición sea falsa (o verdadera).

`while`→ marca el comienzo del bucle y va seguido de la condición de parada del mismo.

`do`→ a partir de esta palabra reservada, se encontrarán todas las acciones a ejecutar mientras se ejecute el bucle (en C, se omite esta palabra).

`done`→ marca el fin de las acciones que se van a repetir mientras estemos dentro del bucle (en C, se omite esta palabra).

Ejemplo en lenguaje C:

```
intnum;
num = 0;
while (num<=10) {
        printf("Repetición numero %d\n", num);
        num = num + 1;
};
```

Anexo.- Licencias de recursos.

Datos del recurso (1)
Autoría: Scott Schram. Licencia: CC by 2.0. Procedencia: http://www.flickr.com/photos/schram/21742249/
Autoría: Francisco Palacios. Licencia: CC by -NC-ND 2.0. Procedencia: http://www.flickr.com/photos/wizard_/3303810302/
Autoría: fsse8info. Licencia: CC by -SA 2.0. Procedencia: http://www.flickr.com/photos/fsse-info/3276664015/

TEMA 2

Contenido

[INSTALACIÓN Y USO DE ENTORNOS DE DESARROLLO]

José Luis Comesaña Cabeza --- 2011/2012

Entornos de Desarrollo del curso de "*Desarrollo de Aplicaciones Web*"

INSTALACIÓN Y USO DE ENTORNOS DE DESARROLLO

CASO PRÁCTICO.
Tras el éxito del anterior proyecto, en BK están recibiendo más peticiones de creación de software que nunca.
Ana y Antonio, que ya hace unas semanas que están estudiando el Ciclo de Diseño de Aplicaciones Multiplataforma, piensan que este es un buen momento para participar activamente en los proyectos, pues a sus compañeros no les vendría nada mal un poco de ayuda.
¿Cómo influirá el conocimiento de esta herramienta en el futuro de Ana y Antonio?
A través de esta unidad, veremos si nuestros amigos van logrando ganarse un puesto en la empresa... y de paso, la confianza de Ada.
La fase de codificación es compleja, pero Ana y Antonio están aprendiendo a dominar los llamados entornos integrados de desarrollo de software...
Ada confía en ellos, pero aún es pronto... Por lo menos, ya conocen las fases por las que tiene que pasar todo el desarrollo de aplicaciones... pero eso no será suficiente.
María, sin embargo, no piensa lo mismo y decide darles una oportunidad trabajando en la fase de codificación de un nuevo proyecto de la empresa.
Ana se muestra muy ilusionada y no piensa desperdiciar esta gran oportunidad. Sabe que tiene a su disposición los llamados entornos de desarrollo que le facilitarán su futura tarea.

1. Concepto de Entorno de Desarrollo. Evolución Histórica.

CASO PRÁCTICO.
Todos en la empresa están sorprendidos del entusiasmo de Ana ante los nuevos proyectos que B.K tiene por delante. Juan, que acabó el Ciclo Superior de DAI hace algunos años, se muestra inquieto porque es consciente de que en sólo unos cuatro años han salido muchas herramientas nuevas en el mercado y necesita reciclarse... Escucha a Ana decir que está estudiando los entornos de desarrollo... Yo también debería ponerme al día, piensa...

En la unidad anterior hablábamos de las fases en el proceso de desarrollo de software.

Una de ellas era la fase de codificación, en la cual se hacía uso de algún lenguaje de programación para pasar todas las acciones que debía llevar a cabo la aplicación a algún lenguaje que la máquina fuera capaz de entender y ejecutar.

También se hizo alusión a herramientas de apoyo al proceso de programación.

En esta unidad vamos a analizar, instalar y ejecutar estas herramientas para entender su acción y efecto.

Muchas personas aprender a programar utilizando un editor de texto simple, compilador y depurador. Pero la mayoría, finalmente, terminan haciendo uso de algún entorno de desarrollo integrado (IDE) para crear aplicaciones.

Un entorno integrado de desarrollo (IDE), es un tipo de software compuesto por un conjunto de herramientas de programación. En concreto, el IDE se compone de:

- ✓ Editor de código de programación.
- ✓ Compilador.
- ✓ Intérprete.
- ✓ Depurador.
- ✓ Constructor de interfaz gráfico.

Los primeros entornos de desarrollo integrados nacen a principios de los años 70, y se popularizan en la década de los 90. Tienen el objetivo de ganar fiabilidad y tiempo en los proyectos de software. Proporcionan al programador una serie de componentes con la misma interfaz gráfica, con la consiguiente comodidad, aumento de eficiencia y reducción de tiempo de codificación.

Normalmente, un IDE está dedicado a un determinado lenguaje de programación. No obstante, las últimas versiones de los IDEs tienden a ser compatibles con varios lenguajes (por ejemplo, Eclipse, NetBeans, Microsoft Visual Studio...) mediante la instalación de plugins adicionales.

En este tema, nuestro interés se centra en conocer los entornos de desarrollo, los tipos, en función de su licencia y del lenguaje de programación hacia el cual están enfocados. Instalaremos NetBeans bajo Ubuntu y veremos cómo se configura y cómo se generan ejecutables, haciendo uso de sus componentes y herramientas.

REFLEXIONA
Según datos, casi todas las personas que empiezan a programar utilizan un editor simple de textos y un compilador-depurador instalado en su equipo. Sin embargo, prácticamente todas acaban utilizando un entorno de desarrollo.

1.1Evolución Histórica

En las décadas de utilización de la tarjeta perforada como sistema de almacenamiento el concepto de Entorno de Desarrollo Integrado sencillamente no tenía sentido.

Los programas estaban escritos con diagramas de flujo y entraban al sistema a través de las tarjetas perforadas. Posteriormente, eran compilados.

El primer lenguaje de programación que utiliza un IDE fue el BASIC (que fue el primero en abandonar también las tarjetas perforadas o las cintas de papel).

Éste primer IDE estaba basado en consola de comandos exclusivamente (normal por otro lado, si tenemos en cuenta que hasta la década de los 90 no entran en el mercado los sistemas operativos con interfaz gráfica). Sin embargo, el uso que hace de la gestión de archivos, compilación, depuración… es perfectamente compatible con los IDE actuales.

A nivel popular, el primer IDE puede considerarse que fue el IDE llamado Maestro. Nació a principios de los 70 y fue instalado por unos 22000 programadores en todo el mundo. Lideró el campo durante los años 70 y 80.

El uso de los entornos integrados de desarrollo se ratifica y afianza en los 90 y hoy en día contamos con infinidad de IDE, tanto de licencia libre como no.

Tabla de los IDE más relevantes hoy en día:

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.

DESTACADO
No hay unos entornos de desarrollo más importantes que otros. La elección del IDE más adecuado dependerá del lenguaje de programación que vayamos a utilizar para la codificación de las aplicaciones y el tipo de licencia con la que queramos trabajar.

2. Funciones de un Entorno de Desarrollo

CASO PRÁCTICO

Juan, que asume por fin su desconocimiento, habla con Ana para que le pase sus apuntes de entornos de desarrollo. Ésta se muestra encantada, y le anima a matricularse al ciclo DAM a distancia. Juan se muestra reacio (ya he estudiado el ciclo... y durante cuatro años he cumplido con éxito en la empresa). Pero piensa que quizás debería reciclarse si no quiere quedarse atrás en los proyectos Juan aprendió a programar usando un editor simple de textos, ¿qué ventajas tendrá programando con un IDE?.

Como sabemos, los entornos de desarrollo están compuestos por una serie de herramientas software de programación, necesarias para la consecución de sus objetivos. Estas herramientas son:

- ✓ Un editor de código fuente.
- ✓ Un compilador y / o un intérprete.
- ✓ Automatización de generación de herramientas.
- ✓ Un depurador.

Las funciones de los IDE son:

- ✓ Editor de código: coloración de la sintaxis.
- ✓ Auto-completado de código, atributos y métodos de clases.
- ✓ Identificación automática de código.
- ✓ Herramientas de concepción visual para crear y manipular componentes visuales.
- ✓ Asistentes y utilidades de gestión y generación de código.
- ✓ Archivos fuente en unas carpetas y compilados a otras.
- ✓ Compilación de proyectos complejos en un solo paso.
- ✓ Control de versiones: tener un único almacén de archivos compartido por todos los colaboradores de un proyecto. Ante un error, mecanismo de auto-recuperación a un estado anterior estable.
- ✓ Soporta cambios de varios usuarios de manera simultánea.
- ✓ Generador de documentación integrado.
- ✓ Detección de errores de sintaxis en tiempo real.

Otras funciones importantes son:

- ✓ Ofrece refactorización de código: cambios menores en el código que facilitan su legibilidad sin alterar su funcionalidad (por ejemplo cambiar el nombre a una variable).
- ✓ Permite introducir automáticamente tabulaciones y espaciados para aumentar la legibilidad.
- ✓ Depuración: seguimiento de variables, puntos de ruptura y mensajes de error del intérprete.
- ✓ Aumento de funcionalidades a través de la gestión de sus módulos y plugins.
- ✓ Administración de las interfaces de usuario (menús y barras de herramientas).
- ✓ Administración de las configuraciones del usuario.

AUTOEVALUACIÓN:

Un entorno integrado de desarrollo está compuesto por:

- ○ Editor de código y traductor.
- ○ Interfaz gráfica, editor de código y depurador.
- ◉ Editor de código, compilador, intérprete, depurador e interfaz gráfica.
- ○ Interfaz gráfica, editor de código y depurador.

3. Entornos Integrados Libres y Propietarios

CASO PRÁCTICO

Juan ha buscado por Internet distintos entornos de desarrollo para aplicarlos en la fase de codificación.

—Cuidado, —le dice Ada—. Ya sabes que es de vital importancia el tema de la Licencia de Software. Hay Entornos de desarrollo de licencia libre y otros no, y este aspecto es fundamental si no queremos tener problemas.

Entornos Integrados Libres

Son aquellos con licencia de uso público.

No hay que pagar por ellos, y aunque los más conocidos y utilizados son Eclipse y NetBeans, hay bastantes más.

IDE	Lenguajes que soporta	Sistema Operativo
NetBeans.	C/C++, Java, JavaScript, PHP, Python.	Windows, Linux, Mac OS X.
Eclipse.	Ada, C/C++, Java, JavaScript, PHP.	Windows, Linux, Mac OS X.
Gambas.	Basic.	Linux.
Anjuta.	C/C++, Python, Javascript.	Linux.
Geany.	C/C++, Java.	Windows, Linux, Mac OS X.
GNAT Studio.	Fortran.	Windows, Linux, Mac OS X.

DESTACADO

El aspecto de la licencia del IDE que se elija para el desarrollo de un proyecto es una cuestión de vital importancia. En su elección prevalecerá la decisión de los supervisores del proyecto y de la dirección de la empresa.

PARA SABER MÁS

En el siguiente enlace encontrarás un documento muy interesante, en inglés, donde se detallan todos los entornos de desarrollo existentes en la actualidad con todas sus características: licencias, sistemas operativos donde pueden ser instalados y configurados, lenguajes que soporta, desarrolladores y última versión estable.

Entornos de desarrollo actuales. http://en.wikipedia.org/wiki/Integrated_development_environment

Entornos Integrados Propietarios

Son aquellos entornos integrados de desarrollo que necesitan licencia. No son free software, hay que pagar por ellos. El más conocido y utilizado es Microsoft Visual Studio, que usa el framework .NET y es desarrollado por Microsoft.

IDE	Lenguajes que soporta	Sistema Operativo
Microsoft Visual Studio.	Basic, C/C++, C#.	Windows.
FlashBuilder.	ActionScript.	Windows, Mac OS X.
C++ Builder.	C/C++.	Windows.
Turbo C++ profesional.	C/C++.	Windows.
JBuilder.	Java.	Windows.
JCreator.	Java.	Windows, Linux, Mac OS X.
Xcode.	C/C++, Java.	Mac OS X.

AUTOEVALUACIÓN

Relaciona los siguientes entornos de desarrollo con sus características, escribiendo el número asociado a la característica en el hueco correspondiente.

Entorno de desarrollo.	Relación	Características
Microsoft Visual Studio	2	1. Libre. Soporta C/C++, Java, PHP, Javascript, Python
NetBeans	1	2. Propietario. Soporta Basic, C/C++, C#
C++ Builder	3	3. Propietario. Soporta C/C++

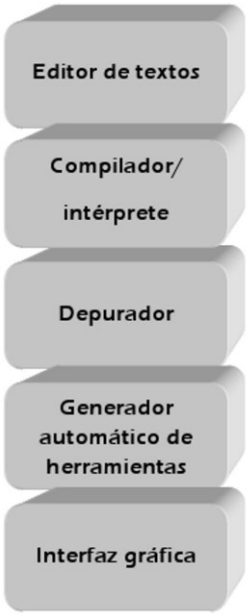
4. Estructura de Entornos de Desarrollo

CASO PRÁCTICO

Juan aprendió a programar utilizando un editor de textos, un compilador y un depurador. Todas estas herramientas se instalaban de forma independiente. A Ana le cuesta creer que los programadores tuvieran que buscar estas herramientas e instalarlas por separado. –En un entorno se integran todas estas cosas y muchas más, y sin salir del mismo puedes programar en varios lenguajes y puedes documentar y…. —Ya lo veo, —le replica Juan—.¿Cuántos componentes tiene el entorno en total?

Los entornos de desarrollo, ya sean libres o propietarios, están formados por una serie de componentes software que determinan sus funciones.

Estos componentes son:



Editor de textos: Se resalta y colorea la sintaxis, tiene la función de autocompletar código, ayuda y listado de parámetros de funciones y métodos de clase. Inserción automática de paréntesis, corchetes, tabulaciones y espaciados.

Compilador/intérprete: Detección de errores de sintaxis en tiempo real. Características de refactorización.

Depurador: Botón de ejecución y traza, puntos de ruptura y seguimiento de variables. Opción de depurar en servidores remotos.

Generador automático de herramientas: Para la visualización, creación y manipulación de componentes visuales y todo un arsenal de asistentes y utilidades de gestión y generación código.

Interfaz gráfica: Nos brinda la oportunidad de programar en varios lenguajes con un mismo IDE. Es una interfaz agradable que puede acceder a innumerables bibliotecas y plugins, aumentando las opciones de nuestros programas.

PARA SABER MÁS

En el siguiente enlace accederás a una página web donde se detallan todos los componentes del entorno de desarrollo, junto con sus funciones.

Estructura de Entornos de Desarrollo http://es.scribd.com/doc/41884812/Entornos-de-Desarrollo-Integrados

5. Instalación de Entornos Integrados de Desarrollo.

CASO PRÁCTICO

Juan está decidido a aprender a usar un entorno de desarrollo. Después de documentarse, piensa que lo idóneo es trabajar con un IDE libre. Además, el tema del sistema operativo que soporta es importante. Juan quiere trabajar bajo Linux, y se decide por el entorno NetBeans. Ahora bien, ¿Qué hay que hacer para instalarlo?.

Vamos a realizar la instalación de NetBeans, en su versión 6.9.1 sobre Ubuntu 10.10. Tiene alguna complicación, porque se va a trabajar desde la terminal de Ubuntu. Te pedimos que prestes atención a los comandos.

5.1 INSTALACIÓN DEL JDK

DESTACADO

La instalación del IDE NetBeans, ya sea en Linux, Windows o Mac OS X, requiere la instalación previa del JDK compatible con la versión de NetBeans que se quiera instalar.

Lo primero es instalar el JDK en el sistema operativo. Esta será la plataforma del entorno, imprescindible para que éste pueda ser instalado en el sistema operativo y funcionar.

Se ha elegido como sistema operativo Linux. El proceso de instalación sólo podrá ser realizado por el root, que es el súper-usuario. Por ello, la instalación se realizará desde la consola de comandos:

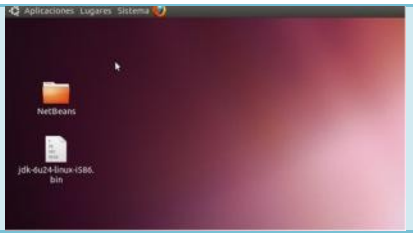
Versión de JDK elegida:

`JDK-6u24-linux-i586.`

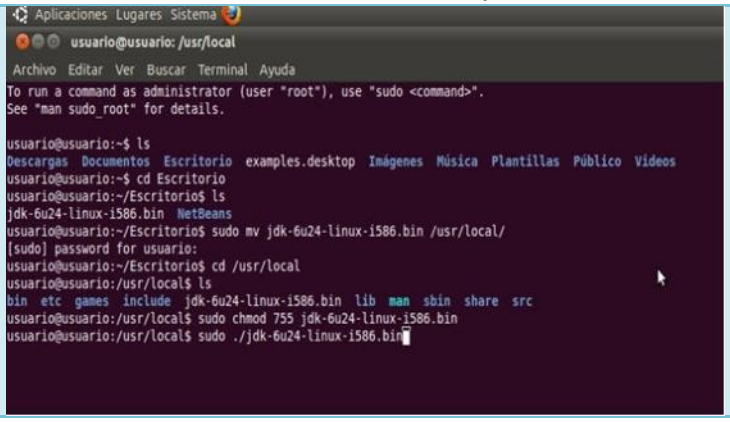
Órdenes en la consola de comandos:

- ✓ Obtener el archivo, que se adjunta como recurso en la presente unidad.
- ✓ Mover el archivo a /usr/local.
- ✓ Darle permisos de ejecución, como root del sistema.
- ✓ Ejecutarlo, como root.

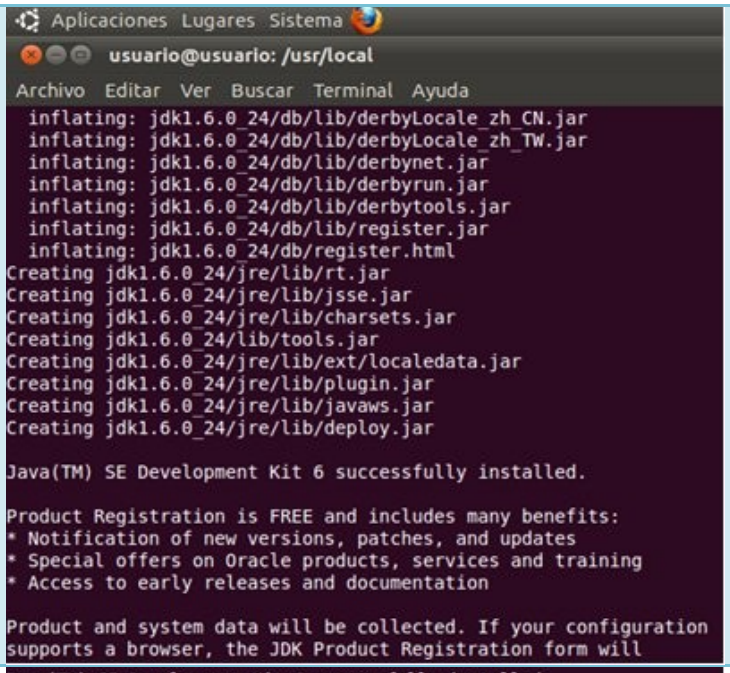
El proceso de instalación en Linux consta de una serie de pasos, y se explican con detalle en el siguiente proceso:

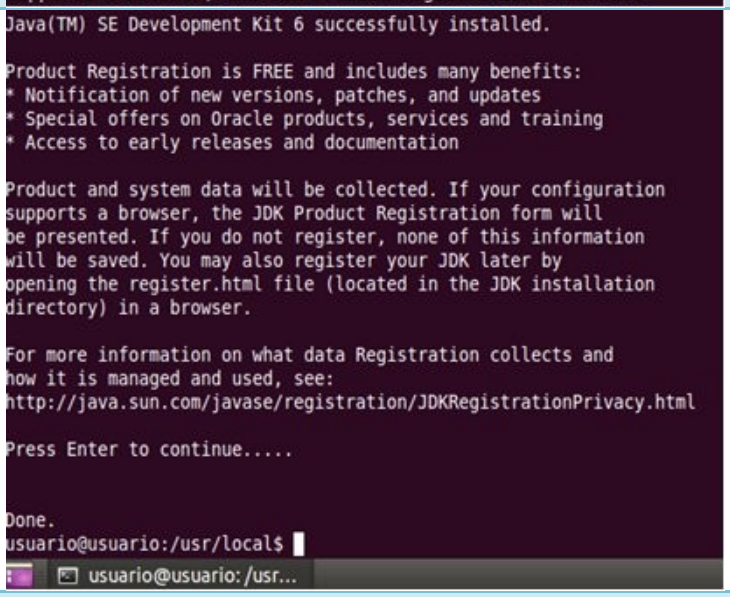
Instalación JDK en Ubuntu 10.10.	
Descarga el JDK de la siguiente URL: http://www.oracle.com/technetwork/java/javase/downloads/index.html	
El archivo de JDK utilizado es: jdk-6u24-linux-i586.bin	
Guardar el archivo en el escritorio de Linux.	
Mover el archivo al directorio /usr/local *El movimiento del archivo a esta ruta sólo puede ser realizado por el root del sistema.* *Para poder ejecutarlo como un usuario normal, basta poner el comando **sudo** antes de la orden. Esto implica que todas las operaciones a partir de este momento deberemos realizarlas desde la terminal del sistema operativo.*	
Para acceder a la terminal, pulsamos sobre la pestaña de: Aplicaciones - Accesorios - Terminal	
Las acciones a realizar serán las siguientes:	
Entramos en el escritorio:	$ cd Escritorio $ sudo mv jdk-6u24-linux-i586.bin /usr/local
Darle permiso de ejecución al archivo jdk y ejecutarlo	$ cd /usr/local $ sudo chmod 755 jdk-6u24-linux-i586.bin

$ sudo ./jdk-6u24-linux-i586.bin

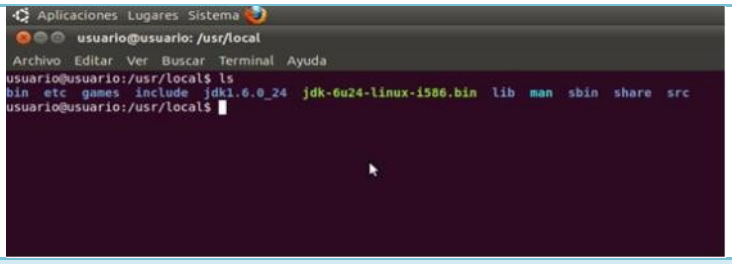


Comienza la instalación…

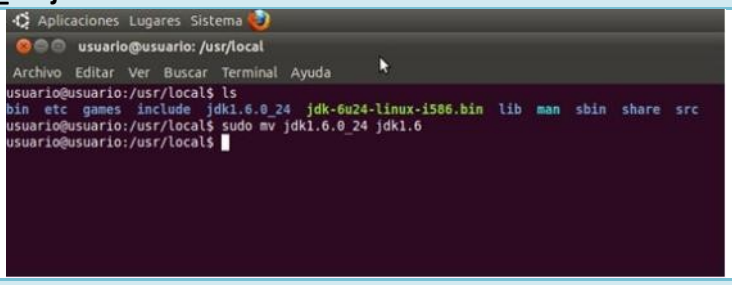




Renombramos la carpeta que se ha creado durante la instalación del archivo.

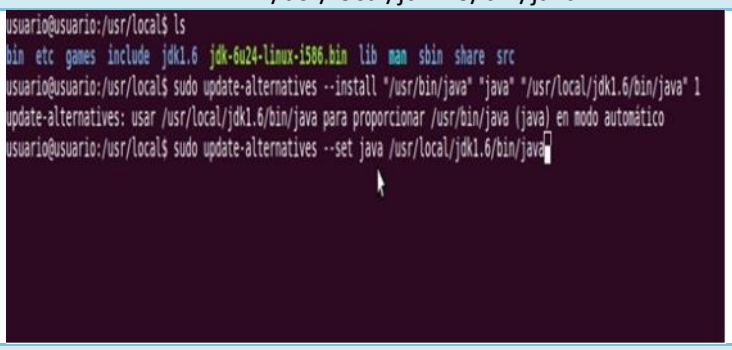


$ sudo mv jdk1.6.0_24 jdk1.6



Ejecutamos los siguientes comandos:

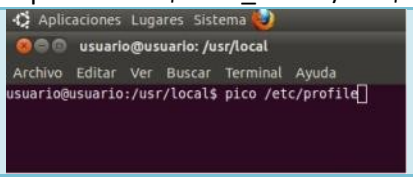
$ sudo update-alternatives --install "/usr/bin/java" "java" "/usr/local/jdk1.6/bin/java" 1
$ sudo update-alternatives --set java /usr/local/jdk1.6/bin/java



Editamos el archivo /etc/profile y agregamos las siguiente líneas al final del mismo:

export JAVA_HOME=/usr/local/jdk1.6
export PATH=$JAVA_HOME/bin:$PATH

Para editar el archivo podemos usar el comando:
$ pico /etc/profile



o utilizar el comando:
$ nano /etc/profile

```
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
 GNU nano 2.2.4                                  Archivo: /etc/profile

# /etc/profile: system-wide .profile file for the Bourne shell (sh(1))
# and Bourne compatible shells (bash(1), ksh(1), ash(1), ...).

if [ -d /etc/profile.d ]; then
  for i in /etc/profile.d/*.sh; do
    if [ -r $i ]; then
      . $i
    fi
  done
  unset i
fi

if [ "$PS1" ]; then
  if [ "$BASH" ]; then
    PS1='\u@\h:\w\$ '
    if [ -f /etc/bash.bashrc ]; then
      . /etc/bash.bashrc
    fi
  else
    if [ "`id -u`" -eq 0 ]; then
      PS1='# '
    else
      PS1='$ '
    fi
  fi
fi

umask 022
```

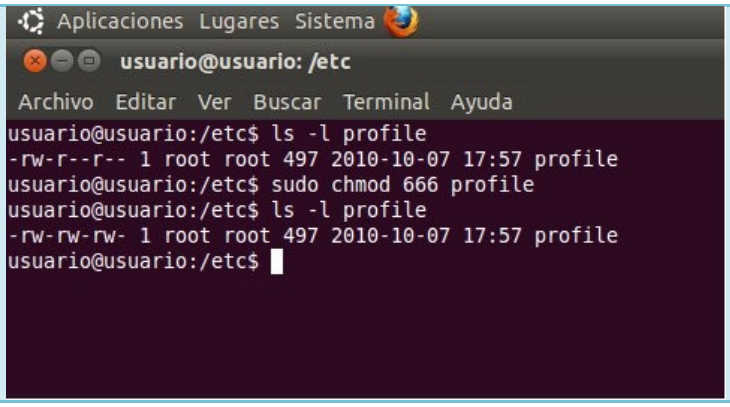
Nos colocamos al final del archivo y escribimos esas dos líneas:

```
# /etc/profile: system-wide .profile file for the Bourne shell (sh(1))
# and Bourne compatible shells (bash(1), ksh(1), ash(1), ...).

if [ -d /etc/profile.d ]; then
  for i in /etc/profile.d/*.sh; do
    if [ -r $i ]; then
      . $i
    fi
  done
  unset i
fi

if [ "$PS1" ]; then
  if [ "$BASH" ]; then
    PS1='\u@\h:\w\$ '
    if [ -f /etc/bash.bashrc ]; then
      . /etc/bash.bashrc
    fi
  else
    if [ "`id -u`" -eq 0 ]; then
      PS1='# '
    else
      PS1='$ '
    fi
  fi
fi

umask 022
export JAVA_HOME=/usr/local/jdk1.6
export PATH=$JAVA_HOME/bin:$PATH
```

Guardamos el archivo y nos dice que no tenemos permisos para modificarlo.

Por tanto, tenemos que darle a /etc/profile permiso de modificación:

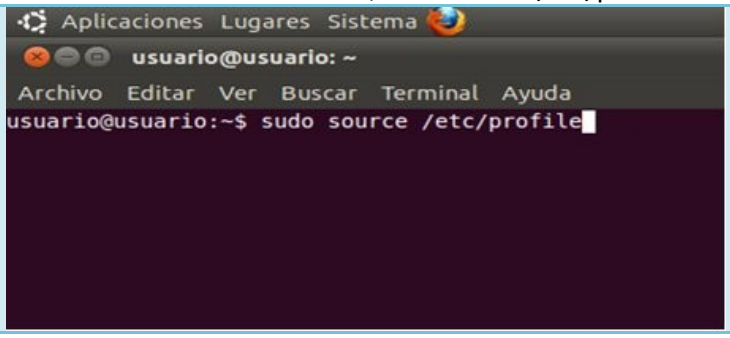
```
Aplicaciones  Lugares  Sistema
usuario@usuario: /etc
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
usuario@usuario:/etc$ ls -l profile
-rw-r--r-- 1 root root 497 2010-10-07 17:57 profile
usuario@usuario:/etc$
```



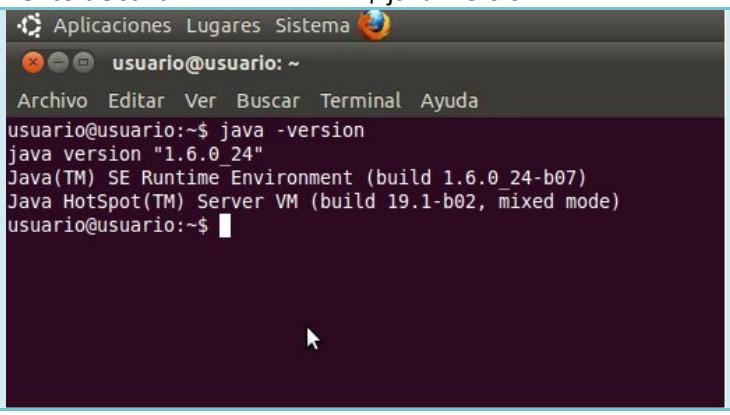
Ya sí podemos modificar el archivo agregándole las dos líneas al final del mismo (Repetir el paso de antes y guardar el archivo)

Salimos de la terminal, tecleando el comando exit, y volvemos a entrar en ella.

Teclear lo siguiente: $ sudo source /etc/profile



Probar el funcionamiento de Java. $ java -version



DESTACADO

JDK son las siglas de Java Development Kit: Kit de desarrollo de Java. Consiste en la plataforma del entorno, imprescindible para que éste pueda ser instalado y ejecutado.

5.2 INSTALACIÓN DE NETBEANS

CASO PRÁCTICO

Juan ya ha instalado el JDK.

—Uff, me ha costado un poco… —le comenta a Ana. —Hace tiempo que no trabajaba en la terminal de Linux y se me habían olvidado algunas órdenes básicas.

Ana le comenta que ya tiene el equipo preparado para instalar NetBeans. Decide pasarle los apuntes del ciclo a distancia para que Juan no tenga que perder mucho tiempo buscando los comandos necesarios.

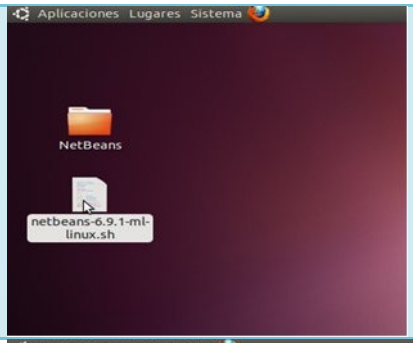
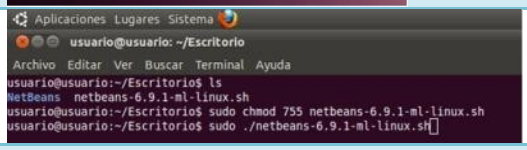
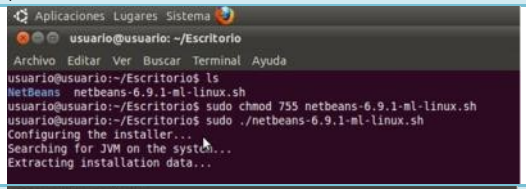
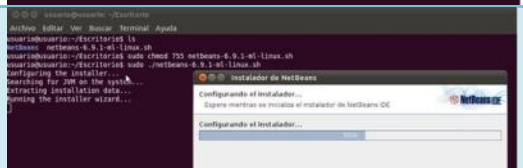
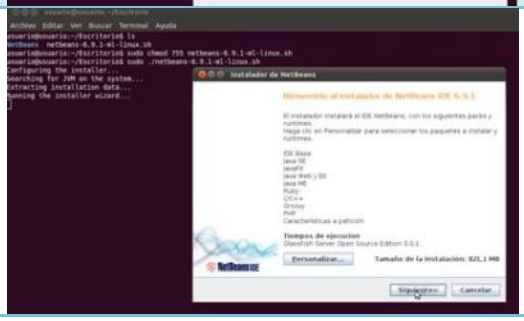
Una vez tenemos instalado el JDK en nuestro equipo, ya tenemos preparado el contexto en el que se instalará el entorno NetBeans.

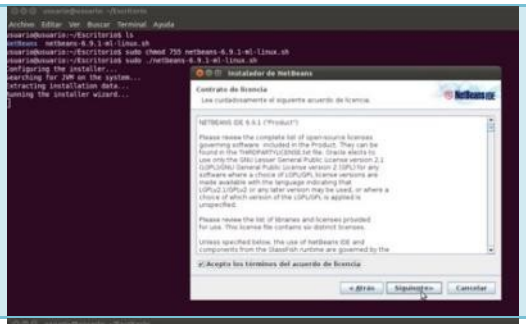
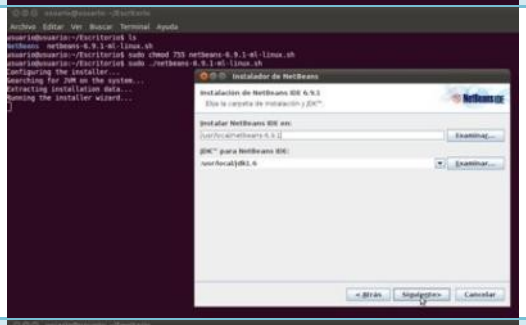
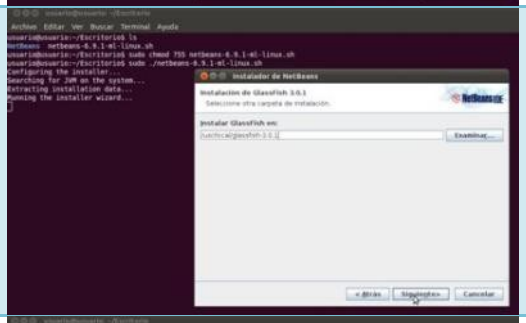
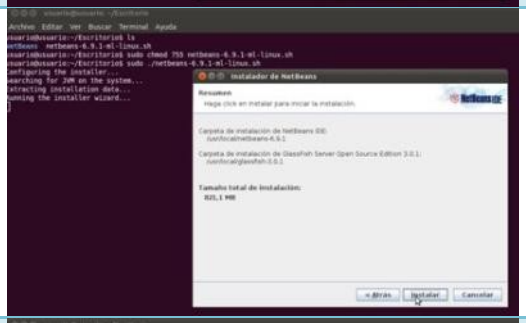
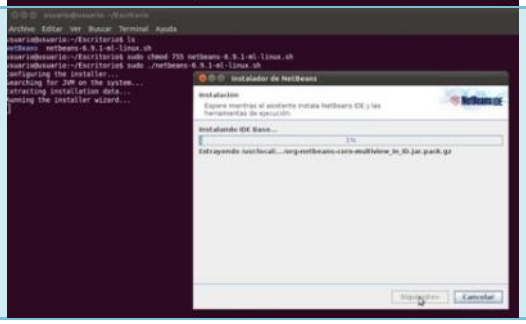
La versión elegida es NetBeans 6.9.1. El archivo se puede descargar libremente desde el sitio web oficial y la instalación sólo puede ser realizada por el root. (Cuando estudies este módulo puede que haya una versión más reciente. De todas formas, es muy probable que las condiciones de instalación no sean las mismas que las aquí descritas. Recuerda repasar las recomendaciones de instalación que estarán en la página de NetBeans).

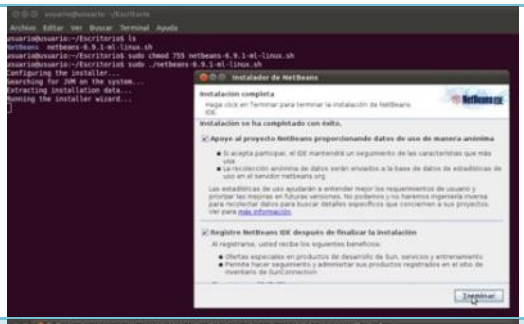
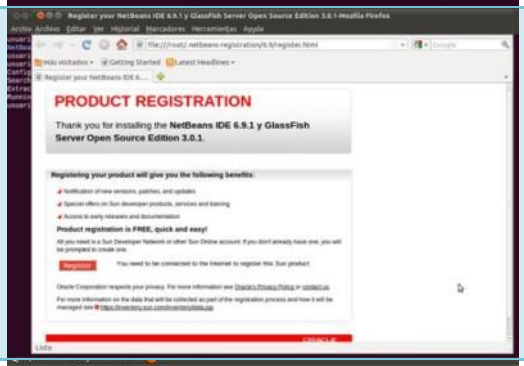
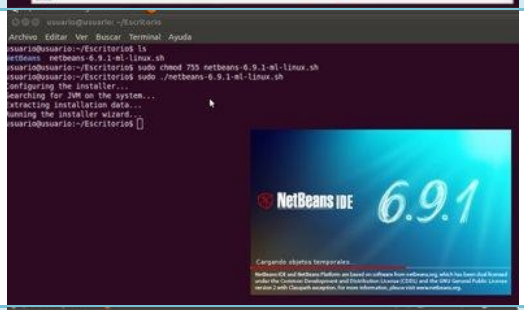
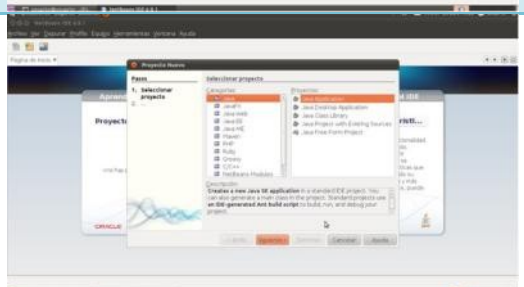
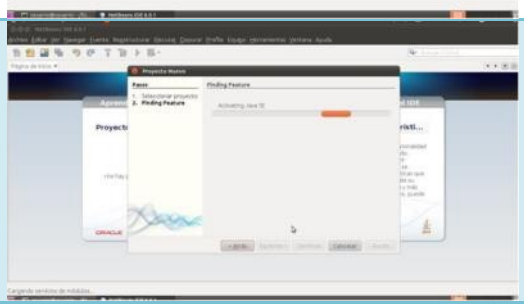
Eso nos fuerza a realizarla en la consola de comandos, y es un poco más compleja que en el caso del JDK.

Al igual que en el caso anterior, hay que darle al archivo permiso de ejecución y ejecutarlo.

El proceso de instalación se explica con detalle a continuación:

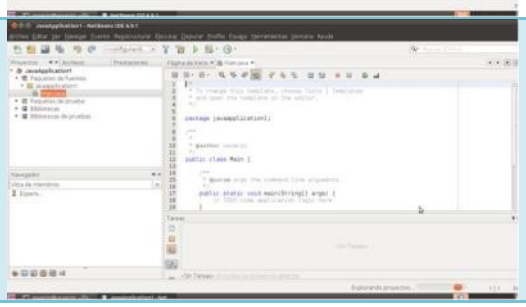
Instalación NetBeans 6.9.1 en Ubuntu 10.10.	
Descargar NetBeans de la siguiente URL: http://www.oracle.com/technetwork/java/javase/downloads/index.html	
La versión de NetBeans utilizada es: NetBeans 6.9.1 y el archivo de instalación es: Netbeans-6.9.1-ml-linux.sh	
Guardar el archivo en el escritorio de Linux.	
Le damos permiso de ejecución y lo instalamos	
El código es:	$ sudo chmod 755 netbeans-6.9.1-ml-linux.sh $ sudo ./netbeans-6.9.1-ml-linux.sh
Comienza el proceso de instalación:	
Durante la instalación nos aparecen sus imágenes gráficas correspondientes:	
La instalación en sí es muy sencilla: basta con seleccionar "siguiente" en todas las opciones:	

Aceptamos la licencia…	
Podemos seleccionar el lugar donde se instalará NetBeans. **Aquí se ha dejado el lugar por defecto: seleccionamos siguiente.**	
Volvemos a seleccionar siguiente:	
Terminada la fase de configuración previa, seleccionamos instalar:	
El proceso de instalación dura unos instantes…	

En la penúltima ventana de instalación nos preguntan si queremos registrarnos y colaborar con el proyecto de forma anónima. Esta elección es personal y aunque se ha seleccionado, no es necesario, aunque es una opción interesante.	
Pulsamos sobre terminar.	
Después de registrarnos (si así lo hemos querido), se abre la ventana de NetBeans:	
Una opción muy útil de NetBeans es la incorporación de tutoriales on-line sobre los aspectos más destacados de este entorno de desarrollo:	
Si queremos hacer un nuevo proyecto, basta con seleccionar: **Archivo-Nuevo Proyecto** **y aparece la siguiente ventana:**	
Seleccionamos la opción que nos interese y pulsamos sobre siguiente.	

Le daríamos un nombre y la ubicación donde se va a guardar:
Finalmente, pulsamos Terminar.

La apariencia primera del proyecto sería la siguiente:



AUTOEVALUACIÓN

En tu opinión, ¿Por qué crees que la instalación del JDK sólo la puede realizar el root del sistema?

- (•) Porque se trata de un archivo binario de sistema.
- ( ) Porque ningún archivo puede ser ejecutado por un usuario que no sea el root.
- ( ) Porque estamos trabajando en la terminal del sistema.

PARA SABER MÁS

De los IDE propietarios, es muy utilizado el Microsoft Visual Studio. En el siguiente vídeo podrás ver un proceso de instalación de este entorno:
http://www.youtube.com/watch?v=F2fDz2alP-w

6. Configuración y personalización de entornos de desarrollo.

CASO PRÁCTICO

Juan está consternado. NetBeans parece albergar tanta información que no sabe por dónde empezar. Le gustaría personalizar la configuración de su primer proyecto en el IDE (que va a ser una aplicación de Java). ¿Cómo lo hace? ¿Qué parámetros puede configurar?

Una vez tenemos instalado nuestro entorno de desarrollo podemos acceder a personalizar su configuración.

Al abrir un proyecto existente, o bien crear un nuevo proyecto, seleccionaremos un desplegable con el nombre de "configuración" desde el que podremos personalizar distintas opciones del proyecto.

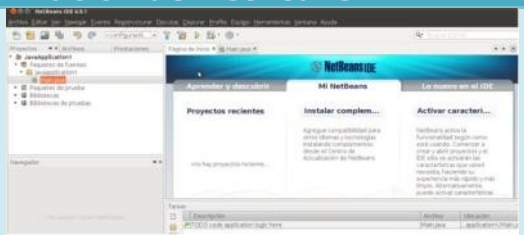
Podemos personalizar la configuración del entorno sólo para el proyecto actual, o bien para todos los proyectos, presentes y futuros.

Parámetros configurables del entorno:

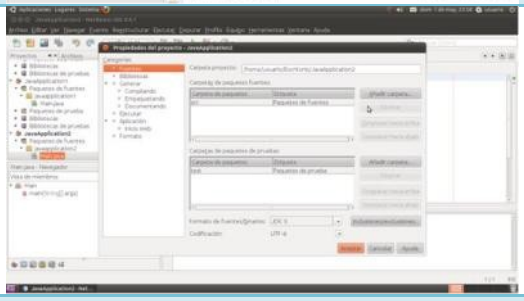
- ✓ Carpeta/s donde se alojarán todos los archivos de los proyectos (es importante la determinación de este parámetro, para tener una estructura de archivos ordenada).
- ✓ Carpetas de almacenamiento de paquetes fuente y paquetes prueba.
- ✓ Administración de la plataforma del entorno de desarrollo.
- ✓ **Opciones de la compilación de los programas**: compilar al grabar, generar información de depuración...
- ✓ **Opciones de empaquetado de la aplicación**: nombre del archivo empaquetado (con extensión .jar, que es la extensión característica de este tipo de archivos empaquetados) y momento del empaquetado.
- ✓ Opciones de generación de documentación asociada al proyecto.
- ✓ Descripción de los proyectos, para una mejor localización de los mismos.
- ✓ Opciones globales de formato del editor: número de espaciados en las sangrías, color de errores de sintaxis, color de etiquetas, opción de autocompletado de código, propuestas de insertar automáticamente código...
- ✓ Opciones de combinación de teclas en teclado.
- ✓ Etc.

Configuración y personalización de NetBeans.

Accedemos a NetBeans y entramos en la página principal de la aplicación.



Para entrar a la aplicación podemos seleccionar "Nuevo Proyecto" y, una vez abierto, personalizar la configuración de NetBeans para ese proyecto.
En la barra de iconos de la aplicación, seleccionamos el desplegable de configuración
Seleccionamos "personalizar" y nos aparecerá la siguiente ventana:



Aquí vemos todo lo que podemos personalizar de la aplicación:

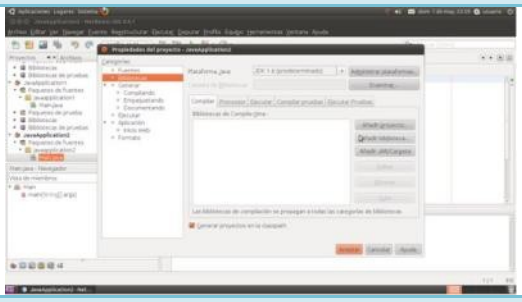
- ✓ **Fuentes.**
- ✓ **Bibliotecas.**
- ✓ **Generación de código.**
- ✓ **Ejecución de código.**

- ✓ **Opciones de la aplicación.**
- ✓ **Formato del código en el editor de textos.**

FUENTES:
Podemos modificar:

- ✓ **La carpeta que contendrá el proyecto**
- ✓ **La carpeta que almacenará los paquetes fuentes**
- ✓ **La carpeta que contendrá los paquetes prueba**

BIBLIOTECAS:

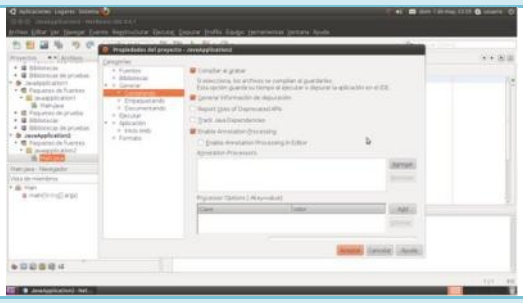


Desde esta ventana podemos elegir la plataforma de la aplicación.
Toma por defecto el JDK, pero se puede cambiar si se quiere, siempre y cuando sea compatible con la versión de NetBeans utilizada.
También en esta ventana se puede configurar el paquete de pruebas que se realizará al proyecto.

GENERACIÓN DE CÓDIGO - COMPILANDO
Las opciones que nos permite modificar en cuanto a la compilación del programa son:

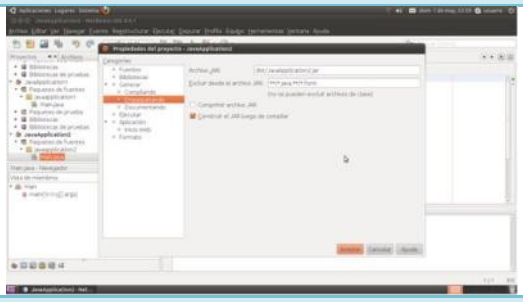
- ✓ **Compilar al grabar: al guardar un archivo se compilará automáticamente.**
- ✓ **Generar información de depuración: para obtener la documentación asociada.**
- ✓ **Enable annotation processsing: permitir anotaciones durante el proceso.**

También podemos agregar anotaciones concretas para el proceso de compilación y añadir opciones de proceso que, según las características del proyecto, puedan ser de interés para nosotros.



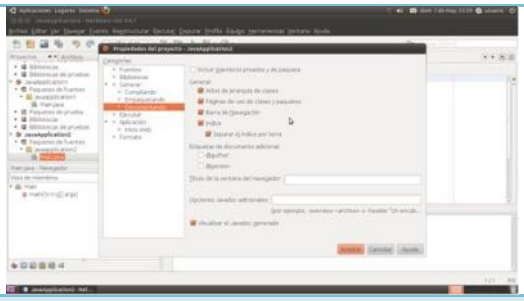
GENERACIÓN DE CÓDIGO - EMPAQUETANDO
Las aplicaciones resultado de la compilación del código deben ser empaquetadas antes de su distribución, con objeto de tener un único archivo, generalmente comprimido, que contenga en su interior todos los archivos de instalación y configuración necesarios para que la aplicación pueda ser instalada y desarrollada con éxito por el usuario cliente.

Como vemos en la imagen, en esta opción podemos modificar el lugar donde se generará el archivo resultante del empaquetado, así como si deseamos comprimirlo.
También podemos elegir que el archivo empaquetado se construya tras la compilación, que es lo habitual (por eso esta opción aparece como predeterminada)



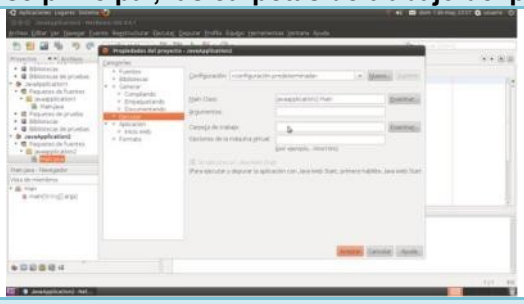
GENERACIÓN DE CÓDIGO - DOCUMENTANDO
Como ya vimos en la unidad anterior, la documentación de aplicaciones es un aspecto clave que no debemos descuidar nunca. NetBeans nos ofrece una ventaja muy considerable al permitirnos obtener documentación de la fase de codificación de los programas de forma automática.

