UD 01 - DESARROLLO DE SOFTWARE

1a. Tipos de software

Software: Conjunto de programas informáticos que actúan sobre el hardware para ejecutar lo que el usuario desee.

Se distinguen tres tipos de software:

- **Sistema operativo**: Software base que debe estar instalado para que las aplicaciones se puedan ejecutar.
- **Software de programación**: herramientas que permitan desarrollar programas informáticos.
- **Aplicaciones**: Conjunto de programas con una finalidad más o menos concreta. Los programas son un conjunto de instrucciones escritas en un lenguaje de programación.

1b. Relación hardware-software

La relación hardware-software puede plantearse desde dos puntos de vista:

- El sistema operativo: Coordina el hardware de forma transparente durante el funcionamiento del ordenador. Es el intermediario entre este y las aplicaciones, consumidoras de recursos hardware (Tiempo de CPU, espacio de RAM, interrupciones, dispositivos de E/S)
- Las aplicaciones: El hardware del equipo debe interpretar y ejecutar el código en el que están escritos las aplicaciones. El código es entendible por el humano; el hardware entiende señales eléctricas (ausencia o presencia de tensión -binario-). Hay, por tanto, un proceso de traducción de código para que el ordenador ejecute las sentencias escritas en el lenguaje de programación.

2. Desarrollo de software

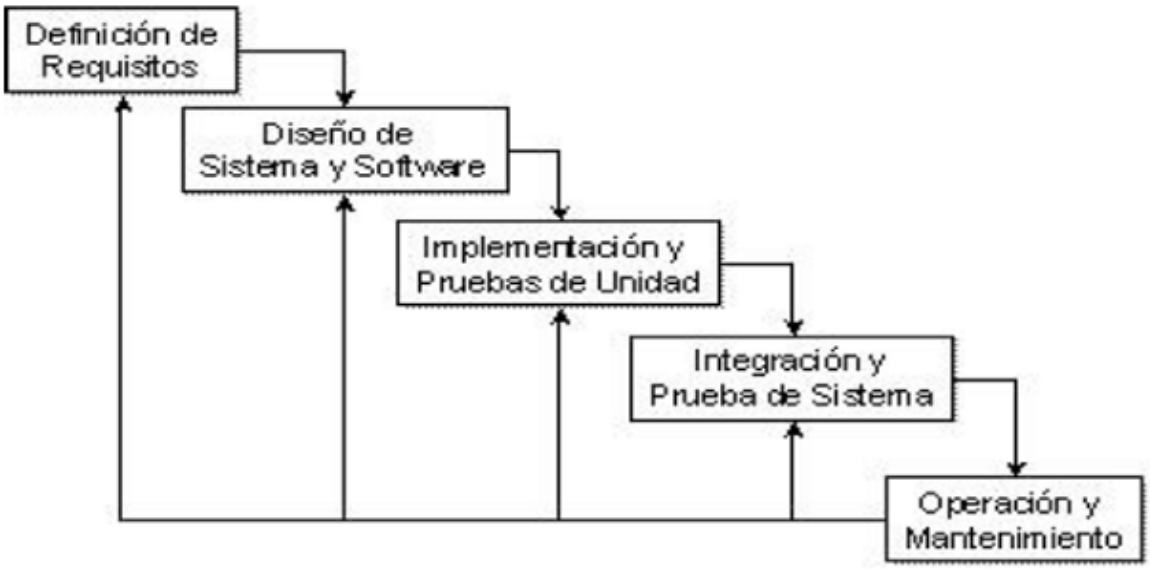
Proceso desde que se concibe una idea hasta que el programa está implementado y funcionando. Debe tener etapas de obligado cumplimiento para garantizar que los programas sean eficientes, fiables, seguros y respondan a la necesidad del usuario final. Exige de coordinación y disciplina del grupo de trabajo que lo desarrolla.
Estas etapas se adoptan siguiendo un modelo o ciclo de vida del software.

2.1 Ciclos de vida del software

Lo más conocidos y utilizados son:

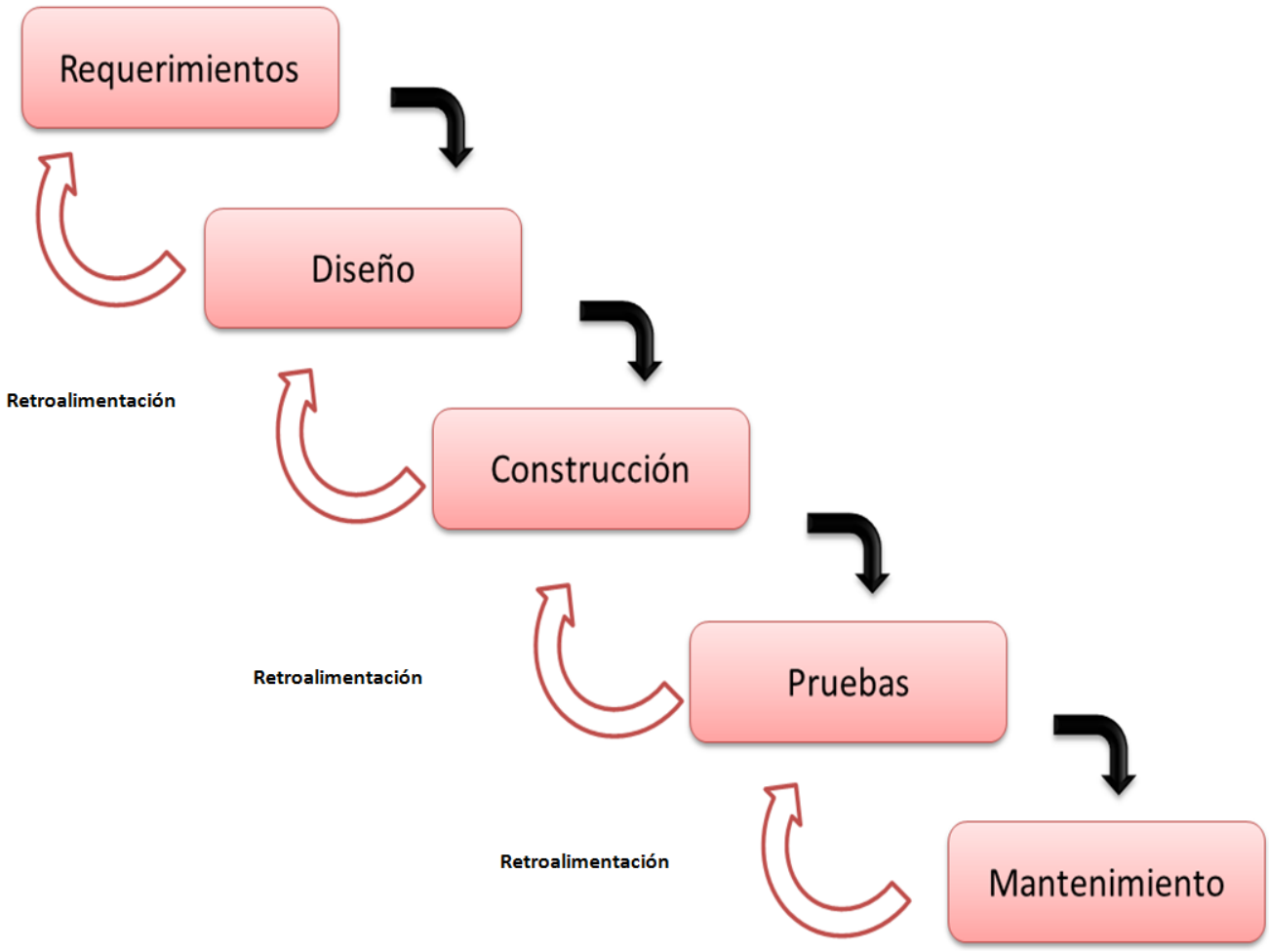
1. **Modelo en Cascada**

- Modelo clásico de vida del software. Las etapas pasan de una a otra sin retorno posible (se presupone que no habrá errores, ni variaciones del software)
- Prácticamente imposible de utilizar, requiere conocer todos los requisitos del sistema. Solo **aplicable a pequeños desarrollos**



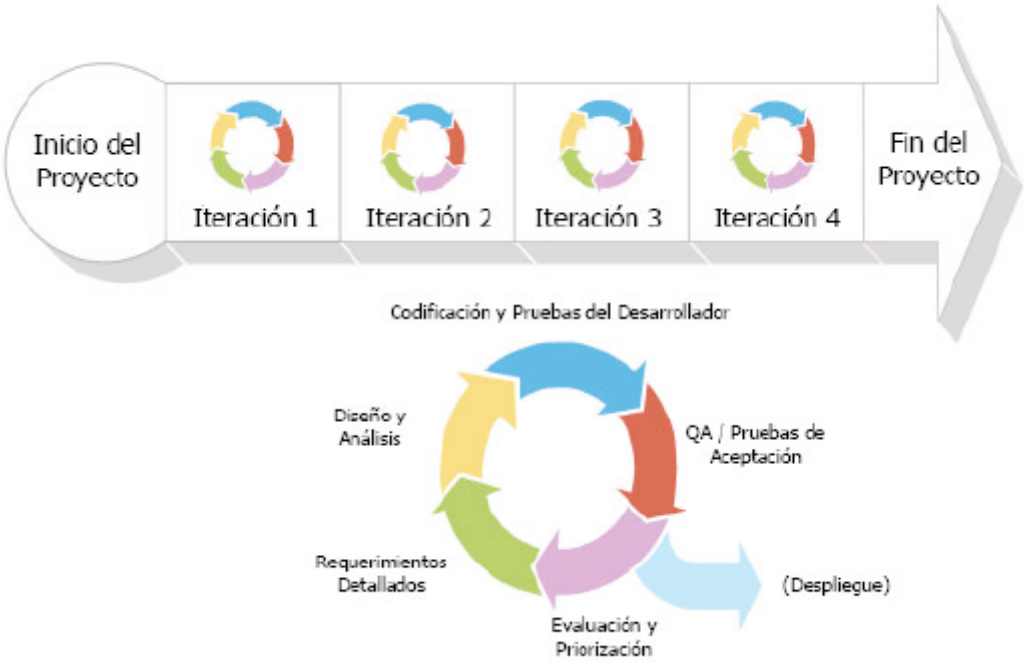
2. **Modelo en Cascada con Realimentación**

- Variación del modelo en cascada, con realimentación entre etapas de forma que se pueda volver atrás para corregir, modificar o depurar.
- Si se prevén muchos cambios durante el desarrollo no es el más adecuado.
- Modelo perfecto si el **modelo es rígido** (pocos cambios, poca evolución) y los **requisitos están claros**

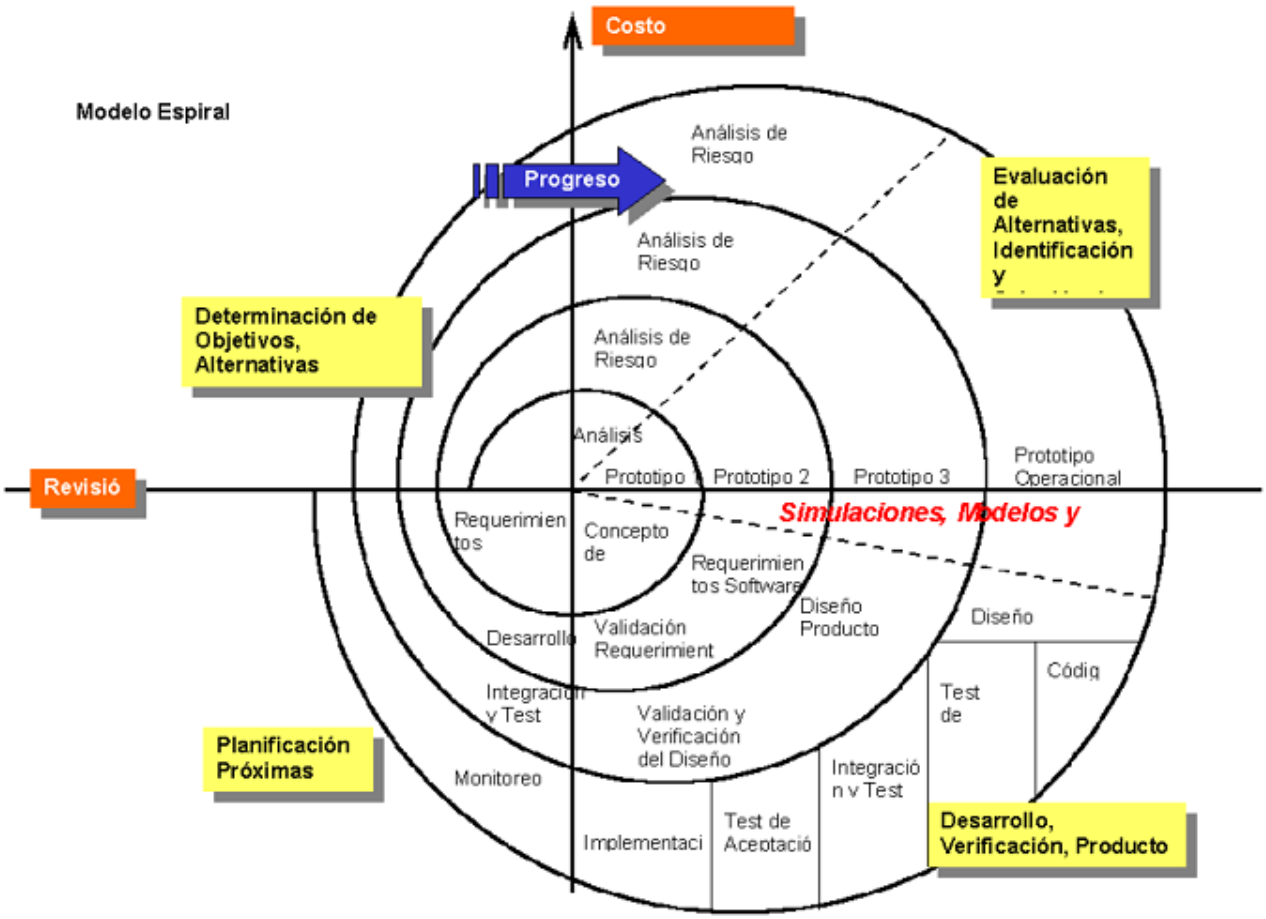


3. **Modelos Evolutivos**
Más modernos, tienen en cuenta la naturaleza cambiante y evolutiva del software
3.1. **Modelo Iterativo-Incremental**
- Basado en el modelo en cascada con realimentación.
- Las fases se repiten y refinan. Van propagando su mejora a las fases siguientes.