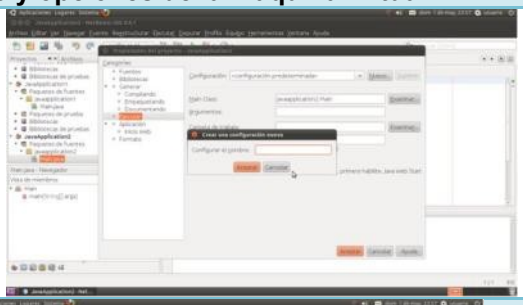
Dentro del documento que se va a generar podemos elegir que se incluyan todas las opciones anteriores. Esto es lo más recomendable, por eso aparecen todas marcadas de forma predeterminada y lo mejor es dejarlo como está.



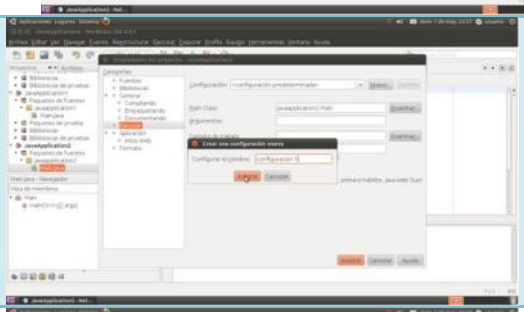
EJECUTANDO CÓDIGO

Esta opción nos permite definir una nueva configuración de ejecución de código, elegir la clase principal, las carpetas de trabajo del proyecto y opciones de la máquina virtual.



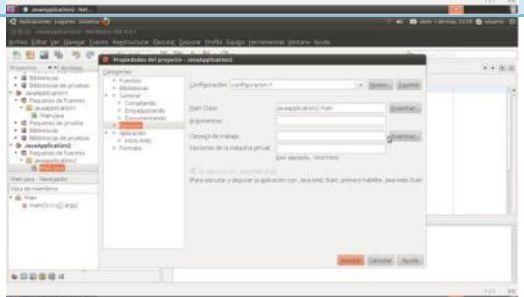


En la ventana de "Configurar el nombre" escribimos el nombre que tendrá nuestra configuración personalizada.

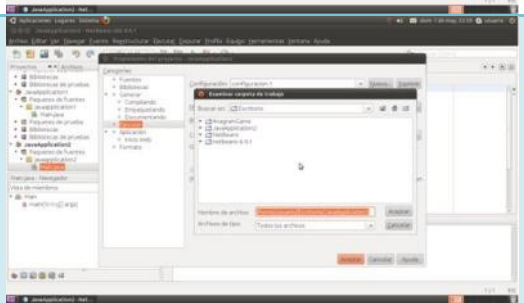


En este caso, escribimos "configuración 1" y pulsamos "aceptar"

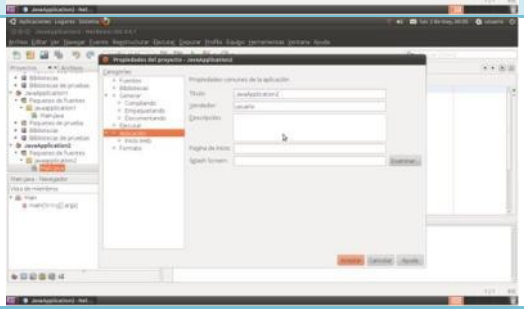
A partir de este momento, todas las opciones de configuración que seleccionemos que guardarán en "configuración 1"



Ahora podemos elegir la aplicación sobre la cual queremos aplicar la configuración personalizada de "configuración 1"

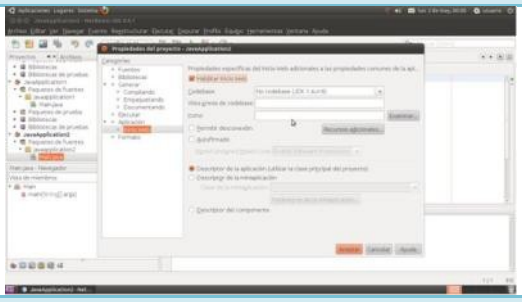


OPCIONES DE LA APLICACIÓN



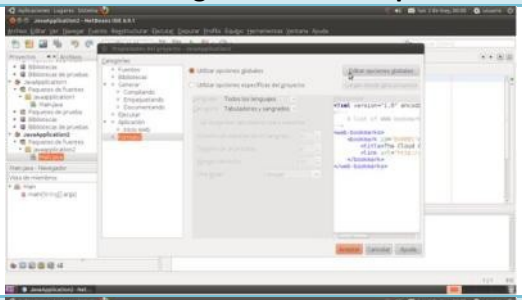
Como vemos, podemos dar una descripción al proyecto, cambiarle el nombre, etc…
Es conveniente hacerlo, ya que el nombre de los nuevos proyectos se generar automáticamente por NetBeans al inicio de la sesión.

En cuanto las opciones del inicio web:

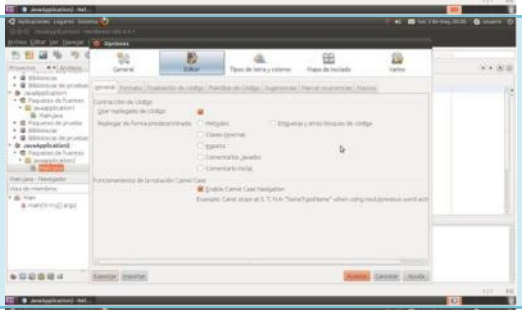


FORMATO
Aquí podemos personalizar aspectos globales del formato del código fuente en la aplicación.

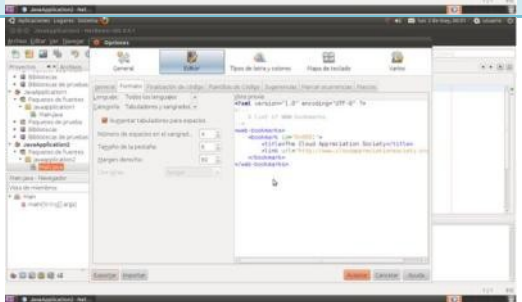
Podemos personalizar las opciones sólo para el proyecto actual o bien para todos los proyectos que estén basados en NetBeans a partir de ahora (utilizar opciones globales)



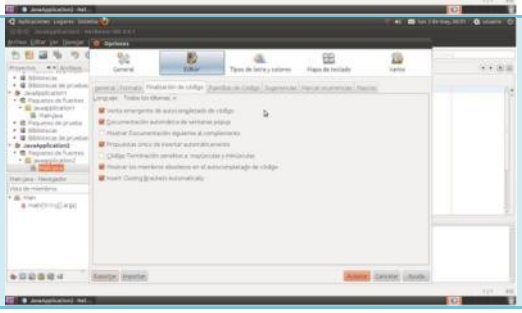
Si seleccionamos Editar opciones globales nos encontramos con la siguiente ventana, que tiene una barra superior de pestañas para configurar cada apartado del formato de forma independiente:



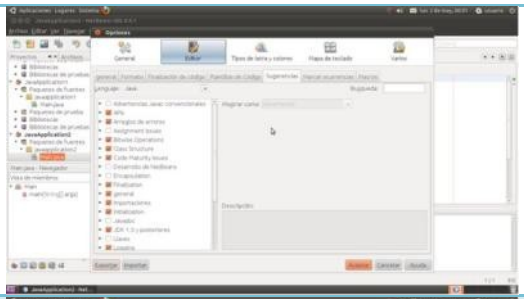
Pestaña Formato:
Se puede configurar los tamaños de los espaciados, pestañas, etc…



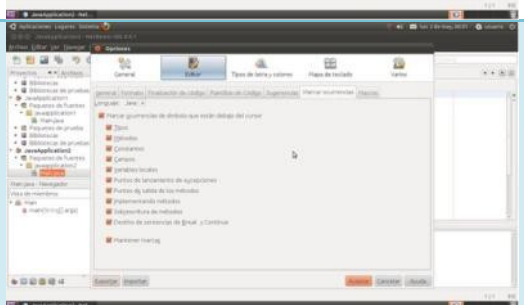
En la pestaña de Finalización de código:



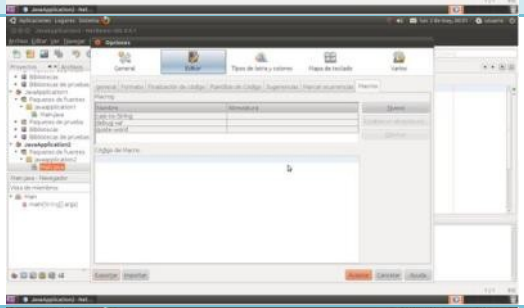
En la pestaña de sugerencias:



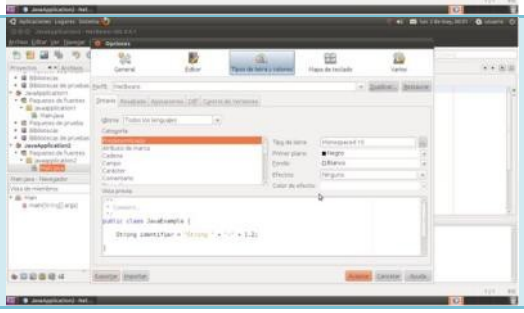
En la pestaña de Marcar ocurrencias:



En la pestaña de macros:

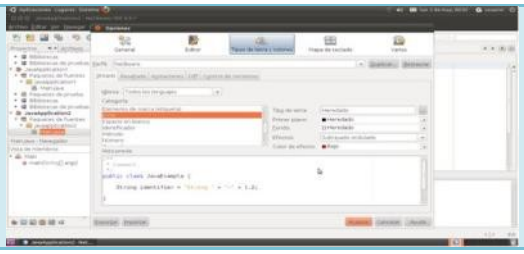


En cuanto al icono de Tipos de letra y colores:



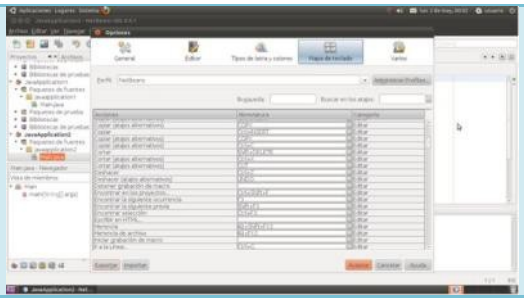
Consiste en elegir el tipo de letra y colores que prefiramos para el texto del código, así como efectos (si es que así lo deseamos)

También podemos configurar el tipo de letra y color de los errores del código (por defecto, de color rojo)



Y lo mismo con los números, espacios en blanco, etc…

En cuanto a los métodos abreviados de teclado (combinación de teclas equivalente a las acciones en NetBeans), podemos modificar aquellas acciones que hagamos con más frecuencia por aquella combinación de teclas que nos sea más fácil recordar.



7. Gestión de módulos

CASO PRÁCTICO

Después de haber probado a configurar algunos aspectos del entorno, ahora Juan desea empezar a programar. Tiene un trabajo pendiente en JavaScript, pero observa que, tristemente, este lenguaje no es soportado por NetBeans.

—¿Cómo que no? —Le dice Ana. —Basta con encontrar el módulo de JavaScript (estructuras del lenguaje más bibliotecas asociadas) y añadirlo como complemento al entorno. Entonces sí que podrás programar (también) en ese lenguaje.

A Juan le parece fascinante…

Con la plataforma dada por un entorno de desarrollo como NetBeans podemos hacer uso de módulos y plugins para desarrollar aplicaciones.

En la página oficial de NetBeans encontramos una relación de módulos y plugins, divididos en categorías.

Seleccionando la categoría Lenguajes de Programación, encontraremos aquellos módulos y plugins que nos permitan añadir nuevos lenguajes soportados por nuestro IDE.

Un módulo es un componente software que contiene clases de Java que pueden interactuar con las APIs del entorno de desarrollo y el manifest file, que es un archivo especial que lo identifica como módulo.

Los módulos se pueden construir y desarrollar de forma independiente. Esto posibilita su reutilización y que las aplicaciones puedan ser construidas a través de la inserción de módulos con finalidades concretas. Por esta misma razón, una aplicación puede ser extendida mediante la adición de módulos nuevos que aumenten su funcionalidad.

Existen en la actualidad multitud de módulos y plugins disponibles para todas las versiones de los entornos de desarrollo más utilizados. En las secciones siguientes veremos dónde encontrar plugins y módulos para NetBeans 6.9.1 que sean de algún interés para nosotros y las distintas formas de instalarlos en nuestro entorno.

También aprenderemos a desinstalar o desactivar módulos y plugins cuando preveamos que no los vamos a utilizar más y cómo podemos estar totalmente actualizados sin salir del espacio de nuestro entorno.

Veremos las categorías de plugins disponibles, su funcionalidad, sus actualizaciones…

REFLEXIONA

¿Cómo crees que influye el hecho de tener módulos y plugins disponibles en el éxito que tenga un IDE?

- (•) Contribuyen al éxito del entorno
- ( ) No influyen en el éxito del entorno

7.1 Añadir

CASO PRÁCTICO

Ya sabemos que podemos añadir funcionalidades a nuestro entorno. Pero ni Juan ni Ana saben cómo hacerlo. Piden ayuda a María, que decide ayudarles.

—Añadir módulos y plugins es muy sencillo, prestad atención.

Añadir un módulo va a provocar dotar de mayor funcionalidad a nuestros proyectos desarrollados en

NetBeans. Para añadir un nuevo módulo tenemos varias opciones:

a) Añadir algún módulo de los que NetBeans instala por defecto.
b) Descargar un módulo desde algún sitio web permitido y añadirlo.

c) Instalarlo on-line en el entorno.

Por supuesto, una cuarta posibilidad es crear el módulo nosotros mismos (aunque eso no lo veremos aquí).

Sin embargo, lo más usual es añadir los módulos o plugins que realmente nos interesan desde la web oficial de NetBeans. El plugin se descarga en formato .nbm que es el propio de los módulos en NetBeans. Posteriormente, desde nuestro IDE, cargaremos e instalaremos esos plugins. A esta manera de añadir módulos se le conoce como adición off-line.

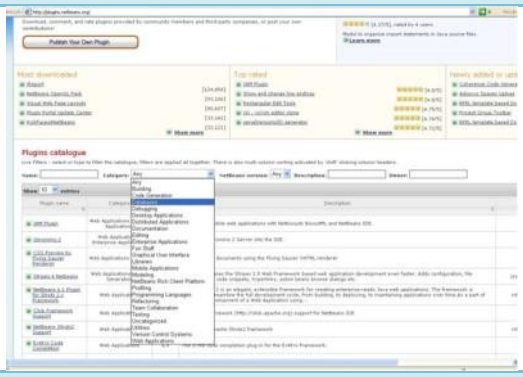
También es habitual instalarlos on-line, sin salir del IDE.

La adición on-line requiere tener instalado el plugin Portal Update Center en NetBeans 6.9.1 y consiste en instalar complementos desde nuestro mismo IDE, sin tener que descargarlos previamente.

A modo de ejemplo, a continuación se explican los pasos para añadir un módulo o plugin, de forma off-line (descargando el archivo e instalándolo posteriormente) y de forma on-line.

Adición de módulo en NetBeans.

Hay dos formas de añadir módulos y plugins en NetBeans:
***Off-line*: Buscar y descargar plugins desde la página web oficial de la plataforma:**
http://plugins.netbeans.org/
Ejemplo:
Vamos a buscar un plugin para jugar al sudoku desde nuestro IDE. No es muy educativo, pero sirva como ejemplo la manera en que se va a realizar el proceso (será igual en todos los casos):
Entramos en la zona de descargas de plugins para NetBeans y en la zona del catálogo, escribiremos la palabra sudoku:

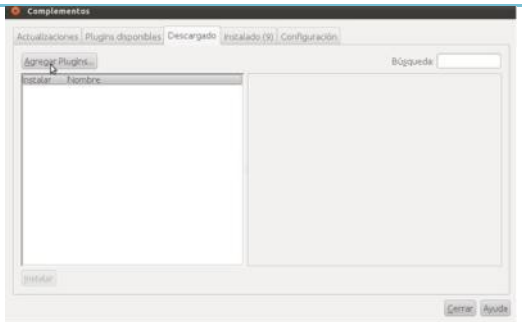


Se nos abre una ventana con las características del plugin y la opción de descargarlo. Elegimos la carpeta donde queramos que se guarde.

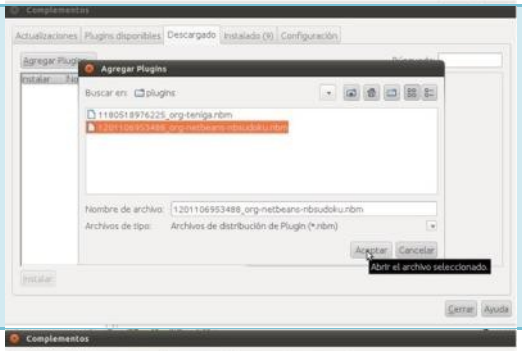
Entramos en NetBeans:

Creamos nuevo proyecto y seleccionamos el tipo de proyecto que queramos (por ejemplo, aplicación de Java).

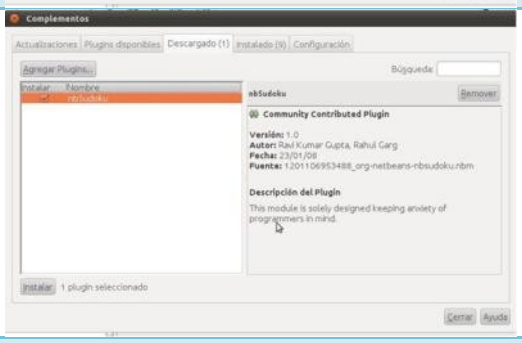
Herramientas - Complementos:



En la pestaña "descargado" seleccionamos "Agregar Plugins"



Seleccionamos la carpeta donde habíamos guardado el plugin del sudoku y le damos a "aceptar"



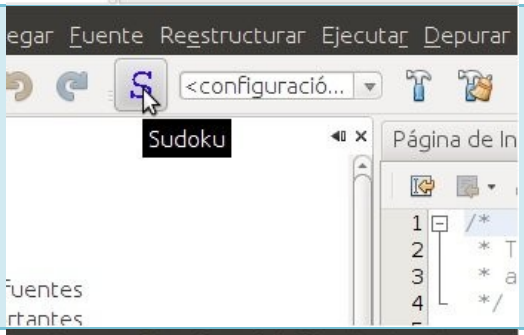
Estando el plugin seleccionado, pulsamos "instalar".

Empieza la instalación:

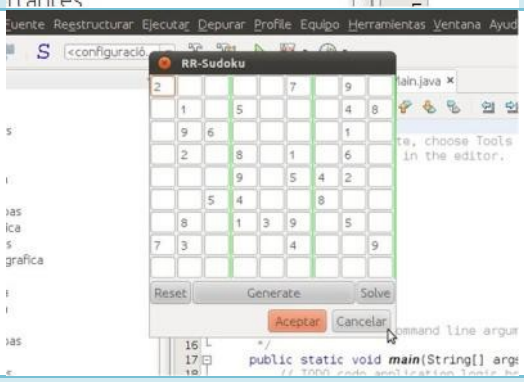
Pulsamos siguiente. Después, aceptamos la licencia:

Pulsamos "instalar"

Seleccionamos "Terminar"
Observamos el icono que aparece en la barra de iconos superior del sitio:



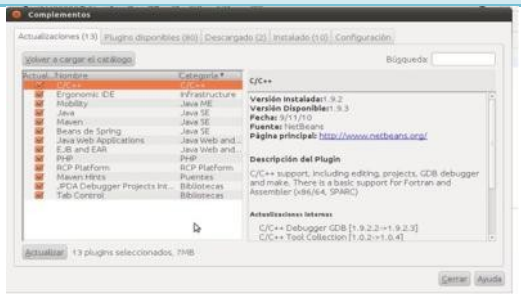
Si lo pulsamos, ya podemos jugar un ratito al sudoku para despejarnos:



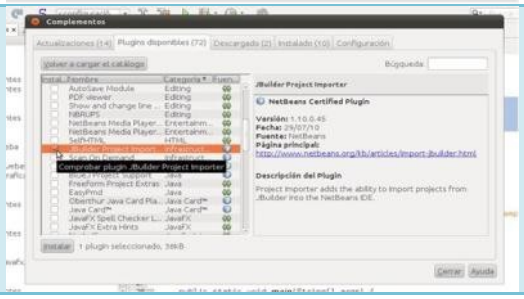
On-Line: Instalarlos desde el propio entorno de desarrollo:
Ahora vamos a instalar otros plugins con mayores utilidades que el anterior… vamos a hacer dos ejemplos instalando dos plugins diferentes:

- ✓ **Pdf Viewer: Nos permitirá abrir archivos en pdf desde el propio IDE, emergiendo una nueva ventana en el sitio específica para ello.**
- ✓ **Importador de bibliotecas y proyectos de JBuilder.**

Estando en NetBeans, seleccionamos Herramientas - Complementos:

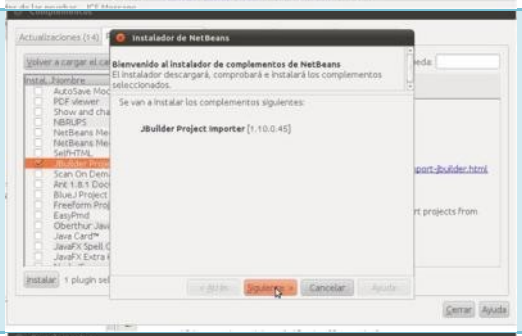


En la pestaña de plugins disponibles: seleccionamos JBuilder - Instalar

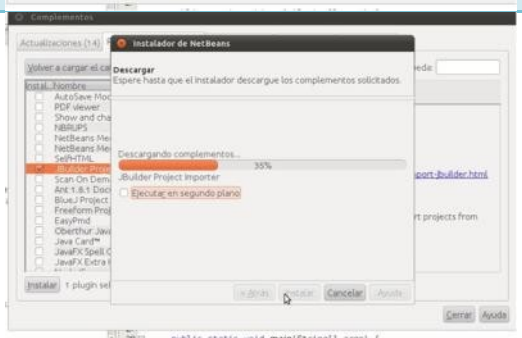


Se abre la siguiente ventana:

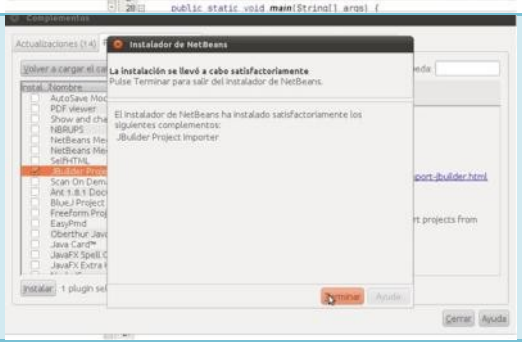
Aceptamos los términos de la licencia y pulsamos sobre Instalar.



Pulsamos siguiente

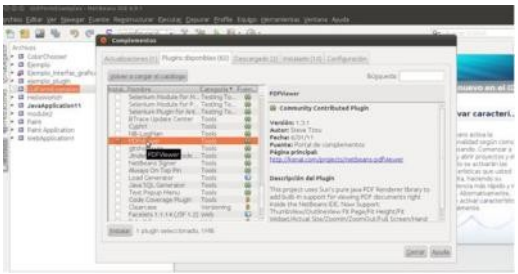


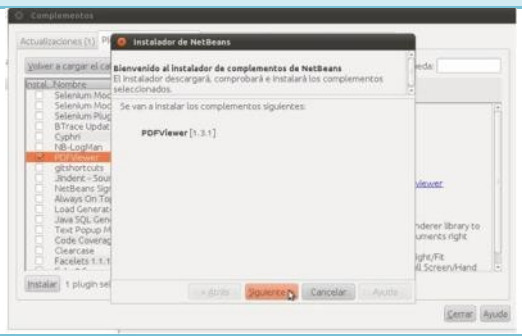
Pulsamos sobre Terminar.

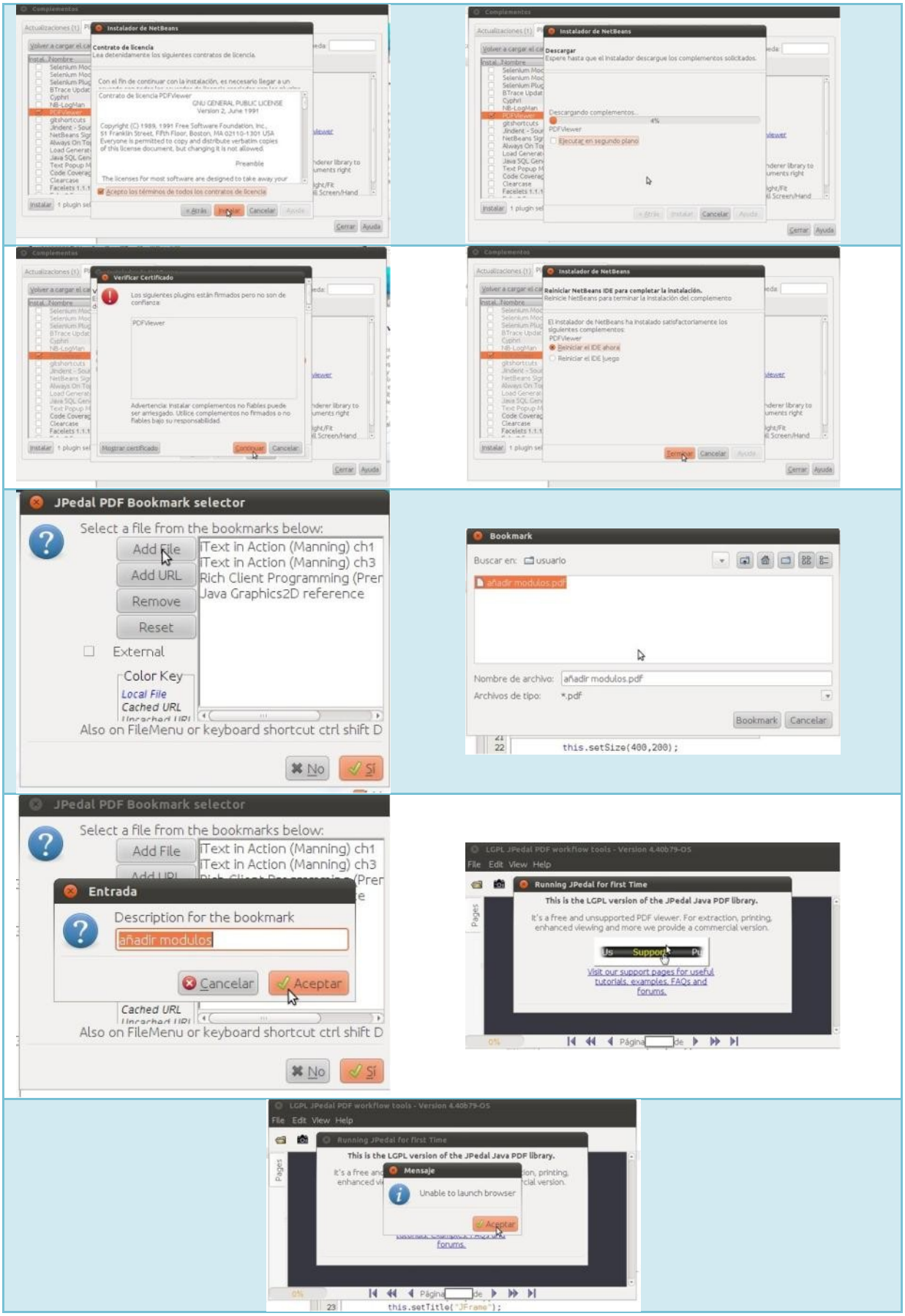


Ya tenemos el plugin instalado.

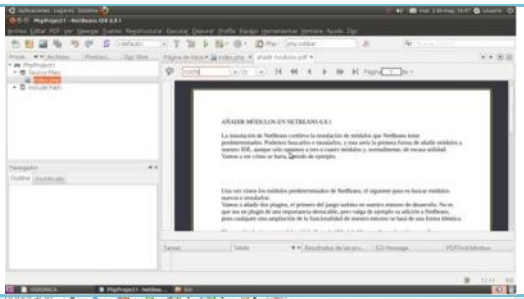
Con pdf Viewer:



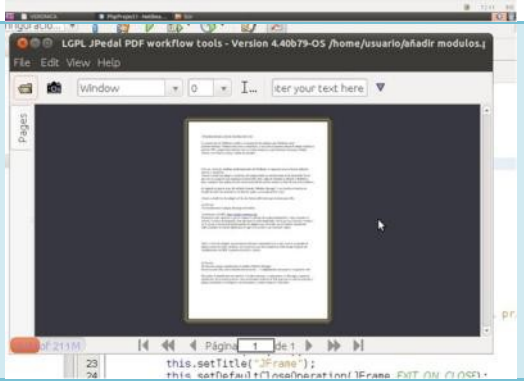



Instalador de NetBeans
Contrato de licencia
Lea detenidamente los siguientes contratos de licencia.
Acepto los términos de todos los contratos de licencia
Instalador de NetBeans
Descargar
Espere hasta que el instalador descargue los complementos solicitados.
Descargando complementos...
Ejecutar en segundo plano
Verificar Certificado
Los siguientes plugins están firmados pero no son de confianza:
PDFViewer
Instalador de NetBeans
Reiniciar NetBeans IDE para completar la instalación.
Reinicie NetBeans para terminar la instalación del complemento
JPedal PDF Bookmark selector
Select a file from the bookmarks below:
Add File
Add URL
Remove
Reset
External
Color Key
Local File
Cached URL
Also on FileMenu or keyboard shortcut ctrl shift D
iText in Action (Manning) ch1
iText in Action (Manning) ch3
Rich Client Programming (Prer
Java Graphics2D reference
No
Sí
Bookmark
Buscar en: usuario
añadir modulos.pdf
Nombre de archivo: añadir modulos.pdf
Archivos de tipo: *.pdf
Bookmark
Cancelar
this.setSize(400,200);
Entrada
Description for the bookmark
añadir modulos
Cancelar
Aceptar
LGPL JPedal PDF workflow tools - Version 4.40b79-OS
File Edit View Help
Running JPedal for first Time
This is the LGPL version of the JPedal Java PDF library.
It's a free and unsupported PDF viewer. For extraction, printing, enhanced viewing and more we provide a commercial version.
Visit our support pages for useful tutorials, examples, FAQs and forums.
Mensaje
Unable to launch browser
this.setTitle("JFrame");

Vemos el icono de pdf en la barra de herramientas.



Vemos también cómo NetBeans utiliza una ventana del entorno reservada al documento que se lee en pdf.



DEBES CONOCER

Navegar y familiarizarse por la plataforma web que NetBeans pone a disposición de los desarrolladores es fundamental para estar al día de las últimas funcionalidades que podemos añadir a nuestro entorno mediante la instalación de plugins.

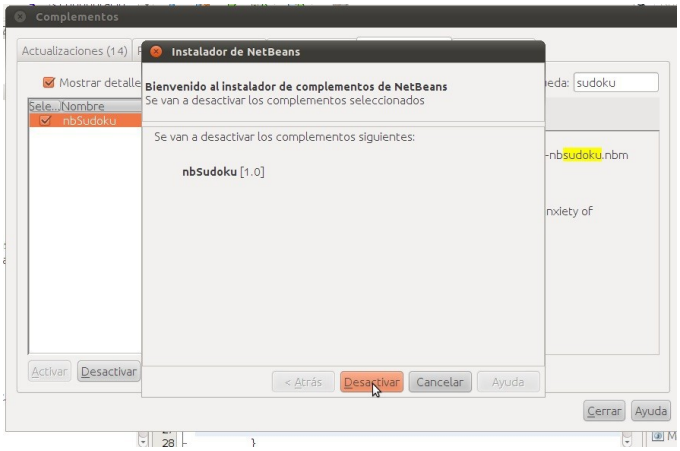
Búsqueda online de plugins para NetBeans http://plugins.netbeans.org/

7.2 Eliminar

Cuando consideramos que algún módulo o plugin de los instalados no nos aporta ninguna utilidad, o bien que el objetivo para el cual se añadió ya ha finalizado, el módulo deja de tener sentido en nuestro entorno. Es entonces cuando nos planteamos eliminarlo.

Eliminar un módulo es una tarea trivial que requiere seguir los siguientes pasos:

1. Encontrar el módulo o plugin dentro de la lista de complementos instalados en el entorno.
2. A la hora de eliminarlo, tenemos dos opciones:
   a) Desactivarlo: El módulo o plugin sigue instalado, pero en estado inactivo (no aparece en el entorno).
   b) Desinstalarlo: El módulo o plugin se elimina físicamente del entorno de forma permanente.



Esta es la ventana, desde el gestor de complementos de NetBeans, que nos aparece cuando queremos eliminar un módulo del entorno.

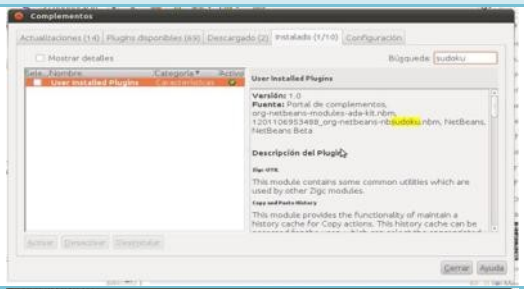
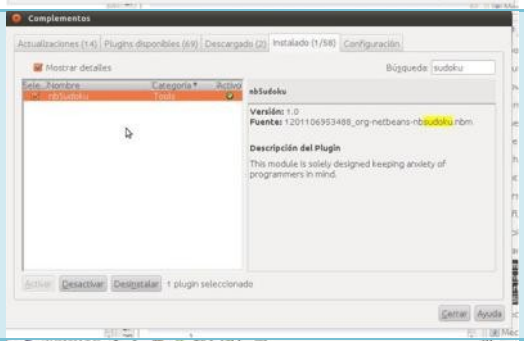
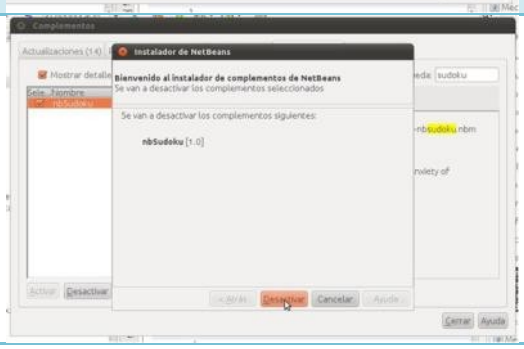
Siempre nos pedirá elegir entre dos opciones:

desactivar o desinstalar.

En este ejemplo, se opta por desactivar el complemento, como podemos ver en la

imagen.

Para ver el ejemplo completo de desactivación de un complemento, se indican los pasos a seguir:

Eliminar módulos en NetBeans	
Vamos a ver la secuencia de pasos a seguir para eliminar el plugin del juego del sudoku del entorno. **El proceso es muy sencillo: basta con conseguir la lista de complementos instalados (Herramientas - Complementos). Localizamos el complemento que queremos eliminar escribiendo su nombre en el lugar destinado para ello y seleccionamos una de entre las dos opciones posibles: desinstalarlo o desactivarlo**	
En la pestaña de complementos instalados, escribimos el nombre del plugin (sudoku) en la barra de búsqueda:	
Cuando lo encuentra, en la ventana aparecen las dos posibilidades de eliminación:	
En este caso, hemos optado por desactivarlo.	

AUTOEVALUACIÓN:

Para añadir un módulo desde la web oficial de NetBeans:

a) Hay que instalar el plugin Update Center.
b) Hay que conectar con la web desde Netbeans y instalar on-line.
c) Hay que encontrar el complemento, descargarlo y luego instalarlo en el IDE.
d) No se pueden descargar los complementos desde ahí.

7.3 Funcionalidades

CASO PRÁCTICO

—Para que sepas qué puedes encontrar en los complementos de NetBeans, te recomiendo que tengas claras las funcionalidades que ofrece, teniendo en cuenta que se van ampliando día a día, —le comenta Ana a Juan.

Los módulos y plugins disponibles para los entornos de desarrollo, en sus distintas versiones, tienen muchas y muy variadas funciones.

Podemos clasificar las distintas categorías de funcionalidades de módulos y plugins en los siguientes grupos:

1. Construcción de código: facilitan la labor de programación.
2. Bases de datos: ofrecen nuevas funcionalidades para el mantenimiento de las aplicaciones.
3. Depuradores: hacen más eficiente la depuración de programas.
4. Aplicaciones: añaden nuevas aplicaciones que nos pueden ser útiles.
5. Edición: hacen que los editores sean más precisos y más cómodos para el programador.
6. Documentación de aplicaciones: para generar documentación de los proyectos en la manera deseada.
7. Interfaz gráfica de usuario: para mejorar la forma de presentación de diversos aspectos del entorno al usuario.
8. Lenguajes de programación y bibliotecas: para poder programar bajo un Lenguaje de Programación que, en principio, no soporte la plataforma.
9. Refactorización: hacer pequeños cambios en el código para aumentar su legibilidad, sin alterar su función.
10. Aplicaciones web: para introducir aplicaciones web integradas en el entorno.
11. Prueba: para incorporar utilidades de pruebas al software.

REFLEXIONA
¿Qué categoría de funcionalidad de NetBeans te parece más interesante? ¿Por qué?

- ○ Todas son igual de interesantes porque aumentan la funcionalidad.
- ◉ Depende de la tarea a realizar y el nivel del usuario.

PARA SABER MÁS
En el siguiente vídeo, se hace un repaso de la adición de nuevas funcionalidades a NetBeans:
Adicionar funcionalidades a NetBeans www.youtube.com/watch?v=8icMxyazHHk

7.4 Herramientas concretas

- ✓ ***Importador de Proyectos de NetBeans***: permite trabajar en lenguajes como JBuilder.
- ✓ ***Servidor de aplicaciones GlassFish***: Proporciona una plataforma completa para aplicaciones de tipo empresarial.
- ✓ ***Soporte para Java Enterprise Edition***: Cumplimiento de estándares, facilidad de uso y la mejora de rendimiento hacen de NetBeans la mejor herramienta para crear aplicaciones de tipo empresarial de forma ágil y rápida.
- ✓ ***Facilidad de uso a lo largo de todas las etapas del ciclo de vida del software***.
- ✓ ***NetBeans Swing GUI builder***: simplifica mucho la creación de interfaces gráficos de usuarios en aplicaciones cliente y permite al usuario manejar diferentes aplicaciones sin salir del IDE.
- ✓ ***NetBeans Profiler***: Permite ver de forma inmediata ver cómo de eficiente trabajará un trozo de software para los usuarios finales.
- ✓ ***El editor WSDL*** facilita a los programadores trabajar en servicios Web basados en XML.
- ✓ ***El editor XML Schema Editor*** permite refinar aspectos de los documentos XML de la misma manera que el editor WSDL revisa los servicios Web.
- ✓ Aseguramiento de la seguridad de los datos mediante el ***Sun Java System Acces Manager***.
- ✓ ***Soporte beta de UML*** que cubre actividades como las clases, el comportamiento, la interacción y las secuencias.
- ✓ ***Soporte bidireccional***, que permite sincronizar con rapidez los modelos de desarrollo con los cambios en el código conforme avanzamos por las etapas del ciclo de vida de la aplicación.

- ✓ Etc.

PARA SABER MÁS

Amplía las herramientas concretas que ofrece NetBeans para el desarrollo de aplicaciones multiplataforma.

Visita la web oficial:

Información herramientas concretas de NetBeans http://netbeans.org/kb/kb.html

AUTOEVALUACIÓN

¿En qué fases del desarrollo de software ayudan los entornos integrados de desarrollo?

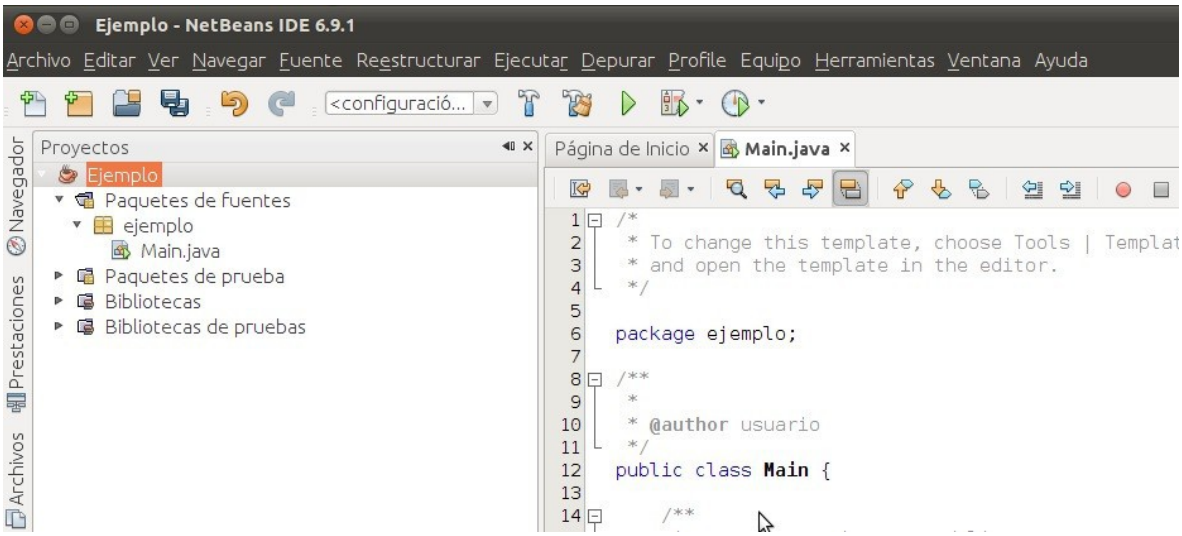
- (•) En codificación, pruebas, documentación, explotación y mantenimiento.
- ( ) En codificación y documentación.
- ( ) En análisis y documentación.

8. Uso básico de entornos de desarrollo

CASO PRÁCTICO

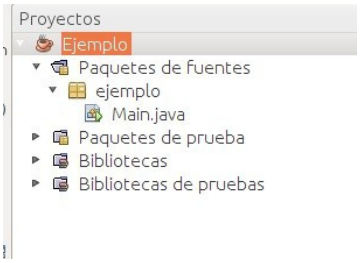
—En qué partes se divide el espacio principal del entorno? Vamos a echar un vistazo, —le comenta Juan a Antonio. (A Juan le gusta explicárselo a su compañero, ahora que va descubriendo las ventajas de los IDE…).

En el sitio principal del entorno de desarrollo de NetBeans nos encontramos con la siguiente ventana, que aparece cuando seleccionamos archivo, nuevo proyecto, java:



Vemos que el espacio se divide en dos ventanas principales.

Ventana Izquierda: ventana de proyectos.



Aquí irá apareciendo la relación de proyectos, archivos, módulos o clases que vayamos abriendo durante la sesión.

Cada proyecto comprende una serie de archivos y bibliotecas que lo componen.

El principal archivo del proyecto Java es el llamado Main.java.

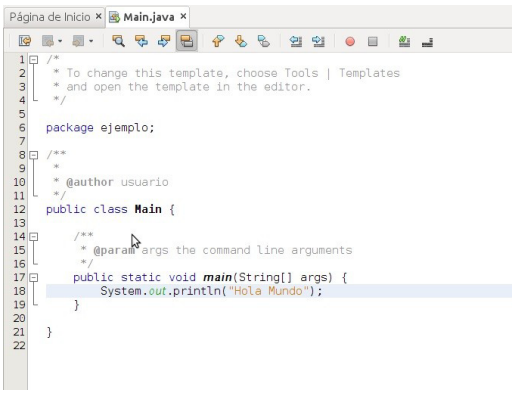
Ventana derecha: espacio de escritura de los códigos de los proyectos.

Aquí aparece el esqueleto propio de un programa escrito en lenguaje Java.

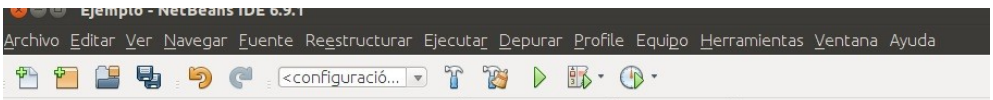
Se ha añadido el código:

System.out.println(“Hola Mundo”);

Y veremos su significado en las siguientes páginas. De momento, saber que para escribir cualquier código, hay que hacerlo en esta ventana.



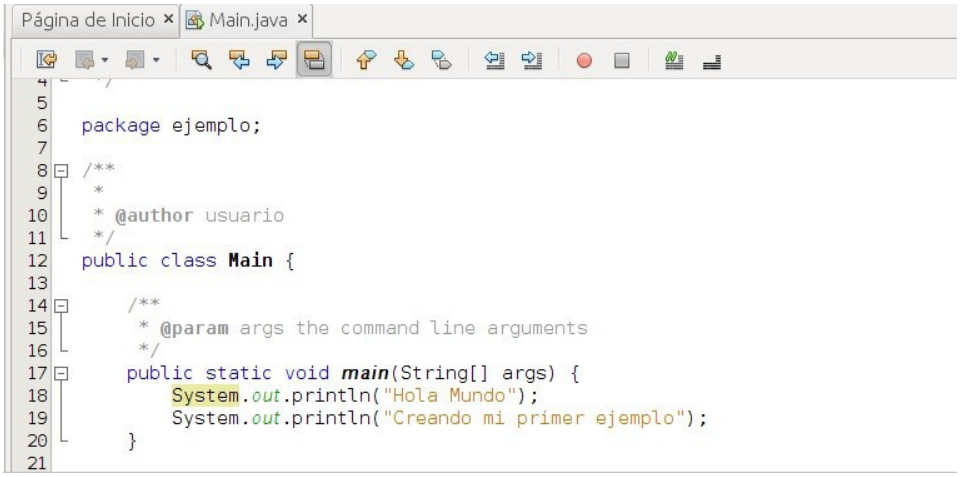
BARRA DE HERRAMIENTAS: Desde aquí podremos acceder a todas las opciones del IDE.



8.1 Edición de Programas

CASO PRÁCTICO

—Vamos a hacer el primer ejemplo —comenta Ana, entusiasmada—.
Después de todo, no debemos perder de vista la finalidad de la herramienta, ESCRIBIR PROGRAMAS!



En este sencillo ejemplo se ve una modificación de las líneas de código en la ventana de codificación del archivo Main.java del proyecto **ejemplo** que acabamos de crear.

Las dos líneas que aparecen resaltadas se han escrito sobre la ventana y, tal y como significan en lenguaje Java, su ejecución implicará que sendos mensajes encerrados entre comillas y entre paréntesis saldrán impresos.

No hay que decir que la programación en Java no es objeto del presente módulo, pero puedes probar con algunos ejemplos en Java que tengas de otros módulos.

Mientras escribimos en el editor de textos nos percatamos de varias características de NetBeans que ya hemos señalado en páginas anteriores:

- ✓ Autocompletado de código.
- ✓ Coloración de comandos.
- ✓ Subrayado en rojo cuando hay algún error y posibilidad de depuración y corrección de forma visual, mediante un pequeño icono que aparece a la izquierda de la línea defectuosa.

DEBES CONOCER

El proceso de edición de un programa desde que arranca el entorno hasta que está libre de errores sintácticos.

```
package ejemplo;

import javax.swing.JFrame;
import javax.swing.JLabel;

public class Main
extends JFrame {

     public Main()  {
          JLabel lblSaludo = new JLabel( "Hola Mundo. Creando mi primer ejemplo")
          add(lblSaludo);

          this.setSize(200,200);
          this.setTitle("JFrame");
          this.setDeafaultCloseOperation(JFrame.EXIT_ON_CLOSE);
     }
```

```
    public static void main(String[] args)  {
        Main main = new Main();
    }
}
```

8.2 Generación de Ejecutables

Una vez tenemos el código plasmado en la ventana de comandos y libre de errores de sintaxis, los siguientes pasos son: compilación, depuración, ejecución.