- Proyecto se desarrolla en pequeñas porciones (incremental) en sucesivas iteraciones **(sprints)** al final de las cuales se puede ver lo que se ha desarrollado y, antes de comenzar la siguiente iteración (sprint) se pueden ver los requerimientos que no se conocen o estan mal implementados o incluso introducir nuevos requerimientos (adaptativo).
- Cada sprint debe dar un resultado completo (incremento del valor del producto final) listo para entregar al cliente.



3.2. **Modelo en Espiral**
- Combinación del modelo iterativo-incremental con el modelo en cascada.
- El software se construye repetidamente con versiones cada vez mejores porque incrementan la funcionalidad en cada versión. Modelo bastante complejo.



2.2 Herramientas de apoyo al desarrollo de software

Herramientas CASE (Computed Aided Software Engineering) son aplicaciones que se usan en el desarrollo de software para ahorrar costes y tiempos del proceso. Buscan automatizar las fases de desarrollo de software ya que permiten:

- Mejorar planificación
- Agilizar proceso
- Reutilizar partes en proyectos futuros
- Que las aplicaciones respondan a estándares
- Mejorar el mantenimiento de los programas
- Mejorar el proceso de desarrollo, visualizando fases de forma gráfica.

Desarrollo rápido de aplicaciones RAD (Rapid Application Development) es proceso de desarrollo de software que incluye el desarrollo iterativo, la construcción de prototipos y el uso de utilidades CASE.

Según la fase del ciclo de vida en la que ofrecen ayuda pueden clasificarse en:

- **U-CASE** En planificación y análisis de requisitos
- **M-CASE** En análisis y diseño
- **L-CASE** En programación de software, detección de errores de código, depuración de programas y pruebas y en generación de documentación. Podríamos citar herramientas de generación semiautomática de código, editores UML, de refactorización de código, de sistemas de control de versiones. Ej.: ArgoUML, Use Case Maker, ObjectBuilder...

3. Lenguajes de programación

3.1. Definición, características y clasificación

Lenguaje de programación: Idioma creado de forma artificial, formado por un conjunto de símbolos y normas que se aplican sobre un alfabeto para obtener código que el hardware de la computadora pueda entender y ejecutar.

Puede definirse como el conjunto de **ALFABETO** (símbolos permitidos), **SINTAXIS** (normas de construcción permitidas con el alfabeto), **SEMÁNTICA** (significado de las construcciones para hacer acciones válidas.)

- Nos permiten comunicarnos con el hardware del ordenador.
- Su elección depende de las características del problema a resolver
- Evolucionan hacia la mayor usabilidad de los mismos
- Java, C, C++, PHP y Visual Basic concentran 60% del interés de la comunidad (Mentira, eso era hace siglos... pero aprendámoslo por si acaso)

**A. Según el nivel de abstracción

- Lenguaje máquina
- Lenguaje ensamblador
- De bajo nivel: Próximos al funcionamiento de la computadora
- De alto nivel: Próximos al razonamiento humano

LENGUAJE MÁQUINA

- El ordenador lo entiende directamente
- Instrucciones como combinaciones de unos y ceros
- Es el primer lenguaje utilizado pero hoy día no se usa
- Único para cada procesador

LENGUAJE ENSAMBLADOR

- Sustituyó al lenguaje máquina y se traduce a este para poder ejecutarse
- Se programa con mnemotécnicos (instrucciones complejas)
- Las instrucciones son sentencias que hacen referencia a la ubicación física
- Difícil de usar

LENGUAJE DE ALTO NIVEL BASADO EN CÓDIGO

- Sustituyó al ensamblador
- Se programa con sentencias y órdenes derivadas del idioma inglés
- Cercanos al razonamiento humano

LENGUAJES VISUALES

- Sustituirían a los de alto nivel (Que te has creído tú eso)
- Se programa diseñando gráficamente con el ratón
- El código es autogenerado
- Portables de un equipo a otro
- Deben traducirse a lenguaje máquina

B. Según la técnica de programación utilizada

- Lenguajes de programación estructurada (Pascal, C)
- Lenguajes de programación orientada a objetos (C++, Java, Ada, Delphi, VB.NET, PowerBuilder...)
- Lenguajes de programación visuales (Visual Basic Net, Borland Delphi,...)

https://www.monografias.com/trabajos38/tipos-lenguajes-programacion/tipos-lenguajes-programacion2#interp

3.2. Lenguajes de programación estructurados

- Técnica que permite el uso de Sentencias secuenciales (una detrás de otra en el orden en el que fueron escritas), sentencias selectivas (condicionales), sentencias repetitivas (iteraciones o bucles)

  Comentemos algunas cosas
  Iteraciones: `if`, `else` , `then` (En bash para separar la condición de lo que debe ejecutarse)
  Bucles: `while`, `do`, `done` (En bash para indicar que el bucle finalizó)
- A partir de ellos se evoluciona a otros más complejos (orientado a objetos, orientado a eventos)
- Los requerimientos actuales son más complejos de lo que esta técnica puede hacer.
- Fue sustituida por la programación modular ("divide y vencerás")

VENTAJAS: Son fáciles de leer, sencillos, rápidos, fácil mantenimiento, estructura clara
INCONVENIENTES: El programa se concentra en un único bloque (difícil de mantener), no puede reutilizarse eficazmente el código al no estar codificado en módulos y cloques

3.3. Lenguajes de programación orientados a objetos

- Los programas se componen de objetos independientes entre sí que colaboran para realizar acciones.
- Clase: Colección de objetos con características similares. Mediante métodos los objetos se comunican con otros cambiando el estado de los mismos. Los objetos tienen atributos que los diferencian unos de otros.
  VENTAJAS: Los objetos son reutilizables por proyectos futuros. Los errores son más fácil de localizar y de depurar
  INCONVENIENTES: Programación no tan intuitiva.

4. Fases en el desarrollo y ejecución del software

Las etapas que siempre se deben construir son:

- **ANÁLISIS DE REQUISITOS**: Especificar requisitos funcionales y no funcionales del sistema
- **DISEÑO**: Dividir sistema en partes y determinar la función de cada una
- **CODIFICACIÓN**: Elegir lenguaje de programación y codificar
- **PRUEBAS**: Probar programas para detectar errores y depurar
- **DOCUMENTACIÓN**: Documentar y guardar información de todas las etapas
- **EXPLOTACIÓN**: Instalar, configurar y probar aplicación en equipos del cliente
- **MANTENIMIENTO**: Contacto con el cliente para actualizar y modificar la aplicación en el futuro

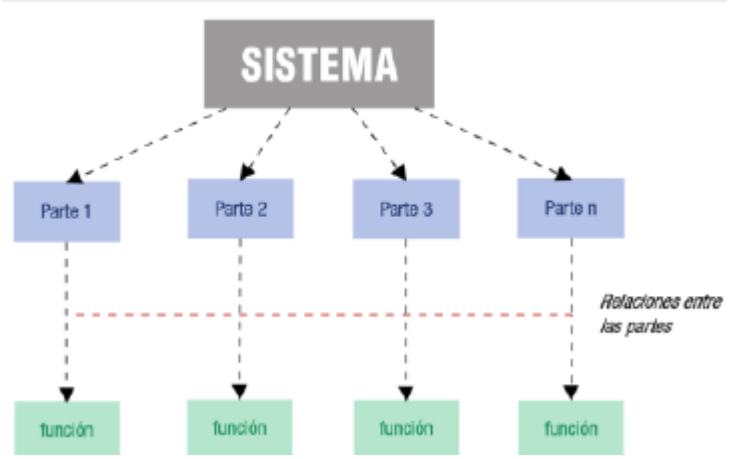
4.1. Análisis

¿Qué hay qué hacer?

- La fase más importante. Todo lo demás depende de lo bien que se haya detallado esta. Es la más complicada, no se puede automatizar y depende de la capacidad del analista. Es fundamental la comunicación entre analista y cliente para que cumpla las expectativas.
- Se analizan requisitos funcionales (funciones de la aplicación, respuesta ante las entradas, comportamiento en situaciones inesperadas) y no funcionales (tiempos de respuesta, legislación aplicable, tratamiento ante simultaneidad de peticiones).
- En el documento ERS (Especificación de Requisitos de Software) se especifican: La planificación de las reuniones, relación de los objetivos del cliente - sistema, relación de requisitos funcionales y no funcionales, objetivos prioritarios y temporización, reconocimiento de requisitos mal planteados o que conllevan contradicciones.

4.2. Diseño

¿Cómo hacerlo?
Debe dividirse el sistema en partes y establecer relaciones entre ellas. Decidir qué hará exactamente cada parte.
Debe crearse modelo funcional-estructura de los requerimientos del sistema para poder dividirlo y afrontar partes por separado.



Tomar decisiones importantes como:

- Entidades y relaciones de BBDD
- Lenguaje de programación a usar
- Selección del SGBD
- Diagrama de clases
- Diagrama de colaboración
- Diagrama de paso de mensajes
- Diagrama de casos de uso

4.3. Codificación.

Elegir un lenguaje de programación, codificar la información y llevarlo al código fuente.

Tiene que cumplir con los datos impuestos en el análisis y el diseño de la aplicación.

Todo código debería cumplir:

- Modularidad: Dividido en trozos más pequeños
- Corrección: Que haga lo que se pide
- Fácil de leer: Fácil desarrollo y mantenimiento
- Eficiencia: Buen uso de los recursos
- Portabilidad: Que se implemente en cualquier equipo

4.3.1. Fases en la obtención de código

El código pasará por
CÓDIGO FUENTE Escrito por el programador en editor de texto con lenguaje de programación de alto nivel y contiene instrucciones necesarias.

- Debe partirse de las etapas de análisis y diseño
- Es importante elaboración previa de un **algoritmo** (pasos a seguir para obtener solución del problema), diseñado en pseudocódigo (expresión de ideas en lenguaje a medio camino entre humano y programación). Gracias al algoritmo, la codificación será más rápida y directa.
- Elegir lenguaje de alto nivel apropiado
- Codificar el algoritmo

Finalmente se obtiene un documento con todos los módulos (parte de la aplicación con funcionalidad concreta), funciones (parte de código muy pequeña con finalidad concreta), bibliotecas y procedimientos (funciones que no devuelven ningún valor).

Será necesario traducirlo a código binario (código objeto)

Importante tener en cuenta su licencia distinguiendo entre código fuente abierto y código fuente cerrado.

CÓDIGO OBJETO: Código binario resultado de compilar el código fuente. Consiste en bytecode (código binario) distribuido en varios archivos, cada uno es un programa fuente compilado. El código objeto es código intermedio, no es inteligible inicialmente por el ser humano, tampoco por la computadora.

Compilación: Traducción de una sola vez con código objeto usando compilador. *Interpretación*: Traducción y ejecución del programa línea a línea usando intérprete. Los programas interpretados no producen código objeto -> Paso de fuente a ejecutable es directo. Es un proceso más lento que en compilación pero recomendable cuando se es inexperto al dar detección de errores más detallada.

CÓDIGO EJECUTABLE: Resultante de enlazar código objeto con rutinas y bibliotecas necesarias mediante un software llamado linker. Es un único archivo que puede ser directamente ejecutado por computadora, sin aplicación externa. El sistema operativo carga el código ejecutable en la RAM. Este código es el conocido como código máquina, inteligible por la computadora.

La generación de ejecutables. Editor se escribe código fuente en algún lenguaje; Código fuente se compila a código objeto o bytecode; Bytecode a través de máquina virtual pasa a código máquina ejecutable por computadora.

4.3.2. Máquinas virtuales

Tipo especial de software que separa el funcionamiento del ordenador de los componentes de hardware instalados.
Así puede desarrollar y ejecutar una aplicación sobre cualquier equipo, independientemente de sus características.

Funciones:

- Aplicaciones portables
- Reservar memoria y liberar la no utilizada
- Comunicarse con el sistema donde se instala la aplicación (host) para controlar los hardware implicados
- Cumplir normas de seguridad de aplicaciones
- Se aisla la aplicación de los detalles físicos del equipo. Con el bytecode se puede ejecutar en cualquier máquina si se tiene independencia independencia respecto al hardware que se vaya a usar. La máquina funciona como capa de software de bajo nivel, puente entre bytecode y dispositivos físicos. Verifica todo el bytecode y protege las direcciones de memoria.

4.3.2.1 Frameworks

Los **frameworks** son estructuras de ayuda al programador que permiten desarrollar proyectos sin partir desde cero. Plataforma software que tiene definidos programas soporte, bibliotecas, lenguaje interpretado.. para desarrollar y unir los módulos o partes de un proyecto.
Permite pasar más tiempo con requerimientos del sistema y especificaciones técnicas en lugar de perder tiempo en detalles de programación.

Ventajas:

- Desarrollo rápido de software
- Reutilización de código
- Diseño uniforme
- Portabilidad de aplicaciones

Inconvenientes:

- Dependencia del código respecto al framework (cambiar de framework = reescribir gran parte de la aplicación)
- Recursos del sistema por tener el framework

Ejemplos: .NET para desarrollar aplicaciones Windows (Visual Studio .net para desarrollarlas, motor "Net framework" para ejecutarlas). Spring en Java.

4.3.2.2 Entornos de ejecución

Servicio de máquina virtual que sirve como base software para la ejecución de programas. Puede pertenecer al sistema operativo o instalarse como software independiente que funcionará por debajo de la aplicación. Es un conjunto de utilidades que permiten la ejecución del programa.

Los entornos se encargan:

- Configurar memoria principal disponible en el sistema
- Enlazar archivos del programa con bibliotecas existentes (subprogramas para desarrollar o comunicar componentes que ya existen previamente) y con nuestros subprogramas
- Depurar programas comprobando errores semánticos del lenguaje

El funcionamiento:

- Formado por la máquina virtual y los APIs (bibliotecas de clases estándar para que la aplicación se ejecute). Se distribuyen conjuntamente porque deben ser compatibles entre sí.

El entorno es intermediario entre lenguaje fuente y sistema operativo.
Para desarrollar nuevas aplicaciones necesitaremos un "Entorno de desarrollo"

El entorno en tiempo de ejecución de Java es el JRE (Java Runtime Environment). Se compone de utilidades que permiten la ejecución. Y está formado (como hemos comentado) por la máquina virtual de Java (JMV) que interpreta el código fuente de Java y las bibliotecas de clase estándar del API de Java.

4.6. Pruebas

Se verá en UD-03 así que solo comentar que la realización de pruebas es

- Imprescindible para asegurar validación y verificación del código
- El periodo de prueba debe ser pactado con el cliente
- La prueba final se denomina "Beta Test" y se realiza por el entorno de producción dónde el software va a ser usado por el cliente (en equipos del cliente, con funcionamiento normal de la empresa)

4.7. Documentación

El proyecto debe documentarse en todas las fases del mismo para pasar de una a otra de forma clara y definida. Correcta documentación permite reutilización de parte de los programas en otras aplicaciones, si estos se han desarrollado con diseño modular.

En el desarrollo de software debería elaborarse:

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 codificación de los programas. Las pruebas realizadas.	Descripción de la funcionalidad de la aplicación. Forma de comenzar a ejecutar la aplicación. Ejemplos de uso del programa. 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 técnico en informática (analistas y programadores).	A los usuarios que van a usar la aplicación (clientes).	Al personal informático responsable de la instalación, en colaboración con los usuarios que van a usar la aplicación (clientes).
¿Cuál es su objetivo?	Facilitar un correcto desarrollo, realizar correcciones en los programas y permitir un mantenimiento futuro.	Dar a los usuarios finales toda la información necesaria para utilizar la aplicación.	Dar toda la información necesaria para garantizar que la implantación de la aplicación se realice de forma segura, confiable y precisa.

4.8. Explotación

Fase en que los usuarios finales conocen la aplicación y empiezan a utilizarla. Es la instalación, puesta a punto y funcionamiento en el equipo final del cliente.

Sucede cuando las pruebas demuestran que el software es fiable y sin errores y se han documentado todas las fases.
Algunos autores consideran explotación y mantenimiento como una única etapa.

Instalación: Programas transferidos al computador del usuario, configurados y verificados. Recomendable que clientes estén presentes y explicárselo. (Momento de las Beta Test con cargas normales de trabajo)

Configuración: Asignamos parámetros de funcionamiento normal de la empresa y se prueba que la aplicación es operativa. Podrían también hacerlo los usuarios a veces con la guía de instalación o podría hacerse automáticamente. Si el software es "a medida", es aconsejable que lo haga la empresa que fabrica el software.

Fase de producción normal. La aplicación pasa a manos de los usuarios finales y comienza la explotación del software.

4.9 Mantenimiento

Proceso de control, mejora y optimización del software.

La construcción del software no termina con la entrega. La etapa de mantenimiento es la más larga de todo el ciclo de vida.

Tipos de cambios:

- **Perfectivos**: Mejorar la funcionalidad del software
- **Evolutivos**: Nuevas necesidades del cliente. Necesarias modificaciones, expansiones o eliminaciones.
- **Adaptativos**: Modificaciones, actualizaciones... para nuevas tendencias de mercado, nuevos componentes de hardware, seguridad...
- **Correctivos**: Aplicación tendrá errores en el futuro

5. Entornos de desarrollo

Aunque puede aprenderse a programar usando editor de texto (escritura de código), compilador (detectar errores sintácticos) y depurador (seguir las variables de código al ejecutar) por separado; en el terreno profesional se utiliza un entorno integrado de desarrollo IDE. Este tiene editor de código fuente, compilador y/o intérprete, depurador, interfaz gráfico, automatización de generación de herramientas.

Los primeros entornos nacen en los 70 y se popularizan en los 90. Normalmente para un lenguaje aunque tienden a ser compatibles con varios mediante instalación de plugins (Eclipse, NetBeans, VSCode...)

Al principio no se usaban los IDEs. Se programaba con tarjetas perforadas (tarjeta que almacenaba información que era leído por un lector) los programas estaban escritos con diagramas de flujo (gráfico que plasma entradas y salidas de las variables de un programa)

El primer IDE, Maestro, (principios de los 70) fue usado por BASIC (que abandonó las tarjetas perforadas). Estaba basado en consola de comandos (sistemas operativos con interfaz gráfica no salen hasta los 90) pero era uso parecido a los IDEs actuales.

Debe elegirse el entornos de desarrollo dependiendo del lenguaje de programación que se quiera emplear.

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.

Tipos de entornos de desarrollo libres más relevantes en la actualidad.

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.

Tipos de entornos de desarrollo propietarios más relevantes en la actualidad.

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, Linux, Mac OS X.
JCreator.	Java.	Windows.
Xcode.	C/C++, Java.	Mac OS X.

Las funciones de los IDE son:

- Editor de código: Coloración de la sintaxis
- Autocompletado de código, atributos y métodos de clases
- Identificación de código
- Herramientas de concepción visual para manipular componentes visuales
- Asistentes de gestión y generación de código
- Archivos fuentes en unas carpetas y compilados en otras
- Compilación de proyectos complejos en un solo paso
- Control de versiones
- Cambios de usuarios de forma simultánea
- Generador de documentación
- Detección de errores
- Refactorización de código
- Introducción de tabulaciones y espaciados
- Depuración: Seguimiento de variables, puntos de ruptura, mensajes de error del intérprete
- Funcionalidades a través de gestión de sus módulos y plugins
- Administración de interfaces y configuraciones de usuario

Los entornos de desarrollo tienen en su estructura los siguientes componentes:

- **EDITOR DE TEXTOS**: Resalta y colorea la sintaxis, autocompleta código, lista parámetros de funciones y métodos, inserta automáticamente paréntesis, tabulaciones...
- **COMPILADOR / INTÉRPRETE**: Detección de errores de sintaxis en tiempo real. Compilador traduce de una sola pasada genera archivo ejecutable que puede ser ejecutado sin necesidad del código original, cambios necesitan recompilar; Intérprete ejecuta línea por línea y no genera ningún ejecutable durante el proceso de interpretación, pueden ser más lentos, sus cambios no es necesario recompilar.
- **DEPURADOR**: Botón de ejecución y traza, puntos de ruptura, seguimiento de variables. Posibilidad de depurar en servidores remotos.
- **GENERADOR DE HERRAMIENTAS**: Creación y manipulación de componentes visuales, uso de asistentes y utilidades de gestión y generación de código.
- **INTERFAZ GRÁFICA**: Programar varios lenguajes con el mismo IDE (bibliotecas, plugins...)

UD 02 - INSTALACIÓN Y USO DE ENTORNOS DE DESARROLLO

1. Introducción y ejemplos de IDEs

Los **entornos de desarrollo** son un tipo de software que permite desarrollar aplicaciones facilitando la labor a través de herramientas y tareas predefinidas. Los hay generales, pudiendo abarcar diversos lenguajes y tipos de aplicaciones y también específicos.

Los entornos de desarrollo se pueden clasificar:

- en función del tipo de licencia: software libre o software privado
- en función de la plataforma: multiplataforma o no
- en función del lenguaje: PHP, Java, múltiples...

Comunes DAM / DAW

- **Netbeans**: Entorno multilenguaje más extendido y más integrado con Java.
- **Eclipse**: Multilenguaje. Primero en integrar la SDK de Android. Perdiendo protagonismo porque Oracle ha elegido Netbeans como IDE y Android apuesta por Android Studio.
- **Visual Studio:** Aplicaciones en múltiples lenguajes Node.js, Javascript, C++, PHP, .NET
- **Oracle SQL Developer**: Desarrollo y gestión de BBDD de Oracle. Desarrollar aplicaciones PL/SQL, realizar consultas a BBDD y gestionar con lenguaje SQL.
- **Monodevelop**: Proyecto de código abierto con herramientas basadas en Linux que permiten desarrollar usando tecnología .NET.

Propios de DAM

- **Android Studio**: Por Google para aplicaciones Android
- **Xcode**: Desarrollo para macOS (macOS, iOS, watchOS, tvOS). Es gratuito pero solo se puede instalar en computadoras con macOS.
- **APPinventor**: Aplicaciones sencillas para Android
- **Unity**: Motor de videojuegos y plataforma de desarrollo. La mayoría son versiones de pago; pero existe una "Personal" para desarrollo de videojuegos propios. La mayoría de videojuegos de movil y PC están en Unity.
- **JMonkey**: Motor de videojuegos libre en tres dimensiones. Escrito en Java, tiene su propio IDE.

Propios de DAW

- **Atom**: IDE de código abierto lanzado a través de GitHub. Es OpenSource. Disponible para muchas plataformas. Integración con Node.js
- **PhpStorm**: IDE comercial para desarrollo de aplicaciones en PHP. Integración con frameworks como Symfony. Hay licencia gratuita de estudiantes

Otras herramientas

- **Notepad++**: Editor de código. Ficheros en múltiples lenguajes
- **Sublime Text**: Editor de código ligero. Es de pago pero tiene versión de prueba trial.
- **Adobe Dreamweaver**: Desarrollo de páginas web usando editor WYSIWYG
- **Brackets.io**: Editor de texto open source desarrollado por Adobe. Permite ver en tiempo real en el navegador cómo quedan las modificaciones.

Conceptos: IDE vs Framework

- IDE: Entorno de desarrollo integrado que nos facilita el proceso de desarrollo. Tiene un editor de texto, herramientas de desarrollo automático, un depurador y, si lo necesita el lenguaje, un compilador. También tiene entornos de ejecución para validar el software desarrollado (ej. Android Studio tiene terminal virtual de Android)

- Framework: Entorno de trabajo que proporciona herramientas, librerías y patrón base que facilita el proceso de desarrollo. Estructura de código base a partir del cual se puede desarrollar un código mucho más complejo usando las herramientas proporcionadas. Ej.: Angular, Symfony, Bootstrap.

1.1. Funciones de los IDEs

Ya se ha comentado que un IDE tiene:

- Editor de código fuente
- Compilador y/o intérprete
- Automatización de generación de herramientas
- Depurador

Las funciones de los IDEs son:

- Editor de código: coloración de la sintaxis (construcción válida de sentencias en un lenguaje)
- Autocompletado, de código: atributos y métodos de clases
- Identificación automática de código
- Herramientas de concepción visual para crear y manipular componentes visuales
- Asistentes de gestión y generación de código
- Archivos fuentes en unas carpetas y compilados en otras
- Compilación de proyectos complejos en un solo paso
- Control de versiones: Único almacén de archivos compartido por los desarrolladores con mecanismo de autorecuperación a estados anteriores estables
- Cambios de varios usuarios de forma simultánea
- Generación de documentación integrado
- Detección de errores de sintaxis en tiempo real
- Refactorización de código
- Introducir tabulaciones y espaciados
- Depuración: Seguimiento de variables, puntos de ruptura, mensajes de error
- Aumento de funcionalidades con gestión de módulos y plugins
- Administración de las interfaces de usuario (menús y barras)
- Administración de las configuraciones de usuario

1.2. Estructura de los IDEs

- **Editor de texto**: Resulta y colorea la sintaxis, autocompleta código, lista parámetros de funciones y métodos, inserta automáticamente paréntesis, corchetes, tabulaciones y espaciados
- **Compilador/intérprete**: Detección de errores de sintaxis en tiempo real
- **Depurador**: Botón de ejecución y traza, puntos de ruptura y seguimiento de variables. Posibilidad de depurar en servidores remotos.
- **Generador automático de herramientas**: Para visualización, creación y manipulación de componentes visuales
- **Interfaz gráfica**: Programar en varios lenguajes en u nmismo IDE. Interfaz agradable con innumerables bibliotecas y plugins.