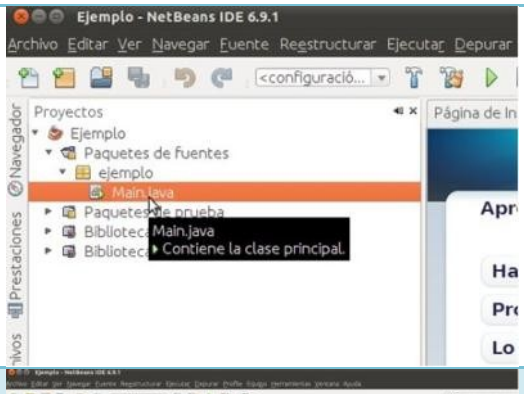
Al ejecutar el ejemplo anterior, el resultado es:

Si a este ejemplo le añadimos la funcionalidad de JFrame, el resultado de la ejecución es:

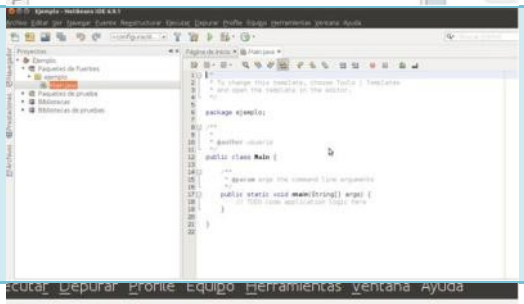
Estos ejemplos aparecen detallados en el siguiente apartado:

Ejemplo de edición de código
En este documento vamos a introducirnos en la edición de programas en NetBeans a través de un ejemplo sencillo de una aplicación de Java.
Lo primero es iniciar la plataforma: 
Seleccionamos archivo - nuevo proyecto. Elegimos una aplicación de Java: 

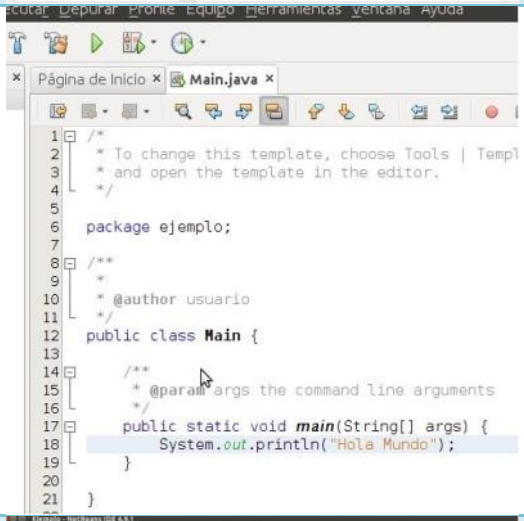
Lo vamos a llamar ejemplo.
Una vez iniciado el proyecto, en la ventana de proyectos (izquierda) vemos cómo se ha cargado el proyecto ejemplo. Lo seleccionamos con el ratón y se despliega, mostrando todos sus archivos componentes. Seleccionamos Main.java (que es el archivo principal del proyecto, el cual vamos a editar):



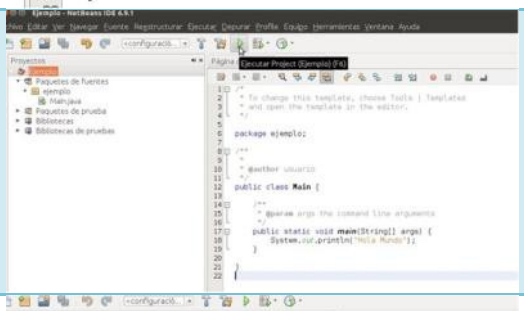
En la ventana de edición (a la derecha) nos aparece el esqueleto de la estructura básica de una aplicación en Java.
Lo que vamos a hacer a lo largo del ejemplo es añadir código.



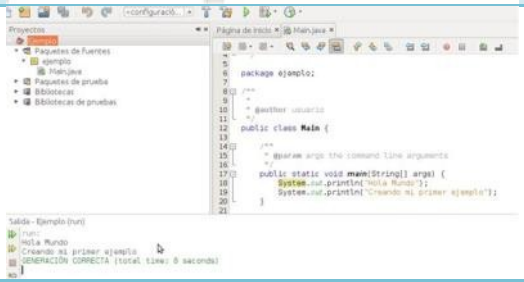
La primera línea de código que vamos a agregar es una orden sencilla en Java, cuya ejecución posterior dará lugar a la aparición de un mensaje por pantalla.



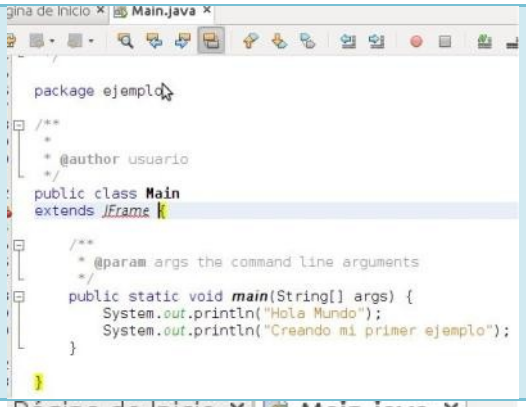
La apariencia del IDE será la siguiente:



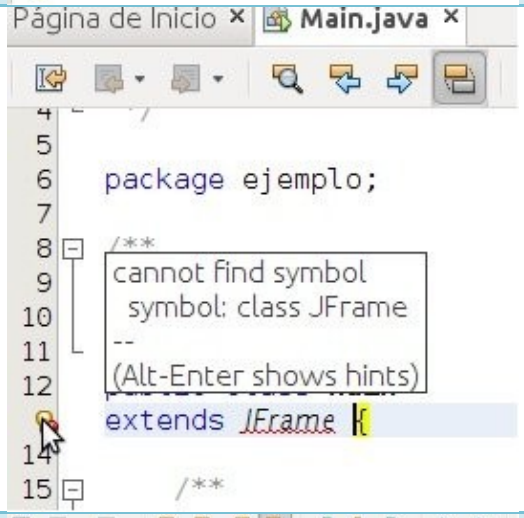
Añadimos otra línea más con otro mensaje "Creando mi primer ejemplo"



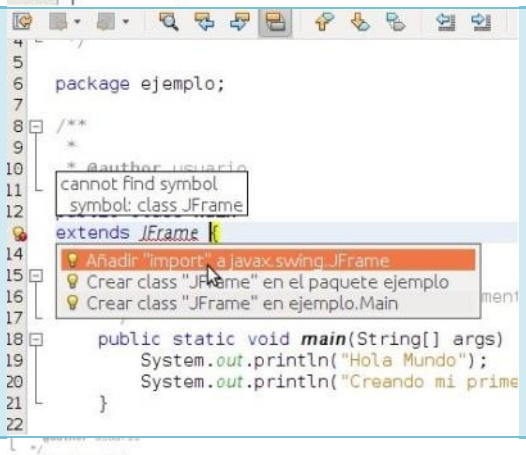
Ahora vamos a modificar la parte de arriba del programa. Añadimos la siguiente línea:



Esta línea nos va a servir para adentrarnos en una de las utilidades más importantes de NetBeans 6.9.1.
NetBeans entiende esta orden como un error (aparece subrayada en una línea roja ondulada y con un pequeño icono al lado izquierdo)

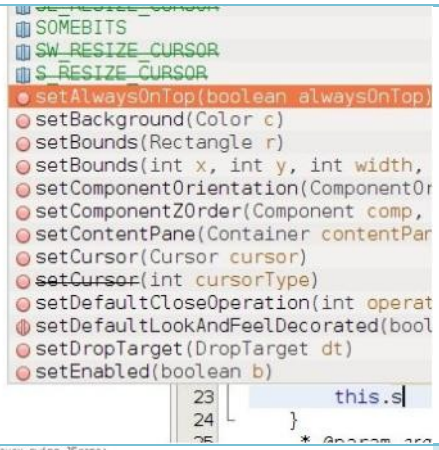


Si pulsamos sobre ese icono con el ratón, NetBeans nos aporta sugerencias para deshacer el error:



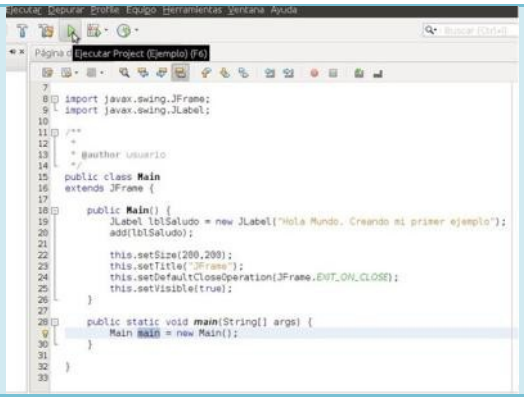
En este caso, elegimos importar JFrame a la librería.
Y seguimos añadiendo código en el editor:

Se nos vuelven a subrayar líneas en rojo, actuamos igual que en el caso anterior y vamos viendo las sugerencias que nos dan para corregir. También vamos viendo las opciones de autocompletado de código:



Llegados a este punto, ya hemos comprobado que el editor no nos da ningún problema más. En el siguiente punto del tema, veremos cómo ejecutar esto.

Vemos también cómo se han importando con éxito las librerías que nos han hecho falta:

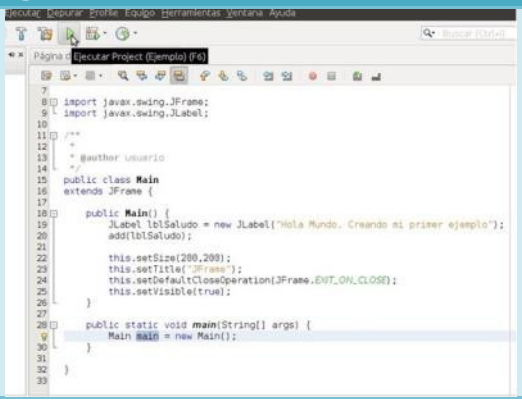


El código completo del ejemplo es el siguiente:

```
package ejemplo;
import javax.swing.JFrame;
import javax.swing.JLabel;
public class Main extends JFrame {
   public Main()  {
      JLabel lblSaludo = new JLabel( "Hola
Mundo. Creando mi primer ejemplo")
      add(lblSaludo);
      this.setSize(200,200);
      this.setTitle("JFrame");
      this.setDefaultCloseOperation(
JFrame.EXIT_ON_CLOSE);
   }
   public static void main(String[] args)  {
      Main main = new Main();
   }
}
```

Ejecución de un programa en NetBeans

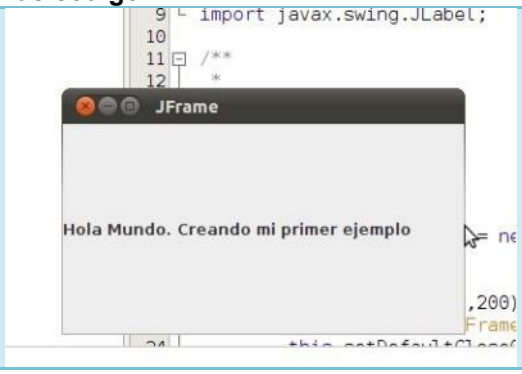
Continuando con el ejemplo anterior, recuerda que habíamos llegado a este punto:



Tenemos el programa escrito en el editor libre de errores sintácticos.
¿Cómo convertir ese programa en ejecutable?
Cabe destacar que, por la sencillez y pequeñez del programa, la ejecución del mismo podría ser directa sin ningún problema.
Sin embargo, debemos acostumbrarnos a seguir los pasos adecuados, que son:

- ✓ **Editor libre de errores → Compilación → Depuración → Ejecución**
- ✓ **Para compilar un programa, debemos seleccionar ejecutar (en la barra superior de herramientas) → Compile File**
- ✓ **Depurar → Barra de herramientas**
- ✓ **Ejecutar → En la barra de herramientas o bien mediante el icono de acceso directo en la parte superior de la ventana de edición de código.**

El resultado que obtenemos (si todo ha ido bien) es:



AUTOEVALUACIÓN:

Los pasos que debemos dar para generar un ejecutable son:

- ○ Ejecución directa
- ○ Ejecución, una vez que el editor esté libre de errores sintácticos.
- ◉ Una vez que el editor esté libre de errores, compilar, depurar y ejecutar.

9. Actualización y mantenimiento de entornos de desarrollo

CASO PRÁCTICO

—Por último, es de vital importancia el mantener y actualizar el entorno de desarrollo —comenta Ana—. Deberíamos tener permanentemente actualizados todos los complementos y realizar un correcto mantenimiento a las bases de datos asociadas a nuestros proyectos.

El mantenimiento del entorno de desarrollo es una tarea fundamental que requiere tener todos sus componentes periódicamente actualizados.

También es de vital importancia realizar copias de seguridad sobre las bases de datos de nuestros proyectos por si ocurriera algún error o proceso defectuoso poder restaurarlos.

El mantenimiento y las actualizaciones se hacen de forma on-line. En NetBeans contamos con el complemento llamado ***Auto Update Services***. Lo podemos encontrar en el siguiente enlace:

Complementos de Netbeans http://plugins.netbeans.org

DESTACADO

Para añadir módulos y plugins on-line, hay que tener este complemento instalado en el entorno.

DESTACADO

La gestión de las bases de datos asociadas a nuestros proyectos es muy importante. Habrá que realizarles copias de seguridad periódicamente, para asegurar su restauración en caso de fallos en el sistema, y mantenerlas actualizadas para su posible portabilidad futura a nuevas versiones del entorno que utilicemos.

REFLEXIONA

¿Cuál es la razón, en tu opinión, de que salgan nuevas versiones de los entornos de desarrollo tan rápidamente?

- ○ Para adaptarse a la evolución del hardware
- ◉ Para incluir y modificar funcionalidades del entorno

TEMA 3

Contenido

[DISEÑO Y REALIZACIÓN DE PRUEBAS]

José Luis Comesaña Cabeza --- 2011/2012

Entornos de Desarrollo del curso de "*Desarrollo de Aplicaciones Web*"

DISEÑO Y REALIZACIÓN DE PRUEBAS.

Caso práctico
Situación
BK programación se encuentra desarrollando la primera versión de la aplicación de gestión hotelera.
Ada, la supervisora de proyectos de BK Programación, se reúne con Juan y María para empezar a planificar el diseño y realización de pruebas sobre la aplicación,
Ana se va a encargar, de ir probando los distintas partes de la aplicación que se van desarrollando. Para ello usará casos de prueba diseñados por Juan, y evaluará los resultados.
Juan evaluará los resultados de las pruebas realizadas por Ana, y en su caso, realizará las modificaciones necesarias en el proyecto.

1.- Planificación de las pruebas.

Caso práctico
Todos en la empresa están inmersos en el desarrollo de la aplicación de gestión hotelera. Para garantizar la corrección del desarrollo, Ada propone establecer la planificación de las pruebas. ¿Por qué hay que probar el software? ¿Es necesario seguir un orden concreto en la realización de pruebas? ¿Qué pruebas se realizan?

Durante todo el proceso de desarrollo de software, desde la fase de diseño, en la implementación y una vez desarrollada la aplicación, es necesario realizar un conjunto de pruebas (*Proceso que permite verificar y revelar la calidad de un producto software. Se utilizan para identificar posibles fallos de implementación, calidad o usabilidad de un programa*), que permitan verificar que el software que se está creando, es correcto y cumple con las especificaciones impuesta por el usuario.

En el proceso de desarrollo de software, nos vamos a encontrar con un conjunto de actividades, donde es muy fácil que se produzca un error humano. Estos errores humanos pueden ser: una incorrecta especificación de los objetivos, errores producidos durante el proceso de diseño y errores que aparecen en la fase de desarrollo.

Mediante la realización de pruebas se software, se van a realizar las tareas de verificación (*Proceso por el que se comprueba que el software cumple los requisitos especificados*) y validación (*Proceso que comprueba si el software hace lo que el usuario deseaba. Tiene que estar verificado*) del software. La **verificación** es la comprobación que un sistema o parte de un sistema, cumple con las condiciones impuestas. Con la verificación se comprueba si la aplicación se está construyendo correctamente. La **validación** es el proceso de evaluación del sistema o de uno de sus componentes, para determinar si satisface los requisitos especificados.

Para llevar a cabo el proceso de pruebas, de manera eficiente, es necesario implementar una estrategia de pruebas. Siguiendo el Modelo en Espiral (*Modelo de ciclo de vida de ciclo de vida del software, en el que las actividades se conforman en una espiral. Cada bucle o iteración representa un conjunto de actividades*), las pruebas empezarían con la prueba de unidad, donde se analizaría el código implementado y seguiríamos en la prueba de integración, donde se prestan atención al diseño y la construcción de la arquitectura del software. El siguiente paso sería la prueba de validación, donde se comprueba que el sistema construido cumple con lo establecido en el análisis de requisitos de software. Finalmente se alcanza la prueba de sistema que verifica el funcionamiento total del software y otros elementos del sistema.

2.- Tipos de prueba.

Caso práctico
Juan y María están implementando la mayor parte de la aplicación. ¿Es correcto lo realizado hasta ahora? ¿Cómo se prueba los valores devueltos por una función o método? ¿Es posible seguir la ejecución de un programa, y ver si toma los caminos diseñados?

No existe una clasificación oficial o formal, sobre los diversos tipos de pruebas de software. En la ingeniería del software, nos encontramos con dos enfoques fundamentales:

- **Prueba de la Caja Negra** (Black Box Testing): cuando una aplicación es probada usando su interfaz externa, sin preocuparnos de la implementación de la misma. Aquí lo fundamental es comprobar que los resultados de la ejecución de la aplicación, son los esperados, en función de las entradas que recibe.
- **Prueba de la Caja Blanca** (White Box Testing): en este caso, se prueba la aplicación desde dentro, usando su lógica de aplicación.

Una prueba de tipo **Caja Negra** se lleva a cabo sin tener que conocer ni la estructura, ni el funcionamiento interno del sistema. Cuando se realiza este tipo de pruebas, solo se conocen las entradas adecuadas que deberá recibir la aplicación, así como las salidas que les correspondan, pero no se conoce el proceso mediante el cual la aplicación obtiene esos resultados.

En contraposición a lo anterior, una prueba de **Caja Blanca**, va a analizar y probar directamente el código de la aplicación. Como se deriva de lo anterior, para llevar a cabo una prueba de Caja Blanca, es necesario un conocimiento específico del código, para poder analizar los resultados de las pruebas.

Tipos de pruebas de software	
Pruebas de unidad	Con ella se va a probar el correcto funcionamiento de un módulo de código
Pruebas de carga	Este es el tipo más sencillo de pruebas de rendimiento. Una prueba de carga se realiza generalmente para observar el comportamiento de una aplicación bajo una cantidad de peticiones esperada. Esta carga puede ser el número esperado de usuarios concurrentes utilizando la aplicación y que realizan un número específico de transacciones durante el tiempo que dura la carga. Esta prueba puede mostrar los tiempos de respuesta de todas las transacciones importantes de la aplicación. Si la base de datos, el servidor de aplicaciones, etc también se monitorizan, entonces esta prueba puede mostrar el cuello de botella en la aplicación.
Prueba de estrés	Esta prueba se utiliza normalmente para romper la aplicación. Se va doblando el número de usuarios que se agregan a la aplicación y se ejecuta una prueba de carga hasta que se rompe. Este tipo de prueba se realiza para determinar la solidez de la aplicación en los momentos de carga extrema y ayuda a los administradores para determinar si la aplicación rendirá lo suficiente en caso de que la carga real supere a la carga esperada.
Prueba de estabilidad	Esta prueba normalmente se hace para determinar si la aplicación puede aguantar una carga esperada continuada. Generalmente esta prueba se realiza para determinar si hay alguna fuga de memoria en la aplicación
Pruebas de picos	La prueba de picos, como el nombre sugiere, trata de observar el comportamiento del sistema variando el número de usuarios, tanto cuando bajan, como cuando tiene cambios drásticos en su carga.

	Esta prueba se recomienda que sea realizada con un software automatizado que permita realizar cambios en el número de usuarios mientras que los administradores llevan un registro de los valores a ser monitorizados
Prueba estructural	El enfoque estructural o de caja blanca se centra en la estructura interna del programa (analiza los caminos de ejecución).
Prueba funcional	El enfoque funcional o de caja negra se centra en las funciones, entradas y salidas que recibe y produce un módulo o función concreta.
Pruebas aleatorias	El enfoque aleatorio consiste en utilizar modelos (en muchas ocasiones estadísticos) que representen las posibles entradas al programa para crear a partir de ellos los casos de prueba.
Pruebas de regresión	Son cualquier tipo de pruebas de software que intentan descubrir las causas de nuevos errores, carencias de funcionalidad, o divergencias funcionales con respecto al comportamiento esperado del software, inducidos por cambios recientemente realizados en partes de la aplicación que anteriormente al citado cambio no eran propensas a este tipo de error. Esto implica que el error tratado se produce como consecuencia inesperada del citado cambio en el programa.

Reflexiona

Resulta habitual, que en una empresa de desarrollo de software se gaste el 40 por ciento del esfuerzo de desarrollo en la prueba ¿Por qué es tan importante la prueba? ¿Qué tipos de errores se intentan solucionar con las pruebas?

Las pruebas son muy importantes, ya que permiten descubrir errores en un programa, fallos en la implementación, calidad o usabilidad del software, ayudando a garantizar la calidad.

Con las pruebas se intenta verificar que cada componente que se ha diseñado, ya sea un método, función, módulo, etc. realiza la función para la que se ha diseñado. También se intenta comprobar, que existen condiciones en las que todos los caminos de una aplicación llegan a ejecutarse.

2.1.- Funcionales.

Estamos ante pruebas de la caja negra. Se trata de probar, si las salidas que devuelve la aplicación, o parte de ella, son las esperadas, en función de los parámetros de entrada que le pasemos. No nos interesa la implementación del software, solo si realiza las funciones que se esperan de él.

Las pruebas funcionales siguen el enfoque de las pruebas de Caja Negra. Comprenderían aquellas actividades cuyo objetivo sea verificar una acción específica o funcional dentro del código de una aplicación. Las pruebas funcionales intentarían responder a las preguntas ¿puede el usuario hacer esto? o ¿funciona esta utilidad de la aplicación?

Su principal cometido, va a consistir, en comprobar el correcto funcionamiento de los componentes de la aplicación informática. Para realizar este tipo de pruebas, se deben analizar las entradas y las salidas de cada componente, verificando que el resultado es el esperado. Solo se van a considerar las entradas y salidas del sistema, sin preocuparnos por la estructura interna del mismo.

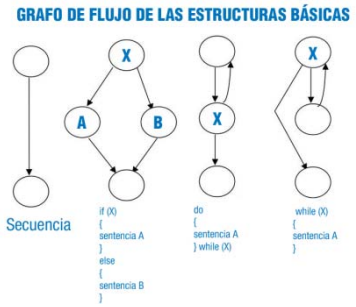
Si por ejemplo, estamos implementando una aplicación que realiza un determinado cálculo científico, en el enfoque de las pruebas funcionales, solo nos interesa verificar que ante una determinada entrada a ese programa el resultado de la ejecución del mismo devuelve como resultado los datos esperados. Este tipo de prueba, no consideraría, en ningún caso, el código desarrollado, ni el algoritmo (*Conjunto ordenado de pasos a seguir para la resolución de un problema*), ni la eficiencia, ni si hay partes del código innecesarias, etc.

Dentro de las pruebas funcionales, podemos indicar tres tipos de pruebas:

- **Particiones equivalentes**: La idea de este tipo de pruebas funcionales, es considerar el menor número posible de casos de pruebas, para ello, cada caso de prueba tiene que abarcar el mayor número posible de entradas diferentes. Lo que se pretende, es crear un conjunto de clases de equivalencia, donde la prueba de un valor representativo de la misma, en cuanto a la verificación de errores, sería extrapolable al que se conseguiría probando cualquier valor de la clase.
- **Análisis de valores límite**: En este caso, a la hora de implementar un caso de prueba, se van a elegir como valores de entrada, aquellos que se encuentra en el límite de las clases de equivalencia.
- **Pruebas aleatorias**: Consiste en generar entradas aleatorias para la aplicación que hay que probar. Se suelen utilizar generadores de prueba, que son capaces de crear un volumen de casos de prueba al azar, con los que será alimentada la aplicación. Esta tipo de pruebas, se suelen utilizar en aplicaciones que no sean interactivas, ya que es muy difícil generar las secuencias de entrada adecuadas de prueba, para entornos interactivos.

Existen otros tipos de pruebas funcionales, aunque todas comparten un mismo objetivo, y es comprobar, solo actuando en la interfaz de la aplicación, que los resultados que produce son los correctos en función de las entradas que se le introducen para probarlos.

2.2.- Estructurales.



Ya hemos visto que las pruebas funcionales se centran en resultados, en lo que la aplicación hace, pero no en cómo lo hace.

Para ver cómo el programa se va ejecutando, y así comprobar su corrección, se utilizan las pruebas estructurales, que se fijan en los caminos que se pueden recorrer:

Las pruebas estructurales son el conjunto de pruebas de la Caja Blanca. Con este tipo de pruebas, se pretende verificar la estructura interna de cada componente de la aplicación, independientemente de la funcionalidad establecida para el mismo. Este tipo de pruebas, no pretenden comprobar la corrección de los resultados producidos por los distintos componentes, su función es comprobar que se van a ejecutar todas la instrucciones del programa, que no hay código no usado, comprobar que los caminos lógicos del programa se van a recorrer, etc.

Este tipo de pruebas, se basan en unos criterios de cobertura lógica, cuyo cumplimiento determina la mayor o menor seguridad en la detección de errores. Los criterios de cobertura que se siguen son:

- **Cobertura de sentencias**: se han de generar casos de pruebas suficientes para que cada instrucción del programa sea ejecutada, al menos, una vez.
- **Cobertura de decisiones**: se trata de crear los suficientes casos de prueba para que cada opción resultado de una prueba lógica del programa, se evalúe al menos una vez a cierto y otra a falso.
- **Cobertura de condiciones**: se trata de crear los suficientes casos de prueba para que cada elemento de una condición, se evalúe al menos una vez a falso y otra a verdadero.
- **Cobertura de condiciones y decisiones**: consiste en cumplir simultáneamente las dos anteriores.
- **Cobertura de caminos**: es el criterio más importante. Establece que se debe ejecutar al menos una vez cada secuencia de sentencias encadenadas, desde la sentencia inicial del programa, hasta su sentencia final. La ejecución de este conjunto de sentencias, se conoce como camino. Como el número de caminos que puede tener una aplicación, puede ser muy

grande, para realizar esta prueba, se reduce el número a lo que se conoce como camino prueba.

- **Cobertura del camino de prueba**: Se pueden realizar dos variantes, una indica que cada bucle se debe ejecutar sólo una vez, ya que hacerlo más veces no aumenta la efectividad de la prueba y otra que recomienda que se pruebe cada bucle tres veces: la primera sin entrar en su interior, otra ejecutándolo una vez y otra más ejecutándolo dos veces.

Autoevaluación
En las pruebas de caja negra:

☐ Es necesario conocer el código fuente del programa, para realizar las pruebas.

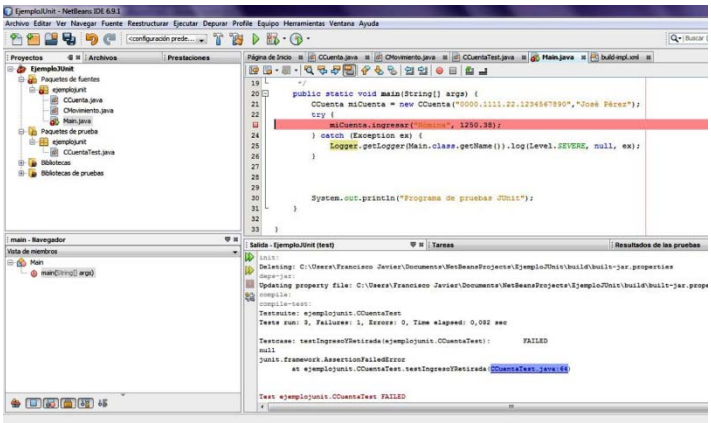
☐ Se comprueba que todos los caminos del programa, se pueden recorrer, al menos una vez.

◉ Se comprueba que los resultados de una aplicación, son los esperados para las entradas que se le han proporcionado.

☐ Es incompatible con la prueba de caja blanca.

2.3.- Regresión.

Durante el proceso de prueba, tendremos éxito si detectamos un posible fallo o error. La consecuencia directa de ese descubrimiento, supone la modificación del componente donde se ha detectado. Esta modificación, puede generar errores colaterales, que no existían antes. Como consecuencia, la modificación realizada nos obliga a repetir pruebas que hemos realizado con anterioridad.



El objetivo de las pruebas de regresión, es comprobar que los cambios sobre un componente de una aplicación, no introduce un comportamiento no deseado o errores adicionales en otros componentes no modificados.

✓ Las pruebas de regresión se deben llevar a cabo cada vez que se hace un cambio en el sistema, tanto para corregir un error, como para realizar una mejora. No es suficiente probar sólo los componentes modificados o añadidos, o las funciones que en ellos se realizan, sino que también es necesario controlar que las modificaciones no produzcan efectos negativos sobre el mismo u otros componentes.

Normalmente, este tipo de pruebas implica la repetición de las pruebas que ya se hayan realizado previamente, con el fin de asegurar que no se introducen errores que puedan comprometer el funcionamiento de otros componentes que no han sido modificados y confirmar que el sistema funciona correctamente una vez realizados los cambios.

En un contexto más amplio, las pruebas de software con éxito, son aquellas que dan como resultado el descubrimiento de errores. Como consecuencia del descubrimiento de errores, se procede a su corrección, lo que implica la modificación de algún componente del software que se está desarrollando, tanto del programa, de la documentación y de los datos que lo soportan. La prueba de regresión es la que nos ayuda a asegurar que estos cambios no introducen un comportamiento no

deseado o errores adicionales. La prueba de regresión se puede hacer manualmente, volviendo a realizar un subconjunto de todos los casos de prueba o utilizando herramientas automáticas.

El conjunto de pruebas de regresión contiene tres clases diferentes de clases de prueba:

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

Para evitar que el número de pruebas de regresión crezca demasiado, se deben de diseñar para incluir sólo aquellas pruebas que traten una o más clases de errores en cada una de las funciones principales del programa. No es práctico ni eficiente volver a ejecutar cada prueba de cada función del programa después de un cambio.

Autoevaluación
La prueba de regresión:

- ☐ Se realiza una vez finalizado cada módulo (*Cada parte, con una funcionalidad concreta, en que se divide una aplicación*) del sistema a desarrollar.
- ☐ Solo utiliza el enfoque de la caja negra.
- ◉ Se realiza cuando se produce una modificación, debido a la detección de algún error, en la fase de prueba.
- ☐ Es incompatible con la prueba de caja blanca.

3.- Procedimientos y casos de prueba.

Caso práctico

En BK, ya tienen el proyecto bastante avanzado. Ahora llega una parte clave: definir las partes del sistema que se van a probar y establecer los casos de prueba para realizarla. Ana va a participar, pero cuando se habla de procedimientos (*Igual que las funciones, pero al ejecutarse no devuelven ningún valor*) y casos de prueba, se siente perdida. A ella le va a tocar ejecutar los casos de prueba.

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

Dada la complejidad de las aplicaciones informáticas, que se desarrollan en la actualidad, es prácticamente imposible, probar todas las combinaciones que se pueden dar dentro de un programa o entre un programa y las aplicaciones que pueden interactuar con él. Por este motivo, en el diseño de los casos de prueba, siempre es necesario asegurar que con ellos se obtiene un nivel aceptable de probabilidad de que se detectarán los errores existentes.

Las pruebas deben buscar un compromiso entre la cantidad de recursos que se consumirán en el proceso de prueba, y la probabilidad obtenida de que se detecten los errores existentes.

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

- ✓ **Enfoque funcional o de caja negra**. En este tipo de prueba, nos centramos en que el programa, o parte del programa que estamos probando, recibe un entrada de forma adecuada y se produce una salida correcta, así como que la integridad de la información externa se mantiene. La prueba no verifica el proceso, solo los resultados.
- ✓ **Enfoque estructural o caja blanca**. En este tipo de pruebas, debemos centrar en la implementación interna del programa. Se trata de comprobar que la operación interna se ajusta a las especificaciones. En esta prueba, se deberían de probar todos los caminos que puede seguir la ejecución del programa.
- ✓ **Enfoque aleatorio**. A partir de modelos obtenidos estadísticamente, se elaboran casos de prueba que prueben las entradas del programa.

4.- Herramientas de depuración.

Caso práctico
Juan y María tiene muchas líneas de código implementadas. Como programadores de la aplicación, juegan un papel decisivo en la fase de pruebas. Al realizar este proyecto utilizando un entorno de desarrollo integrado (NetBeans), cuenta con una herramienta fundamental, el depurador.

Cada IDE (Integrated Development Environment) incluye herramientas de depuración como: inclusión de puntos de ruptura, ejecución paso a paso de cada instrucción, ejecución por procedimiento, inspección de variables, etc.

Todo entorno de desarrollo, independientemente de la plataforma, así como del lenguaje de programación utilizado, suministra una serie de herramientas de depuración, que nos permiten verificar el código generado, ayudándonos a realizar pruebas tanto estructurales como funcionales.

Durante el proceso de desarrollo de software, se pueden producir dos tipos de errores: errores de compilación o errores lógicos. Cuando se desarrolla una aplicación en un IDE, ya sea Visual Studio, Eclipse o Netbeans, si al escribir una sentencia, olvidamos un ";", hacemos referencia a una variable inexistente o utilizamos una sentencia incorrecta, se produce un error de compilación. Cuando ocurre un error de compilación, el entorno nos proporciona información de donde se produce y como poder solucionarlo. El programa no puede compilarse hasta que el programador o programadora no corrija ese error.

El otro tipo de errores son lógicos, comúnmente llamados bugs, estos no evitan que el programa se pueda compilar con éxito, ya que no hay errores sintácticos, ni se utilizan variables no declaradas, etc. Sin embargo, los errores lógicos, pueden provocar que el programa devuelva resultados erróneos, que no sean los esperados o pueden provocar que el programa termine antes de tiempo o no termine nunca.

Para solucionar este tipo de problemas, los entornos de desarrollo incorporan una herramienta conocida como **depurador.** El depurador permite supervisar la ejecución de los programas, para localizar y eliminar los errores lógicos. Un programa debe compilarse con éxito para poder utilizarlo en el depurador. El depurador nos permita analizar todo el programa, mientras éste se ejecuta. Permite suspender la ejecución de un programa, examinar y establecer los valores de las variables, comprobar los valores devueltos por un determinado método, el resultado de una comparación lógica o relacional, etc.

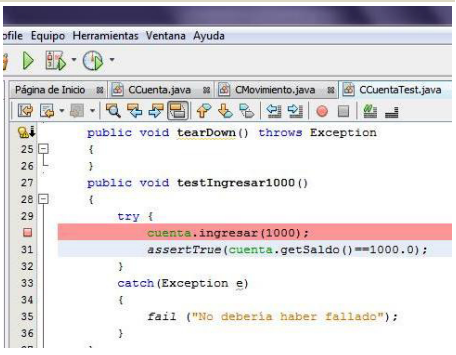
Autoevaluación
¿Qué concepto está relacionado con la prueba de caja negra?

- [x] Es la principal herramienta de validación.
- [ ] Se pueden comprobar los valores que van tomando las variables
- [ ] Se comprueba que todos los caminos del programa, se pueden recorrer, al menos una vez.
- [ ] Es incompatible con la prueba de caja blanca.

Con las pruebas de caja negra no se depura el programa, se comprueba que devuelve los valores esperados en función de las entradas introducidas.

4.1.- Puntos de ruptura.

Dentro del menú de depuración, nos encontramos con la opción insertar punto de ruptura (breakpoint). Se selecciona la línea de código donde queremos que el programa se pare,



para a partir de ella, inspeccionar variables, o realizar una ejecución paso a paso, para verificar la corrección del código

Durante la prueba de un programa, puede ser interesante la verificación de determinadas partes del código. No nos interesa probar todo el programa, ya que hemos delimitado el punto concreto donde inspeccionar. Para ello, utilizamos los puntos de ruptura.

Los puntos de ruptura son marcadores que pueden establecerse en cualquier línea de código ejecutable (no sería válido un comentario, o una línea en blanco). Una vez insertado el punto de ruptura, e iniciada la depuración, el programa a evaluar se ejecutaría hasta la línea marcada con el punto de ruptura. En ese momento, se pueden realizar diferentes labores, por un lado, se pueden examinar las variables, y comprobar que los valores que tienen asignados son correctos, o se pueden iniciar una depuración paso a paso, e ir comprobando el camino que toma el programa a partir del punto de ruptura. Una vez realiza la comprobación, podemos abortar el programa, o continuar la ejecución normal del mismo.

Dentro de una aplicación, se pueden insertar varios puntos de ruptura, y se pueden eliminar con la misma facilidad con la que se insertan.

4.2.- Tipos de ejecución.

Para poder depurar un programa, podemos ejecutar el programa de diferentes formas, de manera que en función del problema que queramos solucionar, nos resulte más sencillo un método u otro. Nos encontramos con lo siguientes tipo de ejecución: paso a paso por instrucción, paso a paso por procedimiento, ejecución hasta una instrucción, ejecución de un programa hasta el final del programa,

- ✓ Algunas veces es necesario ejecutar un programa línea por línea, para buscar y corregir errores lógicos. El **avance paso a paso** a lo largo de una parte del programa puede ayudarnos a verificar que el código de un método se ejecute en forma correcta.
- ✓ El **paso a paso por procedimientos**, nos permite introducir los parámetro que queremos a un método o función de nuestro programa, pero en vez de ejecutar instrucción por instrucción ese método, nos devuelve su resultado. Es útil, cuando hemos comprobado que un procedimiento funciona correctamente, y no nos interese volver a depurarlo, sólo nos interesa el valor que devuelve.
- ✓ En la **ejecución hasta una instrucción**, el depurador ejecuta el programa, y se detiene en la instrucción donde se encuentra el cursor, a partir de ese punto, podemos hacer una depuración paso a paso o por procedimiento.
- ✓ En la **ejecución de un programa hasta el final** del programa, ejecutamos las instrucciones de un programa hasta el final, sin detenernos en las instrucciones intermedias.

Los distintos modos de ejecución, se van a ajustar a las necesidades de depuración que tengamos en cada momento. Si hemos probada un método, y sabemos que funciona correctamente, no es necesario realizar una ejecución paso a paso en él.

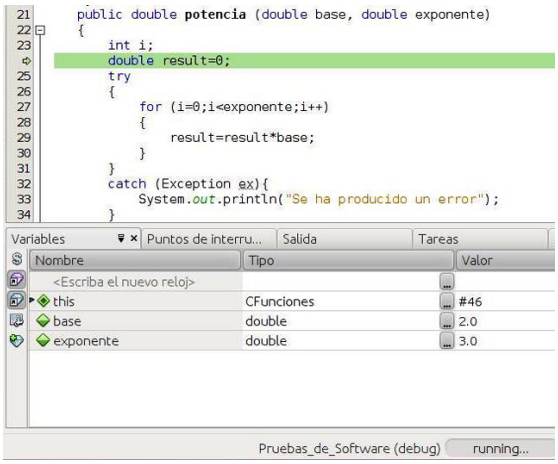
En el IDE NetBeans, dentro del menú de depuración, podemos seleccionar los modos de ejecución especificados, y algunos más. El objetivo es poder examinar todas las partes que se consideren necesarias, de manera rápida, sencilla y los más clara posible.

4.3.- Examinadores de variables.

Durante el proceso de implementación y prueba de software, una de las maneras más comunes de comprobar que la aplicación funciona de manera adecuada, es comprobar que las variables vayan tomando los valores adecuados en cada momento.

Los examinadores de variables, forman uno de los elementos más importantes del proceso de depuración de un programa. Iniciado el proceso de depuración, normalmente con la ejecución paso a paso, el programa avanza instrucción por instrucción. Al mismo tiempo, las distintas variables, van tomando diferentes valores. Con los examinadores de variables, podemos comprobar los distintos valores que adquieren las variables, así como su tipo. Esta herramienta es de gran utilidad para la detección de errores.

En el caso del entorno de desarrollo NetBeans, nos encontramos con un panel llamado Ventana de Inspección. En la ventana de inspección, se pueden ir agregando todas aquellas variables de las que tengamos interés en inspeccionar su valor. Conforme el programa se vaya ejecutando, NetBeans irá mostrando los valores que toman las variables en la ventana de inspección.



Como podemos apreciar, en una ejecución paso a paso, el programa llega a una función de nombre potencia. Esta función tiene definida tres variables. A lo largo de la ejecución del bucle, vemos como la variable **result**, van cambiando de valor. Si con valores de entrada para los que conocemos el resultado, la función no devuelve el valor esperado, "Examinando las variables" podremos encontrar la instrucción incorrecta.

Los depuradores son programas que permiten analizar de manera exhaustiva y en paso a paso, lo que pasa dentro del código de un programa. Gracias a los depuradores es fácil probar las aplicaciones para encontrar los posibles errores, analizando sus causas y posibles soluciones.

En el caso del IDE NetBeans, que es el que utilizamos en el ejemplo, estas utilidades están integradas en el entorno de desarrollo

INICIO DE LA DEPURACIÓN

Para conocer las opciones más habituales de depuración en NetBeans, vamos a depurar un pequeño programa. Este programa dispone de una clase Vector, que implementa algunos métodos, como insertar y ordenar.

Para iniciar la depuración del programa, en el menú principal, seleccionamos **Depurar**.

Como hemos indicado, el objetivo de la depuración es la detección de errores en la aplicación. Para encontrar los posibles errores, o comprobar el buen diseño de una aplicación, debemos navegar por el código diseñado. Podemos analizar el código de varias formas:

Paso a paso: en cuyo caso vamos a ir ejecutando instrucción por instrucción, todo el código.

Ejecutar hasta el cursor: El cursor tendrá el foco en una determinada instrucción. El depurador ejecutará el programa y se parará cuando llegue a esa instrucción. Esto es útil, si ya hemos analizado parte del código y no nos interesa volver a depurarlo, ganando tiempo.

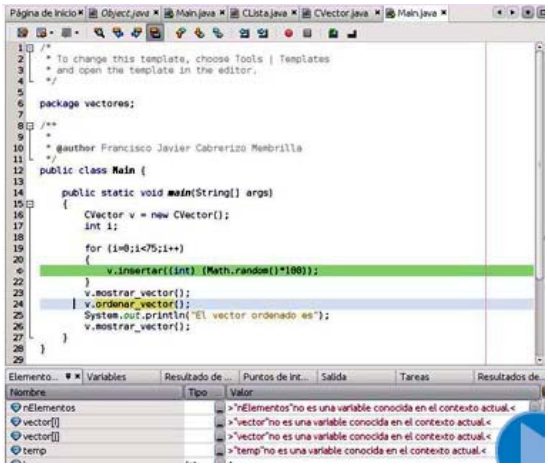
LA DEPURACIÓN

Una vez iniciada la depuración (**paso a paso** o **ejecutar hasta el cursor**), podemos decidir detener la depuración, continuarla, o ejecutar el programa hasta el final.

Podemos elegir **Entrar en el siguiente método**, si queremos analizarlo, o seguir con la ejecución **paso a paso** con la tecla **F7**.

Si el método en cuestión, no necesita depuración podemos **Continuar Ejecución** o pulsar **F8**, con esto el método se ejecuta, pero entramos en su código. Si no nos interesa seguir con la depuración, podemos seleccionar **Finalizar sesión del depurador**, si queremos pausar la depuración **Pausa** y si queremos seguir la ejecución normal, o seguir hasta el siguiente punto de ruptura **Continuar**.

Otras opciones del depurador son:

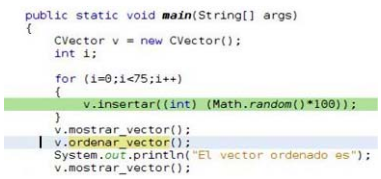


Consulta de la pila (stack) para comprobar las funciones que están siendo llamadas.

Inserción de puntos de interrupción, para indicar la línea de código que queremos analizar con el depurador

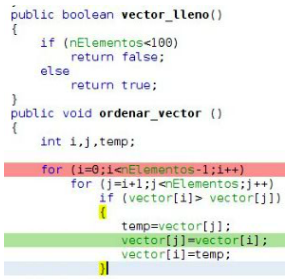
Evaluar expresión, que nos sirve para comprobar los valores que toma dicha expresión.

Si hacemos una depuración **paso a paso**, vamos a ir instrucción por instrucción. Esto nos ayuda a comprobar que los caminos que hemos diseñado en el código se pueden recorrer.



Como vemos en el ejemplo, hay un bucle que se repite 75 veces, si depuramos paso a paso, la ejecución del método ***v.insertar()***, deberíamos realizarla 75 veces. En general, se entra una vez en el método, y se realiza la depuración.

Para pasar a otra parte de código que sí queramos depurar, podemos colocar el cursor en la instrucción en cuestión, y pulsamos **Ejecutar hasta el cursor**, o bien, podemos ejecutar y para en el siguiente punto de ruptura.

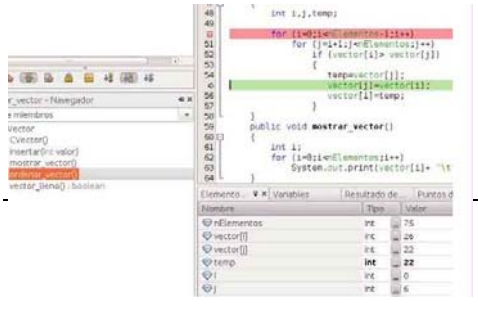


Para insertar un **punto de ruptura**, colocamos el cursor en la instrucción que nos interese, o pulsamos en la parte izquierda de la ventana.

Nos aparecerá la instrucción señalada en rojo. Esto significa, que cuando depuremos, independientemente del tipo de depuración, vamos a parar en esa instrucción. Esto es útil para depurar una parte de código a partir de ese punto de ruptura.

La línea verde, me indica la instrucción por la que va analizando el depurador. En la imagen, también podemos observar la ventana de variables. Esta ventana se utiliza en depuración para inspeccionar el valor que van tomando a lo largo de la ejecución.

En nuestro ejemplo depuramos un método de ordenación de un array, vemos los valores que toman las variables i, j y temp, así como el tipo al que están definidas.



En la imagen, vemos como podemos evaluar las expresiones que queremos. En este caso estamos evaluando los valores que toman las variables i, j y temp, y también comprobamos

los valores almacenados en el vector. Si durante la depuración comprobamos que la aplicación no hace aquello que debiera, podemos cambiarla. Si hemos depurado, y consideramos que la aplicación funciona, podemos pasar a probarla. Para ello, diseñamos casos de prueba en Junit y los aplicamos

Autoevaluación

¿Qué afirmación sobre depuración es incorrecta?

- ☐ En la depuración, podemos inspeccionar las instrucciones que va ejecutando el programa.
- ☐ No es posible conocer los valores que toman las variables definidas dentro de un método.
- ◉ Solo podemos insertar un punto de ruptura en la depuración

En depuración podemos insertar tantos puntos de ruptura como queremos, y que la depuración vaya saltando de uno a otro.

5.- Validaciones.

Caso práctico
Durante todo el proceso de desarrollo, existe una permanente comunicación entre Ada y su equipo, con representantes de la empresa a la que va destinado el proyecto. Ana y Juan van a asistir a la siguiente reunión, donde se va a mostrar a los representantes de la empresa, las fases de proyectos ya implementadas. Será la primera reunión de validación del proyecto.

En el proceso de validación, interviene de manera decisiva el cliente. Hay que tener en cuenta, que estamos desarrollando una aplicación para terceros, y que son estos los que deciden si la aplicación se ajusta a los requerimientos establecidos en el análisis.

En la validación intentan descubrir errores, pero desde el punto de vista de los requisitos *(Comportamiento y casos de uso que se esperan que cumpla el software que se está diseñando).*

La validación del software se consigue mediante una serie de pruebas de caja negra que demuestran la conformidad con 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. Tanto el plan como el procedimiento estarán diseñados para asegurar que se satisfacen todos los requisitos funcionales, que se alcanzan todos los requisitos de rendimiento, que las documentaciones son correctas e inteligible y que se alcanzan otros requisitos, como **portabilidad** (*Capacidad de un programa para ser ejecutado en cualquier arquitectura física de un equipo*), compatibilidad, recuperación de errores, facilidad de mantenimiento etc.

Cuando se procede con cada caso de prueba de validación, puede darse una de las dos condiciones siguientes:

- ✓ Las características de funcionamiento o rendimiento están de acuerdo con las especificaciones y son aceptables o
- ✓ Se descubre una desviación de las especificaciones y se crea una lista de deficiencias. Las desviaciones o errores descubiertos en esta fase del proyecto raramente se pueden corregir antes de la terminación planificada.

6.- Pruebas de código.

Caso práctico
Juan y María prueban cada parte de código que están implementando. Algunos métodos requieren una comprobación de su estructura interna, en otros, valdría con probar los resultados que devuelven. Antonio se pregunta en qué consiste cada prueba, y como se lleva a cabo en la práctica.

La prueba consiste en la ejecución de un programa con el objetivo de encontrar errores. El programa o parte de él, se va a ejecutar bajo unas condiciones previamente especificadas, para una vez observados los resultados, estos sean registrados y evaluados.

Para llevar a cabo las pruebas, es necesario definir una serie de casos de prueba, que van a ser un conjunto de entradas, de condiciones de ejecución y resultados esperados, desarrollados para un objetivo particular.

Para el diseño de casos de prueba, se suelen utilizar tres enfoques principales:

- ✓ Enfoque **estructural** o de caja blanca. Este enfoque se centra en la estructura interna del programa, analizando los caminos de ejecución. Dentro de nuestro proceso de prueba, lo aplicamos con el **cubrimiento**.
- ✓ Enfoque **funcional** o de caja negra. Este enfoque se centra en las funciones, entradas y salidas. Se aplican los **valores límite** y las **clases de equivalencia.**
- ✓ Enfoque **aleatorio**, que consiste en utilizar modelos que represente las posibles entradas al programa, para crear a partir de ellos los casos de prueba. En esta prueba se intenta simular la entrada habitual que va a recibir el programa, para ello se crean datos entrada en la secuencia y con la frecuencia en que podrían aparecer. Para ello se utilizan generadores automáticos de casos de prueba.

Para saber más
Puedes visitar la siguiente página web, donde se detallan los tipos de pruebas que suelen hacer al software y la función de cada una.