2. Entornos de desarrollo comunes

2.1. Netbeans

- Desarrollado principalmente para Java, aunque con módulos permite desarrollar en otros como PHP
- Desarrollado por Sun MicroSystems y después propiedad de Oracle

- Soporta desarrollos de Java J2EE, EJB, aplicaciones móviles
- Permite instalarle módulos para generación de XML, modelado UML, soporte de PHP, C++...
- Para ser instalado necesita el JDK
- Se tienen parámetros configurables del entorno: Carpetas donde se aloja el proyecto, descripción del proyecto para mejor localización, carpetas de almacenamiento de paquetes fuente y pruebas, opciones de compilación, opciones de empaquetado, opciones de generación de documentación, opciones de combinación de teclas en teclado..... Fuentes, Bibliotecas, Generación de código: Compilación, Empaquetado, Ejecución...
- Actualización: De forma online. Hay un Auto Update Services ya incorporado. En eclipse hay un "Check for updates" para actualización automática. Los IDEs se actualizan para incluir y modificar las funcionalidades del entorno.

Pasos adecuados:

Para generar ejecutables debemos, una vez que esté libre de errores sintácticos: COMPILAR, DEPURAR y EJECUTAR. Así se ejecutan los programas en el propio IDE.

Si quiere generarse un ejecutable: "Clean and Build", el .jar está en la carpeta `dist`

Módulos en Netbeans

- Componente software que contiene clases de Java que pueden interactuar con las API del entorno de desarrollo y el manifest file que es un archivo especial que lo identifica como módulo.
- Los módulos pueden construirse y desarrollarse de forma independiente, lo que posibilita su reutilización y que las aplicaciones se puedan construir a través de la inserción de módulos con finalidades concretas. Una aplicación puede ser extendida con la adición de módulos nuevos que aumenten su funcionalidad.
- Existen multitud de módulos disponibles para todas las versiones de los IDEs más usados.

Pueden añadirse módulos:

- Que Netbeans instala por defecto
- Descargar un módulo de algún sitio web y añadirlo
- Instalarlo online desde el entorno.

Adición offline: Lo usual es añadirlos desde la web oficial de Netbeans (se descarga en formato .nbm, propio de los módulos de Netbeans). Desde nuestro IDE se cargan e instalan esos plugins
Adición online: Se pueden instalar on-line sin salir del IDE y sin tener qu descargarlos previamente.

El módulo puede ser desactivado o desinstalado cuando ya no lo necesitemos.

Herramientas concretas en Netbeans

- Importador de proyectos: Para trabajar en lenguajes como `JBuilder`
- Servidor de aplicaciones Glassfish
- Soporte para Java EE
- NetBeans Swing GUI Builder: Creación de interfaces de usuario
- NetBeans Profiler: Eficiencia de trozo de software
- Editor WSDL: Servicios web basados en XML

- Editor XML Schema Editor. Refirnar aspectos de los documentos XML
- Aseguramiento de seguridad de datos mediante Sun Java System Access Manager
- Soporte beta de UML
- Soporte bidireccional que sincroniza los modelos de desarrollo con los cambios en el código conforme avanzamos por el ciclo de vida de la aplicación.

Funcionalidades de los módulos

- Construcción de código
- Bases de datos

- Depuradores
- Aplicaciones
- Edición
- Documentación de aplicaciones
- Interfaz gráfica de usuario
- Lenguajes de programación y bibliotecas
- Refactorización
- Aplicaciones web
- Prueba

(Las funcionalidades de Netbeans serán más o menos interesabntes según la tarea a realizar y el nivel de usuario)

2.2. Eclipse

Plataforma de desarrollo software que dispone de IDEs para los lenguajes de programación más usados (Java, PHP, C++) y una gran comunidad de desarrolladores que respaldan el proyecto.
Son de código abierto, gratuitos y multiplataforma.

Eclipse ofrece:

- Creación de nuevos proyectos (Requiere de un workspace; Al contrario que en netbeans que son proyectos)
- Permite importación/exportación de proyectos
- Funciona con proyectos Android
- Potencial por los plugins
- Sistema de depuración del proyecto
- Uso para desarrollo de aplicaciones web
- Actualización automática
- Traducción a 47 idiomas (proyecto Babel)
- Generación de Javadoc
- Soporte JUnit

Con la opción genérica (Eclipse Neon) nos preguntará qué lenguaje vamos a usar para instalar el Eclipse IDE que mejor se adapte.

Los módulos de eclipse se instalan con el Marketplace. También pueden buscarse plugins en otros marketplaces como RedHat y Obeo.
Algunos plugins recomendables:

- UML Designer: Modelado de objetos en POO y XML
- Ant visualization: Presentación gráfica de dependencias entre objetos Ant
- Eclipse visual editor: Plataforma para crear clases visuales y gráficas sin Eclipse. Soporta WYSIWYG.

2.3. Visual Studio

IDE desarrollado por Microsoft y disponible para Windows y macOS. Soporta múltiples lenguajes como C++, C#, Java, Visual Basic, .NET, PHP, Python. También hay versión Express gratuita, que luego en 2017 se ha llamado Visual Studio Community.
Desarrollar aplicaciones en Android, iOS, macOS, Windows y Web. Permite navegar por el código, escribirlo y modificarlo fácilmente.

Visual Studio Code es un IDE que permite escribir código de forma sencilla y en múltiples lenguajes. Se descarga desde la página oficial de visualstudio.

2.4. SQLDeveloper

Gestión de BBDD Oracle. Desarrollo de aplicaciones en PL/SQL. Solo disponible para Windows.

3. Entornos de desarrollo multiplataforma

3.1. Android Studio

Desarrollado por Google

Tiene:

- Edición de código inteligente
- Integración GitHub
- Desarrollo multidispositivo
- Dispositivos virtuales en todos los tamaños y orientaciones
- Generación de APK basado en Gradle
- Requisitos: 2 GB Ram mínimo, 4 GB recomendado, 400 GB espacio en disco duro, 1 GB Android SDK, 1200x800 de resolución, JDK...
  Instalar Android Studio, SDK Build Tools, Google repository, Google API.

Quizás al principio no detecte bien el SDK y haya que indicárselo.

3.2. XCode

Poco que decir.
Android copa cerca del 88% del mercado de dispositivos móviles. Android Studio es el IDE por excelencia en desarrollo multiplataforma.
Desarrollar aplicaciones para iOS es mucho más rentable.

3.3. Unity

Se puede descargar versión gratuita (pero con limitación no comercial).
Elegir al crear el nuevo proyecto si lo queremos en 2D o 3D.

3.4. JMonkey Engine

Motor para desarrollo de juegos 3D basado en Java. Compatible con LWJGL (Lightweight Java Game Library) y tiene comunidad de desarrolladores que lo respalda.
Tiene su propio IDE para trabajar con JMonkeyEngine. Se descarga de manera conjunta con el motor de videojuegos y las librerías de desarrollo.

4. Entornos de desarrollo web

Inicialmente se basaba en un bloc de notas que, con código basado en etiquetas, generaba archivo interpretado que los navegadores podían entender.

4.1. Aptana Studio

- Basado en Eclipse
- Funciona en Windows, macOS, GNU/Linux
- Lenguajes PHP, Python, Ruby, CSS, HTML, Javascript.
- Soporte librerías de Javascript (jQuery, Ext JS, DOJO, YUI)

4.2. PHP Storm

- IDE comercial para PHP
- Soporta Symfony, Drupal, Wordpress, Zend Framework, Yii, CakePHP...
- Uno de los más completos para trabajar con PHP (149 $ al año pero con licencia de estudiante gratis)
- Soporta tecnologías frontend (HTML5, CSS, Typescript, Javascript) y permite visualizar cambios en caliente, depurarlo, hacer test de pruebas, reformatear...

5. Herramientas CASE para desarrollo, prueba y documentación

5.1. De los navegadores

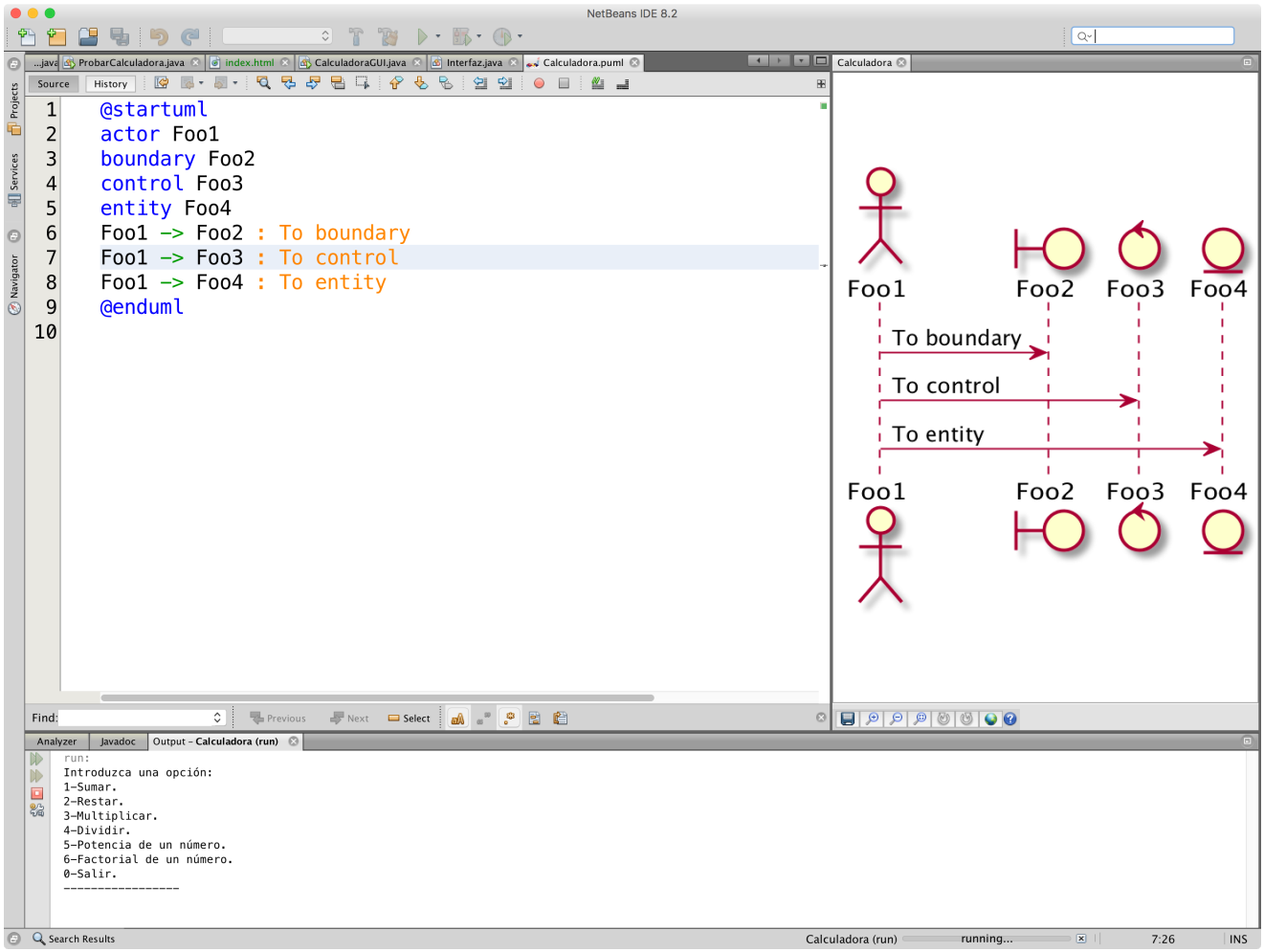
Firefox, Chrome, Safari, Edge tienen herramientas para modiifcarlos desarrollos de forma dinámica:

- Habilitar / Deshabilitar lenguajes
- Modificar CSS, HTML en tiempo real...

Safari: Inspector web "Mostrar inspector web"
Edge: F12, Menú de herramientas para desarrolladores
Firefox: Extensión Firebug, pero ahora tiene utilidad nativa de Desarrollador web
Chrome: Tiene también herramientas responsive Responsive Web Design Tester

5.2. Plugin UML para Netbeans

PlantUML se puede añadir en Netbeans.
Usando sintaxis se puede generar diagramas UML. Por ejemplo... un diagrama de comportamiento



5.3. Documentación de código fuente

JavaDoc

En Netbeans Execute -> Generate JavaDoc

UD 03 - DISEÑO Y REALIZACIÓN DE PRUEBAS

1. Definición de prueba. Cuándo hacer pruebas. La estrategia de pruebas

Prueba: Proceso que permite verificar y revelar la calidad de un producto software. Permite identificar los fallos de implementación, calidad o usabilidad de un programa.

Es necesario realizar pruebas en las diferentes etapas (diseño, implementación y una vez que la aplicación ha sido desarrollada) que permitan verificar que el software que se crea es correcto y cumple las especificaciones pedidas.

Pueden aparecer errores humanos como:

- incorrecta especificación de los objetivos
- errores del proceso de diseño
- errores de la fase de desarrollo.

Con las pruebas se realizan tareas de **VERIFICACION** y de **VALIDACION** :

- Verificación: Si sistema o sus partes cumplen las condiciones impuestas ¿Se construye correctamente?
- Validación: Si el sistema o sus partes satisfacen los requisitos ¿Es lo que se pide?

Debe implementarse una **ESTRATEGIA DE PRUEBAS** que, seguirían el modelo **Modelo en Espiral** (Modelo de ciclo de vida del software con actividades en espiral, cada iteración es un conjunto de actividades) :

- Empiezan con las **pruebas unitarias** (Se analiza el código implementado)
- Siguen con las **pruebas de integración** (Diseño y construcción de la arquitectura de software)
- Prosiguen las **pruebas de validación** (Se comprueba que cumpla el análisis de requisitos)
- Finalmente, **prueba de sistema** (Funcionamiento total del software y otros elementos)

2. Tipos de prueba y de casos de prueba

Se puede hablar de dos enfoques a la hora de pruebas y de casos de prueba:

- **Enfoque o Pruebas funcionales (De caja negra - Black Box Testing)**: Es la principal herramienta de VALIDACIÓN. Se analiza, para una concreta qué salida se espera recibir. Se utiliza la interfaz externa. No se indaga en la implementación de la misma. No es necesario conocer la estructura, ni el funcionamiento. **No se verifica el proceso, solo los resultados** . Se aplica con los valores límite y las clases de equivalencia.
- **Enfoque o Pruebas estructurales (De caja blanca - White Box Testing)**: Se prueba la aplicación desde dentro, usando su lógica de aplicación, sus distintos caminos de ejecución. Es necesario un conocimiento específico del código para poder analizar los resultados. **Se deberían probar todos los caminos que puede seguir la ejecución del programa** . Se aplica con el cubrimiento.
- **Pruebas aleatorias**: Modelos (muchas veces estadísticos) que representen las posibles entradas al programa para crear los casos de prueba

Pruebas de regresión: Probar un producto de software después de que se le hayan realizado modificaciones para garantizar que no se hayan introducido nuevos errores como resultado de los cambios. (errores, carencias de funcionalidad, divergencias funcionales con respecto al comportamiento esperado del software).

Detallamos un poco más los tipos de pruebas:

Tipo de prueba	Explicación
Pruebas de unidad	Funcionamiento de un módulo de código
Pruebas de carga	Observar el comportamiento bajo una cantidad de peticiones esperada. Podría mostrar los tiempos de respuesta de todas las transacciones importantes de la aplicación. Si se monitoriza la BBDD, el servidor... podría verse el cuello de botella. Es la prueba de rendimiento más sencilla.
Pruebas de estrés	Sirve para saber si la aplicación rendirá en casos en los que la carga real de peticiones supere a las esperadas. Se dobla el número de peticiones y se ejecuta una prueba de carga hasta que se rompe. Se usa para determinar la solidez de la aplicación en momentos de carga extrema.
Pruebas de estabilidad	Determinar si puede soportarse una carga esperada de forma continua. Usada para ver si hay fugas de memoria en la aplicación.
Pruebas de picos	Observar el comportamiento del sistema variando de forma drástica (bajadas y subidas) el número de usuarios.

2.1. El input de las pruebas funcionales (¿qué hace?)

Tendríamos varias opciones de inputs según la prueba que se quiera llevar a cabo:

- **Particiones equivalentes**: En ellas se considera el menor número posible de casos de pruebas, consiguiendo que cada caso abarque el mayor número posible de entradas diferentes. El objetivo es crear un conjunto de **clases de equivalencia** donde la prueba de un valor representativo de la misma, sea extrapolable al que se conseguiría probando cualquier valor de la clase.

- **Análisis de valores límite**: Como entrada se introducen los que están en el límite de las clases de equivalencia.
- **Pruebas aleatorias**: Se usan generadores aleatorios de entradas para crear un volumen de casos al azar con los que alimentar la aplicación. Suelen usarse en aplicaciones no interactivas (es más difícil generarlas en este tipo de entornos)

2.2. Pruebas estructurales (¿cómo lo hace?)

Verificar la **estructura interna de cada componente de la aplicación**, **independientemente de la funcionalidad establecida** para el mismo.

No quiere comprobar la corrección de los resultados producidos, sino comprobar que:

- Se ejecutan todas las instrucciones del programa
- No hay código no usado
- Los caminos lógicos del programa se recorren
- ...

Para estas pruebas se realizan **CRITERIOS DE COBERTURA LÓGICA**, cuyo cumplimiento determina la mayor o menor seguridad en la detección de errores. Estos criterios pueden ser:

- **Cobertura de sentencias**: Generar casos de prueba suficientes para que cada instrucción del programa se ejecute al menos una vez.
- **Cobertura de decisiones**: Generar casos de prueba suficientes 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**: Generar casos de prueba suficientes para que cada elemento de una condición se evalúe al menos una vez a cierto y otra a falso.
- **Cobertura de condiciones y decisiones**: Cumplir simultáneamente la cobertura de decisiones y la cobertura de condiciones.
- **Cobertura de caminos**: Criterio más importante. Se debe ejecutar al menos una vez cada secuencia de sentencias encadenadas, desde la sentencia inicial a la sentencia final. Como el número de caminos puede ser muy grande, se reduce el número a lo que se conoce como *camino prueba*.
- **Cobertura del camino de prueba**: Podría haber dos variantes:
  - Una indica que cada bucle debe ejecutarse solo una vez (hacerlo más veces no aumentaría la efectividad de la prueba)
  - Otra indica que se pruebe cada bucle tres veces (la primera sin entrar en su interior, la otra ejecutándolo una vez, la otra ejecutándolo dos veces)

2.3. Pruebas de regresión (¿se ha roto algo después de cambiar lo que fallaba?)

Las modificaciones de código pueden generar errores colaterales que no existían antes.
Deben repetirse las pruebas realizadas con anterioridad para comprobar que no se introduce comportamiento indeseado o errores adicionales.

Deben **llevarse a cabo cada vez que se hace un cambio en el sistema** (ya sean `fixes` o mejoras de la funcionalidad)

Pueden realizarse manualmente, volviendo a realizar un subconjunto de todos los casos de prueba o utilizando herramientas automáticas.

Las pruebas de regresión contiene tres clases diferentes de clases de prueba:

- Muestra representativa de pruebas que ejecute 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 de software que han cambiado.

Las pruebas de regresión se deben diseñar para incluir solo aquellas pruebas que traten una o más clases de errores en cada una de las funciones del programa. **No es práctico ni eficiente volver a ejecutar cada prueba de cada función del programa después de un cambio.**

3. Tipos de errores en el proceso de desarrollo. Las herramientas de depuración ante los errores lógicos

En el proceso de desarrollo se producen:

- Errores de compilación: Olvidar un símbolo, referenciar a variable inexistente, utilizar sentencia incorrecta. El IDE aporta información de dónde se produce y el programa no puede compilarse hasta no ser corregido el erroores.
- Errores lógicos o bugs: Provoca que el programa devuelva resultados erróneos, termine antes de tiempo, se ejecute infinitamente... En estos casos debe recurrirse a un depurador.

Los IDE (Integrated Development Environment) suministran herramientas de depuración para verificar el código generado y realizar pruebas estructurales y funcionales.

El depurador (debugger) permite supervisar la ejecución de los programas para localizar y eliminar los errores lógicos. Con el depurador se puede suspender la ejecución del 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..

- **Puntos de ruptura (Breakpoints)**: Marcadores que pueden establecerse en cualquier línea de código ejecutable (no admite comentarios o líneas en blanco). El programa se ejecuta hasta la línea marcada, momento en el que se pueden examinar las variables, iniciar depuración paso a

paso comprobando el camino que toma el programa... Finalmente se puede abortar el programa o continuar la ejecución normal del mismo.
- **Examinadores de variables**: Permite ver el valor que van tomando las variables en el avance paso a paso.

Tipos de ejecución del depurador

- **Avance paso a paso**: Se ejecuta línea por línea
- **Paso a paso por procedimientos**: Se introduce el parámetro que se quiere a un método o función y nos devuelve el resultado. (Hemos comprobado que el método funciona correctamente y solo nos interesa el valor que devuelve)
- **Ejecución hasta instrucción**: El depurador se ejecuta hasta la instrucción en la que se encuentra el cursor y, en ese punto, hacer depuración paso a paso o por procedimientos.
- **Ejecución hasta el final** sin detenerse en instrucciones intermedias.

Los modos de ejecución deben ajustarse a las necesidades de depuración que tengamos en cada momento. Lo importante es que se puedan examinar todas las partes que se consideren necesarias de forma rápida, sencilla y clara.

Cómo debuggear en el prehistórico IDE Netbeans....

En NetBeans hay un panel llamado ventana de inspección en la que se pueden añadir todas las variables de las que se tenga interés en inspeccionar su valor.

En el menú de depuración de Netbeans se pueden seleccionar los modos de ejecución especificados antes y algunos más.

¿Cómo debuggear?
Pinchamos en **Depurar** .
Vamos analizando el código "Paso a paso", "Ejecutar hasta el cursor".
Se puede detener la ejecución, continuarla o ejecutar el programa hasta el final.
Se puede elegir "Entrar en el siguiente método" o seguir con la ejecución paso a paso con F7.
Si no necesita depuración podemos "Continuar ejecución" pulsando F8.
Si no nos interesa seguir con la depuración podemos "Finalizar sesión del depurador"
Si queremos pausar la depuración "Pausa"
Si queremos continuar la ejecución normal o seguir hasta el siguiente punto de ruptura "Continuar"

Línea roja indica punto de ruptura. Línea verde indica por donde está pasando el depurador.

El depurador también tiene:

- Consulta de la pila (stack) para ver qué funciones se llaman
- Inserción de puntos de ruptura
- Evaluar expresión

Una vez depurado, si consideramos que no hace lo que debe lo cambiamos.
Si una vez depurado hace lo que debe, puede pasarse a probar diseñando casos de prueba en JUnit y aplicándolos.

4. La validación

Se intenta descubrir errores desde el punto de vista de los requisitos (Comportamiento y casos de uso que se esperan que cumpla el software que se está diseñando). Interviene de forma decisiva el cliente.

Se realizan una serie de pruebas de caja negra que demuestran la conformidad con los requisitos.

Plan de prueba: Clases de pruebas que se han de llevar a cabo
Procedimiento de prueba: Define los casos de pruebas específicos para descubrir errores según los requisitos

Se busca que se satisfacen todos los requisitos funcionales, que se alcanzan todos los requisitos de rendimiento, que las documentaciones son correctas e inteligibles, que se alcanzan requisitos de portabilidad, compatibilidad, recuperación de errores, facilidad de mantenimiento...

Finalmente;

- Puede ser que las características están de acuerdo con las especificaciones o puede ser que se descubra una desviación se elabore una lista de deficiencias.
- Las desviaciones o errores descubiertos en esta fase del proyecto, raramente se pueden corregir antes de la terminación planificada.

5. Pruebas de código

5.1. Cubrimiento

Comprobar que todas las funciones, sentencias, decisiones y condiciones se ejecutan.

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

Cubrimiento de la función: Durante la ejecución del programa la función debe ser llamada al menos una vez.

- Cubrimiento de sentencias: `prueba(1,1)` lo cumpliría
- Cubrimiento de decisión: `prueba(1,1)` Y `prueba(0,1)` lo cumplirían
- Cubrimiento de condición: `prueba(1,1)` , `prueba(1,0)`, `prueba(0,0)`.

Hay otros criterios para comprobar el cubrimiento:

- Secuencia lineal de código y salto
- JJ-Path Cubrimiento (Cantidad de rutas de control ejecutadas en las pruebas)
- Cubrimiento de entrada y salida (Evaluar si se han probado todas las combinaciones posibles de valores de entrada y de salida)
  Herramientas comerciales como Clover o Jacoco.

5.2. Valores límite

Los casos de prueba con mayor probabilidad de éxito son los de los valores límite.
Es una técnica complementaria a las particiones equivalentes pero se generan no un conjunto de valores sino unos pocos en el límite del rango de valores aceptado por el componente a probar.

```
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;
}
```

Los valores límite son diferentes porque un método admite como parámetro un `double` y el otro un `int`. Como el parámetro de entrada tiene que ser mayor que 5. En uno valores límite 4,99 y 5,01. En otro valores límite 4 y 6.

Si el parámetro de entrada está entre -4 y+4, los valores límite son -5, -4, -3, 3, 4 y 5

Si el parámetro de entrada es entre 5 y 10 siendo valor `single` (precisión simple, `float`), los valores límite pueden ser 4,99; 5; 5,01; 9,99; 10; 10,01)

5.3. Clases de equivalencia

Cada caso de prueba quiere cubrir el mayor número de entradas posibles.

El dominio de valores de entrada se divide en número finito de clases de equivalencia.
La prueba de un valor representativo de cada clase, permite suponer que el resultado será el mismo que con cualquier otro valor de equivalencia.

Ejemplos:

- Cuando un parámetro de entrada debe estar en un determinado rango hay tres clases de equivalencia: Por debajo, en y por encima.
- Si una entrada requiere un valor en un conjunto hay dos clases de equivalencia: En el conjunto o fuera de él.
- Si una entrada es booleana hay dos clases de equivalencia: Sí y no.

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

Clases de equivalencia:
1- Por debajo x <= 0
2- En x > 0 y x < 100
3- Encima x >= 100

Casos de prueba
1 - Por debajo x = 0
2.- En x = 50
3.- Por encima x = 100

6. Normas de calidad

Estándar usado ahora: ISO/IEC 29119 (O hace mil años, cuando se hizo el temario)

Pretende:

- Unificar en una única norma todos los estándares dando vocabulario, procesos, documentación y técnicas para cubrir todo el ciclo de vida de 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.
- Corregir el hecho de que estándares anteriores no cubren facetas de la fase de pruebas como la organización del proceso y gestión de las pruebas y tienen pocas pruebas funcionales y no-funcionales, etc.

Contenido ISO/EIC 29119:

- 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) .

ChatGPT dice que...

Algunas organizaciones y profesionales de pruebas de software pueden optar por seguir las pautas y estándares establecidos por la ISO/IEC 29119, mientras que otros pueden utilizar metodologías y prácticas diferentes, como las propuestas por el ISTQB (International Software Testing Qualifications Board) u otras.

ISTQB: Es una organización que proporciona un esquema de certificación profesional en el campo de las pruebas de software.