Pruebas de software http://alarcos.inf-cr.uclm.es/doc/ISOFTWAREI/Tema09.pdf

Autoevaluación
Durante la validación:

- ☐ Procedemos a depurar el programa.
- ☐ Sometemos el código a pruebas de cubrimiento.
- ◉ Comprobamos que la aplicación cumple los requerimientos del cliente.

En esta prueba, es el cliente, junto con el equipo de desarrollo, quienes comprueban que lo desarrollado cumple las especificaciones establecidas.

6.1.- Cubrimiento.

Esta tarea la realiza el programador o programadora y consiste en comprobar que los caminos definidos en el código, se pueden llegar a recorrer.

Este tipo de prueba, es de caja blanca, ya que nos vamos a centrar en el código de nuestra aplicación.

Con este tipo de prueba, lo que se pretende, es comprobar que todas las funciones, sentencias, decisiones, y condiciones, se van a ejecutar.

Por ejemplo

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

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 esta 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.

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.

6.2.- Valores límite.

```
public double funcion1 (double x)
{
   if (x > 5)
      return x;
   else
      return -1;
}

public int funcion2 (int x)
{
   if (x > 5)
      return x;
   else
      return -1;
}
```

En el código Java adjunto, aparecen dos funciones que reciben el parámetro x. En la **función1**, el parámetro es de tipo real y en la **función2**, el parámetro es de tipo entero.

Como se aprecia, el código de las dos funciones es el mismo, sin embargo, los casos de prueba con valores límite va a ser diferente.

La experiencia ha demostrado que los casos de prueba que obtienen una mayor probabilidad de éxito, son aquellos que trabajan con valores límite.

Esta técnica, se suele utilizar como complementaria de las particiones equivalentes, pero se diferencia, en que se suelen seleccionar, no un conjunto de valores, sino unos pocos, en el límite del rango de valores aceptado por el componente a probar.

Cuando hay que seleccionar un valor para realizar una prueba, se escoge aquellos que están situados justo en el límite de los valores admitidos.

Por ejemplo, supongamos que queremos probar el resultado de la ejecución de una función, que recibe un parámetro x:

- ✓ Si el parámetro x de entrada tiene que ser mayor estricto que 5, y el valor es real, los valores límite pueden ser 4,99 y 5,01.
- ✓ Si el parámetro de entrada x está comprendido entre -4 y +4, suponiendo que son valores enteros, los valores límite serán -5, -4 ,-3,3, 4 y 5.

Autoevaluación

Si en un bucle while la condición es while (x>5 && x < 10), siendo x un valor single, sería valores límite

☐ 4 y 11

- 4,99 y 11
- 4,99 y 9,99.

6.3.- Clases de equivalencia.

Las clases de equivalencia, es un tipo de prueba funcional, en donde cada caso de prueba, pretende cubrir el mayor número de entradas posible.

El dominio de valores de entrada, se divide en número finito de clases de equivalencia. Como la entrada está dividida en un conjunto de clases de equivalencia, la prueba de un valor representativo de cada clase, permite suponer que el resultado que se obtiene con él, será el mismo que con cualquier otro valor de la clase.

Cada clase de equivalencia debe cumplir:

- ✓ Si un parámetro de entrada debe estar comprendido entre un determinado rango, hay tres clases de equivalencia: por debajo, en y por encima.
- ✓ Si una entrada requiere un valor entre los de un conjunto, aparecen dos clase de equivalencia: en el conjunto o fuera de él.
- ✓ Si una entrada es booleana, hay dos clases: sí o no.
- ✓ Los mismos criterios se aplican a las salidas esperadas: hay que intentar generar resultados en todas y cada una de las clases.

En este ejemplo, las clases de equivalencia serían:

```
public double funcion1 (double x)
{
    if (x > 0 && x < 100)
        return x+2;
    else
        return x-2;
}
```

1. Por debajo: x<=0
2. En: x>0 y x< 100
3. Por encima: x>=100

y los respectivos casos de prueba, podrían ser:

1. Por debajo: x=0
2. En: x=50
3. Por encima: x=100

7.- Normas de calidad.

Caso práctico
Las aplicaciones que desarrolla BK Programación, se caracterizan por cumplir todos los estándares de calidad de la industria. Como no podía ser de otro modo, a la hora de realizar las pruebas, también van a seguir los estándares más actuales del mercado. Ada se va a encargar de supervisar el cumplimiento de los estándares más actuales.

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

- ✓ 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 IEEE de pruebas de software.:
  - → IEEE estándar 829, Documentación de la prueba de software.
  - → IEEE estándar 1008, Pruebas de unidad
  - → Otros estándares **ISO /** IEC 12207, 15289
- ✓ Otros estándares sectoriales

Sin embargo, estos estándares no cubren determinadas facetas de la fase de pruebas, como son la organización el proceso y gestión de las pruebas, presentan pocas pruebas funcionales y no funcionales etc. Ante esta problemática, la industria ha desarrollado la norma ISO/IEC 29119.

La norma ISO/IEC 29119 de prueba de software, pretende unificar en una única norma, todos los estándares, de forma que proporcione vocabulario, procesos, documentación y técnicas para cubrir todo el ciclo de vida del software. Desde estrategias de prueba para la organización y políticas de prueba, prueba de proyecto al análisis de casos de prueba, diseño, ejecución e informe. Con este estándar, se podrá realizar cualquier prueba para cualquier proyecto de desarrollo o mantenimiento de software.

Debes conocer
Norma ISO/IEC 29119.
La norma ISO/IEC 29119 la compone 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) .

Para saber más

En el siguiente enlace podrás visitar la página internacional, donde se detallan las normas a seguir, por las pruebas de software:

Normas para la prueba de software http://softwaretestingstandard.org/

Autoevaluación

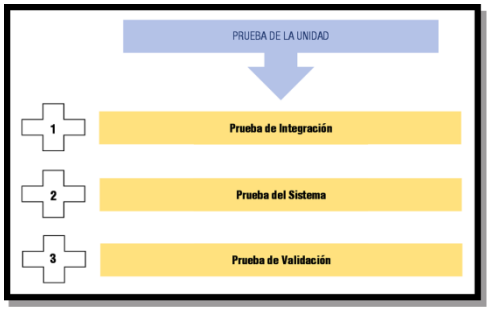
¿Qué norma de calidad intenta unificar los estándares para pruebas de software?

- ☐ BS 7925-1.
- ☐ IEEE 1008.
- ◉ ISO/IEC 29119.

8.- Pruebas unitarias.

Caso práctico

Antonio está un poco confuso. La aplicación que están diseñando Juan y María es ya de cierta envergadura y se pregunta por la labor ingente que queda, solo para probar todos los componentes de la aplicación. María le tranquiliza y le comenta que los Entornos de Desarrollo actuales, incorporan herramientas que realizan las pruebas de cada método, de forma automática.



Las pruebas unitarias, o prueba de la unidad, tienen por objetivo probar el correcto funcionamiento de un módulo de código. El fin que se persigue, es que cada módulo funciona correctamente por separado.

Posteriormente, con la prueba de integración, se podrá asegurar el correcto funcionamiento del sistema.

Una unidad es la parte de la aplicación más pequeña que se puede probar. En programación procedural, una unidad puede ser una función o procedimiento, En programación orientada a objetos, una unidad es normalmente un método.

Con las pruebas unitarias se debe probar todas las funciones o métodos no triviales de forma que cada caso de prueba sea independiente del resto.

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

- ✓ **Automatizable**: no debería requerirse una intervención manual.
- ✓ **Completas**: deben cubrir la mayor cantidad de código.
- ✓ **Repetibles o Reutilizables**: no se deben crear pruebas que sólo puedan ser ejecutadas una sola vez.
- ✓ **Independientes**: la ejecución de una prueba no debe afectar a la ejecución de otra.
- ✓ **Profesionales**: las pruebas deben ser consideradas igual que el código, con la misma profesionalidad, documentación, etc.

El objetivo de las pruebas unitarias es aislar cada parte del programa y demostrar que las partes individuales son correctas. Las pruebas individuales nos proporcionan cinco ventajas básicas:

1. **Fomentan el cambio**: Las pruebas unitarias facilitan que el programador cambie el código para mejorar su estructura, puesto que permiten hacer pruebas sobre los cambios y así asegurarse de que los nuevos cambios no han introducido errores.
2. **Simplifica la integración**: Puesto que permiten llegar a la fase de integración con un grado alto de seguridad de que el código está funcionando correctamente.
3. **Documenta el código**: Las propias pruebas son documentación del código puesto que ahí se puede ver cómo utilizarlo.
4. **Separación de la interfaz y la 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.
5. **Los errores están más acotados y son más fáciles de localizar**: dado que tenemos pruebas unitarias que pueden desenmascararlos.

8.1.- Herramientas para Java.

Entre las herramientas que nos podemos encontrar en el mercado, para poder realizar las pruebas, las más destacadas serían:

- ✓ **Jtiger:**
  - → Framework de pruebas unitarias para Java (1.5).
  - → Es de código abierto.
  - → Capacidad para exportar informes en HTML, XML o texto plano.
  - → Es posible ejecutar casos de prueba de Junit mediante un plugin.
  - → Posee una completa variedad de aserciones como la comprobación de cumplimiento del contrato en un método.
  - → Los metadatos (*Conjunto de datos que se utilizan para describir otros datos*) de 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 página web con toda la información necesaria para comprender su uso, y utilizarlo con IDE como Eclipse.
  - → El Framework (*Estructura conceptual y tecnológica de soporte definida, normalmente con módulos de software concretos, con base en la cual otro proyecto de software puede ser organizado y desarrollado*) incluye pruebas unitarias sobre sí mismo.
- ✓ **TestNG:**
  - → Esta inspirado en JUnit y NUnit.
  - → Está diseñado para cubrir todo tipo de pruebas, no solo las unitarias, sino también las funcionales, las de integración ...
  - → Utiliza las anotaciones de Java 1.5 (desde mucho antes que Junit).
  - → Es compatible con pruebas de Junit.
  - → Soporte para el paso de parámetros a los métodos de pruebas.
  - → Permite la distribución de pruebas en maquinas esclavas.
  - → Soportado por gran variedad de plug-ins (Eclipse, NetBeans, IDEA ...)
  - → Los clases de pruebas no necesitan implementar ninguna interfaz ni extender ninguna otra clase.
  - → Una vez compiladas la pruebas, estas se pueden invocar desde la linea de comandos con una tarea de Ant o con un fichero XML.
  - → Los métodos de prueba se organizan en grupos (un método puede pertenecer a uno o varios grupos).
- ✓ **Junit:**
  - → 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.

8.2.- Herramientas para otros lenguajes.

En la actualidad, nos encontramos con un amplio conjunto de herramientas destinadas a la automatización del prueba, para la mayoría de los lenguajes de programación más extendidos en la actualidad. Existen herramientas para C++, para PHP, FoxPro, etc.

Cabe destacar las siguientes herramientas:

- **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. Junit
- **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).

Autoevaluación

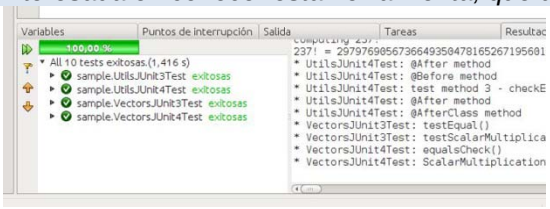
Las herramientas de automatización de pruebas más extendida para Java es:

- (•) Junit.
- ( ) FoxUnit.
- ( ) Simple Test.

9.- Automatización de la prueba.

Caso práctico

Juan está realizando pruebas de la unidad, es decir, comprueba el correcto funcionamiento de los métodos que ha implantado. Para ello, utiliza las herramienta de prueba incorporadas en el entorno de desarrollo. En sucaso, ya que está utilizando NetBeans, se decanta por Junit. Ana está muy interesada en conocer esta herramienta, que ayuda notablemente en el proceso de pruebas



Las pruebas de software son parte esencial del ciclo de desarrollo. La elaboración y mantenimiento de unidad, pueden ayudarnos a asegurar que los los métodos individuales de nuestro código, funcionan correctamente. Los entorno de desarrollo, integran frameworks, que permiten automatizar las pruebas.

En el caso de entornos de desarrollo para Java, como NetBeans y Eclipse, nos encontramos con el framework JUnit. JUnit es una herramienta de automatización de pruebas que nos permite de manera rápida y sencilla, elaborar pruebas. La herramienta nos permite diseñar clases de prueba, para cada clase diseñada en nuestra aplicación. Una vez creada las clases de prueba, establecemos los métodos que queremos probar, y para ello diseñamos casos de prueba. Los criterios de creación de casos de prueba, pueden ser muy diversos, y dependerán de lo que queramos probar.

Una vez diseñados los casos de prueba, pasamos a probar la aplicación. La herramienta de automatización, en este caso Junit, nos presentará un informe con los resultados de la prueba (imagen anterior). En función de los resultados, deberemos o no, modificar el código.

Uso de JUNIT en NetBeans

INTRODUCCIÓN

Los entornos de desarrollo más extendidos, que se utilizan para implementar aplicaciones Java, tales como NetBeans o Eclipse, incorporan un plugin para trabajar con Junit

Junit nos va a servir para realizar pruebas unitarias de clases escritas en Java, dentro de un entorno de pruebas. Es un framework con muy pocas clases fácil de aprender y de utilizar.

Una vez que hemos diseñado nuestra aplicación, y la hemos depurado, procedemos a probarla. En el caso del ejemplo, disponemos de una clase, de nombre ***cVector***, donde se han definido una serie de métodos.

El objetivo va a ser el diseño y ejecución de algunos casos de prueba.

INICIO DE JUNIT

Para iniciar Junit, seleccionada en la ventana de proyectos la clase a probar, abrimos el menú contextual y seleccionamos ***Herramientas > Crear pruebas Junit***.

Nos aparece un formulario donde nos da a elegir entre JUnit 3.x y JUnit 4.x. Son las dos versiones de JUnit disponibles en NetBeans 6.9.1. En nuestro caso, elegimos ***JUnit 3.x.***

Puesto que vamos a probar la clase CVector, por convenio es recomendable llamar a la clase de prueba CVectorTest. Esta clase se va a insertar en un nuevo paquete de nuestro proyecto, denominado ***Paquete de prueba***.

Como se aprecia en el formulario, JUnit va a generar los métodos que aparecen seleccionados. En nuestro caso lo vamos a dejar tal cual, aunque luefo van a ser modificados en el código.

Al pulsar el botón *Aceptar* nos aparecen un nueva clase de nombre *CVectorTest*, que contiene los métodos que estaban seleccionados en el formulario anterior, con un código prototipo. Es en ese código en el que el programador creará sus casos de prueba.

```
public class CVectorTest extends TestCase {

    public CVectorTest(String testName) {
        super(testName);
    }

    @Override
    protected void setUp() throws Exception {
        super.setUp();
    }

    @Override
    protected void tearDown() throws Exception
        super.tearDown();
    }

    /**
     * Test of insertar method, of class CVect
     */
```

El diseño de los casos de prueba, requiere que se establezcan criterios que garanticen que esa prueba tiene muchas probabilidades de encontrar algún error no detectado hasta el momento.

CASOS DE PRUEBA

En primer lugar, vamos a ver la función de los Inicializadores y Finalizadores. El método SetUp y el método tearDown, se utilizan para inicializar y finalizar las condiciones de prueba, como puede ser la creación de un objeto, inicialización de variables, etc. En algunos casos, no es necesario utilizar estos métodos, pero siempre se suelen incluir.

El ***método setUp*** es un método de inicialización de la prueba y se ejecutan antes de cada caso de prueba, en la clase de prueba. Este método no es necesario para ejecutar pruebas, pero si es necesario para inicializar algunas variables antes de iniciar la prueba.

El ***método tearDown*** es un método finalizador de prueba, y se ejecutará después de cada test en la clase prueba. Un método finalizador no es necesario para ejecutar las pruebas, pero si necesitamos un finalizador para limpiar algún dato que fue requerido en la ejecución de los casos de prueba.

En segundo lugar, es necesario conocer las aserciones. Los método assertXXX(), se utilizan para hacer las pruebas. Estos métodos, permiten comprobar si la salida del método que se está probando, concuerda con los valores esperados. Las principales son:

AssertTrue() evalúa una expresión booleana. La prueba pasa si el valor de la expresión es true.

assertFalse() evalúa una expresión booleana. La prueba pasa si el valor de la expresión es false.

AssertNull() verifica que la referencia a un objeto es nula.

assertNotNull() verifica que la referencia a un objeto es no nula.

AssertSame() compara dos referencias y asegura que los objetos referenciados tienen la misma dirección de memoria. La prueba pasa si los dos argumentos son el mismo objeto o pertenecen al mismo objeto.

assertNotSame() Compara dos referencias a objetos y asegura que ambas apuntan a diferentes direcciones de memoria. La prueba pasa si los dos argumentos suplidos son objetos diferentes o pertenecen a objetos distintos.

assertEquals() Se usa para comprobar igualdad a nivel de contenidos. La igual de tipos primitivos se compara usando "==", la igual entre objetos se compara con el método equals(). La prueba pasa si los valores de los argumentos son iguales.

fails() causa que la prueba falle inmediatamente. Se puede usar cuando la prueba indica un error o cuando se espera que el método que se está probando llame a una excepción.

En este punto, nos disponemos a diseñar los métodos que necesitemos para los casos de prueba. Ejemplos de casos de prueba pueden ser:

```
public void test_posicion (int pos){
	vectorTest.insertar(3);
	vectorTest.insertar(5);
	try{
		assertTrue (vectorTest.posicion(7)==1);
	}catch (Exception e){
		fail("El método test_posicion no funciona");
	}
}
/**
* Test del método ordenar_vector, de la clase CVector. Si ordena bien, la comparación entre el
* vector vOrdenado y prueba, debe ser cierta y la prueba un éxito
*/
public void test_ordenar_vector(){
	int [] vOrdenado = {7,36,45,52,85};
	int [] prueba = new int[5];
	try{
		vectorTest.insertar(36);
		vectorTest.insertar(85);
		vectorTest.insertar(45);
		vectorTest.insertar(52);
		vectorTest.insertar(7);
		vectorTest.ordenar_vector();
		for (int i=0;i<5;i++)
			prueba[i]=vectorTest.posicion(i);
		assertEquals(vOrdenado,prueba);
	}catch (Exception e){
		fail("El método vector _lleno no funciona");
	}
}
/**
* Test del método vector_lleno(), de la clase cVector. Debería devolver true, al
* insertar 100 elementos
*/
public void test_vector_lleno(){
	try{
		for(int i=0;i<100;i++)
		vectorTest.insertar(i);
		assertTrue(vectorTest.vector_lleno());
	}
	catch(Exception e){
		fail(·El método vector_lleno no funciona");
	}
}
```

Estos tres métodos intentan probar algunos métodos de la clase CVector. Para ello, teniendo seleccionado el proyecto, accederemos al menú contextual y pulsamos la opción Probar.

Como se puede comprobar, la prueba sobre el método vector_lleno ha sido un éxito, pero ha fallado la prueba sobre ordenar_vector. Con esta información, debemos comprobar que el caso de prueba está diseñado, en cuyo caso, lo que se ha encontrado es un error en el diseño del método ordenar_vector, y hay que rediseñarlo. La ventaja de utilizar herramientas automatizadas, es que se facilita la regresión, ya que tenemos diseñado el caso de prueba para el método, así que una vez

rediseñado, podemos volver a probarlo con el mismo caso de prueba.

Para saber más

En el siguiente enlace nos encontramos con un ejempo completo de prueba de la unidad con NetBeans

Creación de Casos de Prueba en NetBeans con Junit
http://netbeans.org/kb/docs/java/junit-intro.html#Exercise_21

10.- Documentación de la prueba.

Caso práctico
BK Programación, al igual que en todas la fase de diseño de un sistema, en la fase de prueba realiza la documentación. Ada, como coordinadora, le pide a Ana que ayuda a realizar la documentación de la prueba, y le pide que se repase la metodología Métrica v.3 y ayude a María y a Juan en la labor de documentación.

Como en otras etapas y tareas del desarrollo de aplicaciones, la documentación de las pruebas es un requisito indispensable para su correcta realización. Unas pruebas bien documentadas podrán también servir como base de conocimiento para futuras tareas de comprobación.

Las metodologías actuales, como Métrica v.3 (*Metodología de Planificación, Desarrollo y Mantenimiento de sistemas de información. Métrica v3 se puede usar libremente, con la única restricción de citar la fuente de su propiedad intelectual, que es el Ministerio de Presidencia*), proponen que la documentación de la fase de pruebas se basen en los estándares ANSI / IEEE sobre verificación y validación de software.

En propósito de los estándares ANSI/IEEE es describir un conjunto de documentos para las pruebas de software. Un documento de pruebas estándar puede facilitar la comunicación entre desarrolladores al suministrar un marco de referencia común. La definición de un documento estándar de prueba puede servir para comprobar que se ha desarrollado todo el proceso de prueba de software.

Los documentos que se van 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.

Para saber más
En el siguiente enlace podrás visitar la página de Ministerio de Política Territorial y Administración Pública, dedicada a Métrica v.3

Métrica v.3
http://administracionelectronica.gob.es/?_nfpb=true&_pageLabel=P60085901274201580632&langPae=es

Autoevaluación
La documentación de la prueba:

- ☐ Es una labor voluntaria que se puede realizar al final del proceso de pruebas
- ☐ Cada equipo de pruebas decide qué documenta y cómo.
- ◉ En España se usa Métrica v.3

Anexo I.- Norma ISO/IEC 29119.

La norma 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) .

TEMA 4

Contenido

[OPTIMIZACIÓN Y DOCUMENTACIÓN]

José Luis Comesaña Cabeza --- 2011/2012

Entornos de Desarrollo del curso de "*Desarrollo de Aplicaciones Web*"

OPTIMIZACIÓN Y DOCUMENTACIÓN

CASO PRÁCTICO.
BK programación se encuentra desarrollando la aplicación de gestión hotelera.
Como parte del proceso de desarrollo surgen una serie de problemas: hay un código que presenta algunas implementaciones que se pueden mejorar aplicando refactorización, nos encontramos que los miembros del equipo de desarrollo están modificando constantemente métodos o clases, creando diferentes versiones de las mismas y falta una documentación clara y precisa del código.
Ada propone a Juan y a María (programadores principales de la aplicación) que use los patrones generales de refactorización para conseguir un código de mejor calidad, más fácil de entender, más fácil de probar y con menor probabilidad de generar errores. Ana va a intentar ayudar a Juan, y Carlos a María, pero no conocen nada de refactorización, ni entiende la necesidad de realizar este trabajo "extra".
Como todos los miembros de BK Programación trabajan sobre los mismo proyectos, Ada debe coordinar el trabajo de todos ellos, por lo que propone que cada uno de ellos utilice un cliente de control de versiones, de forma que ella, de manera centralizada, pueda gestionar la configuración del software.
Los miembros con menor experiencia, Carlos y Ana, van a ir generando, utilizando herramientas de documentación automatizadas, la documentación de las clases y del código generado por Juan y María.
Ada va a encargarse de la Gestión de Configuraciones del Software, ya que es una labor fundamental para cualquier jefe de proyecto. Asimismo, debe garantizar que el código generado por Juan y María esté refactorizado.

1. Refactorización.

CASO PRÁCTICO.
Gran parte de las clases, métodos y módulos que forman parte de la aplicación de Gestión Hotelera han sido implementados, surge ahora una pregunta. ¿Podemos mejorar la estructura del código y que sea de mayor calidad, sin que cambie su comportamiento? ¿Cómo hacerlo? ¿Qué patrones hay que seguir?

La refactorización es una disciplina técnica, que consiste en realizar pequeñas transformaciones en el código de un programa, para mejorar la estructura sin que cambie el comportamiento ni funcionalidad del mismo. Su objetivo es mejorar la estructura interna del código. Es una tarea que pretender limpiar el código minimizando la posibilidad de introducir errores.

Con la refactorización se mejora el diseño del software, hace que el software sea más fácil de entender, hace que el mantenimiento del software sea más sencillo, la refactorización nos ayuda a encontrar errores y a que nuestro programa sea más rápido.

Cuando se refactoriza se está mejorando el diseño del código después de haberlo escrito. Podemos partir de un mal diseño y, aplicando la refactorización, llegaremos a un código bien diseñado. Cada paso es simple, por ejemplo mover una propiedad desde una clase a otra, convertir determinado código en un nuevo método, etc. La acumulación de todos estos pequeños cambios pueden mejorar de forma ostensible el diseño.

Puedes visitar la siguiente página web (en inglés), donde se presenta el proceso de refactorización de aplicaciones Java con Netbeans.
http://wiki.netbeans.org/Refactoring

1.1 Concepto

CASO PRÁCTICO.
Juan va a empezar a refactorizar parte del código que ha generado. Ana no sabe que es refactorizar, así que Juan le va a explicar las bases del proceso de refactorización.

FACTORIZACIÓN DE POLINOMIOS
$X^2 - 1 = (X + 1)(X - 1)$

El concepto de refactorización de código, se base en el concepto matemático de factorización de polinomios.

Podemos definir el concepto de refactorización de dos formas:

- ✓ **Refactorización**: Cambio hecho en la estructura interna del software para hacerlo más fácil de entender y fácil de modificar sin modificar su comportamiento.

  Ejemplos de refactorización es "Extraer Método" y "Encapsular Campos". La refactorización es normalmente un cambio pequeño en el software que mejora su mantenimiento.

- ✓ **Campos encapsulados**: Se aconseja crear métodos getter y setter, (de asignación y de consulta) para cada campo que se defina en una clase. Cuando sea necesario acceder o modificar el valor de un campo, basta con invocar al método getter o setter según convenga.
- ✓ **Refactorizar**: Reestructurar el software aplicando una serie de refactorizaciones sin cambiar su comportamiento.

El propósito de la refactorización es hacer el software más fácil de entender y de modificar. Se pueden hacer muchos cambios en el software que pueden hacer algún pequeño cambio en el comportamiento observable.

Solo los cambios hechos para hacer el software más fácil de entender son refactorizaciones.

Hay que diferenciar la refactorizacón de la optimización. En ambos procesos, se pretende mejorar la estructura interna de una aplicación o componente, sin modificar su comportamiento. Sin embargo, cuando se optimiza, se persigue una mejora del rendimiento, por ejemplo mejorar la velocidad de ejecución, pero esto puede hacer un código más difícil de entender.

Hay que resaltar que la refactorización no cambia el comportamiento observable del software. El software sigue cumpliendo la misma función que hacía antes. Ningún usuario, ya sea usuario final u otro programador, podrá determinar qué cosas han cambiado.

Con la refactorización de código, estamos modificando un código que funciona correctamente, ¿merece la pena el esfuerzo de refactorizar un código ya implementado?

1.2 Limitaciones.

CASO PRÁCTICO.
Ana se muestra muy interesada por conocer esta técnicas avanzadas se programación, sin embargo Juan le pone los pies en el suelo, explicándole que son técnicas con muchas limitaciones y poca documentación en la que basarse.

La refactorización es un técnica lo suficientemente novedosa para conocer cuáles son los beneficios que aporta, pero falta experiencia para conocer el alcance total de sus limitaciones. Se ha constatado que la refactorización presente problemas en algunos aspectos del desarrollo.

Un área problemática de la refactorización son las bases de datos. Una base de datos presenta muchas dificultades para poder ser modificada, dado la gran cantidad de interdependencias que soporta. Cualquier modificación que se requiera de las bases de datos, incluyendo modificación de esquema y migración de datos, puede ser una tarea muy costosa. Es por ello que la refactorización de una aplicación asociada a una base de datos, siempre será limitada, ya que la aplicación dependerá del diseño de la base de datos.

Otra limitación, es cuando cambiamos interfaces. Cuando refactorizamos, estamos modificando la estructura interna de un programa o de un método. El cambio interno no afecta al comportamiento ni a la interfaz. Sin embargo, si renombramos un método, hay que cambiar todas las referencias que se hacen a él. Siempre que se hace esto se genera un problema si es una interfaz pública. Una solución es mantener las dos interfaces, la nueva y la vieja, ya que si es utilizada por otra clase o parte del proyecto, no podrá referenciarla.

Hay determinados cambios en el diseño que son difíciles de refactorizar. Es muy difícil refactorizar cuando hay un error de diseño o no es recomendable refactorizar, cuando la estructura a modificar es de vital importancia en el diseño de la aplicación.

Hay ocasiones en las que no debería refactorizar en absoluto. Nos podemos encontrar con un código que, aunque se puede refactorizar, sería más fácil reescribirlo desde el principio. Si un código no funciona, no se refactoriza, se reescribe.

La refactorización:

- ☐ Se utiliza como técnica complementaria de realización de pruebas.
- ☐ Genera un código más difícil de entender y de mantener.
- ◉ Utiliza una serie de patrones, de aplicación sobre el código fuente.
- ☐ Es una técnica de programación no recogida por los entornos de desarrollo.

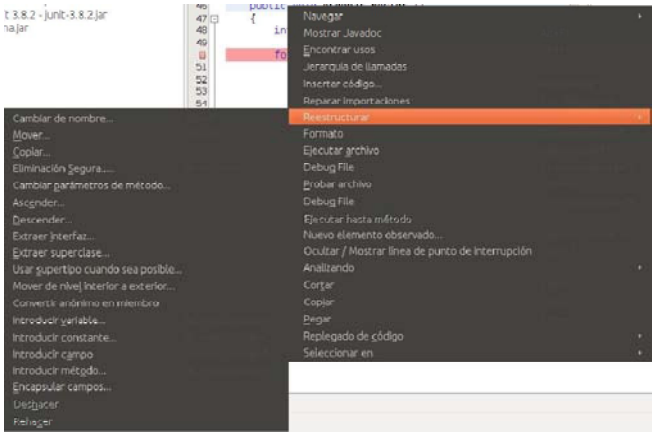
1.3 Patrones de refactorización más habituales.

CASO PRÁCTICO.
A pesar de la poca documentación y lo novedoso de la técnica, Juan le enseña a Ana algunos de los patrones más habituales de refactorización, que vienen ya integrados en la mayoría de los entornos de desarrollos más extendidos en el mercado.

En el proceso de refactorización, se siguen una serie de patrones preestablecidos, los más comunes son los siguientes:

- ✓ **Renombrado** (rename): Este patrón nos indica que debemos cambiar el nombre de un paquete, clase, método o campo, por un nombre más significativo.
- ✓ **Sustituir bloques de código por un método**: Este patrón nos aconseja sustituir un bloque de código, por un método. De esta forma, cada vez que queramos acceder a ese bloque de código, bastaría con invocar al método.
- ✓ **Campos encapsulados**: Se aconseja crear métodos getter y setter, (de asignación y de consulta) para cada campo que se defina en una clase. Cuando sea necesario acceder o modificar el valor de un campo, basta con invocar al método getter o setter según convenga.
- ✓ **Mover la clase**: Si es necesario, se puede mover una clase de un paquete a otro, o de un proyecto a otro. La idea es no duplicar código que ya se haya generado. Esto impone la actualización en todo el código fuente de las referencias a la clase en su nueva localización.
- ✓ **Borrado seguro**: Se debe comprobar, que cuándo un elemento del código ya no es necesario, se han borrado todas las referencias a él que había en cualquier parte del proyecto.

- ✓ **Cambiar los parámetros del proyecto**: Nos permite añadir nuevos parámetros a un método y cambiar los modificadores de acceso.
- ✓ **Extraer la interfaz**: Crea un nueva interfaz de los métodos public non-static seleccionados en una clase o interfaz.
- ✓ **Mover del interior a otro nivel**: Consiste en mover una clase interna a un nivel superior en la jerarquía.



1.4 Analizadores de código.

CASO PRÁCTICO.
María se va a centrar en el uso de analizadores de código con la ayuda de Carlos. Carlos no sabe lo que son los analizadores de código ni para qué sirven.

Cada IDE incluye herramientas de refactorización y analizadores de código. En el caso de software libre, existen analizadores de código que se pueden añadir como complementos a los entornos de desarrollo. Respecto a la refactorización, los IDE ofrecen asistentes que de forma automática y sencilla, ayudan a refactorizar el código.

El análisis estático de código, es un proceso que tiene como objetivo, evaluar el software, sin llegar a ejecutarlo.

Esta técnica se va a aplicar directamente sobre el código fuente, para poder obtener información que nos permita mejorar la base de código, pero sin que se modifique la semántica.

Los analizadores de código, son las herramientas encargadas de realizar esta labor. El analizador estático de código recibirá el código fuente de nuestro programa, lo procesará intentando averiguar la funcionalidad del mismo, y nos dará sugerencias, o nos mostrará posibles mejoras.

Los analizadores de código incluyen analizadores léxicos y sintácticos que procesan el código fuente y de un conjunto de reglas que se deben aplicar sobre determinadas estructuras. Si el analizador considera que nuestro código fuente tiene una estructura mejorable, nos lo indicará y también nos comunicará la mejora a realizar.

Las principales funciones de los analizadores es encontrar partes del código que puedan reducir el rendimiento, provocar errores en el software, tener una excesiva complejidad, complicar el flujo de datos, crear problemas de seguridad.

El análisis puede ser automático o manual. El automático, los va a realizar un programa, que puede formar parte de la funcionalidad de un entorno de desarrollo, por ejemplo el **FindBugs** en NetBeans, o manual, cuando es una persona.

El análisis automático reduce la complejidad para detectar problemas de base en el código, ya que los busca siguiendo una serie de reglas predefinidas. El análisis manual, se centra en apartados de nuestra propia aplicación, como comprobar que la arquitectura de nuestro software es correcta.

Tomando como base el lenguaje de programación Java, nos encontramos en el mercado un conjunto de analizadores disponibles:

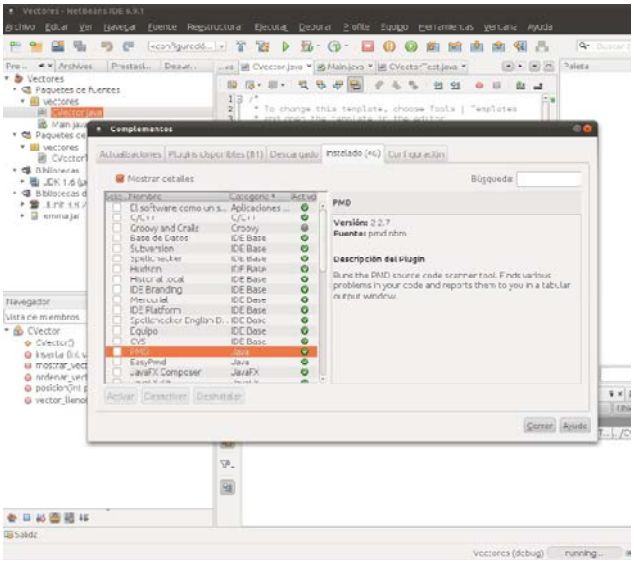
PMD. Esta herramienta basa su funcionamiento en detectar patrones, que son posibles errores en tiempo de ejecución, código que no se puede ejecutar nunca porque no se puede llegar a él, código que puede ser optimizado, expresiones lógicas que pueden ser simplificadas, malos usos del lenguaje, etc.

CPD. Forma parte del PMD. Su función es encontrar código duplicado.

1.4.1 Uso.

Los analizadores de código estático, se suelen integrar en los Entornos de Desarrollo, aunque en algunos casos, hay que instalarlos como plug-in , tras instalar el IDE. En el caso de NetBeans 6.9.1, vamos a instalar el plug-in para PMD. Para ello lo descargamos de sourceforge. Se descomprime el plug-in para NetBeans y se añade a los complementos de nuestro entorno de desarrollo. Para ello podemos utilizar el siguiente enlace:

http://sourceforge.net/projects/pmd/files/



Como se indicó en el apartado anterior, PMD es un analizador de código estático, capaz de detectar automáticamente un amplio rango de defectos y de inseguridades en el código. PMD se centra en detectar defectos de forma preventiva.

Una vez que tenemos desarrollado nuestro código, si queremos analizarlo con PMD, y obtener el informe del análisis, pulsamos el botón derecho del ratón sobre el directorio que contiene los ficheros de código, en la vista de proyectos y elegimos: Herramientas - Ejecutar PMD.

Mientras se analiza el código, el plug-in mostrará una barra de progreso en la esquina inferior derecha. El informe producido contiene la localización, el nombre de la regla que no se cumple y la recomendación de cómo se resuelve el problema.

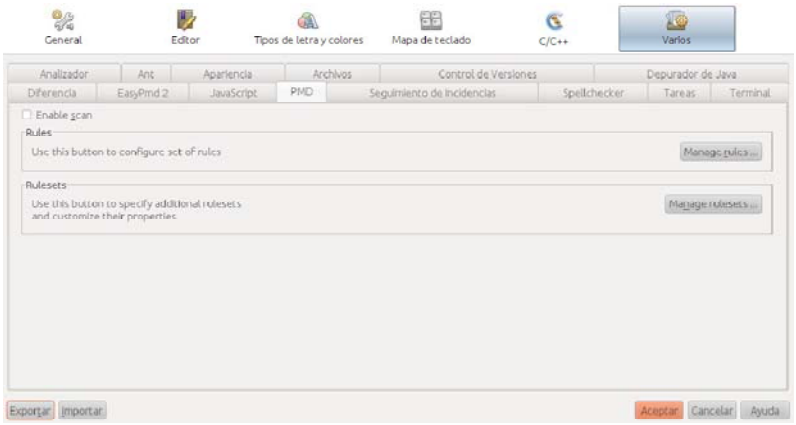
El informe PMD, nos permite navegar por el fichero de clases y en la línea donde se ha detectado el problema.

En el número de línea, veremos una marca PMD. Si se posiciona el ratón encima de ella, veremos un tooltip con la descripción del error.

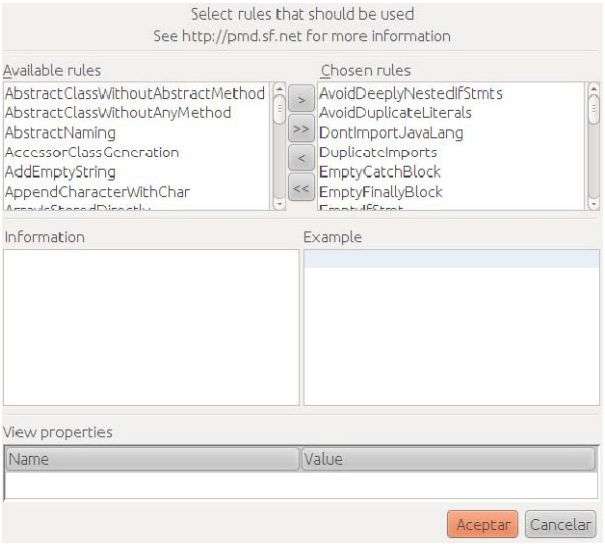
1.4.2 Configuración.

La configuración de los analizadores de código, nos va a permitir definir nuestras propias reglas, añadiéndolas a las que el analizador nos proporciona por defecto.

En el caso del analizador de código PMD, lo primero que vamos a hacer, es conocer las reglas que son aplicables. Elegimos las reglas que son aplicables accediendo a Herramientas - Opciones - Varios - PMD



Enable Scan habilita el escaneado automático del proyecto, a intervalos regulares. Si pulsamos en **Manage Rules** nos iremos a un cuadro de diálogo que nos permitirá activar o desactivar reglas específicas.



Si seleccionamos el nombre de una regla, podemos ver una descripción del problema y un ejemplo de aplicación.

– Si lo que queremos es introducir nuestras propias reglas, hacemos clic sobre **Manage Rulesets.** Se nos mostrará un nuevo formulario donde podemos importar un fichero XML o JAR con reglas.

Para crear ficheros XML con conjuntos de reglas, podemos obtener información en la siguiente página web.

http://pmd.sourceforge.net/howtomakearuleset.html

1.5 Refactorización y pruebas.

CASO PRÁCTICO.
Juan ayudado por Ana, se ha dedicado a refactorizar algunas partes del código que había diseñado. Al modificar el código diseñado, se encuentra en al necesidad de realizar pruebas de nuevo.

En la actualidad, la refactorización y las pruebas, son dos aspectos del desarrollo de aplicaciones, que por sus implicaciones y su interrelación, se han convertido en conceptos de gran importancia para la industria. Muchas cuestiones y problemas siguen sin ser explorados en estos dos ámbitos. Uno de los enfoques actuales, que pretende integrar las pruebas y la refactorización, es el Desarrollo Guiado por Pruebas (TDD, Test Driven Development).

Con el Desarrollo Guiado por Pruebas (TDD), se propone agilizar el ciclo de escritura de código, y realización de pruebas de unidad.

Cabe recordar, que el objetivo de las pruebas de unidad, es comprobar la calidad de un módulo desarrollado. Existen utilidades que permiten realizar esta labor, pudiendo ser personas distintas a las que los programan, quienes los realicen. Esto provoca cierta competencia entre los programadores de la unidad, y quienes tienen que realizar la prueba. El proceso de prueba, supone siempre un gasto de tiempo importante, ya que el programador realiza revisiones y depuraciones del mismo antes de enviarlo a prueba.

Durante el proceso de pruebas hay que diseñar los casos de prueba y comprobar que la unidad realiza correctamente su función. Si se encuentran errores, éstos se documentan, y son enviados al programador para que lo subsane, con lo que debe volver a sumergirse en un código que ya había abandonado.

Con el Desarrollo Guiado por Pruebas, la propuesta que se hace es totalmente diferente. El programador realiza las pruebas de unidad en su propio código, e implementa esas pruebas antes de escribir el código a ser probado.

Cuando un programador recibe el requerimiento para implementar una parte del sistema, empieza por pensar el tipo de pruebas que va a tener que pasar la unidad que debe elaborar, para que sea correcta. Cuando ya tiene claro la prueba que debe de pasar, pasa a programar las pruebas que debe pasar el código que debe de programar, no la unidad en sí. Cuando se han implementado las pruebas, se comienza a implementar la unidad, con el objeto de poder pasar las pruebas que diseñó previamente.

Cuando el programador empieza a desarrollar el código que se le ha encomendado, va elaborando pequeñas versiones que puedan ser compiladas y pasen por alguna de las pruebas. Cuando se hace un cambio y vuelve a compilar también ejecuta las pruebas de unidad. Y trata de que su programa vaya pasando más y más pruebas hasta que no falle en ninguna, que es cuando lo considera listo para ser integrado con el resto del sistema.

Para realizar la refactorización siguiendo TDD, se refactoriza el código tan pronto como pasa las pruebas para eliminar la redundancia y hacerlo más claro. Existe el riesgo de que se cometan errores durante la tarea de refactorización, que se traduzcan en cambios de funcionalidad y, en definitiva, en que la unidad deje de pasar las pruebas. Tratándose de reescrituras puramente sintácticas, no es necesario correr ese riesgo: las decisiones deben ser tomadas por un humano, pero los detalles pueden quedar a cargo de un programa que los trate automáticamente.

1.6 Herramientas de ayuda a la refactorización.

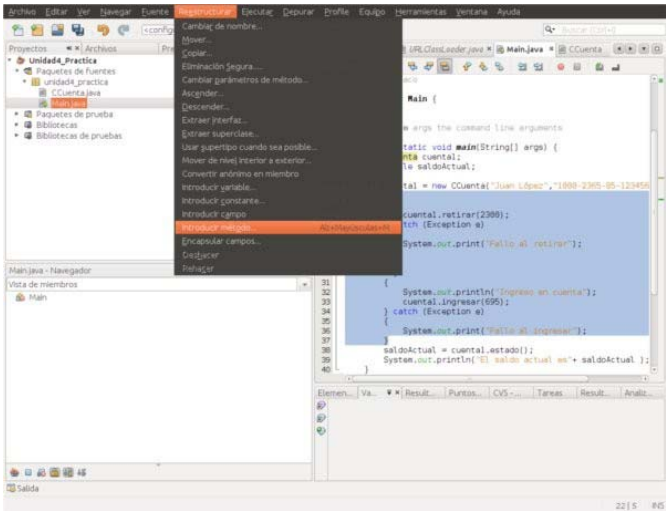
CASO PRÁCTICO.
Ana ya conoce los principios y patrones básicos de refactorización, ahora Juan le va a enseñar las herramientas de NetBeans para refactorizar de forma automática y sencilla, código Java.

Los entornos de desarrollo actuales, nos proveen de una serie de herramientas que nos facilitan la labor de refactorizar nuestro código. En puntos anteriores, hemos indicado algunos de los patrones

que se utilizan para refactorizar el código. Esta labor se puede realizar de forma manual, pero supone una pérdida de tiempo, y podemos inducir a redundancias o a errores en el código que modificamos.

En el Entorno de Desarrollo NetBeans, la refactorización está integrada como una función más, de las utilidades que incorpora.

A continuación, vamos a usar los patrones más comunes de refactorización, usando las herramientas de ayuda del entorno.



- ✓ **Renombrar**. Ya hemos indicado en puntos anteriores, que podemos cambiar el nombre de un paquete, clase, método o campo para darle un nombre más significativo. NetBeans nos permite hacerlo, de forma que actualizará todo el código fuente de nuestro proyecto donde se haga referencia al nombre modificado.
- ✓ **Introducir método**. Con este patrón podemos seleccionar un conjunto de código, y reemplazarlo por un método.
- ✓ **Encapsular campos**. NetBeans puede automáticamente generar métodos getter y setter para un campo, y opcionalmente actualizar todas las referencias al código para acceder al campo, usando los métodos getter y setter.

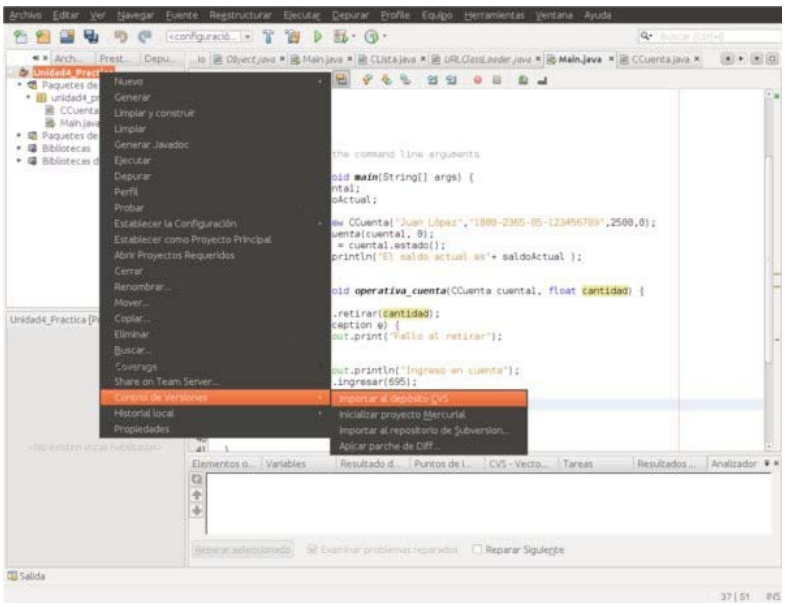
¿Cuál no es un patrón de refactorización?

- ◉ Eliminar parámetros de un método
- ○ Renombrado
- ○ Sustitución de un bloque de sentencias por un método.
- ○ Mover clase

2. Control de versiones.

CASO PRÁCTICO
Juan y María están implementando la mayor parte de la aplicación. Continuamente está modificando el código generado, bien para mejorar algunos aspecto, o por qué han refactorizado. Hay diferentes versiones de una misma clase, de un mismo método. ¿Qué ocurre si se borra accidentalmente algún fichero? ¿Qué pasa si María modifica una clase que necesita Juan?

Con el término versión, se hace referencia a la evolución de un único elemento, o de cada elemento por separado, dentro de un sistema en desarrollo.



Siempre que se está realizando una labor, sea del tipo que sea, es importante saber en cada momento de que estamos tratando, qué hemos realizado y qué nos queda por realizar.

En el caso del desarrollo de software ocurre exactamente lo mismo. Cuando estamos desarrollando software, el código fuente está cambiando continuamente, siendo esta particularidad vital. Esto hace que en el desarrollo de software actual, sea de vital importancia que haya sistemas de control de versiones. Las ventajas de utilizar un sistema de control de versiones son múltiples. Un sistema de control de versiones bien diseñado facilita al equipo de desarrollo su labor, permitiendo que varios desarrolladores trabajen en el mismo proyecto (incluso sobre los mismo archivos) de forma simultánea, sin que se que pisen unos a otros. Las herramientas de control de versiones proveen de un sitio central donde almacenar el código fuente de la aplicación, así como el historial de cambios realizados a lo largo de la vida del proyecto. También permite a los desarrolladores volver a un versión estable previa del código fuente si es necesario.

Una versión, desde el punto de vista de la evolución, 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 hay que disponer de un método, para designar las diferentes versiones de manera sistemática u organizada.

En los entornos de desarrollo modernos, los sistemas de control de versiones son una parte fundamental, que van a permitir construir técnicas más sofisticadas como la Integración Continua.

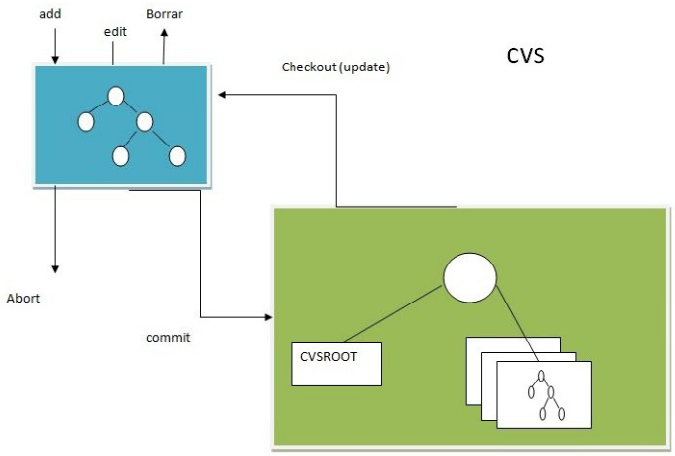
En los proyectos Java, existen dos sistemas de control de versiones de código abierto, CVS y Subversion. La herramienta CVS es una herramienta de código abierto que es usada por gran cantidad de organizaciones. Subversion es el sucesor natural de CVS, ya que se adapta mejor que CVS a las modernas prácticas de desarrollo de software.

Para gestionar las distintas versiones que se van generando durante el desarrollo de una aplicación, los IDE, proporcionan herramientas de Control de Versiones y facilitan el desarrollo en equipo de aplicaciones.

2.1 Estructura de herramientas de control de versiones.

CASO PRÁCTICO.
Para Ana el concepto de control de versiones le resulta muy abstracto. Juan va a intentar explicarle cómo funciona el mecanismo de control de versiones, tomando como ejemplo una herramienta que él conoce: CVS.

Las herramientas de control de versiones, suelen estar formadas por un conjunto de elementos, sobre los cuales, se pueden ejecutar órdenes e intercambiar datos entre ellos. Como ejemplo, vamos a analizar ha herramienta CVS.



Una herramienta de control de versiones, como CVS, es un sistema de mantenimiento de código fuente (grupos de archivos en general) extraordinariamente útil para grupos de desarrolladores que trabajan cooperativamente usando alguna clase de red. CVS permite a un grupo de desarrolladores trabajar y modificar concurrentemente ficheros organizados en proyectos. Esto significa que dos o más personas pueden modificar un mismo fichero sin que se pierdan los trabajos de ninguna. Además, las operaciones más habituales son muy sencillas de usar.

CVS utiliza una arquitectura cliente-servidor: un servidor guarda la versión actual del proyecto y su historia, y los clientes conectan al servidor para sacar una copia completa del proyecto, trabajar en esa copia y entonces ingresar sus cambios. Típicamente, cliente y servidor conectan utilizando Internet, pero cliente y servidor pueden estar en la misma máquina. El servidor normalmente utiliza un sistema operativo similar a Unix, mientras que los clientes CVS pueden funcionar en cualquier de los sistemas operativos más difundidos.

Los clientes pueden también comparar diferentes versiones de ficheros, solicitar una historia completa de los cambios, o sacar una "foto" histórica del proyecto tal como se encontraba en una fecha determinada o en un número de revisión determinado. Muchos proyectos de código abierto permiten el "acceso de lectura anónimo", significando que los clientes pueden sacar y comparar versiones sin necesidad de teclear una contraseña; solamente el ingreso de cambios requiere una contraseña en estos escenarios. Los clientes también pueden utilizar el comando de actualización con el fin de tener sus copias al día con la última versión que se encuentra en el servidor. Esto elimina la necesidad de repetir las descargas del proyecto completo.

El sistema de control de versiones está formado por un conjunto de componentes:

- ✓ **Repositorio**: Es el lugar de almacenamiento de los datos de los proyectos. Suele ser un directorio en algún ordenador.
- ✓ **Módulo**: En un directorio especifico del repositorio. Puede identificar una parte del proyecto o ser el proyecto por completo.
- ✓ **Revisión**: Es cada una de las versiones parciales o cambios en los archivos o repositorio completo. La evolución del sistema se mide en revisiones. Cada cambio se considera incremental.
- ✓ **Etiqueta**: Información textual que se añada a un conjunto de archivos o a un módulo completo para indicar alguna información importante.
- ✓ **Rama**: Revisiones paralelas de un módulo para efectuar cambios sin tocar la evolución principal. Se suele emplear para pruebas o para mantener los cambios en versiones antiguas.

Las órdenes que se pueden ejecutar son:

- ✓ **checkout**: obtiene un copia del trabajo para poder 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 Repositorio.

CASO PRÁCTICO.
Juan le ha explicado a Ana la los fundamentos del control de versiones, pero quiere profundizar más en el concepto de repositorio, ya que para él es la parte fundamental del control de versiones.

El repositorio es la parte fundamental de un sistema de control de versiones. Almacena toda la información y datos de un proyecto.

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

El repositorio centraliza todos los componentes de un mismo sistema, incluyendo las distintas versiones de cada componente. Con el repositorio, se va a conseguir un ahorro de espacio de almacenamiento, ya que estamos evitando guardar por duplicado, los elementos que son comunes a varias versiones. El repositorio nos va a facilitar el almacenaje de la información de la evolución del sistema, ya que, aparte de los datos en sí mismo, también almacena información sobre las versiones, temporización, etc.

El entorno de desarrollo integrado NetBeans usa como sistema de control de versiones CVS. Este sistema, tiene un componente principal, que es el repositorio. En el repositorio se deberán almacenar todos los ficheros de los proyectos, que puedan ser accedidos de forma simultánea por varios desarrolladores.

Cuando usamos un sistema de control de versiones, trabajamos de forma local, sincronizándonos con el repositorio, haciendo los cambios en nuestra copia local, realizando el cambio, se acomete el cambio en el repositorio. Para realizar la sincronización, en el entorno NetBeans, lo realizamos de varias formas:

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

Si tenemos un proyecto CVS versionado, con el que hemos trabajado, podemos abrirlo en el IDE y podremos acceder a las características de versionado. El IDE escanea nuestros proyectos abiertos y si contienen directorios CVS, el estado del archivo y la ayuda-contextual se activan automáticamente para los proyectos de versiones CVS.

¿Qué afirmación sobre control de versiones es correcta?

- ○ Solo puede existir una única versión de una clase
- ○ El almacenamiento de versiones es local a cada máquina
- ◉ El repositorio centraliza el almacenamiento de los datos

2.3 Herramientas de control de versiones.

CASO PRÁCTICO.
María quiere que Carlos conozca las herramientas de control de versiones que integra NetBeans, ya que es el Entorno que utilizan para su desarrollo. Dado que estamos utilizando un Entorno de Desarrollo Integrado, Carlos debe conocer las herramientas que incorpora NetBeans.

Durante el proceso de desarrollo de software, donde todo un equipo de programadores están colaborando en el desarrollo de un proyecto software, los cambios son continuos. Es por ello necesario que existan en todos los lenguajes de programación y en todos los entornos de programación, herramientas que gestionen el control de cambios.

Si nos centramos en Java, actualmente destacan dos herramientas de control de cambios: CVS y Subversion. CVS es una herramienta de código abierto ampliamente utilizada en numerosas organizaciones. Subversion es el sucesor natural de CVS, está rápidamente integrándose en los nuevos proyectos Java, gracias a sus características que lo hacen adaptarse mejor a las modernas prácticas de programación Java. Estas dos herramienta de control de versiones, se integran perfectamente en los entornos de desarrollado para Java, como NetBeans y Eclipse.

Otras herramientas de amplia difusión son:

- ✓ **SourceSafe**: Es una herramienta que forma parte del entorno de desarrollo Microsoft Visual Studio.
- ✓ **Visual Studio Team Foundation Server**: Es el sustituto de Source Safe. Es un productor que ofrece control de código fuente, recolección de datos, informes y seguimiento de proyectos, y está destinado a proyectos de colaboración de desarrollo de software.
- ✓ **Darcs**: Es un sistema de gestión de versiones distribuido. Algunas de sus características son: la posibilidad de hacer commits locales (sin conexión), cada repositorio es una rama en sí misma, independencia de un servidor central, posibilidad de renombrar ficheros, varios métodos de acceso como local, ssh, http y ftp, etc.
- ✓ **Git**: Esta herramienta de control de versiones, diseñada por Linus Torvalds,
- ✓ **Mercurial**: Esta herramienta funciona en Linux, Windows y Mac OS X, Es un programa de línea de comandos. Es una herramienta que permite que el desarrollo se haga distribuido, gestionando de forma robusta archivos de texto y binarios. Tiene capacidades avanzadas de ramificación e integración. Es una herramienta que incluye una interfaz web para su configuración y uso.

¿Qué herramienta no es una herramienta de Control de Versiones

- ○ Subversion
- ○ CVS
- ○ Mercurial
- ◉ PMD

2.4 Planificación de la gestión de configuraciones.

CASO PRÁCTICO.
El equipo de desarrollo de BK Programación decide reunirse para planificar la gestión de configuraciones, ya que la aplicación de Gestión Hotelera es amplia y compleja, y continuamente se están diseñando nuevos módulos, clases o métodos.

La Gestión de Configuraciones del software (GCS) es un conjunto de actividades desarrolladas para gestionar los cambios a lo largo del ciclo de vida.
La planificación de la Gestión de Configuraciones del software, está regulado en el estándar IEEE 828.

Cuando se habla de la gestión de configuraciones, se está haciendo referencia a la evolución de todo un conjunto de elementos. Una configuración es una combinación de versiones particulares de los componentes que forman un sistema consistente. 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 puede cambiar porque se añaden, eliminan o se modifican elementos. También puede cambiar, debido a la reorganización de los componentes, sin que estos cambien.

Como consecuencia de lo expuesto, es necesario disponer de un método, que nos permita designar las diferentes configuraciones de manera sistemática y planificada. De esta forma se facilita el desarrollo de software de manera evolutiva, mediante cambios sucesivos aplicados a partir de una configuración inicial hasta llegar a una versión final aceptable del producto.

La Gestión de Configuraciones de Software se va a componer de cuatro tareas básicas:

1. **Identificación**. Se trata de establecer estándares de documentación y un esquema de identificación de documentos.
2. **Control de cambios**. Consiste en la evaluación y registro de todos los cambios que se hagan de la configuración software.
3. **Auditorías de configuraciones**. Sirven, junto con las revisiones técnicas formales para garantizar que el cambio se ha implementado correctamente.
4. **Generación de informes**. El sistema software está compuesto por un conjunto de elementos, que evolucionan de manera individual, por consiguiente, se debe garantizar la consistencia del conjunto del sistema.

La planificación de la Gestión de Configuraciones de Software va a comprender diferentes actividades:

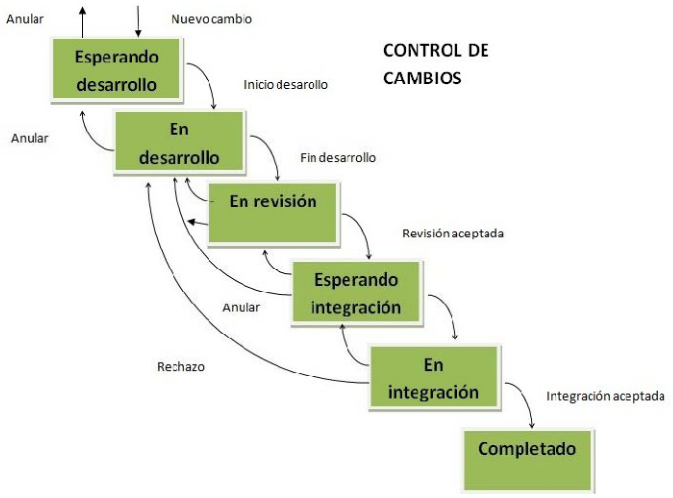
1. Introducción (propósito, alcance, terminología).
2. Gestión de GCS (organización, responsabilidades, autoridades, políticas aplicables, directivas y procedimientos).
3. Actividades GCS (identificación de la configuración, control de configuración, etc.).
4. Agenda GCS (coordinación con otras actividades del proyecto).
5. Recursos GCS (herramientas, recursos físicos y humanos).
6. Mantenimiento de GCS.

2.5 Gestión del cambio.

CASO PRÁCTICO.
Para gestionar el control de versiones de forma centralizada, Ada va a supervisar los cambios de versión que el equipo de Juan y de María están efectuando de forma independiente. Ada va a realizar una gestión del cambio centralizada y organizada.

Las herramientas de control de versiones no garantizan un desarrollo razonable, si cualquier componente del equipo de desarrollo de una aplicación puede realizar cambios e integrarlos en el repositorio sin ningún tipo de control. Para garantizar que siempre disponemos de una línea base para continuar el desarrollo, es necesario aplicar controles al desarrollo e integración de los cambios. El control de cambios es un mecanismo que sirve para la evaluación y aprobación de los cambios hechos a los elementos de configuración del software.

Pueden establecerse distintos tipos de control:



1. Control individual, antes de aprobarse un nuevo elemento. Cuando un elemento de la configuración está bajo control individual, el programador responsable cambia la documentación cuando se requiere. El cambio se puede registrar de manera informal, pero no genera ningún documento formal.
2. Control de gestión u organizado, conduce a la aprobación de un nuevo elemento. Implica un procedimiento de revisión y aprobación para cada cambio propuesto en la configuración. Como en el control individual, el control a nivel de proyecto ocurre durante el proceso de desarrollo pero es usado después de que haya sido aprobado un elemento de la configuración software. El cambio es registrado formalmente y es visible para la gestión.
3. Control formal, se realiza durante el mantenimiento. Ocurre durante la fase de mantenimiento del ciclo de vida software. El impacto de cada tarea de mantenimiento se evalúa por un Comité de Control de Cambios, el cuál aprueba la modificaciones de la configuración software.

¿Cuál de las siguientes no es una tarea básica de la Gestión de Configuraciones del Software?

- ☐ Control de cambios
- ☐ Generación de informes
- ☐ Auditorías de configuraciones
- ◉ Gestión del repositorio

2.6 Gestión de versiones y entregas.

CASO PRÁCTICO.
María ha desarrollado varias versiones del módulo de reservas de la aplicación de Gestión Hotelera. Le explica a Carlos como ha sido la evolución de cada versión del módulo, desde la primera versión, hasta la versión actual, que ella cree definitiva.

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

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

Variantes: En este caso, existen varias versiones del componente. El grafo ya no es una secuencia lineal, si no que adopta la forma de un árbol. La numeración de las versiones requerirá dos niveles. El primer número designa la variante (línea de evolución) y el segundo la versión particular (revisión) a lo largo de dicha variante.

La terminología que se usa para referirse a los elementos del grafo son:

- ✓ **Tronco** (trunk): Es la variante principal.
- ✓ **Cabeza** (head): Es la 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, suele ser necesario aplicar un mismo cambio a varias variantes.

Fusión de variantes: En determinados momentos puede dejar de ser necesario mantener una rama independiente. En esta caso se puede fundir con otra (MERGE).

Técnicas de almacenamiento: Como en la mayoría de los casos, las distintas versiones tienen en común gran parte de su contenido, se organiza el almacenamiento para que no se desaproveche espacio repitiendo los datos en común de varias versiones.

- ✓ **Deltas directos**: Se almacena la primera versión completa, y luego los cambios mínimos necesarios 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 los deltas directos.
- ✓ **Marcado selectivo**: Se almacena el texto refundido de todas las versiones como una secuencia lineal, marcando cada sección del conjunto con los números de versiones que corresponde.

En cuanto a la **gestión de entregas**, en primer lugar definimos el concepto de entrega como una instancia de un sistema que se distribuye a los usuarios externos al equipo de desarrollo.

La planificación de la entrega se ocupa de cuándo emitir una versión del sistema como una entrega. La entrega está compuesta por el conjunto de programas ejecutables, los archivos de configuración que definan como se configura la entrega para una instalación particular, los archivos de datos que se necesitan para el funcionamiento del sistema, un programa de instalación para instalar el sistema en el hardware de destino, documentación electrónica y en papel, y, el embalaje y publicidad asociados, diseñados para esta entrega Actualmente los sistemas se entregan en discos ópticos (CD o DVD) o como archivos de instalación descargables desde la red.

2.7 Herramientas CASE para la gestión de configuraciones.

CASO PRÁCTICO.
Ada decide utilizar herramientas CASE, de código abierto, para la gestión de configuraciones. Decide utilizar Bugzilla, ya que puede integrarse con NetBeans.

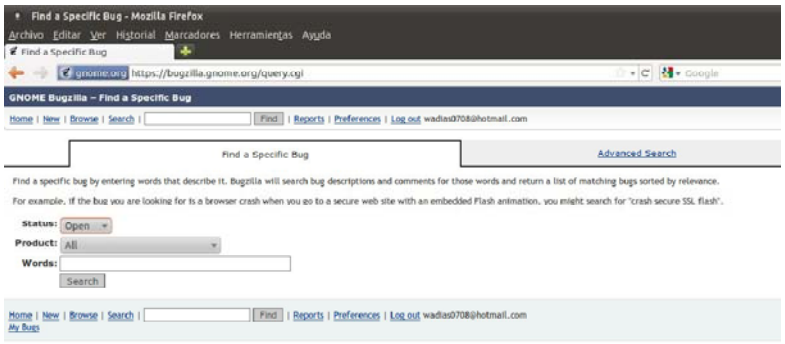
Los procesos de gestión de configuraciones están estandarizados y requieren la aplicación de procedimientos predefinidos, ya que hay que gestionar gran cantidad de datos. Cuando se construye un sistema a partir de versiones de componentes, un error de gestión de configuraciones, puede implicar que el software no trabaje correctamente. Por todo ello, las herramientas CASE de apoyo son imprescindibles para la gestión de configuraciones.

Las herramientas se pueden combinar con entornos de trabajo de gestión de configuraciones. Hay dos tipos de entornos de trabajo de Gestión de Configuraciones:

- ✓ **Entornos de trabajo abiertos**: Las herramientas de cada esta de Gestión de Configuraciones son integradas de acuerdo con procedimientos organizacionales estándar.

  Nos encontramos con bastantes herramientas de Gestión de Configuraciones comerciales y open-source disponibles para propósitos específicos. La gestión de cambios se puede llevar a cabo con herramientas de seguimiento (bug-tracking) como **Bugzilla**, la gestión de versiones a través de herramientas como **RCS** o **CVS**, y la construcción del sistema con herramientas como **Make** o **Imake**. Estas herramientas son open-source y están disponibles de forma gratuita.

- ✓ **Entornos integrados**: Estos entorno ofrecen facilidades integradas para gestión de versiones, construcción del sistema o seguimiento de los cambios. Por ejemplo, está el proceso de control de Cambios Unificado de **Rational**, que se basa en un entorno de Gestión de Configuraciones que incorpora **ClearCase** para la construcción y gestión de versiones del sistema y **ClearQuest** para el seguimiento de los cambios. Los entornos de Gestión de Configuraciones integrado, ofrecen la ventaja de un intercambio de datos sencillos, y el entorno ofrece una base de datos de Gestión de Configuraciones integrada.



La Gestión de Configuraciones:

- ☐ Verifica la evolución de la implementación de una clase
- ☐ Es una tarea que se puede gestionar con herramientas de Control de Versiones
- ◉ Es un conjunto de actividades desarrolladas para gestionar los cambios a lo largo del ciclo de vida del software

2.8 Clientes de control de versiones integrados en el entorno de desarrollo.

CASO PRÁCTICO.
Juan le va a enseñar a Ana los clientes de control de versiones que hay en NetBeans, para que aprenda a utilizarlos e integrarlos en los proyectos que realice de ahora en adelante.

Los entornos de desarrollo actuales integran de forma generalizada, clientes de control de versiones. En algunos caso, como puede ser el IDE Microsoft Visual Studio, se utiliza Visual Studio Team Foundation, de tal forma, que se realiza una gestión centralizada de un proyecto desarrollado por un

equipo, incluyendo la gestión de configuraciones de software, la gestión de versiones, y demás aspectos de un desarrollo en equipo.

Los entorno de desarrollo Open-Source, como Eclipse y NetBeans, los integran de manera directa, o bien instalándolos como plug-in (*complementos*). Los clientes de control de versiones más destacados son:

- ✓ CVS
- ✓ Subversion
- ✓ Mercurial.

Debes visitar el siguiente enlace donde se puede ver la guía de uso de subversión en Netbeans. Está en inglés pero es conveniente que le eches un vistazo para conocer cómo funciona Subversion.

http://netbeans.org/kb/docs/ide/subversion.html

¿Qué cliente de Gestión de Versiones no incorpora NetBeans?

- ◉ VS Team Foundation
- ○ CVS
- ○ Mercurial

3. Documentación.

CASO PRÁCTICO
Juan y María saben de la importancia de tener documentado el código y todo el proceso de desarrollo de software. Al mismo tiempo que codifican y refactorizan, dejan constancia documental de lo que van haciendo.
Como están desarrollando con NetBeans, y con el objetivo de facilitar su trabajo de documentación, van a utilizar JavaDoc. Ana y Antonio consiguen entender el funcionamiento de la aplicación, y la función de cada método, gracias a los comentarios que insertan en el código los dos programadores principales.

El proceso de documentación de código, es uno de los aspectos más importantes de la labor de un programador.

Documentar el código nos sirve para explicar su funcionamiento, punto por punto, de forma que cualquier persona que lea el comentario, puede entender la finalidad del código.

La labor de documentación es fundamental para la detección de errores y para su mantenimiento posterior, que en muchos casos, es realizado por personas diferentes a las que intervinieron en su creación. Hay que tener en cuenta que todos los programas tienen errores y todos los programas sufren modificaciones a los largo de su vida.

La documentación añade explicaciones de la función del código, de las características de un método, etc. Debe tratar de explicar todo lo que no resulta evidente. Su objetivo no es repetir lo que hace el código, sino explicar por qué se hace.

La documentación explicará cual es la finalidad de un clase, de un paquete, qué hace un método, para que sirve una variable, qué se espera del uso de una variable, qué algoritmo se usa, por qué hemos implementado de una manera y de otro, qué se podría mejorar en el futuro, etc.

El siguiente enlace nos muestra el estilo de programación a seguir en Java, así como la forma de documentar y realizar comentarios de un código. (En inglés)
http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html

3.1 Uso de comentarios.

CASO PRÁCTICO.
Carlos está aprendiendo muchas cosas con María. Sin embargo hay algunos métodos y clases que ha implementado María, y que no logra entender.
María le comenta que va a incluir comentarios en su código, y le va a enseñar la forma correcta de hacerlo.

Uno de los elementos básicos para documentar código, es el uso de comentarios. Un comentario es una anotación que se realiza en el código, pero que el compilador va a ignorar, sirve para indicar a los desarrolladores de código diferentes aspectos del código que pueden ser útiles. En principio, los comentarios tienen dos propósitos diferentes:

- ✓ Explicar el objetivo de las sentencias. De forma que el programador o programadora, sepa en todo momento la función de esa sentencia, tanto si lo diseñaron como si son otros los que quieren entenderlo o modificarlo.
- ✓ Explicar qué realiza un método, o clase, no cómo lo realiza. En este caso, se trata de explicar los valores que va a devolver un método, pero no se trata de explicar cómo se ha diseñado.

En el caso del lenguaje Java, C# y C, los comentarios, se implementan de forma similar. Cuando se trata de explicar la función de una sentencia, se usan los caracteres **//** seguidos del comentario, o con los caracteres **/*** y ***/**, situando el comentario entre ellos: ***/* *comentario* */***

Otro tipo de comentarios que se utilizan en Java, son los que se utilizan para explicar qué hace un código, se denominan comentarios JavaDoc y se escriben empezando por **/**** y terminando con ***/** , estos comentarios pueden ocupar varias líneas. Este tipo de comentarios tienen que seguir una estructura prefijada.

Los comentarios son obligatorios con JavaDoc, y se deben incorporar al principio de cada clase, al principio de cada método y al principio de cada variable de clase. No es obligatorio, pero en muchas situaciones es conveniente, poner los comentarios al principio de un fragmento de código que no resulta lo suficientemente claro, a la largo de bucles, o si hay alguna línea de código que no resulta evidente y pueda llevarnos a confusión.

Hay que tener en cuenta, que si el código es modificado, también se deberán modificar los comentarios.

3.2 Alternativas.

CASO PRÁCTICO.
Juan le explica a María, que para documentar el software, existen diferentes formas de hacerlo y distintas herramientas en el mercado que automatizan la tarea de documentación.
En la actualidad, el desarrollo rápido de aplicaciones, en muchos casos, va en detrimento de una buena documentación del código. Si el código no está documentado, puede resultar bastante difícil de entender, y por tanto de solucionar errores y de mantenerlo.

La primera alternativa que surge para documentar código, son los comentarios. Con los comentarios, documentamos la funcionalidad de una línea de código, de un método o el comportamiento de una determinada clase.

Existen diferentes herramientas que permiten automatizar, completar y enriquecer nuestra documentación.

Podemos citar JavaDoc, SchemeSpy y Doxygen, que producen una documentación actualizada, precisa y utilizable en línea, incluyendo además, con SchemeSpy y Doxygen, modelos de bases de datos gráficos y diagramas.

Insertando comentario en el código más difícil de entender, y utilizando la documentación generada por alguna de las herramientas citadas anteriormente, se genera la suficiente información para ayudar a cualquier nuevo programador o programadora

Un comentario en formato JavaDoc.

- Utiliza los caracteres //
- Comienzan con /* y termina por */
- Comienza por /** y terminan por */

3.3 Documentación de clases.

CASO PRÁCTICO.
Juan va a utilizar JavaDoc para documentar las clases que ha desarrollado. Ana se da cuenta de la ayuda tan importante que ofrece, tanto para el futuro mantenimiento de la aplicación como para entender su funcionamiento, tener documentadas las clases.

Las clases que se implementan en una aplicación, deben de incluir comentarios. Al utilizar un entorno de programación para la implementación de la clase, debemos seguir una serie de pautas, muchas de las cuales las realiza el IDE de forma trasparente, en el diseño y documentación del código. Cuando se implementa una clase, se deben incluir comentarios. En el lenguaje Java, los criterios de documentación de clases, son los establecidos por JavaDoc.

Los comentarios de una clase deben comenzar con **/**** y terminar con ***/** . Entre la información que debe incluir un comentario de clase debe incluirse, al menos las etiquetas **@autor** y **@version**, donde **@autor** identifica el nombre del autor o autora de la clase y **@version**, la identificación de la versión y fecha.

Con el uso de los entornos de desarrollo, las etiquetas se añaden de forma automática, estableciendo el **@autor** y la **@version** de la clase de forma transparente al programador-programadora. También se suele añadir la etiqueta **@see**, que se utiliza para referenciar a otras clases y métodos.

Dentro de la la clase, también se documentan los constructores y los métodos. Al menos se indican las etiquetas:

- ✓ **@param**: seguido del nombre, se usa para indicar cada uno de los parámetros que tienen el constructor o método.
- ✓ **@return**: si el método no es void, se indica lo que devuelve.
- ✓ **@exception**: se indica el nombre de la excepción, especificando cuales pueden lanzarse.
- ✓ **@throws**: se indica el nombre de la excepción, especificando las excepciones que pueden lanzarse.

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

3.4 Herramientas.

CASO PRÁCTICO.
Para documentar el código, el equipo de desarrollo de BK Programación, ha decidido utilizar JavaDoc, por lo que todos los componentes del equipo de desarrollo, debe familiarizarse con la herramienta.

Los entornos de programación que implementa Java, como Eclipse o Netbeans, incluyen una herramienta que va a generar páginas HTML de documentación a partir de los comentarios incluidos en el código fuente. La herramienta ya se ha indicado en los puntos anteriores, y es JavaDoc. Para que JavaDoc pueda generar las páginas HTML es necesario seguir una serie de normas de documentación en el código fuente, estas son:

- ✓ Los comentarios JavaDoc deben empezar por /** y terminar por */.
- ✓ Los comentarios pueden ser a nivel de clase, a nivel de variable y a nivel de método.
- ✓ La documentación se genera para métodos public y protected.
- ✓ Se puede usar tag para documentar diferentes aspectos determinados del código, como parámetros Los tags más habituales son los siguientes:

Tipo de tag	Formato	Descripción
Todos.	@see.	Permite crear una referencia a la documentación de otra clase o método.
Clases.	@version.	Comentario con datos indicativos del número de versión.
Clases.	@author.	Nombre del autor.
Clases.	@since.	Fecha desde la que está presente la clase.

Métodos.	@param.	Parámetros que recibe el método.
Métodos.	@return.	Significado del dato devuelto por el método
Métodos.	@throws.	Comentario sobre las excepciones que lanza.
Métodos.	@deprecated.	Indicación de que el método es obsoleto.

TEMA 5

Contenido

[DISEÑO ORIENTADO A OBJETOS. ELABORACIÓN DE DIAGRAMAS ESTRUCTURALES]

José Luis Comesaña Cabeza --- 2011/2012

Entornos de Desarrollo del curso de "*Desarrollo de Aplicaciones Web*"

Diseño orientado a objetos. Elaboración de diagramas estructurales.

Caso práctico
En la empresa siguen trabajando en diferentes aplicaciones con un nivel alto de complejidad, se desarrolla para diferentes plataformas, en entornos de ventanas, para la web, dispositivos móviles, etc. ***Ada*** *lleva un tiempo observando a su equipo, y a pesar de que ya han hablado de las diferentes fases de desarrollo del software, y que están descubriendo nuevos entornos de programación que han facilitado su trabajo enormemente, se ha dado cuenta de que todavía hay una asignatura pendiente, sus empleados no utilizan herramientas ni crean documentos en las fases previas del desarrollo de una aplicación, a pesar de ser algo tan importante como el resto de fases del proceso de elaboración de software. Tampoco construyen modelos que ayuden a hacerse una idea de cómo resultará el proyecto. Estos documentos y modelos son muy útiles para que todo el mundo se ponga de acuerdo en lo que hay que hacer, y cómo van a hacerlo.*
Como ***Ada*** *muy bien conoce, un proyecto de software tendrá éxito sólo si produce un software de calidad, consistente y sobre todo que satisfaga las necesidades de los usuarios que van a utilizar el producto resultante.*
Para desarrollar software de calidad duradera, hay que idear una sólida base arquitectónica que sea flexible al cambio.
Incluso para producir software de sistemas pequeños sería bueno hacer análisis y modelado ya que redunda en la calidad, pero lo que sí es cierto, es que cuanto más grande y complejos son los sistemas más importante es hacer un buen modelado ya que nos ayudará a entender el comportamiento del sistema en su totalidad. Y cuando se trata de sistemas complejos el modelado nos dará una idea de los recursos necesarios (tanto humanos como materiales) para abordar el proyecto. También nos dará una visión más amplia de cómo abordar el problema para darle la mejor solución.
Ada *se da cuenta de que el equipo necesita conocer procedimientos de análisis y diseño de software, así como alguna herramienta que permita generar los modelos y la documentación asociada, así que decide reunir a su equipo para empezar a tratar este tema...*

1.- Introducción a la orientación a objetos.

Caso práctico
Ya en la sala de reuniones...
—Deberíamos empezar por revisar cual es la situación actual. Como ya sabéis existen diferentes lenguajes de programación que se comportan de manera diferente, y esto determina en gran medida el enfoque que se le da al análisis previo. No es lo mismo un lenguaje estructurado que uno orientado a objetos. Tendríamos que conocer las características de ambos enfoques para entender un poco mejor cómo se analizan.
—Es cierto —contesta ***Juan*** *—desde que empecé en el mundo de la informática esto ha cambiado un poco, así que he tenido que ir investigando para adaptarme a los nuevos lenguajes de programación, si queréis, os pongo al día brevemente, ...*

La construcción de software es un proceso cuyo objetivo es dar solución a problemas utilizando una herramienta informática y tiene como resultado la construcción de un programa informático. Como en cualquier otra disciplina en la que se obtenga un producto final de cierta complejidad, si queremos obtener un producto de calidad, es preciso realizar un proceso previo de análisis y especificación del proceso que vamos a seguir, y de los resultados que pretendemos conseguir.

El enfoque estructurado.

Sin embargo, cómo se hace es algo que ha ido evolucionando con el tiempo, en un principio se tomaba el problema de partida y se iba sometiendo a un proceso de división en subproblemas más pequeños reiteradas veces, hasta que se llegaba a problemas elementales que se podía resolver

utilizando una función. Luego las funciones se hilaban y entretejían hasta formar una solución global al problema de partida. Era, pues, un proceso centrado en los procedimientos, se codificaban mediante funciones (*conjunto de sentencias escritas en un lenguaje de programación que operan sobre un conjunto de parámetros y producen un resultado*) que actuaban sobre estructuras de datos (*conjunto de una colección de datos y de las funciones que modifican esos datos, que recrean una entidad con sentido, en el contexto de un problema*), por eso a este tipo de programación se le llama programación estructurada (*paradigma de programación que postula que una programa informático no es más que una sucesión de llamadas a funciones, bien sean del sistema o definidas por el usuario*). Sigue una filosofía en la que se intenta aproximar qué hay que hacer, para así resolver un problema.

Enfoque orientado a objetos.

La orientación a objetos ha roto con esta forma de hacer las cosas. Con este nuevo paradigma (*modelo o patrón aplicado a cualquier disciplina científica u otro contexto epistemológico*) el proceso se centra en simular los elementos de la realidad asociada al problema de la forma más cercana posible. La abstracción (*aislar un elemento de su contexto o del resto de elementos que le acompañan para disponer de ciertas características que necesitamos excluyendo las no pertinentes. Con ello capturamos algo en común entre las diferentes instancias con objeto de controlar la complejidad del software*) que permite representar estos elementos se denomina objeto, y tiene las siguientes características:

- ✓ Está formado por un conjunto de **atributos**, que son los datos que le caracterizan y
- ✓ Un conjunto de **operaciones** que definen su comportamiento. Las operaciones asociadas a un objeto actúan sobre sus atributos para modificar su estado. Cuando se indica a un objeto que ejecute una operación determinada se dice que se le pasa un **mensaje**.

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

Cuando se ejecuta un programa orientado a objetos 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 se libera la memoria.

Todo acerca del mundo de la orientación a objetos se encuentra en la página oficial del Grupo de gestión de objetos:
http://www.omg.org/index.htm

2.- Conceptos de Orientación a Objetos.

Caso práctico
*—De acuerdo, buen resumen **Juan**, sin embargo, los últimos proyectos que han entrado a la empresa se han desarrollo en su totalidad mediante software orientado a objetos, hemos usado PHP con Javascript, pero sobre todo Java, que es un lenguaje basado en objetos, así que sería necesario que analizáramos con un poco más de detenimiento el enfoque orientado a objetos, que características presenta, y qué ventajas tiene sobre otros.*
*—¡Gracias, **Ada**!, también tengo alguna información sobre eso...*

Como hemos visto la orientación a objetos trata de acercarse al contexto del problema lo más posible por medio de la simulación de los elementos que intervienen en su resolución y basa su desarrollo en los siguientes conceptos:

- ✓ **Abstracción**: Permite capturar las características y comportamientos similares de un conjunto de objetos con el objetivo de darles una descripción formal. La abstracción es clave en el proceso de análisis y diseño orientado a objetos, ya que mediante ella podemos llegar a armar un conjunto de clases que permitan modelar la realidad, o el problema que se quiere atacar.
- ✓ **Encapsulación**: Significa reunir todos los elementos que pueden considerarse pertenecientes a una misma entidad, al mismo nivel de abstracción. Esto permite aumentar la cohesión de los componentes del sistema. Algunos autores confunden este concepto con el principio de ocultación, principalmente porque se suelen emplear conjuntamente.
- ✓ **Modularidad**: Propiedad que permite subdividir una aplicación en partes más pequeñas (llamadas módulos), cada una de las cuales debe ser tan independiente como sea posible de la aplicación en sí y de las restantes partes. En orientación a objetos es algo consustancial, ya que los objetos se pueden considerar los módulos más básicos del sistema.
- ✓ **Principio de ocultación**: Aísla las propiedades de un objeto contra su modificación por quien no tenga derecho a acceder a ellas. Reduce la propagación de efectos colaterales cuando se producen cambios.
- ✓ **Polimorfismo**: Consiste en reunir bajo el mismo nombre comportamientos diferentes. La selección de uno u otro depende del objeto que lo ejecute.
- ✓ **Herencia**: Relación que se establece entre objetos en los que unos utilizan las propiedades y comportamientos de otros formando una jerarquía. Los objetos heredan las propiedades y el comportamiento de todas las clases a las que pertenecen.
- ✓ **Recolección de basura**: Técnica por la cual el entorno de objetos se encarga de destruir automáticamente los objetos, y por tanto desvincular su memoria asociada, que hayan quedado sin ninguna referencia a ellos.

2.1.- Ventajas de la orientación a objetos.

Caso práctico
*—Además la orientación a objetos cuenta con una serie de ventajas que nos vienen muy bien a los que nos dedicamos a la construcción de software, sobre todo porque nos facilitan su construcción y mantenimiento al dividir un problema en módulos claramente independientes y que, además, cuando ya tenemos suficientemente probados y completos podemos utilizar en otras aplicaciones, la verdad que ahorra bastante tiempo y esfuerzo... —argumenta **Juan**.*

Este paradigma tiene las siguientes **ventajas** con respecto a otros:

1. Permite desarrollar software en mucho menos tiempo, con menos coste y de mayor calidad gracias a la reutilización (*utilizar artefactos existentes durante la construcción de nuevo software. Esto aporta calidad y seguridad al proyecto, ya que el código reutilizado ya ha sido probado*) porque al ser completamente modular facilita la creación de código reusable dando la posibilidad de reutilizar parte del código para el desarrollo de una aplicación similar.
2. Se consigue aumentar la calidad de los sistemas, haciéndolos más extensibles (*principio de diseño en el desarrollo de sistemas informáticos que tiene en cuenta el futuro crecimiento del sistema. Mide la capacidad de extender un sistema y*

el esfuerzo necesario para conseguirlo) ya que es muy sencillo aumentar o modificar la funcionalidad de la aplicación modificando las operaciones.

3. El software orientado a objetos es más **fácil de modificar y mantener** porque se basa en criterios de modularidad y encapsulación en el que el sistema se descompone en objetos con unas responsabilidades claramente especificadas e independientes del resto.
4. La tecnología de objetos facilita la adaptación al entorno y el cambio haciendo aplicaciones escalables (*propiedad deseable de un sistema, red o proceso que le permite hacerse más grande sin rehacer su diseño y sin disminuir su rendimiento*). Es sencillo modificar la estructura y el comportamiento de los objetos sin tener que cambiar la aplicación.

¿Cuál es la afirmación más adecuada al paradigma de orientación a objetos?

- ☐ Permite crear aplicaciones basadas en módulos de software que representan objetos del entorno del sistema, por lo que no son apropiados para dar solución a otros problemas.
- ☐ Tiene como objetivo la creación de aplicaciones basadas en abstracciones de datos estáticas y de difícil ampliación
- ☐ Permite crear aplicaciones cuyo mantenimiento es complicado porque las modificaciones influyen a todos los objetos del sistema
- ◉ **Permite crear aplicaciones basadas en módulos que pueden reutilizarse, de fácil modificación y que permiten su ampliación en función del crecimiento del sistema**

2.2.- Clases, atributos y métodos.

Caso práctico

—De acuerdo, ahora conocemos las características básicas y ventajas de usar la orientación a objetos, ¿qué más nos haría falta? ¿Quizá sus estructuras básicas?

*—Yo puedo contaros algo sobre eso, —comenta **Juan**— lo estudié en el Ciclo Formativo.*

Los objetos de un sistema se abstraen, en función de sus características comunes, en clases. Una clase está formada por un conjunto de procedimientos y datos que resumen características similares de un conjunto de objetos. La clase tiene dos propósitos: definir **abstracciones** y favorecer la **modularidad**.

Una clase se describe por un conjunto de elementos que se denominan **miembros** y que son:

- ✓ **Nombre**.
- ✓ **Atributos**: conjunto de características asociadas a una clase. Pueden verse como una relación binaria entre una clase y cierto dominio formado por todos los posibles valores que puede tomar cada atributo. Cuando toman valores concretos dentro de su dominio definen el estado del objeto. Se definen por su nombre y su tipo, que puede ser simple o compuesto como otra clase.
- ✓ **Protocolo**: Operaciones (métodos, 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 por un objeto. Los métodos determinan como actúan los objetos cuando reciben un mensaje, es decir, cuando se requiere que el objeto realice una acción descrita en un método se le envía un mensaje. El conjunto de mensajes a los cuales puede responder un objeto se le conoce como *protocolo del objeto*.

Por ejemplo, si tenemos un objeto icono, tendrá como atributos el tamaño, o la imagen que muestra, y su protocolo puede constar de mensajes invocados por el clic del botón de un ratón cuando el usuario pulsa sobre el icono. De esta forma los mensajes son el único conducto que conectan al objeto con el mundo exterior.

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

Un objeto es una concreción de una clase, es decir, en un objeto se concretan valores para los atributos definidos en la clase, y además, estos valores podrán modificarse a través del paso de mensajes al objeto.

- [x] **Verdadero**
- [ ] Falso

2.3.- Visibilidad.

Caso práctico

—Pues creo que ya lo tenemos todo...

*—No creas, —dice **Ada** que siempre sabe algo más, que el resto desconoce— en orientación a objetos, existe un concepto muy importante, que es el de visibilidad, permite definir hasta qué punto son accesibles los atributos y métodos de una clase, por regla general, cuando definimos atributos los ocultamos, para que nadie pueda modificar el estado del objeto, y dejamos los métodos abiertos, porque son los que permiten el paso de mensajes entre objetos...*

El principio de ocultación es una propiedad de la orientación a objetos que consiste en aislar el estado de manera que sólo se puede cambiar mediante las operaciones definidas en una clase. Este aislamiento protege a los datos de que sean modificados por alguien que no tenga derecho a acceder a ellos, eliminando efectos secundarios e interacciones. Da lugar a que las clases se dividan en dos partes:

1. **Interfaz**: captura la visión externa de una clase, abarcando la abstracción del comportamiento común a los ejemplos de esa clase.
2. **Implementación**: comprende cómo se representa la abstracción, así como 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 que se permite a atributos y métodos y que podemos establecer como:

- ✓ **Público**: Se pueden acceder desde cualquier clase y cualquier parte del programa.
- ✓ **Privado**: Sólo se pueden acceder desde operaciones de la clase.
- ✓ **Protegido**: Sólo se pueden acceder desde operaciones de la clase o de clases derivadas (*cuando se utiliza la herencia es la clase que hereda los atributos y métodos de la clase base*) en cualquier nivel.

Como norma general a la hora de definir la visibilidad tendremos en cuenta que:

- ✓ El estado debe ser privado. 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 (si no se utilizan desde clases derivadas) o protegidas (si se utilizan desde clases derivadas).

¿Desde dónde se puede acceder al estado de una clase?

- [ ] Desde cualquier zona de la aplicación.
- [ ] Desde la clase y sus clases derivadas.
- [x] **Solo desde los métodos de la clase.**

2.4.- Objetos. Instanciación.

Caso práctico
Antonio *ha asistido a esta reunión como parte de su formación laboral, pero se encuentra algo perdido entre tantos conceptos:*
—A ver, estamos todo el tiempo hablando de que las clases tienen atributos y métodos, luego, que los objetos se pasan mensajes, que son los que modifican los atributos, entonces, ¿no son lo mismo?, ¿qué diferencia hay?

Una clase es una abstracción que define las características comunes de un conjunto de objetos relevantes para el sistema.

Cada vez que se construye un objeto en un programa informático a partir de una clase se crea lo que se conoce como instancia (*objeto de una clase, creado en tiempo de ejecución con un estado concreto*) de esa clase. Cada instancia en el sistema sirve como modelo de un objeto del contexto del problema relevante para su solución, que puede realizar un trabajo, informar y cambiar su estado, y "comunicarse" con otros objetos en el sistema, sin revelar cómo se implementan estas características.

Un objeto se define por:

- ✓ **Su estado**: es la concreción de los atributos definidos en la clase a un valor concreto.
- ✓ **Su comportamiento**: definido por los métodos públicos de su clase.
- ✓ **Su tiempo de vida**: intervalo de tiempo a lo largo del programa en el que el objeto existe. Comienza con su creación a través del mecanismo de **instanciación** y finaliza cuando el objeto se destruye.

La encapsulación y el ocultamiento aseguran que los datos de un objeto están ocultos, con lo que no se pueden modificar accidentalmente por funciones externas al objeto.

Mientras que un objeto es una entidad que existe en el tiempo y el espacio, una clase representa sólo una abstracción, "la esencia" del objeto, si se puede decir así.
Grady Booch

Ejemplo de objetos:

- ✓ **Objetos físicos**: aviones en un sistema de control de tráfico aéreo, casas, parques.
- ✓ **Elementos de interfaces gráficas de usuario**: ventanas, menús, teclado, cuadros de diálogo.
- ✓ **Animales**: animales vertebrados, animales invertebrados.
- ✓ **Tipos de datos definidos por el usuario**: Datos complejos, Puntos de un sistema de coordenadas.
- ✓ **Alimentos**: carnes, frutas, verduras.

Existe un caso particular de clase, llamada **clase abstracta**, que, por sus características, no puede ser instanciada. Se suelen usar para definir métodos genéricos relacionados con el sistema que no serán traducidos a objetos concretos, o para definir métodos de base para clases derivadas.

3.- UML.

Caso práctico
Ahora que el equipo conoce los fundamentos de la orientación a objetos llega el momento de ver cómo pueden poner en práctica los conocimientos adquiridos.
***Ada** está interesada, sobre todo, en que sean capaces de representar las clases de los proyectos que están desarrollando y como se relacionan entre ellas. Para ello decide comenzar comentando las características de un lenguaje de modelado de sistemas orientados a objetos llamado UML. Este lenguaje permite construir una serie de modelos, a través de diagramas de diferentes visiones de un proyecto.*
—Es importante apreciar como estos modelos, nos van a permitir poner nuestras ideas en común utilizando un lenguaje específico, facilitarán la comunicación, que como sabéis, es algo esencial para que nuestro trabajo en la empresa sea de calidad.

Una empresa de software con éxito es aquella que produce de manera consistente software de calidad que satisface las necesidades de los usuarios. El modelado es la parte esencial de todas las actividades que conducen a la producción de software de calidad.

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 software orientado a objetos. Se ha convertido en el estándar de facto de la industria, debido a que ha sido concebido por los autores de los tres métodos más usados de orientación a objetos: Grady Booch, Ivar Jacobson y Jim Rumbaugh, de hecho las raíces técnicas de UML son:

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

UML permite a los desarrolladores y desarrolladoras visualizar el producto de su trabajo en esquemas o diagramas estandarizados denominados modelos (*representación gráfica o esquemática de una realidad, sirve para organizar y comunicar de forma clara los elementos que involucran un todo. Esquema teórico de un sistema o de una realidad compleja que se elabora para facilitar su comprensión y el estudio de su comportamiento*) que representan el sistema desde diferentes perspectivas.

¿Porqué es útil modelar?

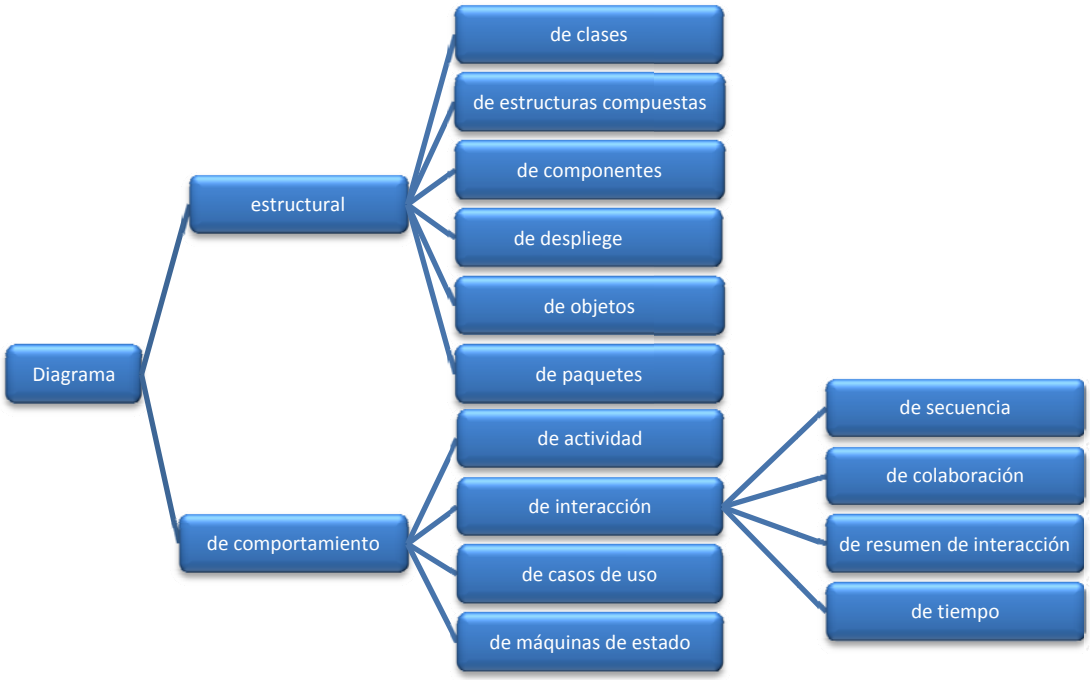
- ✓ Porque permite utilizar un lenguaje común que facilita la comunicación entre el equipo de desarrollo.
- ✓ Con UML podemos documentar todos los artefactos (*información que es utilizada o producida mediante un proceso de desarrollo de software. Pueden ser artefactos un modelo, una descripción o un software*) de un proceso de desarrollo (requisitos *(condiciones que debe cumplir un proyecto software. Suelen venir definidos por el cliente. Permiten definir los objetivos que debe cumplir un proyecto software)*, arquitectura, pruebas, versiones,...) por lo que se dispone de 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 (*conjunto de decisiones significativas acerca de la organización de un sistema software, la selección de los elementos estructurales a partir de los cuales se compone el sistema, y las interfaces entre ellos. Junto con su comportamiento, tal y como se especifica en las colaboraciones entre esos elementos, la composición de estos elementos estructurales y de comportamiento en subsistemas progresivamente mayores y el estilo arquitectónico que guía esta organización, estos elementos y sus interfaces, sus colaboraciones y su composición*), utilizando estas tecnologías podemos incluso indicar qué módulos de software vamos a desarrollar y sus relaciones, o en qué nodos hardware se ejecutarán cuando trabajamos con sistemas distribuidos.
- ✓ Permite especificar todas las decisiones de análisis, diseño e implementación, construyéndose modelos precisos, no ambiguos y completos.