Estándares usados anteriormente

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
-> IEEE estándar 829 Documentación de la prueba de software
-> IEEE estándar 1008 Pruebas de unidad

Estándar *ISO/EIC 12207, 15289*

7. Pruebas unitarias

Las **pruebas unitarias** o **pruebas de unidad** buscan probar el funcionamiento de un módulo de código por separado.
En las pruebas de integración se asegurará el funcionamiento del sistema.

Unidad: Parte de la aplicación más pequeña que se puede probar (Función, procedimiento método).

Las pruebas unitarias deben ser:

- Automatizables (sin intervención manual)
- Completas (elevado cubrimiento)
- Repetibles (ejecutar más de una vez)
- Independientes entre sí
- Profesionales (con profesionalidad, documentación, ..)

Las pruebas unitarias tienen las siguientes ventajas:

- **Fomentan el cambio**: Facilitan que el programador cambie el código para mejorar su estructura al poder hacer pruebas sobre los cambios y asegurarse de que no se han introducido errores
- **Simplifica la integración**: Fase de integración con grado alto de seguridad de que el código funciona correctamente
- **Documenta el código**: Las propias pruebas actúan de documentación del código
- **Separación entre interfaz e implementación**: La interacción entre los casos de prueba y las unidades probadas son las interfaces de estas últimas, se puede cambiar cualquiera de los dos sin afectar al otyro
- **Errores más acotados y son fáciles de localizar** gracias a las pruebas unitarias.

7.1. Herramientas para Java

Jtiger

- Pruebas unitarias Java 5
- Código abierto
- Exportar informes en HTML, XML, Texto plano
- Casos de prueba de JUnit mediante plugin.
- Aserciones para comprobar el cumplimiento del método
- Metadatos (datos que describen otros datos) de los casos de prueba se especifican con anotaciones
- Tarea de Ant para automatizar pruebas
- Documentación completa
- El framework incluye pruebas sobre sí mismo.

TestNG

- Inspirado en JUnit y NUnit
- Diseñado para cubrir todo tipo de pruebas (unitarias, funcionales, integración)
- Anotaciones de Java 5 (WTF... una novedad, claro)
- Compatible con JUnit
- Soportado por Eclipse, Netbeans, IntelliJ
- Las pruebas pueden invocarse desde línea de comandos con tarea de ANT o fichero XML
- Métodos de prueba se organizan en grupos (método puede pertenecer a uno o varios grupos)

JUnit

- 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) de pruebas unitarias creado por Gamma y Beck
- Código abierto
- Implementación de la arquitectura xUnit
- Muy documentado
- Es el que se usa de verdad en Java (Estándar)
- Usa desde JUnit 4 las anotaciones JDK 1.5 (qué novedad....)
- Se pueden crear informes en HTML
- Organización de pruebas en Suites de pruebas

7.2. Herramientas para otros lenguajes

(Basados en xUnit)

CppUnit

- Pruebas unitarias para C++

Nunit

- Framework para .NET

SimpleTest, PHP Unit

- PHP

FoxUnit

- Microsoft visual FoxPro

MOQ

- Para mockeo

8. Automatización de la prueba

Para IDEs como NetBeans, Eclipse, IntelliJ tenemos JUnit.
Permite diseñar clases de prueba para cada clase diseñada en la aplicación.
Se establecen los métodos que se quieren probar y para ello se diseñan casos de prueba.
JUnit presenta un informe con los resultados de la prueba. En función de ellos se debe (o no) modificar el código.

`assertTrue()`: Evalúa expresión booleana. Pasa la prueba si es cierta.
`assertFalse()`: Evalúa expresión booleana. Pasa la prueba si es falsa.
`assertNull()`: Verifica que sea nulo.
`assertNotNull()`: Verifica que no sea nulo.
`assertSame()`: Asegura que los objetos de las dos referencias tienen misma dirección de memoria.
`assertNotSame()`: Asegura que los objetos de las dos referencias NO tienen misma dirección de memoria.
`assertEquals()`: Asegura que son iguales en contenidos (primitivos, objetos con `equals()`
`assertThrows()`: Éxito si se lanza una excepción
`assertAll()`: Agrupar conjunto de asserts
`fails()`: Provoca que la prueba falle inmediatamente.

Asunciones
`assumeTrue(suposicion, mensaje)`: Continua ejecutando la prueba soplo si la condición es verdadera. Si es falsa se considera omitida (no se ejecutan las aserciones siguientes)
`assumeFalse(suposicion, mensaje)`
`assumingThat(suposicion, ejecutable)`

Anotaciones...
`@DisplayName("")` Da un nombre al test (Clase o método)
`@Disabled("")` Deshabilita el test y da motivos
`@BeforeEach`: Lo ejecuta antes de cada
`@AfterEach`: Lo ejecuta después de cada
`@BeforeAll`: Lo ejecuta antes de correr el primero de los test
`@AfterAll`: Lo ejecuta al final de los test
`@EnabledOnOs("")`: Solo activa el test en un SO (o sistemas operativos) concretos
`@EnabledOnJre`: Solo en una versión de JRE indicada
`@EnabledIfSystemProperty(named:"", matches:"")` En función de la propiedad
`@EnabledIfenvironmentProperty(named:"", matches:"")` En función de la variable de entorno

En el viejo Netbeans...

Tengamos una clase `Car` que tiene un constructor `Car(String name, double prize, int stock)`

Queremos testear:

- El método `buyCars(int cars)` que sumará estos coches al stock (y si son negativos o 0 arrojará excepción)
- El método `sellCars(int cars)` que restará estos coches del stock y que si son negativos o 0 arrojará excepción y que si el stock es menor que estos también arrojará excepción.

VALORES CORRECTOS

VALORES NO VÁLIDOS

VALORES LÍMITE. En nuestro caso el valor límite es 0.

File -> New File -> Test for Existing Class (seleccionando la clase).

```
@Test
public void testBuyCarsValid() throws Exception {
        try {
                Car car = new ("Patata", 12000, 300);
                car.buyCars(100);
                assertTrue(car.getStock()==400);
                //assertEquals(400, car.getStock);  // Tercer parámetro de offset
        } catch (Exception e) {
                fail("Excepción no esperada"+e);
        }
}

@Test
public void testBuyCarsNegativeThrowException() throws Exception {
        try {
                Car car = new ("Patata", 12000, 300);
                Exception exception = assertThrow(Exception.class, () -> {car.buyCars(-100)};
                assertEquals("Intento de comprar un número negativo de coches",
exception.getMessage());
        } catch (Exception e) {
                fail("Excepción no esperada"+e);
        }
}

// Comprar 0
// Vender válido
// Vender 0
// Vender negativo
// Vender menor que el stock
```

Suite de tests

Es un contenedor que agrupa un conjunto de pruebas unitarias relacionadas para que se ejecuten juntas. Útil para proyectos grandes con muchas pruebas. La Suite en sí no debería contener ningún test, solo agrupa las clases de prueba.

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({
    TestClass1.class,
    TestClass2.class,
    TestClass3.class
})
public class TestSuite {
    // Esta clase no necesita contener ningún método de prueba, solo sirve para agrupar
pruebas
}
```

Pueden incluirse y excluirse clases con `@SelectClasses`, `@SelectPackages`, `@IncludePackages`, `@ExcludePackages`, `@IncludeClassNamePatters`, `@ExcludeClassNamePatters`, `@IncludeTags`, `@ExcludeTags`, `@SelectMethod`..

TDD

Se llama Test Driven Development (TDD) a la metodología de desarrollo de software que consiste en escribir primero los test unitarios y después el código.
Pasos:

- Escribir prueba para el requisito
- Comprobar si la prueba falla
- Escribir el código que haga pasar la prueba
- Correr la prueba y comprobar qué pasa
- Refactorizar si es necesario

9. Documentación de la prueba

Metodologías actuales como `Métrica v.3` (Metodología de Planificación, Desarrollo y Mantenimiento de Sistemas de Información. Se puede usar libremente solo citando la propiedad intelectual que es el Ministerio de Presidencia) proponen que la documentación de fase de pruebas se base en estándares ANSI / IEEE de verificación y validación de software.

Se busca describir un conjunto de documentos para las pruebas de software (Facilita un marco común).

Documentos a generar (Proceso de Análisis del Sistema):

- Plan de pruebas: Planificación general
- Especificación del diseño de pruebas: Ampliación y detalle del plan de pruebas
- Especificación de un caso de prueba: A partir de la especificación del diseño de pruebas
- Especificación de un procedimiento de prueba: Detallar el modo en que serán ejecutados cada uno de los casos de prueba
- Registro de pruebas: Registro de sucesos durante las pruebas
- Informe de incidente de pruebas: Para cada incidente y defecto detectado. solicitud de mejoras, etc.
- Informe sumario de pruebas: Resumen de las actividades de prueba vinculadas a una o más especificaciones de diseño de pruebas.

UD 04 - OPTIMIZACIÓN Y DOCUMENTACIÓN

1. Refactorización

Refactorización: Cambio hecho en la estructura interna del software para hacerlo más fácil de entender y modificar sin que se altere su comportamiento externo (funcionalidad). Suelen ser cambios pequeños en el software que mejoran su mantenimiento (Ej.: "Extraer método", "Encapsular campos")

Refactorizar: Reestructurar el software aplicando refactorizaciones sin cambiar su comportamiento

La refactorización:

- Busca mejorar la estructura interna del código; limpiarlo sin introducir errores.
- Facilita que el software sea más fácil de entender y de modificar
- Que el mantenimiento sea más sencillo
- Que se encuentren errores
- Que el programa sea más rápido
- Utiliza una serie de patrones, de aplicación sobre el código fuente.

No es lo mismo que la optimización (que persigue una mejora del rendimiento aunque podría llegar a hacer el código más difícil de entender)

1.1. Problemas de la refactorización

La refactorización presenta problemas en:

- Las bases de datos: Tienen dificultades para ser modificadas debido a sus interdependencias y a conllevar modificación del esquema y migración de datos (muy costosa).
- Interfaces. Al modificar la estructura interna no se cambia comportamiento ni la interfaz. Pero si renombramos un método hay que modificar todos los que referencia a él lo cuál es un problema si la interfaz es pública. La solución es mantener las dos interfaces (nueva y vieja)
- Cambios en diseño difíciles de refactorizar como los debidos a errores de diseño o los que son de vital importancia en la aplicación.
- Si un código no funciona, no se refactoriza; se reescribe.

1.2. Patrones

- Renombrado: Cambiar nombre paquete, clase, método, atributo por otro más significativo
- Sustituir bloques de código por un método: Así cada vez que se quiera acceder al bloque de código solo hay que llamar a ese método
- Campos encapsulados: Mediante getters y setters
- Mover la clase: De un paquete a otro, de un proyecto a otro... Evitar duplicar código. Deben actualizarse las referencias a la clase.
- Borrado seguro: Cuando ya no sea necesario el elemento, que se hayan borrado todas las referencias a él en el proyecto.
- Cambiar los parámetros del proyecto: Añadir nuevos parámetros a un método, cambiar los modificadores de acceso.
- Extraer la interfaz
- Mover del interior a otro nivel: Se mueve una clase interna a un nivel superior en la jerarquía.

1.3. Analizadores de código

El analizador estático de código recibe directamente el código fuente y busca evaluarlo sin llegar a ejecutarlo. Intenta averiguar la funcionalidad del mismo y lo procesa dando sugerencias o posibles mejores.

- Tienen analizadores léxicos y sintácticos que procesan el código fuente y un conjunto de reglas que se deben aplicar sobre las estructuras.
- Las funciones de los analizadores son encontrar partes del código que puedan reducir el rendimiento, provocar errores en el software, tener excesiva complejidad, complicar el flujo de datos, crear problemas de seguridad.
- El análisis puede ser automático o manual. El automático lo realizan programas como el **Findbugs** de Netbeans.
- Analizadores como **EasyPMD, PMD** (nadie usa esa mierda, hoy en día SonarLint y SonarQube por favor) que detectan patrones que pueden ser posibles errores en tiempo de ejecución, unreacheable code, código susceptible de ser optimizado, expresiones lógicas que pueden ser simplificadas, malos usos del lenguaje...
- CPD: Forma parte del PMD. Busca código duplicado.

1.3.1. Uso

Los analizadores de código estático se suelen integrar en los Entornos de Desarrollo (muchas veces como plugins). En Netbeans por ejemplo podemos instalar EasyPMD.

En el caso del prehistórico Netbeans y el analizador PMD...

Como Herramientas > Ejecutar PMD se puede ejecutar el análisis y obtener el informe. Después el informe permite navegar por las clases y, en la línea que no cumple, se verá una marca PMD con un tooltip con la descripción del error.

En la configuración de los analizadores de código es posible definir las propias reglas sumándolas a las que el analizador propone por defecto.

Herramientas > Opciones > Varios > easyPMD.

- Con Enable Scan se habilita el escaneado automático del proyecto a intervalos regulares.
- En Rulestet (Manage Rules) pueden verse las existentes y activar-desactivar, añadir (Add standard) y personalizar (Add custom)
- En Reporting se puede establecer qué información visualizar (por cada regla puede verse descripción del problema y ejemplo de aplicación). Con las propias reglas puede usarse "Manage Rulesets" e importar ficheros XML o JAR con reglas.

1..4 Refactorización y pruebas

Refactorización y pruebas están fuertemente relacionadas en el **Test Driven Development**.

- Agilizar ciclo de escritura de código y realizar pruebas de calidad de los módulos
- Evitar conflictos y gasto excesivo entre developers y SQA (ideas y venidas del código...), en este caso el programador realiza las pruebas de unidad en su propio código y las implementa antes de escribir el código a ser probado.

Sería algo así

```
- Lee requerimiento
- Plantea las pruebas que va a tener que pasar el código a elaborar para ser
correcto
- Programa las pruebas
- Comienza a implementar la unidad de código para que pase esas pruebas
- Elabora pequeñas versiones que puedan ser compiladas y pasen por algunas de las
pruebas
- En cada cambio y recompilación reejecuta las pruebas de unidad tratando que cada
vez pase más hasta que no falle ninguna
```

La refactorización siguiendo TDD: Se refactoriza el código tan pronto como pasa las pruebas y hacerlo más claro. Podrían cometerse errores que provocarían que la unidad falle las pruebas. En reescrituras

sintácticas no es necesario correr el riesgo, las decisiones las toma un humano pero los detalles pueden quedar a cargo de programa que lo trate automáticamente.

1.5. Herramientas de ayuda a la refactorización

Herramientas de ayuda de netebans

- **Renombrar**: Actualiza todo el proyecto
- **Encapsular campos**: Genera getters y setters automáticamente y opcionalmente actualiza las referencias al código para usar esos getters y setters.
- **Cambiar parámetros del método**: Añadir o eliminar parámetros y cambiar el tipo
- **Eliminación segura**: Comprueba si hay referencias a elemento y si no la hay, lo borra. Si se borra con referencias, se advierte de que habrá errores.

(Eliminar parámetros del método, según el test, no seria un mecanismo de refactorizar)

2. Control de versiones

Es vital en el desarrollo un sistema de control de versiones porque este:

- Facilita al equipo de desarrollo su labor permitiendo que varios trabajen en mismo proyecto / mismos archivos
- Sitio central en el que almacenar el código fuente
- Historial de cambios realizados
- Volver a versiones estables previas del código
- Sobre ellos se construyen técnicas más sofisticadas como la Integración Continua

Versión: Forma particular de un elemento en un instante dado.
Revisión: Evolución del elemento en el tiempo en el tiempo.

Los IDEs proporcionan herramientas de control de versiones y facilitan el desarrollo en equipo de las aplicaciones.

En un instante dado pueden coexistir varias versiones y hay que disponer de métodos para designar las versiones de forma sistemática u organizada.

Podemos mencionar CSV y Subversión (O al menos eso diría alguien que vive en el año 2000...). Subversion sería el sucesor natural de CVS.

2.1. Estructura de las herramientas de control de versiones CSV (Concurrent Version System)

Sistema de mantenimiento de código fuente (grupos de archivos en general) para desarrolladores que trabajan y modifican concurrentemente ficheros organizados en proyectos.

- Arquitectura cliente-servidor: Servidor guarda versión actual del proyecto y su historia. Clientes conectan al servidor para sacar copia completa del proyecto trabajar en esa copia e ingresar sus cambios.
- Servidor y cliente se conectan por internet aunque podrían estar en la misma máquina. Servidor suele tener sistema operativo UNIX.
- Se permitiría el acceso de lectura anónimo (sin contraseña)

Componentes:

- **Repositorio**: Lugar de almacenamiento de los datos de los proyectos. Suele ser un directorio.

Centraliza los componentes, incluyendo versiones. Se ahorra espacio de almacenamiento (no se guarda por duplicado los elementos comunes a varias versiones). Se facilita el almacenaje de información de la evolución del sistema. Almacena los datos pero también información sobre versiones, temporización. Al usar sistema de control de versiones trabajamos de forma local,

sincronizándonos con el repositorio. Haciendo los cambios en nuestra copia local, se acomete el cambio en el repositorio.

- **Módulo**: Directorio específico del repositorio. Parte del proyecto o proyecto por completo.
- **Revisión**: Cada versión parcial o cambios en archivos o repositorio completo. La evolución del sistema se mide en revisiones.
- **Etiqueta**: Información textual que se añade a un conjunto de archivos o módulo completo
- **Rama**: Revisiones paralelas de un módulo para efectuar cambios sin tocar evolución principal. Pruebas, mantener cambios en versiones antiguas.

Órdenes:

- `checkout`: Copia del trabajo para trabajar en 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

Repositorio CSV

Para conectar con un repositorio en Netbeans -> Team -> CVS -> Checkout
Se introduce localización del repositorio (CVSROOT). Soporta diferentes según sea local o remoto y el método usado para conectarse a él.

Métodos

- `pserver`: contraseña de servidor remoto
- `ext`: acceso usando Remote Shell (RSH) o Secure Shell (SSH)
- `local`: acceso a repositorio local
- `fork`: acceso a repositorio local usando protocolo remoto

En módulos a extraer se especifica el módulo que se quiere extraer (o se hace con Examinar), indicando en Carpeta local donde se quieren extraer.

Para importar hacia el repositorio remoto se selecciona el proyecto no versionado y se elije Equipo - CVS - Importar al depósito. Se especifica la localización con CVSROOT, variando según el método usado para conectarse, y la carpeta local en la que se quiere colocar. Se especifica la localización del repositorio escribiendola en Carpeta del depósito (o con examinar para ir a algun lado dentro del repositorio)

2.2. Herramientas de control de versiones

- CVS.
- Subversion: Los archivos no tienen número de versión independiente. Todo el repositorio tiene único número de versión que identifica el estado común de todos en un instante. Se accede al repositorio a través de redes. Modifican el mismo conjunto de datos. Control de versiones.
- SourceSafe (Microsoft Visual Studio)
- Visual Studio Team Foundation Server. Sustituto de Source Safe. Ofrece código fuente, recolección de datos, seguimiento de proyectos.
- Darcs: Gestión de versiones distribuido. Cada repositorio es una rama en sí misma, independiente del servidor central.
- Git
- Mercurial: Programa de línea de comandos. Desarrollo distribuido. Capacidades avanzadas de ramificación e integración. Opciones dadas a su motor hg. Gran rendimiento y escalabilidad.

Más destacados en IDEs con plugins disponibles: CSV, Subversion, Mercurial. (Y Git ni lo menciona, hay que joderse). en Netbeans y Eclipse. Y otros como Visual Studio Team Foundation en Microsoft Visual Studio.

2.3. Planificación de la gestión de configuraciones

La configuración es una combinación de versiones particulares de componentes que hacen un sistema consistente. Desde la evolución en el tiempo es el conjunto de versiones de los componentes en un instante dado.

Gestión de configuraciones de software (GCS): Conjunto de actividades desarrolladas para gestionar cambios a lo largo del ciclo de vida. Su planificación se regula en estándar IEEE 828.
Se refiere a la evolución de todo un conjunto de elementos.

Debe disponerse de un método que designe las configuraciones de forma sistemática y planificada para facilitar desarrollo de software de forma evolutiva, por cambios sucesivos.

Tareas de la gestión de configuraciones de software

1. **Identificación** (Estándares de documentación. Esquema de identificación)
2. **Control de cambios** (Evaluación y registro de todos los cambios en la configuración)
3. **Auditorías de configuraciones**: Junto con revisiones técnicas formales garantiza que el cambio se ha implementado correctamente
4. **Generación de informes**: Garantizar la consistencia del conjunto del sistema (ya que los componentes evolucionan de forma individual)

Actividades de la planificación gestión de configuraciones de software

- Introducción (propósito, alcance, terminología)
- Gestión de GCS (organización, responsabilidades, políticas, directivas, procedimientos)
- Actividades GSC (identificación de configuración, control de configuración)
- Agenda GSC (coordinación con otras actividades)
- Recursos GCS (herramientas, recursos físicos y humanos)
- Mantenimiento GCS

2.4. Gestión del cambio

- Debe haber un control para que el desarrollo sea razonable (no es posible que cualquier componente del equipo de desarrollo pueda realizar cambios e integrarlos sin ningún control). --> GARANTIZAR LINEA BASE PARA CONTINUAR EL DESARROLLO, con controles al desarrollo y la integración.

Tipos de control:

1. **Control individual**: Antes de aprobarse un nuevo elemento, el programador cambia documentación cuando se requiere. Registra de forma informal el cambio, aunque no sea documento formal.
2. **Control de gestión u organizado**: Con procedimiento de revisión y aprobación de cada cambio propuesto. El cambio es registrado formalmente.
3. **Control formal, durante el mantenimiento**. El impacto de cada tarea de mantenimiento se evalúa por Comité de Control de Cambios, que aprueba las modificaciones de la configuración software.

2.5. Gestión de versiones y entregas (CICD)

De versiones... (""Continuous Integration"")

La evolución de las versiones pueden representarse en forma de grafo donde los nodos son versiones y los arcos son creación de una versión a partir de otra.

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

Variantes: Cuando hay varias versiones del componente el grafo no es lineal sino con forma de árbol. La numeración de versiones tiene dos niveles: Primer número (variante, línea de evolución) y segundo número (versión particular, revisión) a lo largo de dicha variante.

Terminología de los elementos del grafo

- Tronco (trunk): Variante principal
- Cabeza (head): Última versión del tronco
- Ramas (branches): Variantes secundarias
- Delta: Cambio de una revisión respecto a la anterior

Propagación de cambios: Aplicar mismo cambio a varias variables que están en paralelo.

Fusión: Dejar de mantener una rama independiente fundiéndola con otra (merge)

Técnicas de almacenamiento: Ya se ha comentado que suelen tener el mismo contenido las distintas ramas, no repitiendo esos datos comunes para no desaprovechar espacio.

- **Deltas directos**: Primera versión completa y luego cambios mínimos necesarios para reconstruir cada nueva versión a partir de la anterior.
- **Deltas inversos**: Última versión completa y cambios mínimos para reconstruir versión anterior a partir de la siguiente.
- **Marcado selectivo**: Texto refundido de todas las versiones como secuencia lineal. Marcando cada sección del conjunto con los números de versiones que le corresponden.

De entregas... (""Continuous Deployment"")

La entrega es una instancia del sistema que se distribuye a 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 del conjunto de programas ejecutables, los archivos de configuración, los archivos de datos, el instalador, documentación, embalaje, publicidad...

2.6. Herramientas CASE para la gestión de configuraciones

CASE Compute Aided Software Engineering.

Abiertos: Seguimiento configuraciones (bug-tracking) como `Bugzilla`. Seguimiento versiones: `RCS`, `RVS`. Construcción del sistema `Make`, `Imake`

Integrados: Para la gestión de versiones, construcción del sistema y seguimiento de configuraciones (cambios). Proceso de control de cambios unificado de `Rational`, basado en Gestión de configuraciones de `ClearCase` para construcción y gestión de versiones y `ClearQuest` para seguimiento de cambios. Los entornos de Gestión de Configuraciones Integrado ofrecen la ventaja de intercambio de datos sencillos y el entorno ofrece Base de datos de gestión de configuraciones integrada.

3. Documentación

- Permite explicar su funcionamiento, punto por punto para que cualquier persona que lea el comentario pueda entender su finalidad.
- Es fundamental para detección de errores y mantenimiento posterior.
- Da explicaciones de la función del código, de las características de un método. . No debe explicar qué hace el código sino por qué lo hace. (Finalidad de clase, de paquete, qué hace un método, para qué sirve una variable, qué se espera, qué algoritmo se usa, por qué así y node otra forma, qué se podría mejorar)

3.1. Uso de comentarios

Anotación (Entre / * /) que se realiza en el código y que el compilador ignora. Sirve para indicar aspectos a los desarrolladores:

- Explicar objetivo de las sentencias
- Explicar qué realiza (no cómo lo realiza)

En Java los que se usan para explicar qué hace un código se llaman Comentarios Javadoc /** */
Los comentarios "son obligatorios" con JavaDoc. Deben incorporarse al principio de cada clase, método y variable de clase.

Si el código es modificado, los comentarios deben también.

3.2. Alternativas a la falta de buena documentación

Alternativas para documentar código son los comentarios. **JavaDoc, SchemeSpy, Doxygen...** permiten producir una documentación actualizada, precisa y utilizable en línea. (Con SchemeSpyy Doxygen se incluyen modelos de bases de datos gráficos y diagramas)

Con comentarios en código difícil de entender y documentación generada por estas herramientas, se puede ayudar al nuevo desarrollador a entenderlo.

3.3. Documentación de código con JavaDoc