Además UML puede conectarse a lenguajes de programación mediante ingeniería directa (*transformación de un modelo en código a través de su traducción a un determinado lenguaje de programación*) e inversa (*transformación del código en un modelo a través de su traducción desde un determinado lenguaje de programación*), como veremos.

3.1.- Tipos de diagramas UML.

Caso práctico
*Cuando **María** estudió el ciclo formativo no llegó a ver estas tecnologías con tanto detenimiento, así que está asimilándolo todo poco a poco:*
—De acuerdo, UML describe el sistema mediante una serie de modelos que ofrecen diferentes puntos de vista. Pero ¿qué tenemos que hacer para representar un modelo?, ¿en qué consiste exactamente?
—Utilizaremos diagramas, que son unos grafos en los que los nodos definen los elementos del diagrama, y los arcos las relaciones entre ellos.

UML define un sistema como una **colección de modelos** que describen sus diferentes perspectivas. Los modelos se implementan en una serie de diagramas que son representaciones gráficas de una colección de elementos de modelado, a menudo dibujado como un grafo conexo de arcos (relaciones) y vértices (otros elementos del modelo).



Un diagrama UML se compone de cuatro tipos de elementos:

- **Estructuras**: Son los nodos del grafo y definen el tipo de diagrama.
- **Relaciones**: Son los arcos del grafo que se establecen entre los elementos estructurales.
- **Notas**: Se representan como un cuadro donde podemos escribir comentarios que nos ayuden a entender algún concepto que queramos representar.
- **Agrupaciones**: Se utilizan cuando modelamos sistemas grandes para facilitar su desarrollo por bloques.

y se clasifican en:

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

- **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. Dentro de este grupo están los diagramas de interacción.

En la imagen aparecen todos los diagramas organizados según su categoría. Se han destacado aquellos que pertenecen al estándar UML 2.0, más novedosos. En total se describen trece diagramas para modelar diferentes aspectos de un sistema, sin embargo no es necesario usarlos todos, dependerá del tipo de aplicación a generar y del sistema, es decir, se debe generar un diagrama sólo si es necesario.

Un 80% de las aplicaciones se pueden modelar con el 20% de los diagramas UML.

En el siguiente enlace tienes un documento con la descripción de los diagramas UML.
Diagramas UML

3.2.- Herramientas para la elaboración de diagramas UML.

Caso práctico
—Ahora que conocemos los diagramas que podemos generar para describir nuestro sistema, sería buena idea buscar alguna herramienta que nos ayude a elaborarlos. ¡No sería nada práctico andar todo el día con la libreta a cuestas!
—Lo que nos permite conocer a un buen desarrollador es que siempre hace un buen esquema inicial de cada proyecto, y eso puede hacerse en miles de soportes, desde una libreta a un servilleta, cualquier cosa que te permita hacer un pequeño dibujo, no obstante tienes razón. El uso de herramientas, además de facilitar la elaboración de los diagramas, tiene otras ventajas, como la integración en entornos de desarrollo, con lo que podremos generar el código base de nuestra aplicación desde el propio diagrama.
-¡Guau, eso sí es facilitar el trabajo!

La herramienta más simple que se puede utilizar para generar diagramas es lápiz y papel, hoy día, sin embargo, podemos acceder a herramientas CASE que facilitan en gran manera el desarrollo de los diagramas UML. Estas herramientas suelen contar con un entorno de ventanas tipo wysiwyg (**W**hat **Y**ou **S**ee **I**s **W**hat **Y**ou **G**et), permiten documentar los diagramas e integrarse con otros entornos de desarrollo incluyendo la generación automática de código y procedimientos de ingeniería inversa.

Podemos encontrar, entre otras, las siguientes herramientas:

- **Rational Systems Developer de IBM**: Herramienta propietaria que permite el desarrollo de proyectos software basados en la metodología UML. Desarrollada en origen por los creadores de UML ha sido recientemente absorbida por IBM. Ofrece versiones de prueba, y software libre para el desarrollo de diagramas UML.

  Si sientes curiosidad puedes seguir este enlace a la página oficial de Rational Systems Developer:
  http://www-01.ibm.com/software/rational/

- Visual Paradigm for UML (VP-UML): Incluye una versión para uso no comercial que se distribuye libremente sin más que registrarse para obtener un archivo de licencia. Incluye diferentes módulos para realizar desarrollo UML, diseñar bases de datos, realizar actividades de ingeniería inversa y diseñar con Agile. Es compatible con los IDE de Eclipse, Visual Studio .net, IntellijDEA y NetBeans. Multiplataforma, incluye instaladores para Windows y Linux.
  Aquí tienes el enlace a la página oficial de Visual Paradigm.
  http://www.visual-paradigm.com/
- **ArgoUML**: se distribuye bajo licencia Eclipse. Soporta los diagramas de UML 1.4, y genera código para java y C++. Para poder ejecutarlo se necesita la plataforma java. Admite ingeniería directa e inversa.

Aquí tienes el enlace a la página oficial de ArgoUML.

http://argouml.tigris.org/

3.2.1.- Visual Paradigm.

Para realizar el ejemplo de desarrollo de diagramas de clases que veremos a continuación se ha determinado usar la herramienta Visual Paradigm for UML por los siguientes motivos:

- ✓ Incluye una versión para uso no comercial, aunque se debe aclarar que viene con funcionalidad limitada, que se distribuye bajo licencia LGPL. Es posible solicitar una licencia de prueba para treinta días que utilizaremos cuando veamos la parte de ingeniería directa e inversa y generación de código.
- ✓ Es multiplataforma.
- ✓ Compatible con UML 2.0.
- ✓ Admite la generación de informes en formatos PDF, HTML y otros.
- ✓ Incluye un módulo para integrarse con NetBeans.
- ✓ Permite realizar actividades de ingeniería inversa y directa. Esto junto con la consideración anterior permite generar código en un proyecto NetBeans directamente a partir del diseño de clases, ahorrándonos trabajo.

En el siguiente enlace encontrarás información sobre dónde encontrar esta herramienta y su proceso de instalación.

Descarga e instalación de Visual Paradigm

Las herramienta CASE para la elaboración de diagramas UML sirven solo para la generación de los diagramas asociados al análisis y diseño de una aplicación.

- ○ Verdadero
- ◉ **Falso**

.

3.3.- Diagramas de clases.

Caso práctico
*En la empresa ya han instalado Visual Paradigm, **Juan** y **María** están empezando a investigar su funcionamiento, y como utilizarlo desde un proyecto de NetBeans.*
—Empecemos por los diagramas estructurales, entre ellos el más importante es el diagrama de clases, fíjate, representa la estructura estática del sistema y las relaciones entre las clases.

Dentro de los diagramas estructurales, y de todos en general, es el más importante porque representa los elementos estáticos del sistema, sus atributos y comportamientos, y como se relacionan entre ellos. Contiene las clases del dominio del problema, y a partir de éste se obtendrán las clases que formarán después el programa informático que dará solución al problema.

En un diagrama de clases podemos encontrar los siguientes elementos:

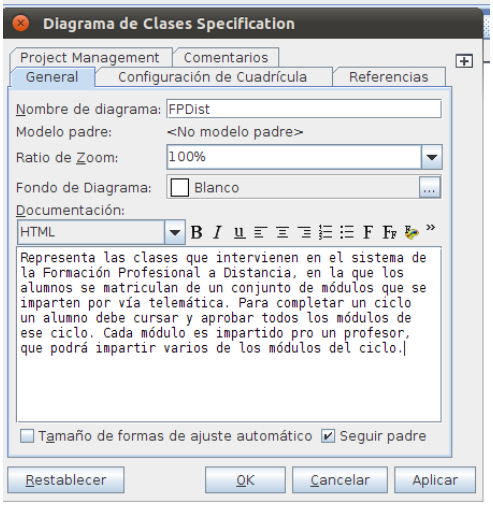
- ✓ **Clases**: recordemos que son abstracciones del dominio del sistema que representan elementos del mismo mediante una serie de características, que llamaremos atributos, y su comportamiento, que serán métodos. Los atributos y métodos tendrán una visibilidad que determinará quien puede acceder al atributo o método. Por ejemplo una clase puede representar a un coche, sus atributos serán la cilindrada, la potencia y la velocidad, y tendrá dos métodos, uno para acelerar, que subirá la velocidad, y otro para frenar que la bajará.

- ✓ **Relaciones**: en el diagrama representan relaciones reales entre los elementos del sistema a los que hacen referencia las clases. Pueden ser de asociación, agregación y herencia. Por ejemplo si tengo una clase persona, puedo establecer una relación conduce entre persona y coche.
- ✓ **Notas**: Se representan como un cuadro donde podemos escribir comentarios que nos ayuden a entender algún concepto que queramos representar.
- ✓ **Elementos de agrupación**: Se utilizan cuando hay que modelar un sistema grande, entonces las clases y sus relaciones se agrupan en paquetes, que a su vez se relacionan entre sí.

Ejercicio resuelto

Crear un diagrama de clases nuevo en Visual Paradigm UML que incluya su nombre y su descripción.

Para crear un diagrama de clases en VP-UML seleccionamos **Archivo >> Nuevo Diagrama** y seleccionamos Diagrama de clases. También podemos acceder al Navegador de diagramas, que se encuentra en el panel de la izquierda y en diagramas de clases hacer clic con el botón secundario y Seleccionar Nuevo diagrama de clases. Cuando generamos un diagrama nuevo tenemos que indicar su nombre y una descripción. Esto es importante para la generación de la documentación posterior.



Cuando creamos un diagrama nuevo aparece en blanco en el panel central de la aplicación. Si es necesario cambiar sus propiedades podemos hacerlo seleccionándolo en el Navegador de diagramas, a través de la opción "Abrir <nombre> Specification" del menú contextual. También podemos abrir el menú contextual, haciendo clic con el botón derecho del ratón sobre el panel central de la aplicación.

3.3.1.- Creación de clases.

Una clase se representa en el diagrama como un rectángulo divido en tres filas, arriba aparece el nombre de la clase, a continuación los atributos con su visibilidad y después los métodos con su visibilidad que está representada por el signo menos "–" para los atributos (privados) y por el signo más "+" para los métodos (públicos).

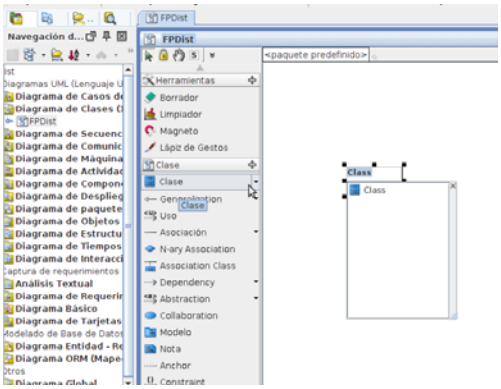
"Una clase es una descripción de un conjunto de objetos que manifiestan los mismos atributos, operaciones, relaciones y la misma semántica."
(Object Modelling and Design [Rumbaught et al., 1991])

"Una clase es un conjunto de objetos que comparten una estructura y un comportamiento comunes."
[Booch G., 1994]

Ejercicio resuelto

Crear una clase nueva en el diagrama de clases del punto anterior.

Cuando generamos un diagrama nuevo aparece un panel con los elementos que podemos añadir al diagrama. Si hacemos clic sobre el icono 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.



Al crear una clase es obligatorio definir nombre, atributos y métodos.

- ☐ Verdadero
- ☒ **Falso**

3.3.2.- Atributos.

Forman la parte estática de la clase. Son un conjunto de variables para las que es preciso definir:

- ✓ Su nombre.
- ✓ Su tipo, puede ser un tipo simple, que coincidirá con el tipo de dato que se seleccione en el lenguaje de programación final a usar, o compuesto, pudiendo incluir otra clase.

Además se pueden indicar otros datos como un valor inicial o su visibilidad. La visibilidad de un atributo se puede definir como:

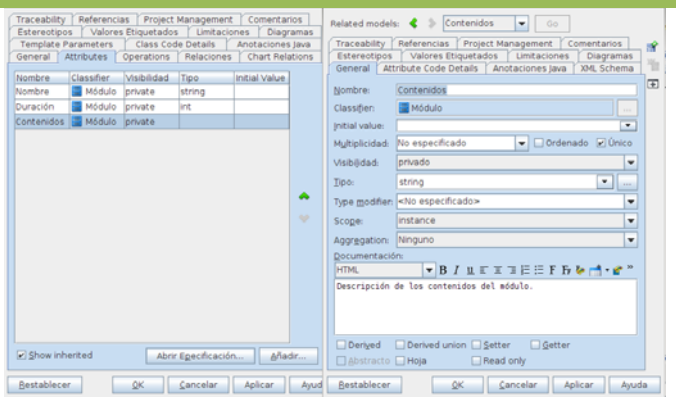
- ✓ **Público**: Se pueden acceder desde cualquier clase y cualquier parte del programa.
- ✓ **Privado**: Sólo se pueden acceder desde operaciones de la clase.
- ✓ **Protegido**: Sólo se pueden acceder desde operaciones de la clase o de clases derivadas en cualquier nivel.
- ✓ **Paquete**: Se puede acceder desde las operaciones de las clases que pertenecen al mismo paquete que la clase que estamos definiendo. Se usa cuando el lenguaje de implementación es Java.

Ejercicio resuelto

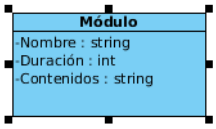
Crear una clase de nombre "Módulo" y que tenga tres atributos:

- ✓ **Nombre, de tipo string.**
- ✓ **Duración de tipo Int.**
- ✓ **Contenidos de tipo string.**

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.

¿Cómo sabemos que los atributos tienen visibilidad privada en el diagrama?

- ○ Porque aparecen acompañados del símbolo más "+"
- ○ Porque aparecen acompañados del símbolo almohadilla "#"
- ○ Porque aparecen acompañados del símbolo "~"
- ◉ **Porque aparece acompañado del símbolo menos "-"**

3.3.3.- Métodos.

Representan la funcionalidad de la clase, es decir, qué puede hacer. Para definir un método hay que indicar como mínimo su nombre, parámetros, el tipo que devuelve y su visibilidad. También se debe incluir una descripción del método que aparecerá en la documentación que se genere del proyecto.

Existen un caso particular de método, el **constructor** de la clase, que tiene como característica que no devuelve ningún valor. El constructor tiene el mismo nombre de la clase y se usa para ejecutar las acciones necesarias cuando se instancia un objeto de la clase. Cunado haya que destruir el objeto se podrá utilizar una función para ejecutar las operaciones necesarias cuando el objeto deje de existir, que dependerán del lenguaje que se utilice.

Ejercicio resuelto

Añadir a la clase creada anteriormente los métodos:

- ✓ matricular(alumno : Alumno) : void
- ✓ asignarDuración(duracion : int) : void

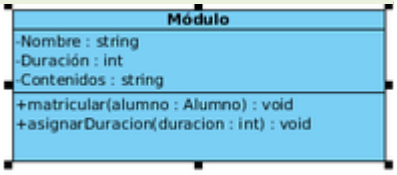
El método más directo para crear un método es en el menú contextual seleccionar "Añadir operación" y escribir la signatura del método:

```
+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 signatura del método indica que es público. Así queda la clase en el diagrama:



¿Cuál es el método que no devuelve ningún tipo de dato?

- El constructor
- Todos los métodos devuelven algo, aunque sea void
- ~<nombre_clase>

3.4.- Relaciones entre clases.

Caso práctico

—Es fácil, ¿lo ves?, por ejemplo, para la aplicación de venta por Internet, tendríamos como clases socio, pedido o artículo, los socios se caracterizan por sus datos personales, los pedidos por su número, fecha, o localidad de destino y los artículos por el código o su descripción.
—Si eso lo veo claro, pero, ¿cómo lo ponemos todo junto? ¿Cómo se conecta el socio con el pedido y el artículo?

Una **relación** es una conexión entre dos clases que incluimos en el diagrama cuando aparece algún tipo de relación entre ellas en el dominio del problema.

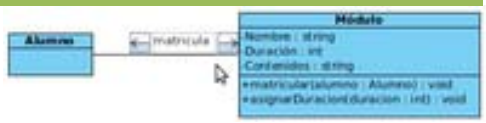
Se representan como una línea continua. Los mensajes "navegan" por las relaciones entre clases, es decir, los mensajes se envían entre objetos de clases relacionadas, normalmente en ambas direcciones, aunque a veces la definición del problema hace necesario que se navegue en una sola dirección, entonces la línea finaliza en punta de flecha.

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

- ✓ De herencia.
- ✓ De composición.
- ✓ De agregación.

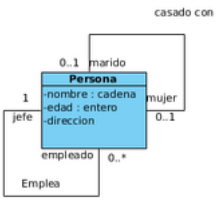
Ejercicio resuelto

Crea una clase nueva llamada Alumno y establece una relación de asociación con el nombre "matrícula" entre ésta y la clase Módulo.



Creamos la clase como hemos visto en puntos anteriores. Para crear una relación utilizamos el elemento asociación de la paleta o bien el icono **Association >> Class** del menú contextual de la clase. Otra forma consiste en hacer clic sobre la clase **Alumno**, seleccionar **Association >> Class** y estirar la línea 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 rellenado en la especificación de la relación los roles y la multiplicidad.



Para obtener las relaciones de un diagrama nos basamos en la descripción de los requisitos del dominio, pero, ¿se pueden crear relaciones en el diagrama que no aparezcan especificadas en la lista de requisitos del problema?

No se puede, las relaciones se deben extraer de la descripción del problema, si no lo hiciéramos así nos estaríamos inventando información.

Sí se puede, a veces se infiere información o se conocen cosas del problema que no aparecen en la descripción de los requisitos.

3.4.1.- Cardinalidad o multiplicidad de la relación

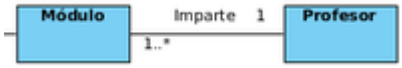
Un concepto muy importante es la **cardinalidad de una relación**, representa cuantos objetos de una clase se van a relacionar con objetos de otra clase. En una relación hay dos cardinalidades, una para cada extremo de la relación y pueden tener los siguientes valores:

Significado de las cardinalidades.	
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, si tengo la siguiente relación:

quiere decir que los alumnos se matriculan en los módulos, en concreto, que un alumno se puede matricular en uno a más módulos y que un módulo puede tener ningún alumno, uno o varios.

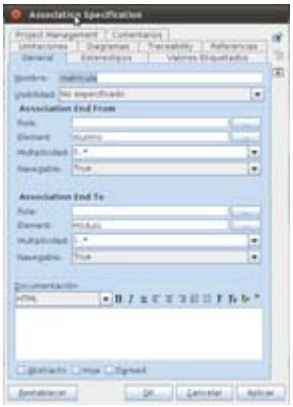
O esta otra:



en la que un profesor puede impartir uno o varios módulos, mientras que un módulo es impartido sólo por un profesor.

Ejercicio resuelto

Establece la cardinalidad de la relación que has creado en el punto anterior para indicar que un alumno debe estar matriculado en al menos un módulo, o varios y que para cada módulo se puede tener ningún alumno, uno o varios.



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.

La **herencia** es una propiedad que permite a los objetos ser construidos a partir de otros objetos, es decir, la capacidad de un objeto para utilizar estructuras de datos y métodos presentes en sus antepasados.

El objetivo principal de la herencia es la *reutilización*, poder utilizar código desarrollado con anterioridad. La herencia supone una clase base y una jerarquía de clases que contiene las clases

derivadas. Las clases derivadas pueden heredar el código y los datos de su clase base, añadiendo su propio código especial y datos, incluso cambiar aquellos elementos de la clase base que necesitan ser diferentes, es por esto que los atributos, métodos y relaciones de una clase se muestran en el nivel más alto de la jerarquía en el que son aplicables.

Tipos:

1. **Herencia simple**: Una clase puede tener sólo un ascendente. Es decir una subclase puede heredar datos y métodos de una única clase base.
2. **Herencia múltiple**: Una clase puede tener más de un ascendente inmediato, adquirir datos y métodos de más de una clase.

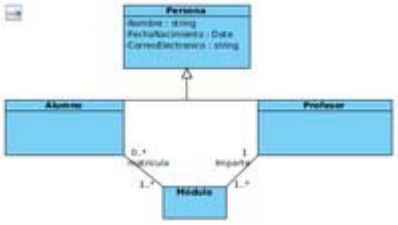
Representación:

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

Ejercicio resuelto

En nuestro diagrama tenemos Alumnos y Profesores. Aún no hemos hablado de su definición y estructura, pero en nuestro sistema tanto un alumno como un profesor tienen unas características comunes como el nombre, la fecha de nacimiento o el correo electrónico por el hecho de ser personas:

Transforma este diagrama para hacer uso de la herencia añadiendo una clase "Persona".



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.

He creado una clase persona cuyos atributo son Nombre, fechaContratación y numeroCuenta. De esta clase derivan por herencia la clase Empleado y JefeDepartamento. ¿Cómo debe declararse un método en la clase Persona que se llame CalculaAntigüedad que se usa sólo para calcular el sueldo de los empleados y jefes de departamento?

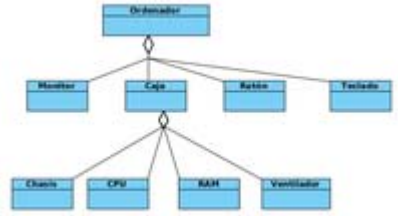
- ☐ Público
- ☐ Privada
- ◉ **Protegida**
- ☐ Paquete

3.4.3.- Agregación y composición.

Muchas veces una determinada entidad existe como un conjunto de otras entidades. En este tipo de relaciones un objeto componente se integra en un objeto compuesto. La orientación a objetos recoge este tipo de relaciones como dos conceptos: la agregación y la composición.

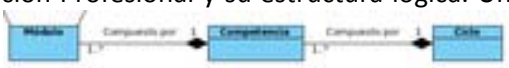
La **agregación** es una asociación binaria que representa una relación todo-parte (pertenece a, tiene un, es parte de). Los elementos parte pueden existir sin el elemento contenedor y no son propiedad suya. Por ejemplo, un centro comercial tiene clientes o un equipo tiene unos miembros. El tiempo de vida de los objetos no tiene porqué coincidir.

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



La **composición** es una agregación fuerte en la que una instancia 'parte' está relacionada, como máximo, con una instancia 'todo' en un momento dado, de forma que cuando un objeto 'todo' es eliminado, también son eliminados sus objetos 'parte'. Por ejemplo: un rectángulo tiene cuatro 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.

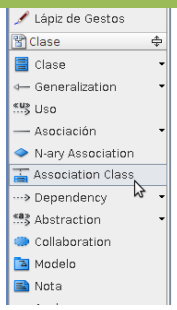
3.4.4.- Atributos de enlace.

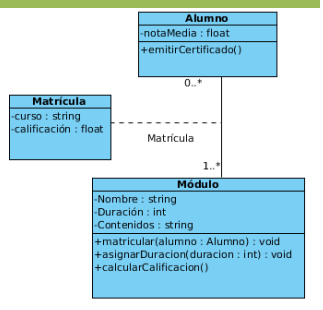
Es posible que tengamos alguna relación en la que sea necesario añadir algún tipo de información que la complete de alguna manera. Cuando esto ocurre podemos añadir atributos a la relación.

Ejercicio resuelto

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:





Siguiendo con el ejemplo anterior, para modelar el cálculo de la nota media de un alumno se añade el método *calcularNotaMedia* a la clase Alumno que realiza la media de las calificaciones de los módulos en los que el alumno se encuentra matriculado para este curso. ¿Qué visibilidad se debería poner a este método?

- (•) **Público**
- ( ) Privado
- ( ) Protegido
- ( ) Paquete

3.5.- Paso de los requisitos de un sistema al diagrama de clases.

Caso práctico
María *y* ***Juan*** *siguen comentando la creación de diagramas de clases.*
—Las reservas se utilizan para relacionar los clientes y las habitaciones, eso es sencillo de ver, pero si tenemos un enunciado un poco más largo, puede no ser tan obvio. Quizá podrías darme algún consejo sobre cómo pasar de los requisitos iniciales de una aplicación a un primer diagrama de clases.
—Es verdad, la cosa se complica un poco cuando tenemos más requisitos, pero la clave está en analizar el texto para obtener nombres y continuar el desarrollo a partir de ahí.

Empezamos identificando objetos que serán las clases del diagrama examinando el planteamiento del problema. Los objetos se determinan subrayando cada nombre o cláusula nominal e introduciéndola en una tabla simple. Los sinónimos deben destacarse. Pero, ¿qué debemos buscar una vez que se han aislado todos los nombres? Buscamos sustantivos que puedan corresponder con las siguientes categorías:

- ✓ **Entidades externas** (por ejemplo: otros sistemas, dispositivos, personas) que producen o consumen información a usar por un sistema computacional.
- ✓ **Cosas** (por ejemplo: informes, presentaciones, cartas, señales) que son parte del dominio de información del problema.
- ✓ **Ocurrencias o sucesos** (por ejemplo: una transferencia de propiedad o la terminación de una serie de movimientos en un robot) que ocurren dentro del contexto de una operación del sistema.
- ✓ **Papeles o roles** (por ejemplo: director, ingeniero, vendedor) desempeñados por personas que interactúan con el sistema.
- ✓ Unidades organizacionales (por ejemplo: división, grupo, equipo) que son relevantes en una aplicación.
- ✓ **Lugares** (por ejemplo: planta de producción o muelle de carga) que establecen el contexto del problema y la función general del sistema.
- ✓ **Estructuras** (por ejemplo: sensores, vehículos de cuatro ruedas o computadoras) que definen una clase de objetos o, en casos extremos, clases relacionadas de objetos.

Cuando estemos realizando este proceso debemos estar pendientes de no incluir en la lista cosas que no sean objetos, como operaciones aplicadas a otro objeto, por ejemplo, "inversión de imagen" producirá un objeto en el ámbito del problema, pero en la implementación dará origen a un método. También es posible detectar dentro de los sustantivos **atributos** de objetos, cosa que también indicaremos en la tabla.

Cuando tengamos la lista completa habrá que estudiar cada objeto potencial para ver si, finalmente, es incluido en el diagrama. Para ayudarnos a decidir podemos utilizar los siguientes criterios:

1. La información del objeto es necesaria para que el sistema funcione.

2. El objeto posee un **conjunto de atributos** que podemos encontrar en cualquier ocurrencia del objeto. Si sólo aparece un atributo normalmente se rechazará y será añadido como atributo de otro objeto.
3. El objeto tiene un **conjunto de operaciones** identificables que pueden cambiar el valor de sus atributos y son comunes a cualquier ocurrencia del objeto.
4. Es una **entidad externa** que consume o produce información esencial para la producción de cualquier solución en el sistema.

El objeto se incluye si cumple todos (o casi todos) los criterios.

Se debe tener en cuenta que la lista no incluye todo, habrá que añadir objetos adicionales para completar el modelo y también, que diferentes descripciones del problema pueden provocar la toma de diferentes decisiones de creación de objetos y atributos.

3.5.1.- Obtención de atributos y operaciones.

Atributos

Definen al objeto en el contexto del sistema, es decir, el mismo objeto en sistemas diferentes tendría diferentes atributos, por lo que debemos buscar en el enunciado o en nuestro propio conocimiento, características que tengan sentido para el objeto en el contexto que se analiza. Deben contestar a la pregunta "*¿Qué elementos (compuestos y/o simples) definen completamente al objeto en el contexto del problema actual?*"

Operaciones

Describen el comportamiento del objeto y modifican sus características de alguna de estas formas:

- ✓ 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

Por último habrá que estudiar de nuevo el enunciado para obtener cómo los objetos que finalmente hemos descrito se relacionan entre sí. Para facilitar el trabajo podemos buscar mensajes que se pasen entre objetos y las relaciones de composición y agregación. Las relaciones de herencia se suelen encontrar al comparar objetos semejantes entre sí, y constatar que tengan atributos y métodos comunes.

Cuando se ha realizado este procedimiento no está todo el trabajo hecho, es necesario revisar el diagrama obtenido y ver si todo cumple con las especificaciones. No obstante siempre se puede refinar el diagrama completando aspectos del ámbito del problema que no aparezcan en la descripción recurriendo a entrevistas con los clientes o a nuestros conocimientos de la materia.

En el siguiente enlace tienes un ejemplo de obtención del diagrama de clases a partir de la descripción de un problema.

Generación del diagrama de clases de un problema dado.

El archivo al proyecto VP-UML resultante:

Enlace al proyecto de Visual Paradigm. (0.17 MB)

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

Caso práctico
—Bueno, ya tenemos el diagrama, es cierto que es bastante útil para aclarar ideas en el equipo y establecer un plan de trabajo inicial. Además es más fácil empezar a programar porque ya tenemos la línea a seguir, ahora solo falta que empecemos a crear clases y a rellenarlas, ¿Verdad?
—Pues sí, pero aún no lo sabes todo, el diagrama de clases aún te va a dar más facilidades...

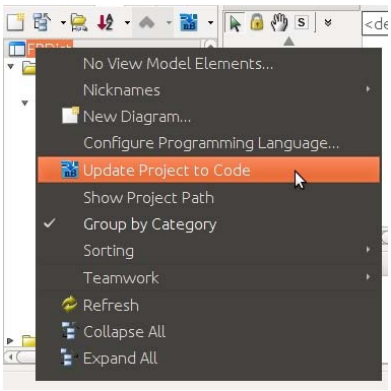
La Generación Automática de Código consiste en la creación utilizando herramientas CASE de código fuente de manera automatizada. El proceso pasa por establecer una correspondencia entre los elementos formales de los diagramas y las estructuras de un lenguaje de programación concreto. El diagrama de clases es un buen punto de partida porque permite una traducción bastante directa de las clases representadas gráficamente, a clases escritas en un lenguaje de programación específico como Java o C++.

Normalmente las herramientas de generación de diagramas UML incluyen la facilidad de la generación, o actualización automática de código fuente, a partir de los diagramas creados.

Ejercicio resuelto

Traduce el diagrama de clases generado en el punto anterior, tanto desde el SDE para NetBeans de Visual Paradigm como desde VP-UML.

Utilizando el SDE integrado de VP-UML en NetBeans:



Antes de hacerlo tendremos que abrir el modelo desde NetBeans, usando el SDE, crear un proyecto nuevo e importar el proyecto VP-UML que hemos creado.

Se puede hacer de dos formas:

- ✓ **Sincronizar con el código**: El código fuente eliminado no se recuperará. Solo se actualizará el código existente.
- ✓ **Forzar sincronizado a código**: Se actualizará todo el código que pueda partir del modelo, incluido el de código eliminado del proyecto NetBeans.

Para generar todas las clases y paquetes de un proyecto VP-UML en NetBeans abrimos el proyecto CE-NB y desplegamos el menú contextual y seleccionamos **Update Project to Code**. También existe la posibilidad de hacerlo directamente desde una clase en particular.

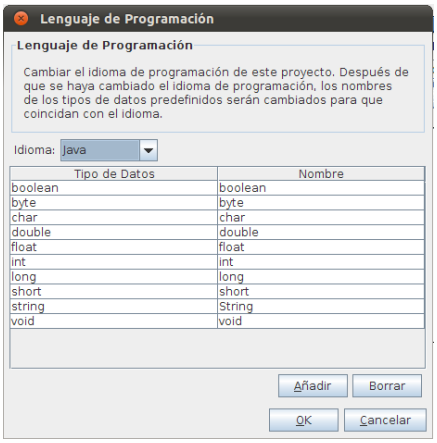
Si se produce algún problema se muestra en la ventana de mensajes, una vez corregido se vuelve a actualizar.

Este procedimiento produce los 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 una ventana en la que podemos configurar el idioma, las clases a generar, y otras características básicas relacionadas con la nomenclatura de atributos y métodos. También permite seleccionar la forma en que se va a implementar la asociación de composición, en nuestro caso hemos elegido la opción por defecto que es a través de un vector.

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. Si existe algún problema en los nombres de clases, atributos o tipos de datos porque no puedan ser utilizados en el lenguaje final o no existan la generación dará un fallo y no se realizará. 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.

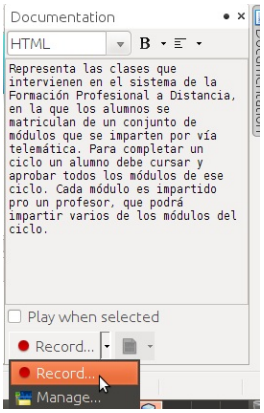
3.7.- Generación de la documentación.

Caso práctico

Los chicos siguen estudiando características de la herramienta VP-UML...

—Sería estupendo que después de generar un diagrama, en el que verdaderamente te has esforzado, pudieras hacer anotaciones sobre la importancia de cada clase, atributo o relación, para que, posteriormente, pudieras compartir esa información o recordarlo rápidamente cuando estuvieras programando el sistema en caso de tener que consultar algo.

—No solo eso, además de generar documentación exhaustiva, la herramienta permite crear informes con ella, pasándola a un formato más cómodo de leer e interpretar en papel por el equipo de desarrollo.

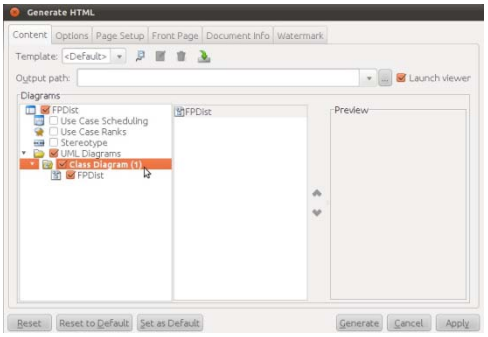


Como en todos los diagramas UML, podemos hacer las anotaciones que consideremos necesarias abriendo la especificación de cualquiera de los elementos, clases o relaciones, o bien del diagrama en sí mismo en la pestaña "Specification".

La ventana del editor cuenta con herramientas para formatear el texto y darle un aspecto bastante profesional, pudiendo añadir elementos como imágenes o hiperenlaces.

También se puede grabar un archivo de voz con la documentación del elemento usando el icono Grabar.

Generar informes



Cuando los modelos están completos podemos generar un informe en varios formatos diferentes (HTML, PDF o Word) con la documentación que hemos escrito. Para generar un informe hacemos:

Desde VP-UML accedemos a Tools >> Reports >> Report writer y seleccionamos el tipo de informe que queremos.

Desde el SDE para NetBeans seleccionamos Modelin >> Reports >> Report writer.

En ambos casos, una vez que elegimos el tipo de informe, obtendremos la siguiente ventana en la que seleccionamos entre otros:

- ✓ Qué diagramas queremos que intervengan y donde se almacenará el informe.
- ✓ La pestaña opciones (Options) permite configurar los elementos que se añadirán al informe, como tablas de contenidos, títulos, etc.
- ✓ Las propiedades de la página.
- ✓ Si se va a añadir una marca de agua.

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

4.- Ingeniería inversa.

Caso práctico

Ada *está muy satisfecha con el cambio que percibe en su equipo, ahora, cuando tiene que enfrentarse a un proyecto nuevo se esfuerzan en escribir los requerimientos antes y comenzar con un proceso de análisis y creación de un diagrama de clases. No obstante, ahora le surge un reto nuevo, una empresa les ha contratado para reconstruir una aplicación que hay que adaptar a nuevas necesidades. Así que los chicos continúan investigando si pueden aplicar el uso de diagramas en este caso también.*

La **ingeniería inversa** se define como el proceso de analizar código, documentación y comportamiento de una aplicación para identificar sus componentes actuales y sus dependencias y 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.

La ingeniería inversa puede ser de varios tipos:

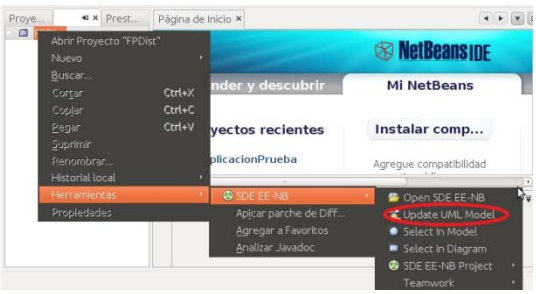
- ✓ **Ingeniería inversa de datos**: Se aplica sobre algún código de bases datos (aplicación, código SQL, etc.) para obtener los modelos relacionales o sobre el modelo relacional para obtener el diagrama entidad-relación.
- ✓ **Ingeniería inversa de lógica o de proceso**: Cuando la ingeniería inversa se aplica sobre el código de un programa para averiguar su lógica (reingeniería), o sobre cualquier documento de diseño para obtener documentos de análisis o de requisitos.
- ✓ **Ingeniería inversa de interfaces de usuario**: Se aplica con objeto de mantener la lógica interna del programa para obtener los modelos y especificaciones que sirvieron de base para la construcción de la misma, con objeto de tomarlas como punto de partida en procesos de ingeniería directa que permitan modificar dicha interfaz.

Ejercicio resuelto

A partir del código que has obtenido en el ejercicio anterior genera el diagrama de clases correspondiente.

Desde el **SDE de VP-UML para NetBeans** tendremos acceso a una herramienta que nos permite la transformación de código Java en diagramas de clases. Para ello:

1. Abrimos el SDE.
2. Seleccionamos el proyecto en el panel de proyectos y abrimos la herramienta SDE-CE NB.
3. Abrir un proyecto nuevo VP-UML en el proyecto de NetBeans.
4. Desde el nodo del proyecto seleccionamos en el menú contextual la opción "**Update UML Model**" lo que iniciará el proceso de ingeniería inversa.



Desde VP-UML

Haremos el proceso de ingeniería inversa desde **Herramienta >> Java Bidireccional >> Código de Inversión**. Al seleccionar esta opción nos preguntará donde se localiza el código fuente. El proceso no genera el diagrama exactamente igual que el original, es capaz de obtener las clases y las relaciones de herencia. El resto de relaciones tendremos que establecerlas nosotros a mano.

Tienes una buena descripción teórica sobre ingeniería inversa en el siguiente documento:

Ingeniería Inversa. (0.53 MB)

Anexo I.- Diagramas UML.

Diagramas estructurales.

- ✓ **Diagramas de clases**: Muestra los elementos del modelo estático abstracto, y está formado por un conjunto de clases y sus relaciones. Tiene una prioridad ALTA.
- ✓ **Diagrama de objetos**: Muestra los elementos del modelo estático en un momento concreto, habitualmente en casos especiales de un diagrama de clases o de comunicaciones, y está formado por un conjunto de objetos y sus relaciones. Tiene una prioridad ALTA.
- ✓ **Diagrama de componentes**: Especifican la organización lógica de la implementación de una aplicación, sistema o empresa, indicando sus componentes, sus interrelaciones, interacciones y sus interfaces públicas y las dependencias entre ellos. Tiene una prioridad MEDIA.
- ✓ **Diagramas de despliegue**: Representan 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 sea de hardware o software, así como las interfaces que las conectan, es decir, muestra como los componentes de un sistema se distribuyen entre los ordenadores que los ejecutan. Se utiliza cuando tenemos sistemas distribuidos. Tiene una prioridad MEDIA.
- ✓ **Diagrama integrado de estructura** (UML 2.0): Muestra la estructura interna de una clasificación (tales como una clase, componente o caso típico), e incluye los puntos de interacción de esta clasificación con otras partes del sistema. Tiene una prioridad BAJA.
- ✓ **Diagrama de paquetes**: Exhibe cómo los elementos del modelo se organizan en paquetes, así como las dependencias entre esos paquetes. Suele ser útil para la gestión de sistemas de mediano o gran tamaño. Tiene una prioridad BAJA.

Diagramas de comportamiento.

- ✓ **Diagramas de casos de uso**: Representan las acciones a realizar en el sistema desde el punto de vista de los usuarios. En él se representan las acciones, los usuarios y las relaciones entre ellos. Sirven para especificar la funcionalidad y el comportamiento de un sistema mediante su interacción con los usuarios y/u otros sistemas. Tiene una prioridad MEDIA.
- ✓ **Diagramas de estado de la máquina**: Describen el comportamiento de un sistema dirigido por eventos. En él aparecen los estados que pueden tener un objeto o interacción, así como las transiciones entre dichos estados. Se lo denomina también diagrama de estado, diagrama de estados y transiciones o diagrama de cambio de estados. Tiene una prioridad MEDIA.
- ✓ **Diagrama de actividades**: Muestran el orden en el que se van realizando tareas dentro de un sistema. En él aparecen los procesos de alto nivel de la organización. Incluye flujo de datos, o un modelo de la lógica compleja dentro del sistema. Tiene una prioridad ALTA.

Diagramas de interacción.

- ✓ **Diagramas de secuencia**: Representan la ordenación temporal en el paso de mensajes. Modela la secuencia lógica, a través del tiempo, de los mensajes entre las instancias. Tiene una prioridad ALTA.
- ✓ **Diagramas de comunicación/colaboración (UML 2.0)**: Resaltan 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. Tiene una prioridad BAJA.

- ✓ **Diagrama de interacción**: Muestra un conjunto de objetos y sus relaciones junto con los mensajes que se envían entre ellos. Es una variante del diagrama de actividad que permite mostrar el flujo de control dentro de un sistema o proceso organizativo. Cada nodo de actividad dentro del diagrama puede representar otro diagrama de interacción. Tiene una prioridad BAJA.
- ✓ **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. Tiene una prioridad BAJA.

Anexo II.- Descarga e instalación de Visual Paradigm.

Descarga e instalación de Visual Paradigm

Obtenemos los archivos desde la página de Visual Paradigm:

http://www.visual-paradigm.com/download/vpuml.jsp?edition=ce

Ofrece dos versiones:

- ✓ **Visual Paradigm for UML (VP-UML)**, versión de prueba de 10 días, ampliable a 30 días mediante registro.
- ✓ **Versión Community-Edition**, para uso no comercial (gratuito).

En cualquier caso necesitamos un código de activación que conseguiremos registrándonos. Se envía al correo electrónico que se indique en el registro.

La versión **Community-Edition** incluye algunas de las funcionalidades de la versión completa, entre las que no se encuentra la generación de código ni la ingeniería inversa, que se verán al final de la unidad por lo que se recomienda empezar por la versión completa de prueba por 30 días, para los que se necesita un código de tipo, conseguiremos el código de activación, que es un archivo de tipo `zvpl`, en este caso llamado `vpsuite.zvpl`.

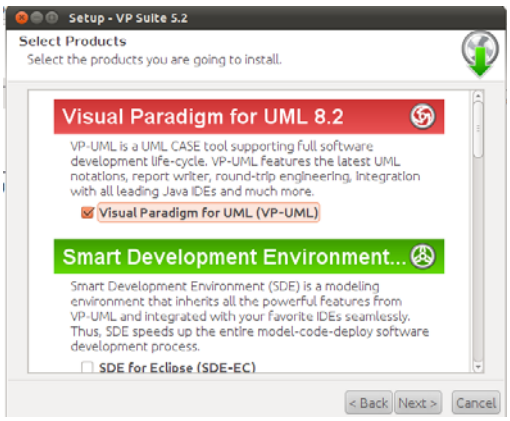
Para la siguiente unidad también usaremos VP-UML, de modo que si fuera necesario tendríamos que conseguir un nuevo código de activación, esta vez de tipo Community-Edition.

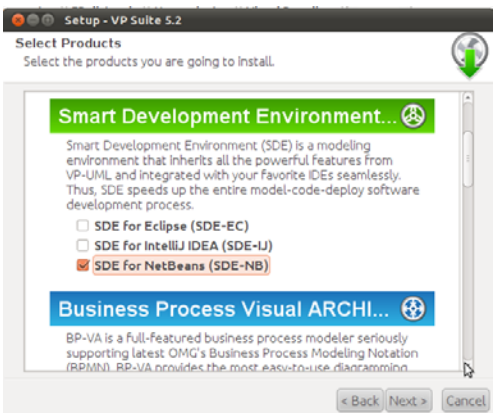
Proceso de instalación

Ejecutaremos el archivo de instalación, que tendrá diferente extensión si es para Windows o para Linux. En nuestro caso suponemos que lo hacemos en un equipo con Ubuntu Desktop 10.10. Se debe tener en cuenta que en el nombre se incluye la versión y la fecha en la que apareció, por lo que estos datos pueden cambiar con el tiempo. Si hacemos la instalación en Windows bastará con hacer doble clic sobre el archivo `.exe`.

```
usuario@equipo:~/VP/ chmod +x VP_Suite_Linux_5_2_20110611.sh
usuario@equipo:~/VP/sudo  ./VP_Suite_Linux_5_2_20110611.sh
```

Durante la instalación tendremos que indicar qué módulos queremos instalar, seleccionaremos Visual Paradigm for UML y el SDE (Smart Development Environment o Entorno de Desarrollo Inteligente), de NetBeans que es el que vamos a usar.





A continuación tendremos que indicar que vamos a utilizar la versión **Enterprise** de ambas herramientas y en que directorio está NetBeans:

Es importante destacar que la instalación debe hacerse sobre una instalación limpia de NetBeans, es decir, que solo podremos instalarlo en el directorio que indicamos una vez.

A continuación se pide un archivo con la **licencia** de la herramienta. Al iniciar la descarga nos pedirá que nos registremos, tras hacerlo podremos solicitar este archivo. Lo insertamos ahora, como hemos instalado dos herramientas nos pedirá dos archivos, pero podemos usar la opción de archivo de licencia combinado, de modo que nos sirva para los dos casos. Si nos lo pide, tendremos que volver a añadirlo después al iniciar Visual Paradigm, con una copia nueva del archivo de clave.

Por último indicamos dónde queremos que ponga los archivos con los proyectos y finalizamos la instalación indicando que no queremos que abra ninguna aplicación.

Iniciar Visual Paradigm

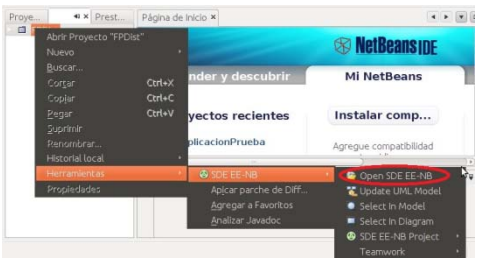
Una vez realizada la instalación tendremos una entrada en el menú **Aplicaciones** llamada **Otras**, si trabajamos con Linux o bien una entrada de menú en el botón Inicio para Visual Paradigm, si es que trabajamos en Windows. En cualquiera de los casos para abrir la herramienta buscamos la opción Visual Paradigm for UML, que se abrevia como **VP-UML**. Al hacer clic se abrirá el programa, y nos preguntará cual es el directorio por defecto para guardar los proyectos, podemos dejar la opción por defecto o seleccionar nuestro propio directorio.

Iniciar VP-UML desde NetBeans

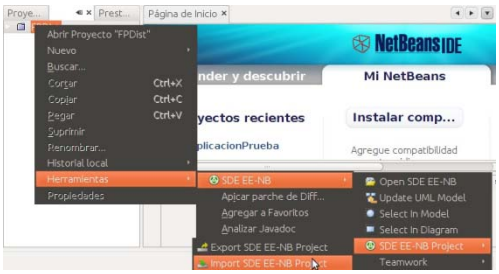
Al hacer la instalación hemos indicado, marcado, que se instale también el SDE para NetBeans, por lo que también tenemos la opción de iniciar la herramienta para usarla integrada con NetBeans. Para abrirlo buscamos dentro del menú de Visual Paradigm la opción **SDE for NetBeans**.

Esto abre la aplicación NetBeans, a la que se ha incorporado una pequeña diferencia, y es que podemos añadir a un proyecto en desarrollo existente un proyecto VP-UML. ¿Cómo lo hacemos?

Estando en la ventana de **Proyectos**, si hacemos clic con el botón secundario sobre un proyecto vemos una serie de opciones, como compilar o construir, ahora, además, abrir el SDE desde **Open SDE EE-NB**, que abre el SDE. La primera vez nos pedirá que importemos un archivo de clave, que podremos obtener con el botón Request Key desde la página oficial. Para ello necesitaremos el correo de registro que hemos utilizado al hacer la instalación.

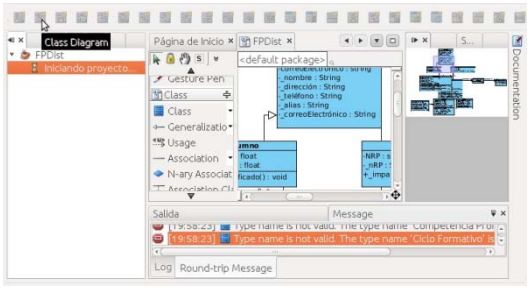


Los proyectos de Visual Paradigm se podrán almacenar en el directorio por defecto, que se denomina `vpproject` y cuelga del directorio principal del proyecto NetBeans, o en otra ubicación. Nosotros nos quedaremos con la opción por defecto.



También podemos importar un proyecto VP-UML que tengamos ya creado seleccionándolo al crear el proyecto existente.

Una vez creado o importado el proyecto, tendremos una serie de botones en la zona superior derecha que nos permitirán crear los diferentes diagramas de UML, y que queden asociados al proyecto NetBeans.



Anexo III.- Generación del diagrama de clases de un problema dado.

Descripción del problema

El Ministerio de Educación ha encargado a **BK Programación** que desarrolle una plataforma de aprendizaje electrónico para que los alumnos de ciclos formativos a distancia tengan acceso a los materiales y puedan comunicarse con sus profesores. Para que los chicos puedan empezar a crear los primeros diagramas de la aplicación Ada les pasa la siguiente descripción del ámbito del problema:

"Los alumnos y alumnas de Ciclos Formativos a Distancia se matriculan de varios módulos formativos al año. Los módulos formativos son impartidos por profesores y profesoras que pondrán los contenidos del módulo a disposición de los alumnos y alumnas. Para superar un módulo hay que hacer una tarea y un examen que se calificarán de uno a diez, y sacar en ambos casos una puntuación superior a cinco. Los exámenes se componen de 30 preguntas que se eligen y ordenan al azar. Las preguntas tienen un enunciado y cuatro posibles respuestas, sólo una de ellas válida. Un ciclo formativo se compone de una serie de competencias profesionales, que tienen una descripción y que, a su vez, están formadas por uno o varios módulos, que tienen un nombre, y un número de horas. Cuando un alumno o alumna supera los módulos correspondientes a una capacidad se le certifica esa capacidad. Cuando se han superado todos los módulos (y por tanto se han adquirido todas las competencias profesionales) se aprueba el ciclo. Cuando un alumno o alumna finaliza el ciclo se emite un certificado de competencias a su nombre donde aparece la descripción de las competencias que forman el ciclo y la nota media obtenida. Si un alumno o alumna no termina de cursar el ciclo completo puede pedir un certificado que acredite las competencias que sí tenga adquiridas. El alumnado y el profesorado se identifican con un alias en el sistema y se comunican a través de correo electrónico. Por motivos administrativos es necesario conocer el nombre y apellidos, dirección completa y teléfono de todas las personas que participan en el sistema, sea como profesores o como alumnos. Para el profesorado, además, se debe conocer su número de registro personal (NRP)"

Extracción de los sustantivos de la descripción del problema.

Primero subrayamos los sustantivos de la descripción del problema (sin repeticiones) y los pasamos a una tabla: (usaremos sólo alumnos y sólo profesores para simplificar el diseño)

Los alumnos de Ciclos Formativos a Distancia se matriculan de varios módulos formativos al año. Los módulos formativos son impartidos por profesores que pondrán los contenidos del módulo a disposición de los alumnos. Para superar un módulo hay que hacer una tarea y un examen que se calificarán de uno a diez, y sacar en ambos casos una puntuación superior a cinco. Los exámenes se componen de 30 preguntas que se eligen y ordenan al azar. Las preguntas tienen un enunciado y cuatro posibles respuestas, sólo una de ellas válida. Un ciclo formativo se compone de una serie de competencias profesionales, que tienen una descripción y que, a su vez, están formadas por uno o varios módulos, que tienen un nombre, y un número de horas. Al sumar las horas de un ciclo obtenemos las horas del módulo. Cuando un alumno supera los módulos correspondientes a una competencia se le certifica esa competencia. Cuando se han superado todos los módulos (y por tanto se han adquirido todas las competencias profesionales) se aprueba el ciclo. Cuando un alumno finaliza el ciclo se emite un certificado de competencias a su nombre donde aparece la descripción de las competencias que forman el ciclo y la nota media obtenida. Si un alumno no termina de cursar el ciclo completo puede pedir un certificado que acredite las competencias que si tenga adquiridas. Los alumnos y profesores se identifican con un alias en el sistema y se comunican a través de correo electrónico. Por motivos administrativos es necesario conocer el nombre y apellidos, dirección completa y teléfono de todas las personas que participan en el sistema, sea como profesores o como alumnos. Para los profesores, además, se debe conocer su número de registro personal (NRP).

Tabla de sustantivos	
Clase/objeto potencial	**Categoría**
Alumno	Entidad externa o rol
Ciclo Formativo a Distancia	Unidad organizacional
Modulo Formativo	Unidad organizacional
Año	Atributo
Profesor	Entidad externa o rol
Contenidos	Atributo
Tarea	Cosa
Examen	Cosa
Uno	Atributo
Diez	Atributo
Pregunta	Cosa
Enunciado	Atributo
Respuesta	Atributo
Competencia Profesional	Unidad organizacional
Descripción	Atributo
Horas	Atributo
Certificado de competencias	Cosa
Nombre	Atributo
Nota media	Atributo
Alias	Atributo
Sistema	Estructura
Nombre	Atributo
Dirección	Atributo
Teléfono	Atributo
Persona	Rol o entidad externa
Número de registro personal	Atributo

Selección de sustantivos como objetos/clases del sistema

Ahora aplicamos los criterios de selección de objetos. En este apartado es necesario destacar que aunque algunos de los sustantivos que tenemos en el anunciado podrían llegar a convertirse en clases y objetos, como los contenidos de un módulo formativo, se descartan en esta fase porque el enunciado no da suficiente información. El proceso de creación de diagramas no es inmediato, sino que está sujeto a revisiones, cambios y adaptaciones hasta tener un resultado final completo.

Tabla de elección de sustantivos como objetos o clases del sistema.	
Clase/objeto potencial	**Criterios aplicables**
Alumno	2,3,4
Ciclo Formativo a Distancia	1,2,3
Módulo Formativo	1,2,3
Profesor	2,3,4
Tarea	1,2,3
Examen	1,2,3
Competencia Profesional	1,2,3
Pregunta	1,2,3
Certificado de competencias	Falla 2,3 rechazado
Sistema	Falla 1,2,3,4 rechazado
Persona	2,3,4

Obtención de los atributos de los objetos.

Buscamos responder a la pregunta ¿Qué elementos (compuestos y/o simples) definen completamente al objeto en el contexto del problema actual?

Tabla de relación de las clases u objetos con sus atributos.	
Clase/objeto potencial	**Atributos**
Alumno	Nombre, dirección, teléfono, alias, correo electrónico.
Ciclo Formativo a Distancia	Nombre, descripción, horas.
Módulo Formativo	Modulo Formativo
Profesor	Nombre, dirección, teléfono, alias, correo electrónico, NRP.
Tarea	Descripción, calificación.
Examen	Descripción, calificación.
Competencia Profesional	Nombre, descripción.
Pregunta	Enunciado, respuestas, respuesta válida.
Persona	Nombre, dirección, teléfono, alias, correo electrónico.

Obtención de los métodos

Buscamos o inferimos en el enunciado verbos, y actividades en general que describan el comportamiento de los objetos o modifiquen su estado.

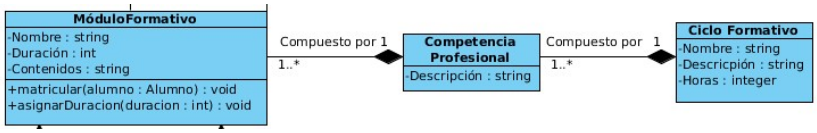
Tabla de clases u objetos del sistema con sus posibles métodos.	
Clase/objeto potencial	**Métodos**
Alumno	CalcularNotaMedia() : void emitirCertificado() : void
Ciclo Formativo a Distancia	

Módulo Formativo	Matricular(Alumno : alumno) : void asignarDuracion(horas: int) : void
Profesor	
Tarea	
Examen	Calificar() añadirPregunta() ordenarPreguntas() crearExamen()
Competencia Profesional	
Pregunta	
Persona	

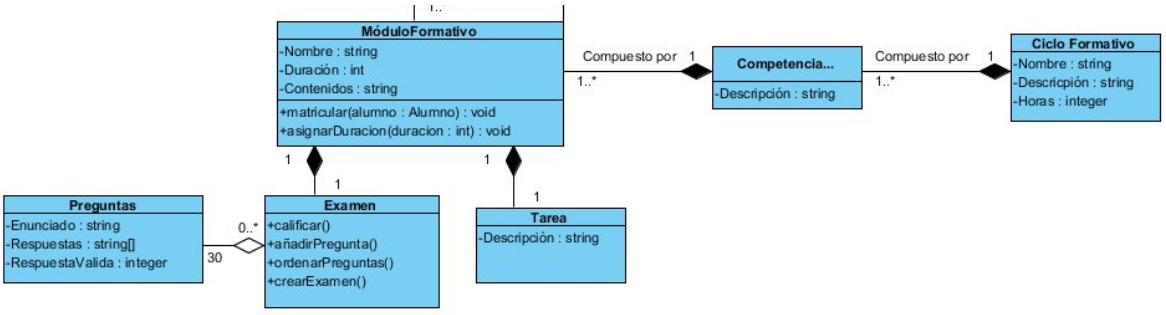
Obtener relaciones

Con las clases ya extraídas y parcialmente definidas (aún faltan por añadir métodos y atributos inferidos de posteriores refinamientos y de nuestro conocimiento) podemos empezar a construir relaciones entre ellas.

Comenzaremos por las clases que hacen referencia a la estructura de los Ciclos, cada Ciclo se compone de una o más competencias profesionales, que no tienen la capacidad de existir por sí mismas, es decir, la competencia no tiene sentido sin su ciclo, por lo que vamos a crear una relación entre ambas clases de composición. De igual manera una competencia profesional se compone de un conjunto de módulos formativos (1 o más) por lo que relacionaremos ambas, también mediante composición.

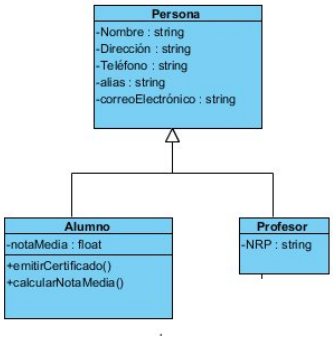


Un módulo formativo a su vez, contiene un examen y una tarea, que tampoco tienen sentido por sí mismos, de modo que también los vamos a relacionarlos mediante composición. El examen por su parte se compone de 30 preguntas, pero éstas pueden tener sentido por sí mismas, y pertenecer a diferentes exámenes, además, el hecho de eliminar un examen no va a dar lugar a que las preguntas que lo forman se borren necesariamente, si leemos con atención el enunciado, podemos deducir que las preguntas se seleccionan de un repositorio del que pueden seguir formando parte [... [Los exámenes se componen de 30 preguntas que se eligen y ordenan al azar...], así que en este caso usaremos la relación de agregación para unirlos.



Por otra parte alumnos y profesores comparten ciertas características, por necesidad del sistema, como son los datos personales, o el correo electrónico, esto induce a pensar que podemos crear una abstracción con los datos comunes, que de hecho, ya hemos obtenido del enunciado en la clase

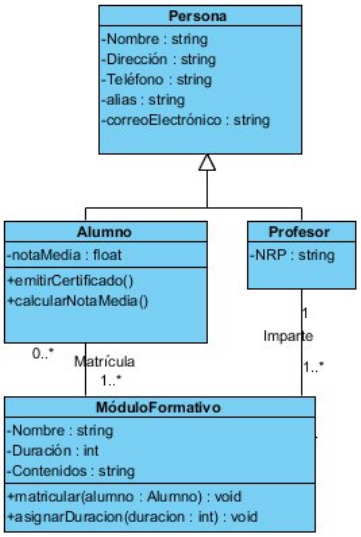
persona, que recoge las coincidencias entre alumnos y profesores y añadir una relación de herencia de la siguiente manera:



Por último queda relacionar a alumnos y profesores con los módulos formativos. Un alumno se matricula de un conjunto de módulos formativos, y un profesor puede impartir uno o varios módulos formativos.

Más concretamente, de cara a la cardinalidad, un alumno puede estar matriculado en uno o varios módulos, mientras que un módulo puede tener, uno o varios alumnos matriculados. Por su parte un profesor puede impartir uno o varios módulos, aunque un módulo es impartido por un profesor.

Éste análisis da como resultado lo siguiente:



Añadir Getters, Setters y constructores

Por último añadimos los métodos que permiten crear los objetos de las clases (constructores) así como los que permiten establecer los valores de los atributos no calculados y leerlos (getters y setters), recuerda que para tener éstos métodos completos es necesario que el atributo tenga establecido su tipo, para que sea tenido en cuenta.

Para añadir los getters y setters en Visual Paradigm, basta con desplegar el menú contextual del atributo y seleccionar la opción Create Getter and Setter.

También hay que añadir los métodos que no se infieren de la lectura del enunciado, por ejemplo los que permiten añadir módulos a las competencias, o competencias a los ciclos. Para comprobar estos métodos puedes descargar el diagrama de clases en un proyecto VP-UML un poco más adelante.

Añadir documentación

Por último se debe rellenar la documentación de cada clase, atributo y método con una descripción de los mismos que será necesaria para la generación de informes posterior. A continuación se listan las clases con su documentación:

- ✓ **Persona**: Generalización para agrupar las características comunes de alumnos y profesores como personas que interactúan con el sistema. De una persona interesa conocer su nombre, dirección, teléfono, alias y correo electrónico.
- ✓ **Alumno**: Es un tipo de persona. Representa a las personas que se matriculan de un ciclo formativo. Un alumno puede estar matriculado durante varios años en un ciclo. Está matriculado de un ciclo siempre que está matriculado en algún módulo que forme parte del ciclo. Para aprobar un Ciclo hay que superar todos los módulos que lo componen. Para superar un módulo hay que realizar la tarea y aprobar el examen, que está compuesto de 30 preguntas de tipo test, con cuatro respuestas posibles una de las cuales es la correcta. De un alumno interesa almacenar su nota media.
- ✓ **Profesor**: Es un tipo de persona. Representa a las personas que imparten los módulos formativos. Evalúan las tareas que realizan los alumnos y se encargan de poner los contenidos a disposición de los alumnos. De un profesor interesa almacenar su número de registro personal.
- ✓ **Ciclo Formativo a Distancia**: Es uno de los núcleos centrales del sistema. Representa los estudios que se pueden realizar a distancia. Un ciclo formativo se compone de un conjunto de competencias profesionales que se componen a su vez de módulos formativos. Se aprueba un ciclo formativo cuando se adquieren todas las competencias que lo forman. De un ciclo formativo se almacena su nombre, descripción y horas totales.
- ✓ **Competencia Profesional**: Representan el conjunto de conocimientos generales que se adquieren cuando se completa un ciclo formativo. Se componen de módulos profesionales y se adquiere una competencia cuando se superan los módulos que la componen. De una competencia se almacena su descripción.
- ✓ **Módulo Formativo**: Unidades formativas que cursa un alumno. Un módulo formativo tiene una serie de contenidos que el alumno debe estudiar, y una tarea y un examen que el alumno debe hacer. Cuando se aprueban la tarea y el examen se supera el módulo. De un módulo se almacena su nombre, duración y contenidos.
- ✓ **Tarea**: Actividad relacionada con los contenidos de un módulo que debe realizar un alumno. Tiene una puntuación de uno a diez y es evaluada por el profesor. La nota se asigna a cada alumno para la matrícula del módulo al que pertenece la tarea. De una tarea se almacena su descripción.
- ✓ **Examen**: Conjunto de treinta preguntas que se evalúa de uno a diez. Un examen puede desordenarse y calificarse calculando cuantas preguntas son correctas.
- ✓ **Pregunta**: Forman los exámenes de un módulo. Se compone del enunciado y cuatro posibles respuestas de las cuales sólo una es válida.

TEMA 6

INDICE

[ENTORNOS DE DESARROLLO]

José Luis Comesaña Cabeza --- 2011/2012

Entornos de desarrollo del curso "*Desarrollo de Aplicaciones Web*"

DISEÑO ORIENTADO A OBJETOS. ELABORACIÓN DE DIAGRAMAS DE COMPORTAMIENTO.

CASO PRÁCTICO

En BK Programación continúan inmersos en el mundo de UML. A pesar de que han trabajado duro y han aprendido bastante acerca de este lenguaje de especificación, Ada se ha dado cuenta de que apenas han empezado a arañar la superficie de todas las posibilidades que les ofrece. De momento ya saben como crear un diagrama de clases bastante completo y como analizar un problema propuesto, sin embargo hay muchos aspectos del problema que no pueden modelar todavía, por ejemplo con solo el diagrama de clases no pueden saber qué se espera del sistema que van a construir, o en qué se deben basar para codificar los métodos, o simplemente, ¿Cómo colaboran los objetos de las clases que han creado para hacer alguna tarea que sea útil?

Ada decide que no pueden parar ahora, y que hay que hacer un esfuerzo final para que los conocimientos del equipo sean globales y puedan enfrentarse a cualquier desarrollo software con solvencia.

Al momento, Ada pone a su equipo manos a la obra.

1. Diagramas de comportamiento.

Caso práctico
—Compañeros, creo que ahora no debemos conformarnos con modelar diagramas de clase y nada más, esto no nos da posibilidades de incluir ninguna información acerca del comportamiento de nuestro sistema. ¿Cómo especificamos la funcionalidad? O ¿qué acciones se realizan?, o ¿las restricciones? Necesitamos seguir estudiando diagramas que nos ayuden a especificar la dinámica del sistema.
¿Empezamos ahora mismo?

En el tema anterior vimos como crear un diagrama de clases para un problema determinado, esto nos ayuda a ver el problema con otra perspectiva y descubrir información nueva, sin embargo no tiene en cuenta elementos como la creación y destrucción de objetos, el paso de mensajes entre ellos y el orden en que deben hacerse, qué funcionalidad espera un usuario poder realizar, o como influyen elementos externos en nuestro sistema. Un diagrama de clases nos da información estática pero no dice nada acerca del comportamiento dinámico de los objetos que lo forman, para incluir éste tipo de información utilizamos los diagramas de comportamiento que incluyen:

- ✓ Diagramas de casos de uso.
- ✓ Diagramas de actividad.
- ✓ Diagramas de máquinas de estado.
- ✓ Diagramas de interacción.
  - ➔ Diagramas de secuencia.
  - ➔ Diagramas de comunicación.
  - ➔ Diagramas de interacción.
  - ➔ Diagramas de tiempo.

En el siguiente enlace tienes una descripción y algunos ejemplos de todos los diagramas UML, puedes usarlo como complemento a lo que vamos a ver en la unidad.
http://jms32.eresmas.net/tacticos/UML/UMLIndex.html

2. Diagramas de casos de uso.

CASO PRÁCTICO.
—Empezaremos por el principio. Cuando estamos diseñando software es esencial saber cuales son los requerimientos del sistema que queremos construir, y necesitamos alguna herramienta que nos ayude a especificarlos de una manera clara, sistemática, y que nuestros clientes puedan entender fácilmente, ya que es imprescindible que nos pongamos de acuerdo en lo que realmente queremos hacer. ¿Alguna idea?
—¿No bastaría con hacer una lista de requerimientos y describirlos exhaustivamente?
—No, una descripción textual puede inducir a errores de interpretación y suele dejar cabos sueltos, y no contempla otra información, como quien realiza las acciones que describimos, por ejemplo. Necesitamos algo más específico.
Lo que Ada necesita son los diagramas de Casos de uso.

Los diagramas de casos de uso son un elemento fundamental del análisis de un sistema desde la perspectiva de la orientación a objetos porque resuelven uno de los principales problemas en los que se ve envuelto el proceso de producción de software: la falta de comunicación entre el equipo de desarrollo y el equipo que necesita de una solución software. Un diagrama de casos de uso nos ayuda a determinar **QUÉ** puede hacer cada tipo diferente de usuario con el sistema, en una forma que los no versados en el mundo de la informática o, más concretamente el desarrollo de software, pueda entender.

Los diagramas de casos de uso documentan el comportamiento de un sistema desde el punto de vista del usuario. Por lo tanto los casos de uso determinan los **requisitos funcionales del sistema** (*acciones fundamentales que debe realizar el software al recibir información, procesarla y producir resultados. Suelen venir definidos por el cliente*), es decir, representan las funciones que un sistema puede ejecutar.

Un diagrama de casos de uso es una visualización gráfica de los requisitos funcionales del sistema, que está formado por casos de uso (se representan como elipses) y los actores que interactúan con ellos (se representan como monigotes). Su principal **función** es dirigir el proceso de creación del software, definiendo qué se espera de él, y su **ventaja** principal es la facilidad para interpretarlos, lo que hace que sean especialmente útiles en la comunicación con el cliente.

Los diagramas de casos de uso se crean en las primera etapa de desarrollo del software, y se enmarcan en el proceso de análisis, para definir de forma detallada la funcionalidad que se espera cumpla el software, y que, además, se pueda comunicar fácilmente al usuario, pero, ¿termina aquí su función?
En absoluto, de los diagramas de casos de uso se desprenden otros (normalmente se realiza antes que el diagrama de clases) que describen tanto la estructura del sistema como su comportamiento, lo que influye directamente en la implementación (*Paso a código de las especificaciones que se han definido durante la fase de análisis y diseño de un sistema software*) del sistema y en su arquitectura (*Conjunto de decisiones significativas acerca de la organización de un sistema software, la selección de los elementos estructurales a partir de los cuales se compone el sistema, y las interfaces entre ellos, junto con su comportamiento, tal y como se especifica en las colaboraciones entre esos elementos, la composición de estos elementos estructurales y de comportamiento en subsistemas progresivamente mayores y el estilo arquitectónico que guía esta organización: estos elementos y sus interfaces, sus colaboraciones y su composición*) final. Por otra parte al describir específicamente qué se espera del software también se usa en la fase de prueba, para verificar que el sistema cumple con los requisitos funcionales, creándose muchos de los casos de prueba (pruebas de caja negra *(Se realizan cuando una aplicación es probada usando su interfaz externa sin preocuparnos de la implementación de la misma)*) directamente a partir de los casos de uso.

2.1 Actores

Caso práctico

—Como decía, uno de los principales problemas de una descripción textual es que no permite especificar adecuadamente qué personas o entidades externas interactúan con el sistema. Ahora tenemos una herramienta que tiene esto muy en cuenta...

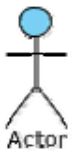
Los actores representan un tipo de usuario del sistema. Se entiende como usuario cualquier cosa externa que interactúa con el sistema. No tiene por qué ser un ser humano, puede ser otro sistema informático o unidades organizativas o empresas.

Siempre hay que intentar independizar los actores de la forma en que se interactúa con el sistema. Por ejemplo, un usuario del sistema puede interpretar diferentes roles según la operación que esté ejecutando, cada uno de estos roles representará un actor diferente, es decir, un actor en un diagrama de casos de uso representa un rol que alguien puede estar jugando, no un individuo particular por lo tanto puede haber personas particulares que puedan estar usando el sistema de formas diferentes en diferentes ocasiones. Suele ser útil mantener una lista de los usuarios reales para cada actor.

Tipos de actores:

- ***Primarios***: interaccionan con el sistema para explotar su funcionalidad. Trabajan directa y frecuentemente con el software.
- ***Secundarios***: soporte del sistema para que los primarios puedan trabajar. Son precisos para alcanzar algún objetivo.
- ***Iniciadores***: no interactúan con el sistema pero desencadenan el trabajo de otro actor.

Los actores se representan mediante la siguiente figura:



Es posible que haya casos de uso que no sean iniciados por ningún usuario, o algún otro elemento software, en ese caso se puede crear un actor “Tiempo” o “Sistema”.

¿Un sistema software externo, como una entidad para la autentificación de claves, podría considerarse como un actor en un diagrama de casos de uso?

- [x] Verdadero
- [ ] Falso

un actor no tiene porqué ser una persona física, es cualquier entidad que interaccione con el sistema

2-2 Casos de uso

Caso práctico

—Vale, pero lo que verdaderamente queremos es identificar la funcionalidad del sistema ¿no?, ¿cómo hace esta herramienta la descripción de la funcionalidad?

Se utilizan casos de uso para especificar tareas que deben poder llevarse a cabo con el apoyo del sistema que se está desarrollando.

Un **caso de uso** especifica 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. El conjunto de casos de uso forma el "**comportamiento requerido**" de un sistema.

El objetivo principal de elaborar un diagrama de casos de uso no es crear el diagrama en sí, sino la descripción que de cada caso se debe realizar, ya que esto es lo que ayuda al equipo de desarrollo a crear el sistema a posteriori. Para hacer esto utilizamos, sobre todo otros diagramas que permiten describir la dinámica del caso de uso, como el diagrama de secuencia que veremos después, y una descripción textual, en la que se deben incluir, al menos, los siguientes datos (a los que se denomina **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.

La representación gráfica de un caso de uso se realiza mediante un óvalo o elipse, y su descripción se suele hacer rellenando una o más tablas como la de la imagen (obtenida de la herramienta Visual Paradigm).

Super Use Case			
Author	usuario		
Date	26-ago-2011 13:56:56		
Brief Description			
Preconditions			
Post-conditions			
Flow of Events		Actor Input	System Response
	1		

"Tras comprobar todos los artículos el pedido queda en el almacén a la espera de ser recogido." ¿Dónde incluirías esta afirmación sobre un caso de uso en un contrato?

- ○ En el flujo de eventos normal. -----
- ○ En el flujo de eventos alternativo.
- ○ En las precondiciones.
- ◉ **En las postcondiciones.**

2.3 Relaciones

Caso práctico

—De acuerdo, y ahora ¿Cómo asociamos a los actores con los casos de uso que pueden realizar?

Los diagramas de casos de uso son grafos no conexos en los que los nodos son actores y casos de uso, y las aristas son las relaciones que existen entre ellos. Representan qué actores realizan las tareas descritas en los casos de uso, en concreto qué actores inician un caso de uso. Pero además existen otros tipos de relaciones que se utilizan para especificar relaciones más complejas, como uso o herencia entre casos de uso o actores.

Existen diferentes tipos de relaciones entre elementos:

- ✓ **Asociación**: representa la relación entre el actor que lo inicia y el caso de uso.

- ✓ **Inclusión**: se utiliza cuando queremos dividir una tarea de mayor envergadura en otras más sencillas, que son utilizadas por la primera. Representa una relación de uso, y son muy útiles cuando es necesario reutilizar tareas.
- ✓ **Extensión**: se utiliza para representar relaciones entre un caso de uso que requiere la ejecución de otro en determinadas circunstancias.
- ✓ **Generalización**: se utiliza para representar relaciones de herencia entre casos de uso o actores.

A continuación las vemos con un poco más de detalle.

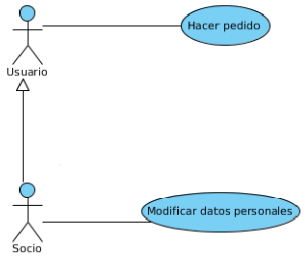
2.3.1 Interacción o asociación.

Hay una asociación entre un actor y un caso de uso si el actor interactúa con el sistema para llevar a cabo el caso de uso o para iniciarlo.

Una asociación se representa mediante un linea continua que une un actor con un caso de uso. Por ejemplo, un usuario de un sistema de venta por Internet puede hacer un pedido, lo que se representa del siguiente modo:

2.3.2 Generalización

Es posible que, igual que con los diagramas de clases, existan casos de uso que tengan comportamientos semejantes a otros que los modifican o completan de alguna manera. El caso base se define de forma abstracta y los hijos heredan sus características añadiendo sus propios pasos o modificando alguno. Normalmente la herencia se utiliza menos en diagramas de casos de uso que en diagramas de clases.



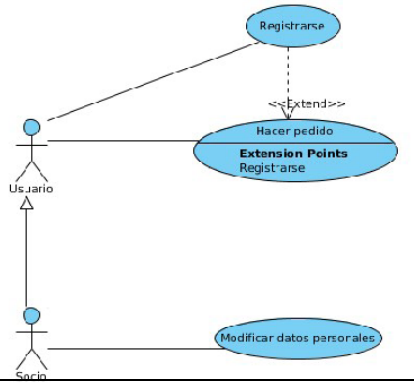
Por ejemplo, el usuario del sistema de venta por Internet puede a su vez darse de alta en la página web para que tengan sus datos registrados a la hora de hacer el pedido, en este caso el usuario es la generalización del socio.

Ambos actores pueden hacer un pedido, pero solo el socio puede modificar sus datos en el sistema.

2.3.3 Extensión.

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.

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.



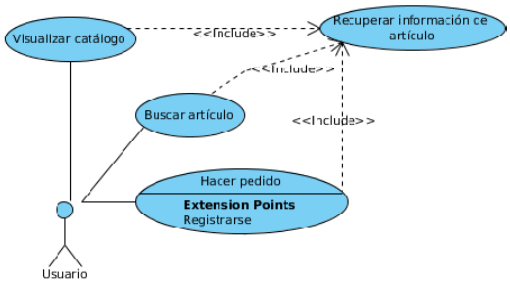
Por ejemplo, cuando un usuario hace un pedido si no es socio se le ofrece la posibilidad de darse de alta en el sistema en ese momento, pero puede realizar el pedido aunque no lo sea.

2.3.4 Inclusión.

Se incluye una relación entre dos casos de uso de tipo "**include**" cuando la ejecución del caso de uso incluido se da en la rutina normal del caso que lo incluye.

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.

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.



Las ventajas de esta asociación son:

- ✓ Las descripciones de los casos de uso son más cortas y se entienden mejor.
- ✓ La identificación de funcionalidad común puede ayudar a descubrir el posible uso de componentes ya existentes en la implementación.

Las desventajas son:

- ✓ La inclusión de estas relaciones hace que los diagramas sean más difíciles de leer, sobre todo para los clientes.

Cuando usamos relaciones de inclusión o extensión no podemos olvidar que los casos de uso extendidos o incluidos deben cumplir con las características propias de un caso de uso, es decir, deben representar un flujo de actividad completo desde el punto de vista de lo que un actor espera que el sistema haga por él, así como no utilizar estas herramientas sólo para descomponer un caso de uso de envergadura en otros más pequeños, piedra angular del diseño estructurado y no del orientado a objetos.

Suponer el siguiente sistema que modela el caso de uso Servir pedido en el que el Empleado de almacén revisa si hay suficientes artículos para hacer el pedido y si todo es correcto, el pedido se embala y se envía:
¿Qué tipo de relación emplearías en el modelo del dibujo?

- ☐ Asociación
- ☐ Generalización
- ☐ Extends
- ◉ **Include**

Así, la consulta de existencias debe realizarse necesariamente y, además, tiene entidad suficiente como para ser un caso de uso por si mismo, que puede usarse en otros casos.

2.4 Elaboración de casos de uso.

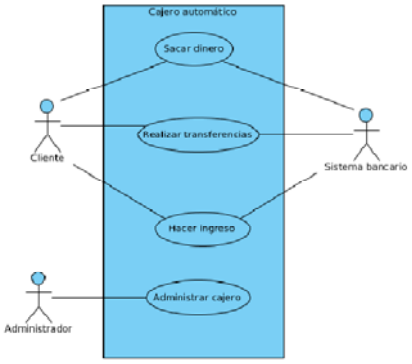
CASO PRÁCTICO

Después de analizar todos los elementos que formen un diagrama de casos de uso el equipo de Ada está preparado para hacer frente a su primer diagrama de casos de uso.

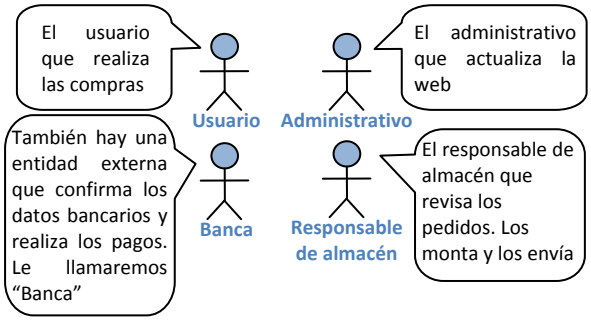
Sea cual sea el tipo de diagrama que estemos creando, cuando lo hacemos realizamos un proceso de abstracción por el cual representamos elementos de la realidad esquemáticamente, y en el diagrama de casos de uso pasa igual, necesitamos abstraer la realidad en un dibujo, en el que representamos qué cosas pueden hacerse en nuestro sistema y quien las va a hacer. Necesitamos diagramas que incluyan suficiente información para que el equipo de desarrollo tome las decisiones más adecuadas en la fase de análisis y diseño para una construcción de software que cumpla con los requerimientos, así como que sean útiles en la fase de implementación en un lenguaje orientado a objetos.

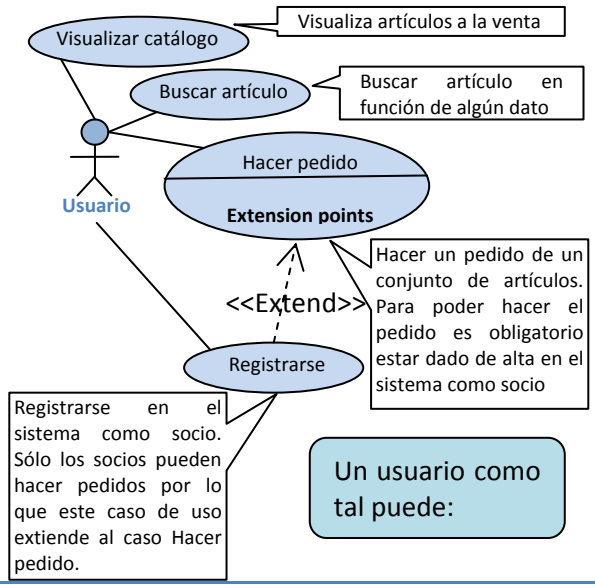
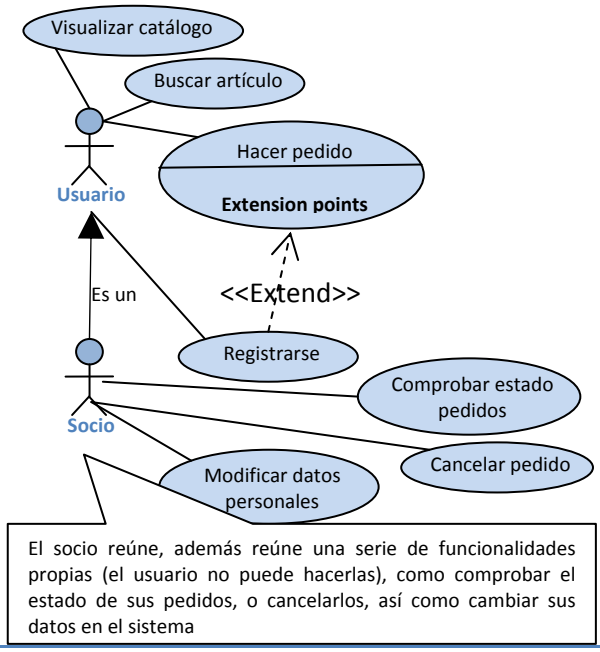
Partiremos de una descripción lo más detallada posible del problema a resolver y trataremos de detectar quien interactúa con el sistema, para obtener los actores diagrama de casos de uso, a continuación buscaremos qué tareas realizan estos actores para determinar los casos de uso más genéricos. El siguiente paso es refinar el diagrama analizando los casos de uso más generales para detectar casos relacionados por inclusión (se detectan fácilmente cuando aparecen en dos o más casos de uso generales), extensión y generalización. Al diagrama generado se le denomina diagrama frontera.

Se conoce como **diagrama frontera** al diagrama de casos de uso que incluye todos los casos de uso genéricos del sistema, que podrán ser desglosados después en nuevos diagramas de casos de uso que los describan si es necesario. Se especifica enmarcando los casos de uso en un recuadro, que deja a los actores fuera.

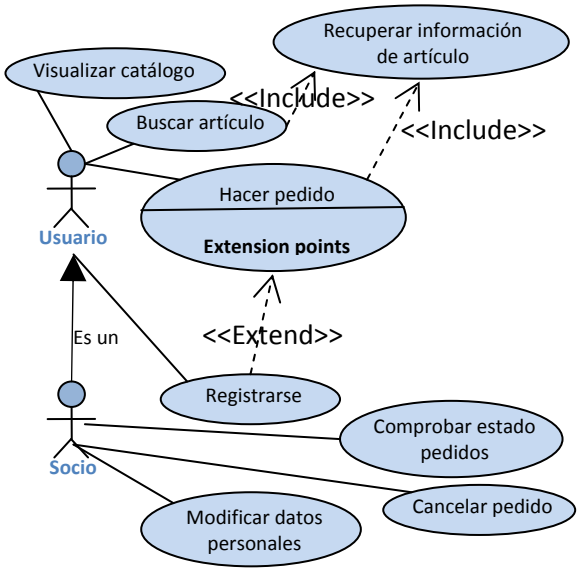


Creación de casos de uso

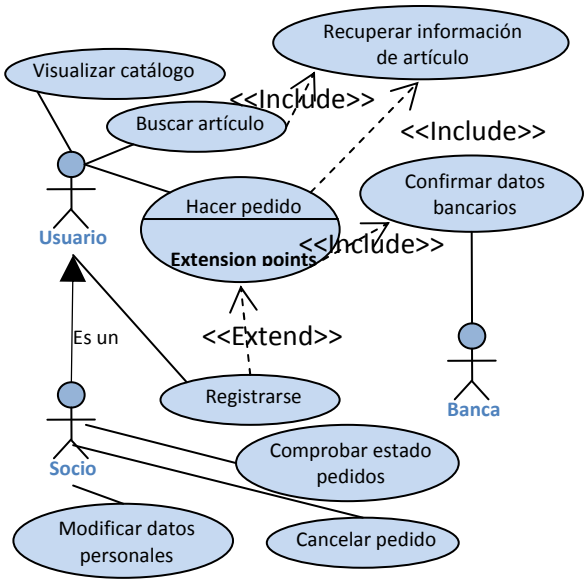
Primeros pasos	Antes de elaborar el diagrama tienes que leer con detenimiento el documento con la especificación del problema a resolver y asegurarte de que entiendes la idea central del problema, crear una tienda virtual en la que se puedan realizar pedidos de los productos a la venta (zapatos). El 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 actores	En nuestro sistema ¿Quién interviene? 
Identificar funcionalidades	Para facilitar la creación del diagrama vamos a ir sacando funcionalidades para cada usuario. Debemos recordar que un 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.
Funcionalidad 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**.
	
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 almacenan cierta información, como sus datos personales y bancarios, email, etc.	

Al revisar un poco la funcionalidad de los casos de uso del usuario podemos comprobar que en los casos **Buscar articulo** 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 de uso "**Recuperar información de artículo**" que se incluye en los otros dos.



Cuando se realiza el pedido es obligatorio hacer una comprobación de los datos bancarios del cliente, que dependen de una entidad externa, por lo que se añade otro caso de uso para esta función, "Comprobar datos bancarios" que trae de la mano la inclusión del actor Banca.

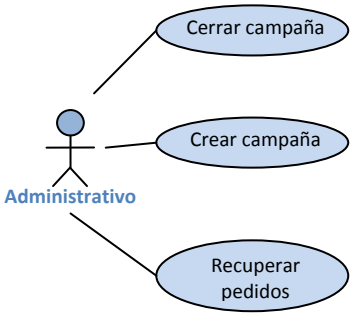


Funcionalidad del administrador web

El objetivo del administrativo 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 nueva, 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.

Se incluye el caso de uso **Recuperar pedidos** en **Crear campaña** por dos motivos:

1. Es una función que puede llevarse a cabo de forma independiente, tanto por el administrador de la web como desde el almacén, y
2. Es de comprobación obligatoria antes de cerrar la temporada.



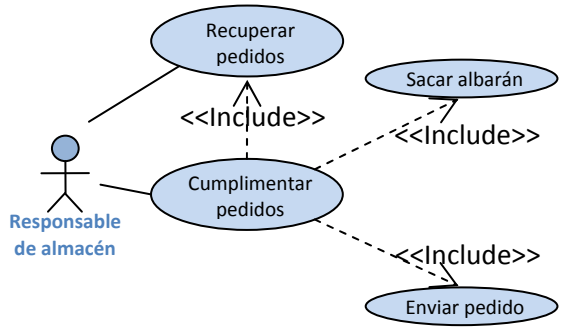
Funcionalidad del responsable de almacén

Es el encargado de leer los pedidos de los usuarios y cumplimentarlos. Ésta 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

El primer paso es **recuperar la lista de pedidos** sin procesar. Esta tarea recupera el pedido más antiguo para ser procesado.
Sacar albarán produce un listado en papel con la información del pedido para el socio.
Enviar pedido es colocar el pedido en la ruta de envío más apropiada para su destino.



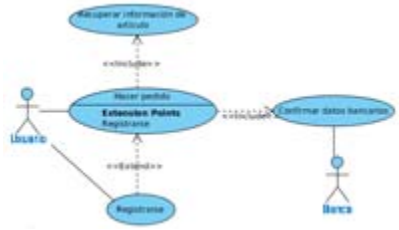
Especificación del problema a modelar.

Descripción del problema: "El tacón de oro".

Los usuarios del sistema navegan 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 siguiente datos: nombre completo, correo electrónico y dirección.
Los socios pueden hacer pedidos de los artículos. 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 cuanto 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 cancelarlos siempre y cuando no hayan sido cumplimentados por el empleado de almacén. Así mismo puede modificar sus datos personales.

Documentación del caso de uso "Hacer pedido".

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 **Hacer 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. Recordamos el aspecto del caso de uso:



Los datos que debemos incluir para elaborar la documentación del caso de uso, el contrato, 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 rellenar:

- ✓ 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 rellenado 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 se 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 rellenar 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 rellenado antes, además, podemos rellenar 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.

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.

Use Case	Hacer pedido		
Author	usuario		
Date	26-ago-2011 13:56:56		
Brief Description	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.		
Preconditions	Existe un catálogo de productos disponibles para pedir. El usuario está registrado. Los datos bancarios son correctos.		
Post-conditions	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.			
Author	usuario		
Date	26-ago-2011 17:56:35		
Brief Description	Cuando el usuario hace un pedido si no está registrado se abre la opción de registro para que se dé de alta.		
Preconditions	El usuario no está registrado. Existe un catálogo de artículos para hacer pedido. Los datos bancarios son correctos.		
Post-conditions	El usuario queda registrado. 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 la cantidad.	
	5		Recupera la información del artículo para obtener su precio y modifica el precio total a

			pagar por el pedido.
	6		Añade la información al pedido en creación.
	7	EL proceso se repite hasta completar la lista de artículos del pedido.	
	8	Acepta el pedido.	
	9		Se comprueba si el usuario es socio.
	10		Se invoca el registro de usuario.
	11		Se comprueban los datos bancarios con una entidad externa.
	12		Se genera: calcula el total, sumando los gastos de envío.
	13		Se realiza el pago a través de la entidad externa.
	14		El pedido queda almacenado con el estado "pendiente".

Flujo alternativo para hacer pedido cuando los datos bancarios no son correctos.			
Author	usuario		
Date	26-ago-2011 18:14:35		
Brief Description	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.		
Preconditions	Existe un catálogo de productos disponibles para pedir. El usuario está registrado. Los datos bancarios no son correctos.		
Post-conditions	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	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".

El resto de casos se uso se documentan de forma similar hasta completar la descripción formal de la funcionalidad del sistema.

2.5 Escenarios

Caso práctico

Ada continua la investigación, junto con el equipo de BK programación, que una vez ha creado su primer diagrama de casos de uso, se da cuenta de que realmente es una herramienta muy útil a la hora de definir la funcionalidad de un sistema. Continuando con la investigación descubren una ventaja adicional, utilizando los flujos de eventos, pueden describir interacciones concretas de los actores con el sistema, estas interacciones son los escenarios.

Un caso de uso debe especificar un comportamiento deseado, pero no imponer cómo se llevará a cabo ese comportamiento, es decir, debe decir QUÉ pero no CÓMO. Esto se realiza utilizando escenarios que son casos particulares de un caso de uso.

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.

Por ejemplo, para el caso de uso hacer pedido podemos establecer diferentes escenarios:

Un posible escenario podría ser:

Realizar pedido de unos zapatos y unas botas.

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 pares de zapatos que ha comprado, con el estado “pendiente”.

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

3. Diagramas de secuencia

Caso práctico
María se ha dado cuenta de que los casos de uso permiten, de una manera sencilla, añadir información sobre qué hace el sistema, sin embargo por completa que sea la descripción de la secuencia de eventos no permite incluir información útil, como los objetos que intervienen en las tareas, y como se comunican.
—Tendríamos que buscar la forma de representar como circula la información, que objetos participan en los casos de uso, qué mensajes envían, y en que momento, esto nos ayudaría mucho a completar después el diagrama de clases.
Como siempre, Ada tiene una solución.
—Tendremos que investigar los diagramas de secuencia.

Los diagramas de secuencia se utilizan para formalizar la descripción de un escenario o conjunto de ellos representando que mensajes fluyen en el sistema así como quien los envía y quien los recibe.

Los objetos y actores que forman parte del escenario se representan mediante rectángulos distribuidos horizontalmente en la zona superior del diagrama, a los que se asocia una linea temporal vertical (una por cada actor) de las que salen, en orden, los diferentes mensajes que se pasan entre ellos. Con esto el equipo de desarrollo puede hacerse una idea de las diferentes operaciones que deben ocurrir al ejecutarse una determinada tarea y el orden en que deben realizarse.

Los diagramas de secuencia completan a los diagramas de casos de uso, ya que permiten al equipo de desarrollo hacerse una idea de qué objetos participan en el caso de uso y como interaccionan a lo largo del tiempo.

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

CASO PRÁCTICO
Ada, orienta a su equipo:
—Bien, ¿qué nos haría falta para poder representar la interacción de los objetos que participan en el caso de uso a lo largo del tiempo?
—Alguna manera de representar los objetos, el paso del tiempo y el paso de mensajes ¿no?

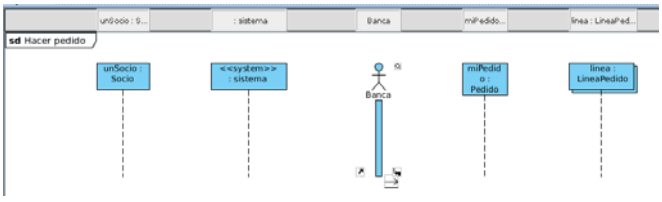
Representación de objetos y linea de vida.

En un diagrama de secuencia se dibujan las entidades que participan dentro de rectángulos que se distribuyen horizontalmente. De cada rectángulo o entidad sale una línea de puntos que representa el paso del tiempo, se les denomina línea de vida y representan que el objeto existe.

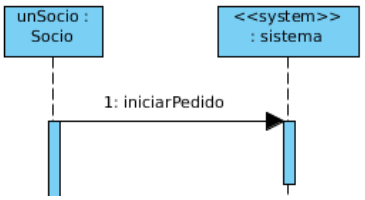
Para formar el nombre de una linea de vida de un objeto se usa el nombre del objeto, que es opcional, seguido del símbolo dos puntos y a continuación el nombre de la clase a la que pertenece.

Una linea de vida puede estar 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 un botón para abrir una ventana o una llamada a una subrutina.

Cuando tenemos un objeto que puede tener varias instancias, aparece como un rectángulo sobre otro, como en es el caso de las lineas del pedido que pueden ser varias.



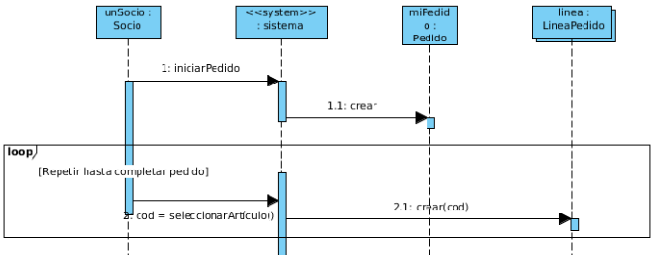
Invocación de métodos.



Los mensajes, que significan la invocación de métodos, se representan como flechas horizontales que van de una linea de vida a otra, indicando con la flecha la dirección del mensaje, los extremos de cada mensaje se conectan con una línea de vida que conecta a las entidades al principio del diagrama. Los mensajes se dibujan desde el objeto que envía el mensaje al que lo recibe, pudiendo ser ambos el mismo objeto y su orden viene determinado por su posición vertical, un mensaje que se dibuja debajo de otro indica que se envía después, por lo que no se hace necesario un número de secuencia.

Iteraciones y condicionales.

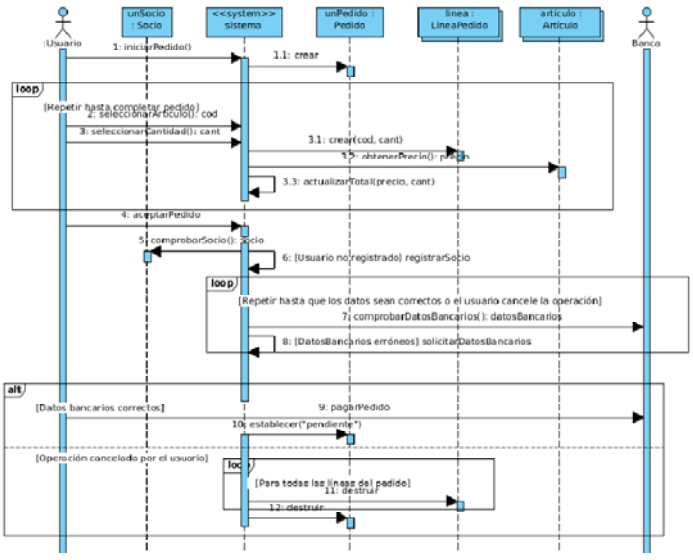
Además de presentar acciones sencillas que se ejecutan de manera secuencial también se pueden representar algunas situaciones más complejas como bucles usando marcos, normalmente se nombra el marco con el tipo de bucle a ejecutar y la condición de parada. También se pueden representar flujos de mensajes condicionales en función de un valor determinado.



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

3.2 Elaboración de un diagrama de secuencia.

Vamos a generar el diagrama de secuencia para el caso de uso **Hacer pedido**. En el 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.

Creación del diagrama de colaboración

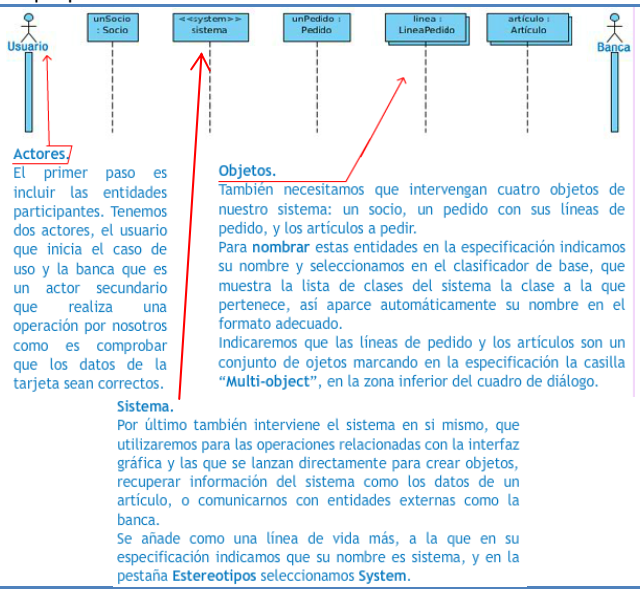
Crear el diagrama de secuencia

- ✓ 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.

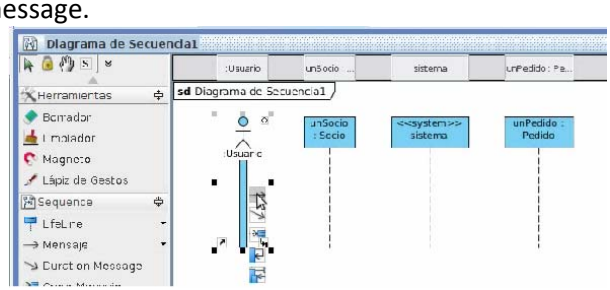
✓ Este es un diagrama muy completo que incluye bastantes elementos de este diagrama, como bucles o condicionantes, veamos como incluirlos con la herramienta Visual Paradigm, como siempre, debes saber que puedes investigar y utilizar otras herramientas que sirvan para este mismo propósito.

Incluir entidades

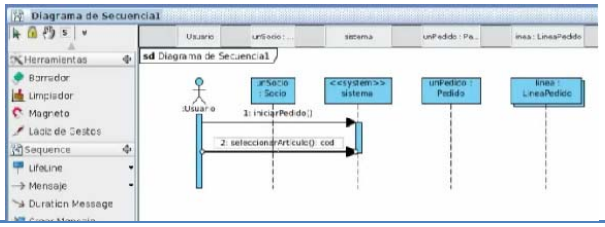


Mensajes

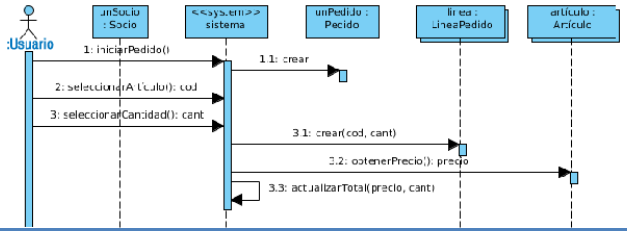
Los mensajes se añaden pinchando sobre la línea de vida que envía el mensaje, y seleccionando la flecha, entonces se abre una ventana en la que elegir la línea destino, ten en cuenta que un objeto puede enviarse un mensaje a si mismo seleccionando la oprición Self-message.



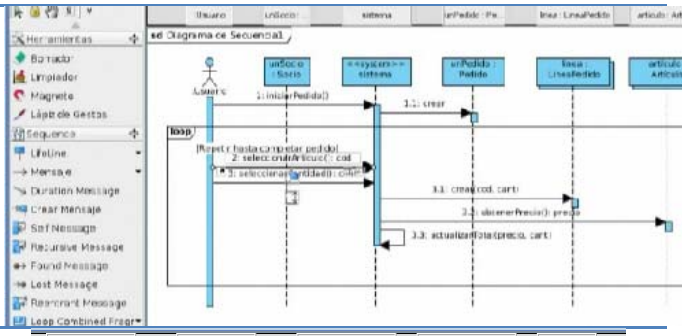
Los mensajes pueden devolver un valor, que podemos escribir en la especificación del mismo en la opción return value.



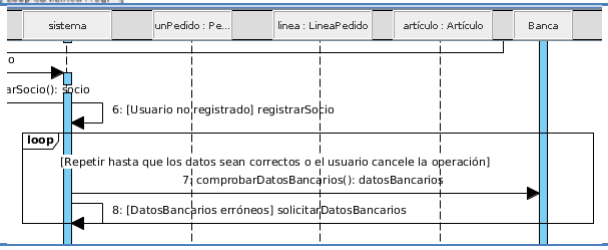
En esta imagen vemos los mensajes correspondientes al apartado de añadir los datos del pedido. Se selecciona el artículo y la cantidad, se crea una nueva línea de pedido y se recuperan los datos del artículo para obtener su precio y actualizar el total. Puesto que este proceso lo podemos repetir en más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



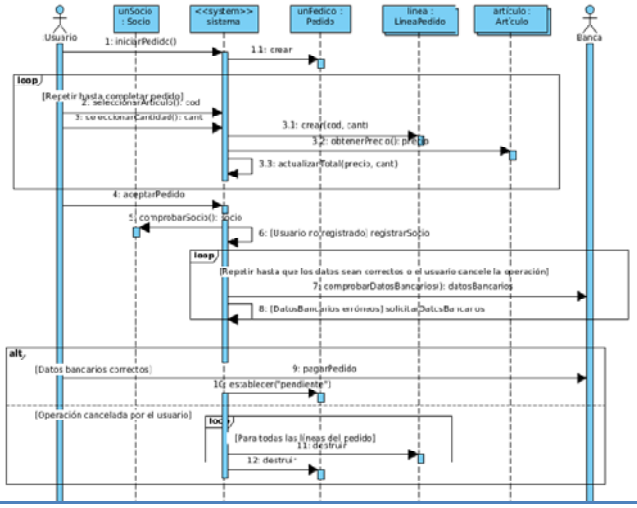
Para añadir un bucle seleccionamos la opción Loop Combined Fragment.



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.



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 sin guardar nada.



¿Cual de estos elementos no forma parte de un diagrama de secuencia?

- ☐ Actor
- ☐ Objeto
- ☐ Bucle
- ☉ **Evento**

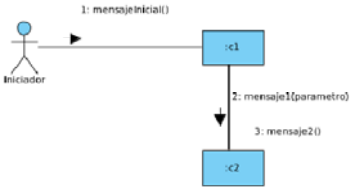
4. Diagramas de colaboración.

CASO PRÁCTICO.
—El diagrama de secuencia ha aportado información muy valiosa sobre la circulación de mensajes en los casos de uso, sin embargo estaría bien poder mostrar esta información de otra forma en la que se apreciase mejor el anidamiento de los mensajes, y el flujo de control entre objetos, ¿no creéis?

Los diagramas de colaboración son un complemento para los de secuencia cuyo objetivo es mostrar las interacciones entre los objetos del diagrama mediante el paso de mensajes entre ellos. Las interacciones entre los 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 pueden enviar mensajes entre ellos. Los objetos se conectan mediante enlaces y se comunican a través de los mensajes.

Como, a diferencia de los diagramas de secuencia, no se incluye una línea temporal los mensajes son numerados para determinar su orden el el tiempo.

En este diagrama de colaboración el actor Iniciador manda un mensaje al objeto c1 que inicia el escenario, a continuación el objeto c1 envía el mensaje mensaje1 que lleva un parámetro al objeto c2 y después el mensaje mensaje2, que no tiene parámetros de nuevo al objeto c2.



Los diagramas de colaboración y secuencia utilizan los mismos elementos pero distribuyéndolos de forma diferente, ¿crees que son semejantes?
Es cierto, ambos diagramas representan la misma información, representan entidades del sistema y los mensajes que circulan entre ellas, además en ambos casos es fácil detectar la secuencialidad bien por el uso de lineas de vida, bien por los números de secuencia. De hecho en algunas aplicaciones para el desarrollo de estos diagramas es posible crear un diagrama a partir del otro.

4.1 Representación de objetos.

CASO PRÁCTICO.
—De acuerdo, mientras investigamos los diagramas de colaboración vamos a ver con un poco más de detalle qué significa la notación que se asigna a los objetos, ¿que diferencia hay entre usar los dos puntos o no hacerlo? ¿Podemos usar el nombre de una clase, solamente, o es obligatorio indicar el nombre del objeto?

Un objeto puede ser cualquier instancia de las clases que hay definidas en el sistema, aunque también pueden incluirse objetos como la interfaz del sistema, o el propio sistema, si esto nos ayuda a modelar las operaciones que se van a llevar a cabo.

Los objetos se representan mediante rectángulos en los que aparece uno de estos nombres.

- ✓ **NombreClase:** directamente se puede utilizar el nombre de la clase a la que pertenece el objeto que participa en la interacción. Pero esta representación hace referencia a la clase, el resto son objetos.

- ✓ **NombreObjeto** : se puede usar el nombre concreto del objeto que participa en la interacción, normalmente aparece subrayado.

- ✓ **:nombreClase** : cuando se coloca el símbolo : delante del nombre de la clase quiere decir que hace referencia a un objeto genérico de esa clase.

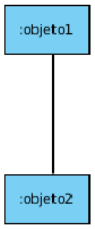
- ✓ **NombreObjeto:nombreClase** : hace referencia al objeto concreto que se nombre añadiendo la clase a la que pertenece.

4.2 Paso de mensajes.

CASO PRÁCTICO.

—Y cuando enviamos un mensaje ¿cómo se representa exactamente?, ¿podemos incluir de alguna forma parámetros en los mensajes o valores devueltos? ¿Y si necesitamos indicar que el mensaje se enviará sólo si se cumple una determinada condición? ¿o que se envía dentro de un bucle?

Para que sea posible el paso de mensajes es necesario que exista una asociación entre los objetos. En la imagen es posible el paso de mensajes entre el objeto objeto1 y objeto2, además de quedar garantizada la navegación y visibilidad entre ambos.



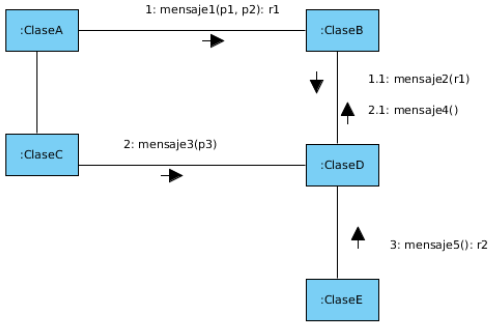
Un mensaje es la especificación de una comunicación entre objetos que transmite información y desencadena una acción en el objeto destinatario. La sintaxis de un mensaje es la siguiente:

```
[secuencia][*] [Condición de guarda]{valorDevuelto} : mensaje (argumentos)
```

donde:

- ✓ **Secuencia**: representa el nivel de anidamiento del envío del mensaje dentro de la interacción. 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.
- ✓ **ValorDevuelto**: 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.

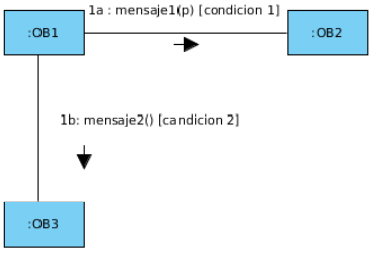
Como se ve en el ejemplo se puede usar la misma asociación para enviar varios mensajes. Vemos que hay dos mensajes anidados, el 1.1 y el 2.1, se ha usado el nombre de los mensajes para indicar el orden real en el que se envían.



Los mensajes 1, 2 y 3 tiene parámetros y los mensajes 1 y 5 devuelve un resultado.

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

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



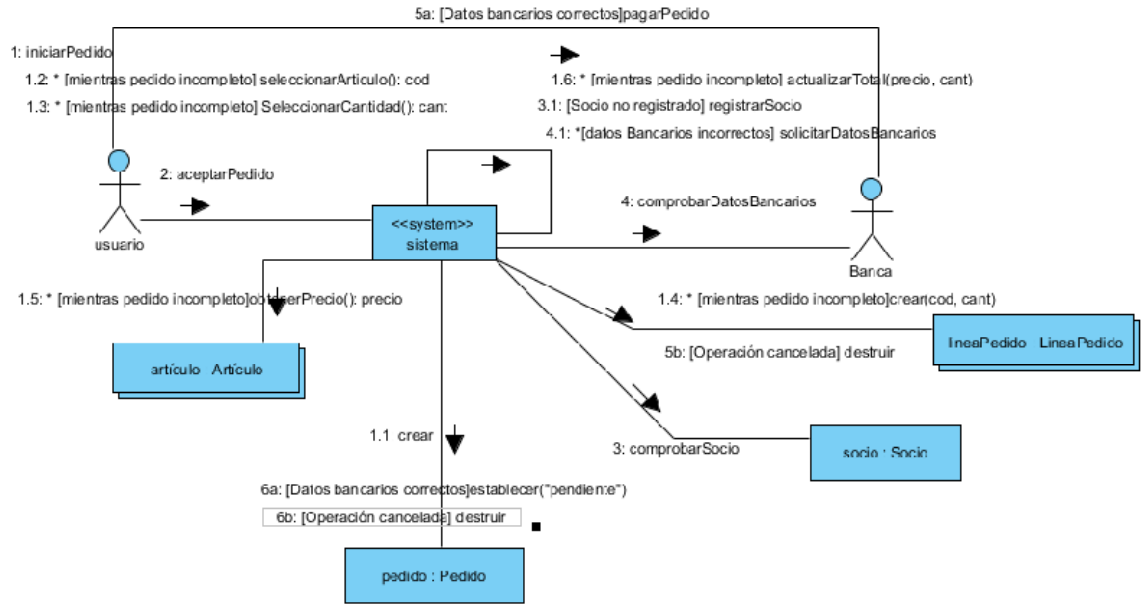
Cuando tenemos una condición se repite el número de secuencia y se añaden las condiciones necesarias, como vemos en la imagen según la condición se enviará el mensaje 1 o el 2, pero no ambos, por lo que coinciden en número de secuencia.

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

Nota: VP-UML modifica el orden en el que aparecen los datos pero no su notación.

4.3 Elaboración de un diagrama de colaboración.

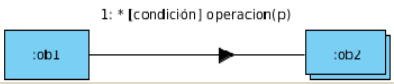
Este es el diagrama de colaboración del caso de uso **Hacer pedido**, se ha creado siguiendo el diagrama de secuencia, por lo que no te debe ser muy difícil seguirlo, de hecho algunas aplicaciones para la creación de estos diagramas permiten la obtención de uno a partir de otro. Debes tener en cuenta que la aplicación modifica un poco la signatura 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 se 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.

Indica qué afirmación no es correcta para el siguiente diagrama:



- ☐ El objeto ob2 es multiobjeto
- ☐ Se envía un mensaje del objeto 1 al objeto 2
- ◉ **El mensaje operación(pp) se ejecutará siempre**
- ☐ La operación se puede ejecutar varias veces

5. Diagramas de actividad.

CASO PRÁCTICO.
Por el momento el equipo de BK no ha tenido problema en seguir lo que Ada les cuenta sobre los diagramas UML. Antonio, que está verdaderamente interesado en el tema hace a Ada la siguiente pregunta:
—¿Que pasaría si quisiera representar sólo las acciones que tienen lugar, prescindiendo de quien las genera, solo el flujo de la actividad del sistema, qué pasa primero, qué ocurre después y qué cosas pueden hacerse al mismo tiempo?
—Pasaría que tendrías que hacer un diagrama de actividad.

El Diagrama de Actividad es una especialización del Diagrama de Estado, organizado respecto de las acciones, que se compone de una serie de actividades y representa cómo 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 actividad.

Se utilizan fundamentalmente 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 llevar a cabo concurrentemente.

Permite visualizar la dinámica del sistema desde otro punto de vista que complementa al resto de diagramas. En concreto define la lógica de control:

- ✓ **En el modelado de los procesos del negocio**: Permiten especificar y evaluar el flujo de trabajo de los procesos de negocio.
- ✓ **En el análisis de un caso de uso**: Permiten comprender qué acciones deben ocurrir y cuáles son las dependencias de comportamiento.
- ✓ **En la comprensión del flujo de trabajo, a través de varios casos de uso**: Permiten representar con claridad las relaciones de flujo de trabajo (workflow) entre varios casos de uso.
- ✓ Aclaran cuando se trata de expresar **aplicaciones multihilos**.

Un diagrama de actividades es un grafo conexo en el que los nodos son estados, que pueden ser de actividad o de acción y los arcos son transiciones entre estados. El grafo parte de un nodo inicial que representa el estado inicial y termina en un nodo final.

¿Por qué decimos que el diagrama de actividades visualiza el comportamiento desde otro punto de vista del resto de diagramas?
En los anteriores diagramas, tratábamos la interacción entre objetos y entidades, cual es el flujo de mensajes, en qué orden y bajo qué condiciones se envían, en los diagramas de actividades se incluye el factor de la concurrencia, y trata sobre todo de expresar el flujo de las actividades, su elemento fundamental, y como se pasa de unas a otras. Además también representa como se influye en los objetos.

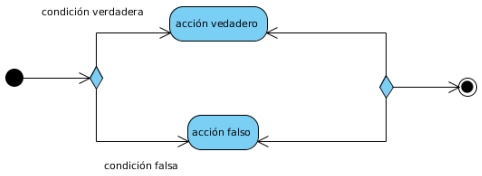
5.1 Elementos del diagrama de actividad.

CASO PRÁCTICO.
—Vale estoy preparado, ¿qué necesito para tener un diagrama de actividad?
Normalmente los diagramas de actividades contienen:

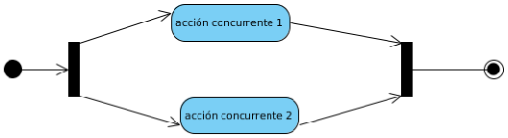
- ✓ **Estados** de actividad y estados de acción.
  - ➔ **Estado de actividad**: Elemento compuesto cuyo flujo de control se compone de otros estados de actividad y de acción.

- **Estado de acción**: Estado que representa la ejecución de una acción atómica, que no se puede descomponer ni interrumpir, normalmente la invocación de una operación. Generalmente se considera que su ejecución conlleva un tiempo insignificante.
- Pueden definirse también otro tipo de estados:
  - **Inicial**.
  - **Final**.

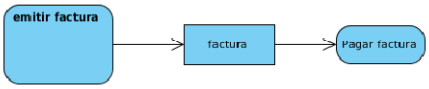
- **Transiciones**: Relación entre dos estados que indica que un objeto en el primer estado realizará ciertas acciones y pasará al segundo estado cuando ocurra un evento específico y satisfaga ciertas condiciones. Se representa mediante una línea dirigida del estado inicial al siguiente. Podemos encontrar diferentes tipos de transacciones:
  - **Secuencial o sin disparadores**: Al completar la acción del estado origen se ejecuta la acción de salida y, sin ningún retraso, el control sigue por la transición y pasa al siguiente estado.
  - **Bifurcación(Descision node)**: Especifica caminos alternativos, elegidos según el valor de alguna expresión booleana. Las condiciones de salida no deben solaparse y deben cubrir todas las posibilidades (puede utilizarse la palabra clave else). Pueden utilizarse para lograr el efecto de las iteraciones.



  - **Fusión (Merge node)**: Redirigen varios flujos de entrada en un único flujo de salida. No tiene tiempo de espera ni sincronización.
  - **División (Fork node)**: Permiten expresar la sincronización o ejecución paralela de actividades. Las actividades invocadas después de una división son concurrentes.
  - **Unión (Join node):** Por definición, en la unión los 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 los flujos de eventos habituales sino flujos de objetos (se representan con una flecha de igual manera) que permiten mostrar los objetos que participan dentro del flujo de control asociado a un diagrama de actividades. Junto a ello se puede indicar cómo cambian los valores de sus atributos, su estado o sus roles.



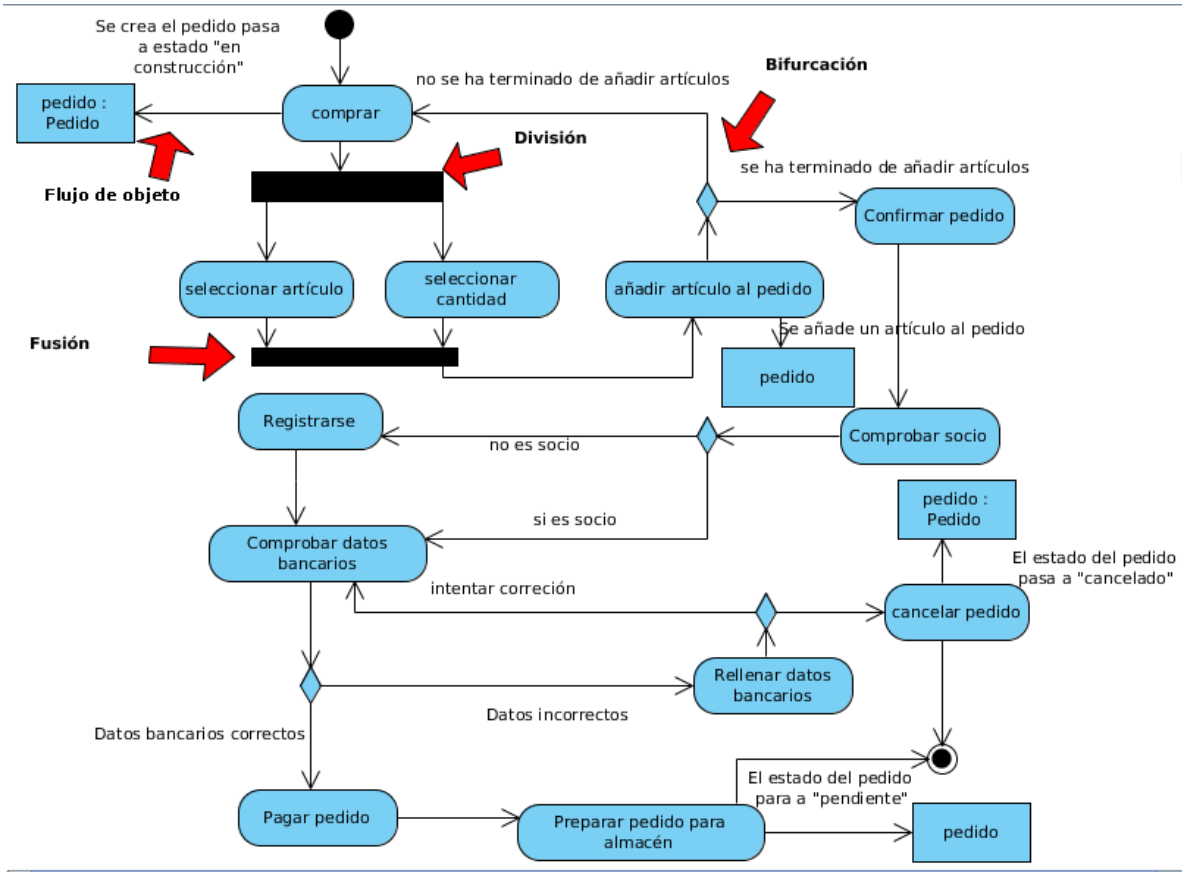
Se utilizan carriles o calles para ver **QUIENES** son los responsables de realizar las distintas actividades, es decir, especifican qué parte de la organización es responsable de una actividad.

- 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 de otras calles.

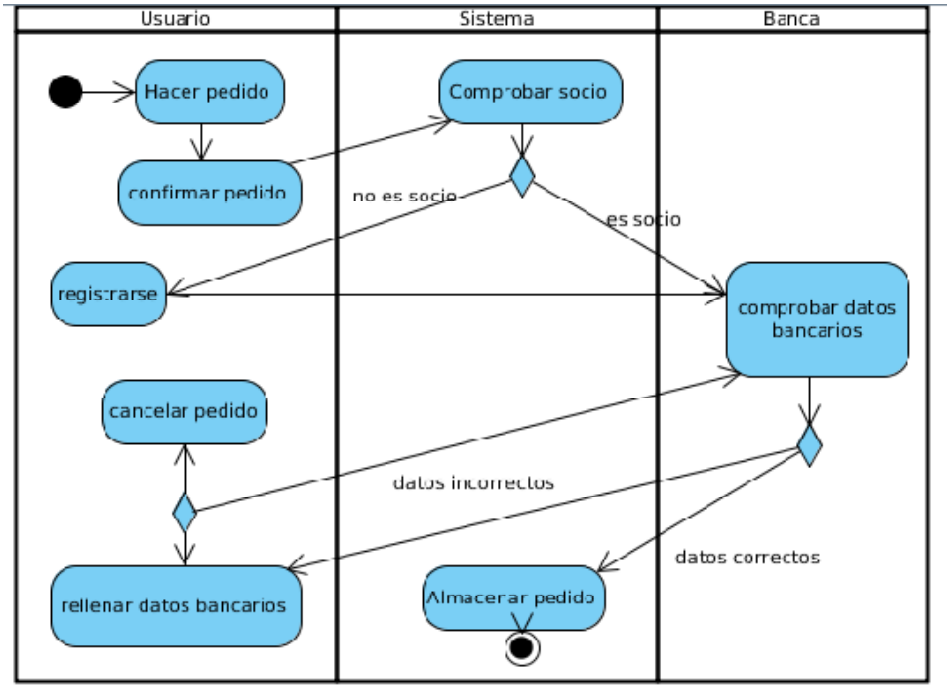
5.2 Elaboración de un diagrama de actividad.

El siguiente diagrama de actividad representa el caso de uso hacer pedido, 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:



PARA AÑADIR LAS CALLES EN VISUAL PARADIGM SE UTILIZA LA HERRAMIENTA DEL PANEL "VERTICAL SWIMLANE"

Los diagramas de actividades, a diferencia del resto, permiten incluir la concurrencia en la representación del diagrama.

- [x] Verdadero
- [ ] Falso

6. Diagramas de estados.

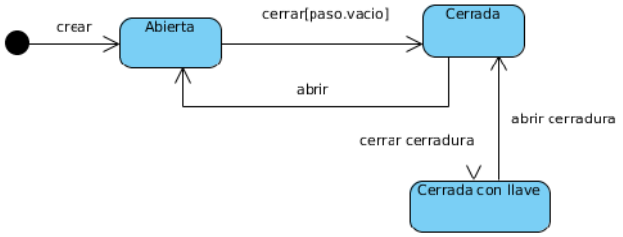
CASO PRÁCTICO.
Ada espera que su equipo continúe con tan buen ánimo para estudiar un tipo de diagrama más, que completará las diferentes visiones de la dinámica de un sistema que proporciona UML. Son los diagramas de estados, que les permitirán analizar cómo va cambiando el estado de los objetos que tienen una situación variable a lo largo del tiempo.

Basado en los statecharts de David Harel. Representan máquinas de estados (autómatas de estados finitos *(Modelo matemático que realiza cómputos sobre una entrada para producir una salida. Se representa mediante un grafo cerrado y dirigido cuyos vértices son estados y sus aristas son transiciones, que van etiquetadas con un símbolo que representa el evento que desencadena la transición. Parte de un estado inicial y termina en uno o varios estados finales.)*) para modelar el comportamiento dinámico basado en la respuesta a determinados eventos de aquellos objetos que requieran su especificación, normalmente por su comportamiento significativo en tiempo real y su participación en varios casos de uso. El resto de objetos se dice que tienen un único estado.

En relación con el diagrama de estados se cumple que:

- ✓ Un objeto está en un estado concreto en un cierto momento, que viene determinado, parcialmente, por los valores de sus atributos.
- ✓ La transición de un estado a otro es momentánea y se produce cuando ocurre un determinado evento.
- ✓ Una máquina de estados procesa un evento cada vez y termina con todas las consecuencias del evento antes de procesar otro. Si ocurren dos eventos simultáneamente se procesan como si se hubieran producido en cualquier orden, sin pérdida de generalidad.

Un diagrama de máquina de estados expresa el comportamiento de un objeto como una progresión a través de una serie de estados, provocada por eventos y las acciones relacionadas que pueden ocurrir. Por ejemplo, aquí tenemos el diagrama de estados de una puerta.



Analiza el diagrama de estados de la puerta, según está dibujado, ¿Se puede abrir una puerta que está cerrada con llave directamente?

○ Verdadero ◉ **Falso**

6.1 Estados y eventos.

CASO PRÁCTICO.
Ada indica a su equipo que para entender bien la dinámica de un diagrama de estados deben comenzar por analizar sus componentes fundamentales: estados y eventos.

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

Elementos de un estado

- ✓ Nombre
- ✓ Acciones entrada/salida.
- ✓ Actividad a realizar.
- ✓ Subestados, cuando el estado es complejo y necesita de un diagrama que lo defina.
- ✓ Eventos diferidos.

Existen dos tipos de estado especiales, estado inicial y estado final.

- ✓ **Estado inicial**: es un pseudoestado que indica el punto de partida por defecto para una transición cuyo destino es el límite de un estado compuesto. El estado inicial del estado de nivel más alto representa la creación de una nueva instancia de la clase.
- ✓ **Estado final**: Estado especial dentro de un estado compuesto que, cuando está activo, indica que la ejecución del estado compuesto ha terminado y que una transición de finalización que sale del estado compuesto está activada.

Un evento es un acontecimiento que ocupa un lugar en el tiempo y espacio que funciona como un estímulo que dispara una transición en una máquina de estados. Existen eventos externos y eventos internos según el agente que los produzca.

Tipos de eventos:

- ✓ **Señales (excepciones)**: la recepción de una señal, que es una entidad a la que se ha dado nombre explícitamente (clase estereotipada), prevista para la comunicación explícita - y asíncrona- entre objetos. Es enviada por un objeto a otro objeto o conjunto de objetos. Las señales con nombre que puede recibir un objeto se modelan designándolas en un compartimento extra de la clase de ese objeto. Normalmente una señal es manejada por la máquina de estados del objeto receptor y puede disparar una transición en la máquina de estados.
- ✓ **Llamadas**: la recepción de una petición para invocar una operación. Normalmente un evento de llamada es modelado como una operación del objeto receptor, manejado por un método del receptor y se implementa como una acción o transición de la máquina de estados.
- ✓ **Paso de tiempo**: representa el paso del tiempo (ocurrencia de un tiempo absoluto respecto de un reloj real o virtual o el paso de una cantidad de tiempo dada desde que un objeto entra en un estado). Palabra clave `after`: `after (2 segundos); after 1 ms desde la salida de devInactivo`.
- ✓ **Cambio de estado**: evento que representa un cambio en el estado o el cumplimiento de una condición. Palabra clave `when`, seguida de una expresión booleana, que puede ser de tiempo o de otra clase: `when (hora = 11:30); when (altitud < 1000)`.

6.2 Transiciones.

CASO PRÁCTICO.

—De acuerdo, los estados son situaciones específicas en las que se puede encontrar un objeto, y los eventos pueden hacer que un objeto cambie de estado, y, ¿cómo representamos eso?

Una transición de un estado A a un estado B, se produce cuando se origina el evento asociado y se satisface cierta condición especificada, en cuyo caso se ejecuta la acción de salida de A, la acción de entrada a B y la acción asociada a la transición.

La signatura de una transición es como sigue:

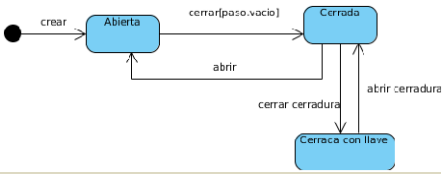
```
Evento(argumentos) [condición] / accion
```

donde todos los elementos son opcionales.

Elementos de una transición:

- ✓ **Estados origen y destino**: la transición se disparará si, estando en el estado origen se produce el evento de disparo y se cumple la condición de guarda (si la hay), pasando a ser activo el estado final.
- ✓ **Evento de disparo**: cuando se produce un evento, afecta a todas las transiciones que lo contienen en su etiqueta. Todas las apariciones de un evento en la misma máquina de estados deben tener la misma signatura. Los tipos de evento los hemos visto en el punto anterior.
- ✓ **Condición de guarda**: expresión booleana. Si es falsa, la transición no se dispara, y si no hay otra transición etiquetada con el mismo evento que pueda dispararse, éste se pierde.
- ✓ **Acción**: computación atómica ejecutable. Puede incluir llamadas a operaciones del objeto que incluye la máquina de estados (o sobre otros visibles), creación o destrucción de objetos o el envío de una señal a otro objeto.

Recordemos el diagrama de estado de la puerta: ¿Qué significa la signatura de la transición "cerrar [paso.vacio]"?



- ☐ Que cuando cerremos la puerta el paso quedará vacío.
- ◉ **Que para cerrar la puerta el paso debe estar vacío.**
- ☐ Que cuando se está cerrando la puerta se vacía el paso.

Los corchetes en un diagrama UML significan una condición que se debe cumplir. No se podrá cerrar la puerta hasta que el paso no esté vacío.

6.3 Creación de un diagrama de estados

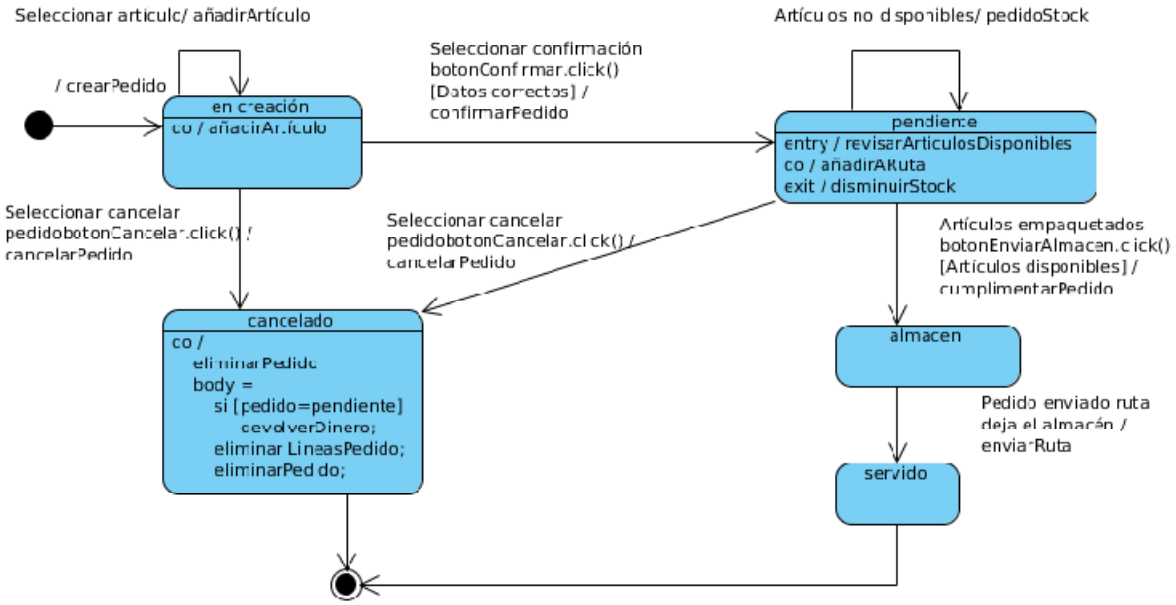
Para ejemplificar la creación de un diagrama de estados vamos a ver el que se asocia al objeto pedido, que cumple con las condiciones que hemos visto al principio, tiene un comportamiento significativo en tiempo real, ya que su situación tanto física, como el sistema, va evolucionando conforme pasa el 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 llegarse 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:

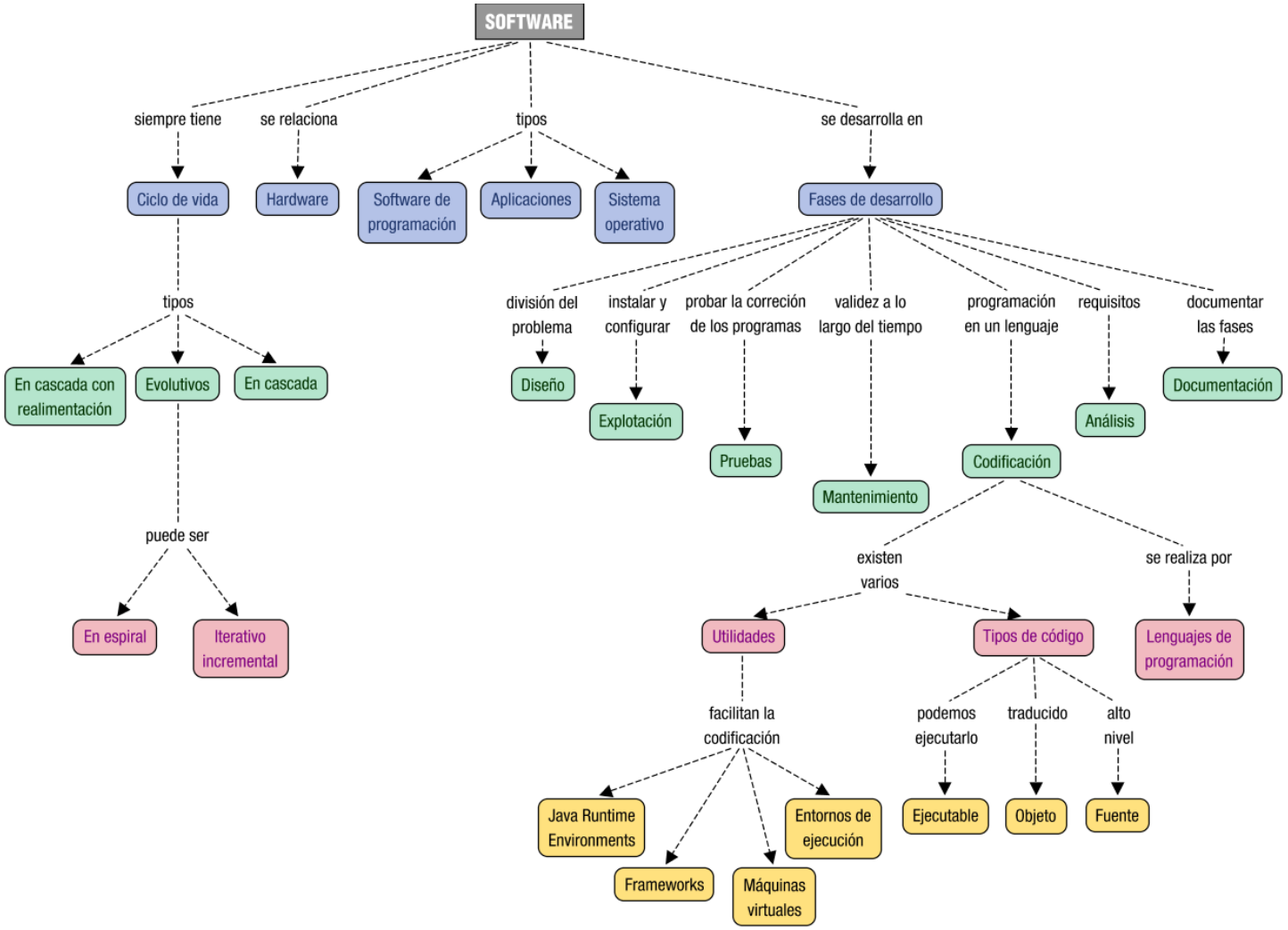


En las **transiciones** se ha incluido el nombre de la transición, el evento que la dispara (normalmente hacer clic en algún botón de la interfaz), si existe condición de guarda se pone entre corchetes y la acción a realizar para llegar al siguiente estado junto al símbolo /. En todos los casos el evento 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.

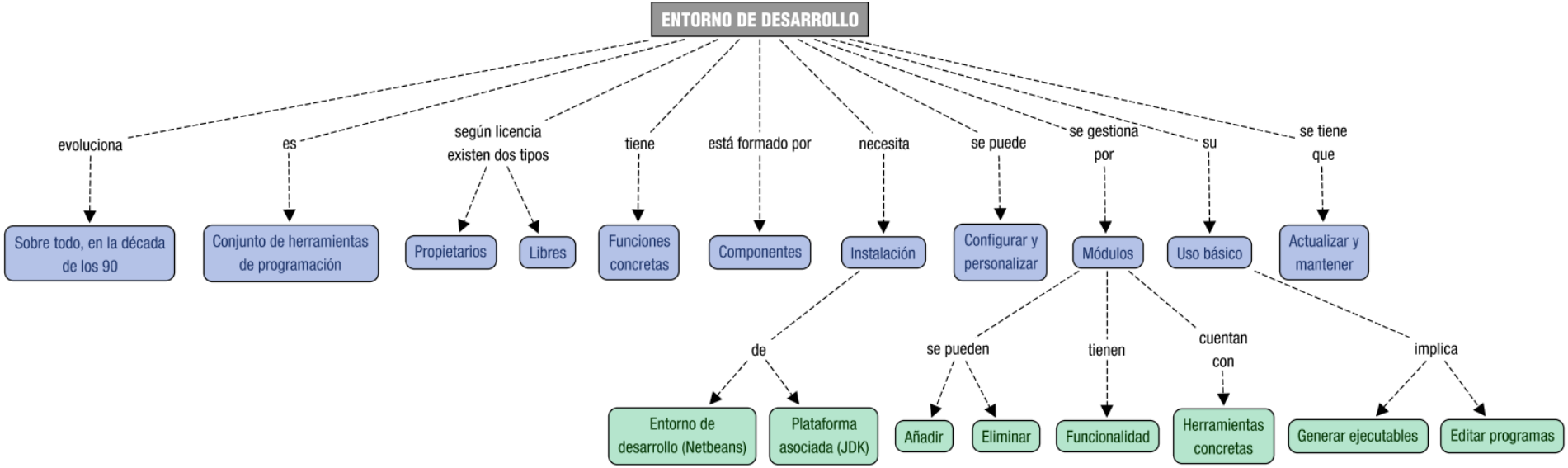
A los **estados** se les ha añadido la acción a realizar, apartado do/ y en algunos casos la acción de entrada, por ejemplo en el caso del estado pendiente, se debe revisar que los artículos a enviar tengan disponibilidad y la de salida, en el ejemplo disminuir el stock.

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.

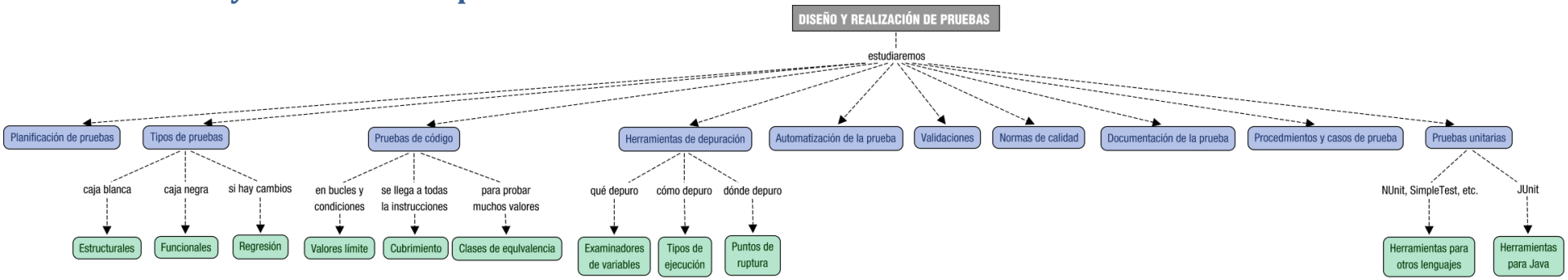
Tema 1 - Software



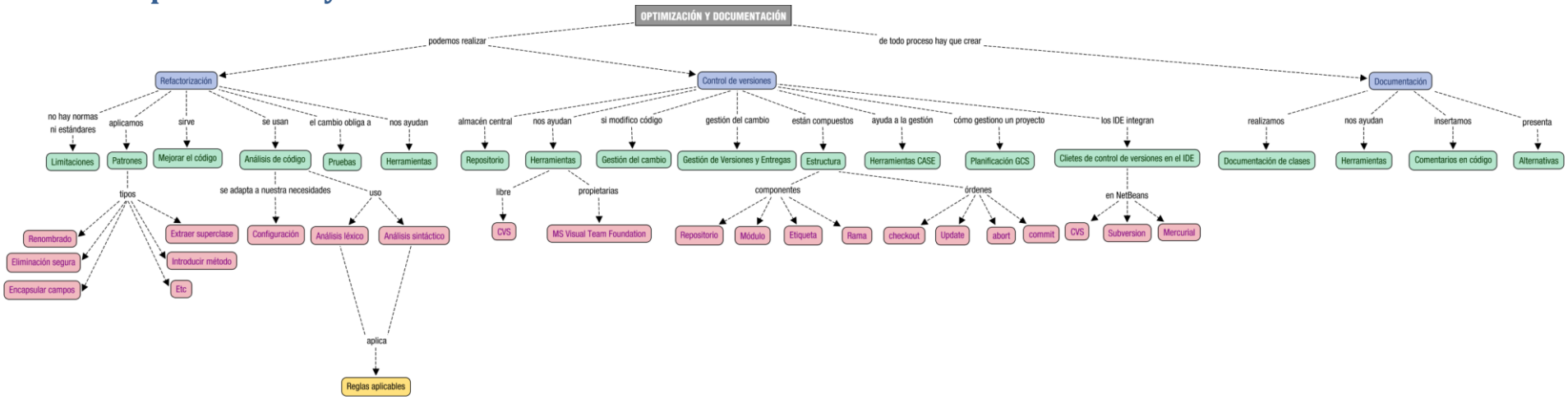
Tema 2 - Entorno de Desarrollo



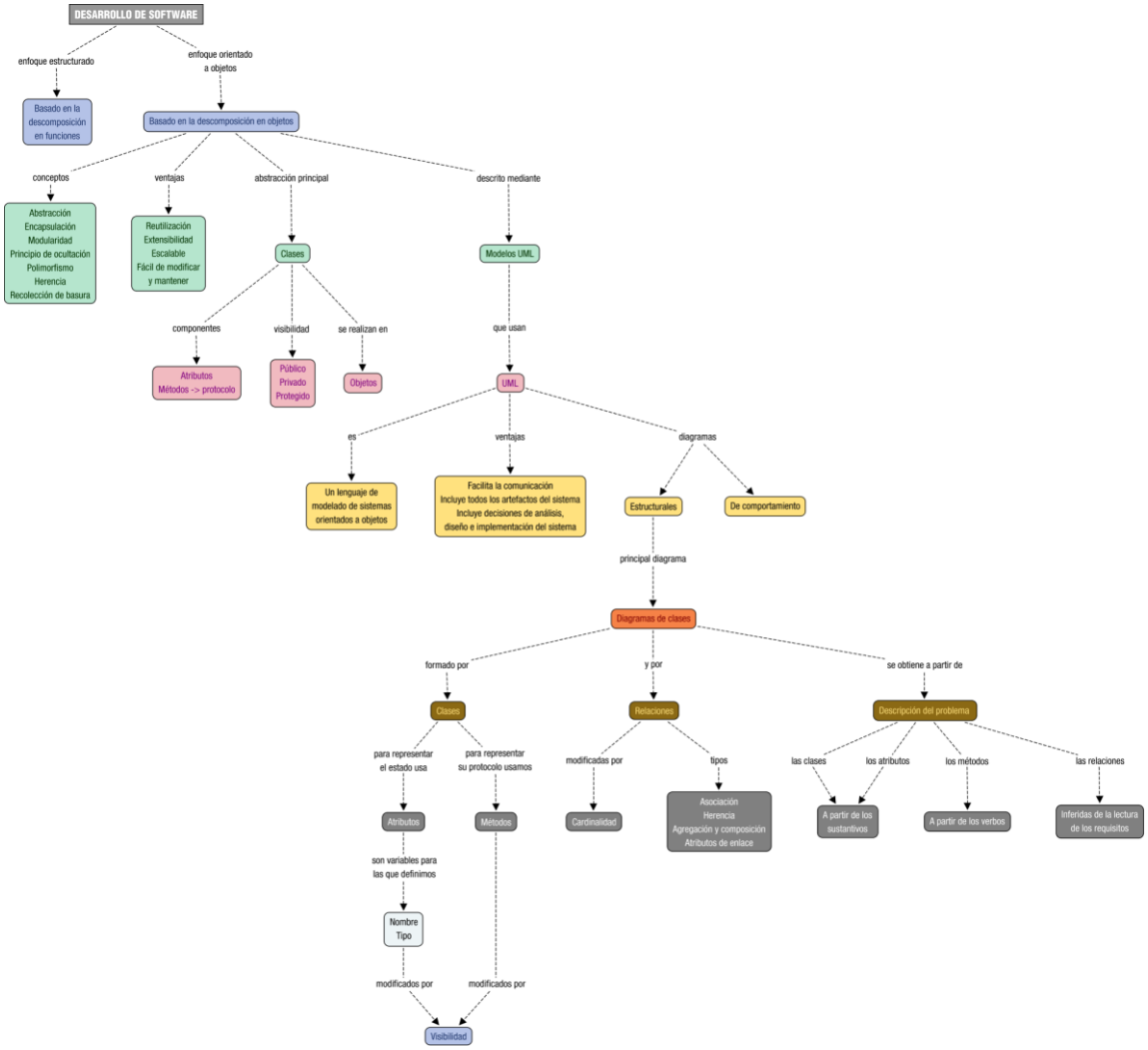
Tema 3 - Diseño y realización de pruebas



Tema 4 - Optimización y Documentación



Tema 5 - Desarrollo de software



Tema 6 - Diagramas de comportamiento