DOCUMENTACIÓN DE CLASES:
Criterios de documentación establecidos por JavaDoc

Comenzar y terminar así
/**
*/

Incluir al menos `@author` y `@version`. Con los IDE se añaden de forma automática.
`@see` para referenciar a otras clases y métodos. También está `@since`, fecha desde la que está presente la clase.

DOCUMENTACIÓN DE MÉTODOS:
`@param` seguido del nombre, parámetro que se le pasa
`@return` si no es `void`, se indica lo que devuelve
`@exception` nombre de la excepción, especificando cual pueden lanzarse. Etiqueta antigua en versiones de Javadoc, se usa para describir excepciones que son subclases de `Throwable` como `RuntimeException` o `IOException`...
`@throws`, nombre de la excepción, más reciente y especificando CUALES excpeciones (en plural) pueden lanzarse
`@deperecated` método obsoleto

DOCUMENTACIÓN DE ATRIBUTOS:
Pueden contener comentarios aunque no hay etiquetas obligatorias en JavaDoc.

UD 05 - DISEÑO ORIENTADO A OBJETOS Y ELABORACIÓN DE DIAGRAMAS

1. Introducción a la orientación a objetos

En la construcción del software podría uno decantarse por un

- **enfoque estructurado**: El problema de partida se toma y se somete a un proceso de división en subproblemas más pequeños hasta llegar a problemas elementales resolubles utilizando una función. Luego estas funciones se entretejen para formar solución global (proceso centrado en procedimientos, codificando mediante funciones que actúan sobre estructuras de datos). Se intenta aproximar qué hay que hacer para resolver un problema.
- **enfoque orientado a objetos**: se simulan los elementos de la realidad asociada al problema de la forma más cercana posible mediante una abstracción llamada objeto (conjunto de atributos por los cuáles se caracteriza y de operaciones que definen su comportamiento. Las operaciones actúan sobre los atributos para modificar su estado. Cuando se indica a un objeto que ejecute determinada operación 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 siendo un objeto una instancia de una clase. Al ejecutar el programa:

- Se crean los objetos a medida que se necesitan
- Los mensajes se mueven de un objeto a otro (o del usuario al objeto) a medida que el programa procesa información o responde a la entrada del usuario
- Cuando los objetos no se necesitan se borran y se libera la memoria

Como **ventajas** frente a otros paradigmas podría destacarse:

MODULARIDAD, ENCAPSULACIÓN, REUTILIZACIÓN, ESCALABILIDAD

- Puede desarrollarse software en mucho menos tiempo, menos coste y más calidad gracias a la reutilización de código (código modular, reusable en aplicaciones similares...)
- La calidad de los sistemas se aumenta, haciéndolos más extensibles (sencillo aumentar o modificar funcionalidad)
- Más fácil de modificar, más mantenible por sus criterios de modularidad y encapsulación
- Mayor adaptación al entorno, haciendo aplicaciones escalables. Sencillo modificar estructura y comportamiento sin cambiar la aplicación.

1.1. Principios de la orientación a objetos

- **Abstracción**: Se capturan las características y comportamientos similares de un conjunto de objetos para darles una descripción formal -> Se arma un conjunto de clases que permiten modelar la realidad.
- **Encapsulamiento / Encapsulación**: Reunir todos los elementos que puedan considerarse pertenecientes a una misma entidad, al mismo nivel de abstracción (No confundir con ocultación)
- **Modularidad**: Subdividir una aplicación en partes más pequeñas (módulos), cada una de las cuales es tan independiente entre sí y del resto de la aplicación como sea posible. Es algo consubstancial en la POO: Los objetos son los módulos más básicos del sistema.
- **Principio de ocultación**: Se aíslan las propiedades del objeto para evitar su modificación por quién no tenga derecho a acceder a ellas (evita propagación de efectos colaterales con los cambios)
- **Polimorfismo**: Se reúnen bajo el mismo nombre comportamientos diferentes
- **Herencia**: Relación entre objetos en los que unos usan propiedades y comportamientos de otros formando una jerarquía. Los objetos heredan las propiedades y el comportamiento de todas las clases a las que pertenecen.

- **Recolector de basura**: El entorno de objetos se encarga de destruirlos automáticamente y de desvincular su memoria asociada cuando ha quedado sin ninguna referencia a ellos.

1.2. Clases, atributos y métodos

Los objetos del sistema se abstraen en **clases**: Conjunto de procedimientos y datos que resumen características similares de un conjunto de objetos. /// Abstracción que define las características comunes de un conjunto de objetos relevantes para el sistema. // Abstracciones del dominio del sistema que representan elementos del mismo mediante una serie de características (atributos) y de comportamientos (métodos)

El objeto es único debido a los valores que tienen asignados sus atributos. La clase define sus características generales y comportamiento.

Sus propósitos son **definir abstracciones** y **favorecer la modularidad**.

Los **miembros** de una clase son:

- **Nombre**: Identificativo. Se relaciona con lo que representa.
- **Atributos**: Características asociadas a la clase. Puede ser una relación binaria entre la clase y un dominio formado por todos los valores que puede tomar cada atributo. Al tomar valores válidos de este dominio definen el **estado** de un objeto. Los atributos se definen por su nombre y por su tipo (pudiendo ser simple o compuesto como otra clase).
- **Protocolo**: Operaciones (métodos, mensajes) que manipulan el estado:
  - Método: Procedimiento o función que se invoca para actuar sobre un objeto. Determina cómo actúan los objetos cuando reciben un mensaje (cuando se requiere que el objeto realice una acción descrita en un método se le envía un mensaje). El conjunto de mensajes a los que puede responder un objeto se conoce como *protocolo* del objeto.
  - Mensaje: Resultado de la acción efectuada por un objeto

Mencionar la **clase abstracta** que es aquella que no puede ser instanciada. Es usada para definir métodos de base para clases derivadas o para definir métodos genéricos relacionados con el sistema que no serán traducidos a objetos concretos (raramente)

1.3. Visibilidad

La visibilidad son los niveles de ocultación posibles (tipo de acceso a atributos y métodos)

- Público `public`: Desde cualquier clase
- Protegido `protected`: Solo operaciones de la clase o clases derivadas (herencia)
- Paquete `default`: Solo operaciones de clase del mismo paquete.
- Privado `private`: Solo operaciones de la clase

Debido al *principio de ocultación* (el estado se aísla de forma que solo pueda cambiarse mediante las operaciones definidas en una clase, evita la modificación por quién no tiene derecho a ello) las clases se dividen en dos partes:

- Interfaz: Visión externa de una clase (abstracción del comportamiento común a los ejemplos de esa clase)
- Implementación: Cómo se representa la abstracción; mecanismos que conducen al comportamiento deseado.

A la hora de definir la visibilidad debe tenerse en cuenta:

- El **estado debe ser privado**. Atributos deben modificarse mediante métodos 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** o **protegidas*

1.4. Objetos. Instanciación

Instancia de clase: Construir un objeto en un programa informático a partir de una clase. Cada instancia es modelo de un objeto del contexto del problema relevante para su solución que puede hacer un trabajo, informar, cambiar su estado y "comunicarse" con otros objetos del sistema, sin revelar cómo se implementan estas características.

Objetos pueden ser... objetos físicos (aviones, casas, parques); elementos UI (menús, teclados, cuadros de diálogo); animales (vertebrados, invertebrados); alimentos (carnes, frutas ,verduras)...

El objeto se define por:

- Su estado (concreción de los atributos de 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 (desde su creación por el mecanismo de instanciación hasta su destrucción)

2. UML. Definición, elementos, clasificación

Unified Modeling Language (UML) o lenguaje unificado de modelado es un conjunto de herramientas que permite MODELAR, CONSTRUIR y DOCUMENTAR los elementos que forman parte de un sistema de software POO.

> Los desarrolladores pueden visualizar el producto de su trabajo en una colección de esquemas o diagramas estandarizados llamados **modelos** que representan el sistema desde diferentes perspectivas

Fue concebido por los autores de los tres métodos más usados de la POO (Booch (Método Booch), Jacobson (OOSE Object-Oriented Software Engineering) y Rumbaugh (OMT Object Modeling Technique))

Es útil porque:

- permite usar un **lenguaje común** para comunicarse en los equipos de desarrollo
- pueden **documentarse todos los artefactos** (toda información utilizada o producida mediante un proceso de desarrollo de software: requisitos, arquitectura, pruebas, versiones) disponiéndose de información que trasciende al proyecto
- **permite indicar información sobre estructuras que trascienden lo representable en un lenguaje de programación**, como la arquitectura del sistema, y con UML pueden indicarse incluso qué módulos de software van a desarrollarse y sus relaciones o en qué nodos hardware se ejecutarán cuando se trata de sistemas distribuidos
- puede conectarse a lenguajes de programación mediante **ingeniería directa** (transformación de modelo en código) e **ingeniería inversa** (transformación del código en un modelo)

2.1. Herramientas para la generación de diagramas UML

Desde lápiz y papel hasta herramientas CASE que suelen contar con ventanas wysiwyg (*What You See Is What You Get*). No solo generan los diagramas asociado al análisis y al diseño de la aplicación, también sirven para documentar los diagramas, integrarse con otros entornos, generar código automáticamente y procesos de ingeniería inversa generando diagramas a partir del código fuente o incluso binario. Mencionamos:

- **Rational Systems Developer de IBM**. **IBM Rational Rhapsody Developer**. Inicialmente creadores UML, después absorbida por IBM. Tiene versiones de prueba y software libre.
- **Visual Paradigm for UML**. Tiene versión para uso no comercial simplemente registrándose para conseguir un archivo de licencia. Incluye módulos para UML, diseño de bases de datos, Agile, ingeniería inversa y es multiplataforma y compatible con Eclipse, VisualStudio, IntelliJ IDEA, NetBeans...
- **ArgoUML**. Con licencia Eclipse. Genera código para Java y C++. Se ejecuta con Java. Tiene ingeniería directa e inversa.

2.2. Elementos de un diagrama UML

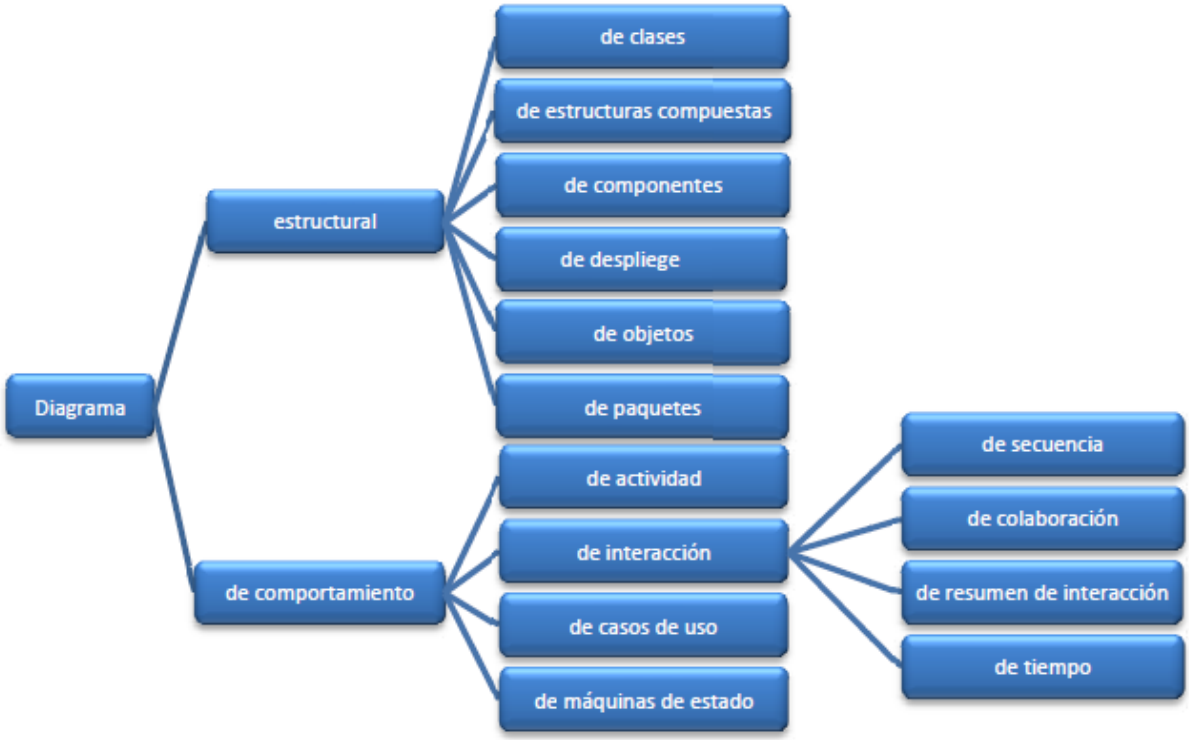
Los modelos suelen ser un grafo conexo de arcos (relaciones) y vértices (elementos del modelo).

- **Estructuras**: Nodos del grafo. Describen el tipo de diagrama
- **Relaciones**: Arcos del grafo que se establecen entre los elementos estructurales

- **Notas**: Cuadro donde se pueden escribir comentarios para entender algún concepto que se quiere representar
- **Agrupaciones**: Al modelar sistemas grandes se usan para separar su desarrollo por bloques

2.3. Clasificación de diagramas UML

Como ejemplo, podrían describirse 13 diagramas para modelar diferentes aspectos de un sistema, aunque no es necesario usarlos todos; dependerá del tipo de aplicación a generar y del sistema, solo debe generarse un diagrama si es necesario.



Existen dos grandes grupos:

- **Diagramas estructurales**: Visión estática del sistema (modelo estático abstracto). Especifican clases y objetos y cómo se distribuyen físicamente en el sistema.
  De prioridad **ALTA**

  - ***Diagrama de clases**: Elementos del modelo estático abstracto. Formado por el conjunto de clases y sus relaciones.
  - **Diagrama de objetos**: Elementos del modelo estático en un momento concreto. Formado por el conjunto de objetos y de sus relaciones. Habitualmente usado para casos especiales de un diagrama de clases o de comunicaciones.

  De prioridad **MEDIA**

  - **Diagrama de componentes**: Organización lógica de la implementación de una aplicación, sistema o empresa, indicando componentes, interrelaciones, interacciones, interfaces públicas y dependencia entre ellos.
  - **Diagrama de despliegue**: Configuración del sistema en tiempo de ejecución. Con nodos de procesamiento y sus componentes. Ejecución de la arquitectura del sistema: Nodos, ambientes operativos (hardware o software) e interfaces que las conectan. Muestra cómo los componentes de un sistema se distribuyen en los ordenadores que los ejecutan. Se usa en sistemas distribuidos.

  De prioridad **BAJA**
  - **Diagrama de paquetes**: Organización de los elementos del modelo en paquetes y dependencias entre estos paquetes. Útiles para sistemas de mediano o gran tamaño.

    - **Diagrama de estructuras compuestas / diagrama integrado de estructuras (UML 2.0)**: Estructura interna de una clasificación (clase, componente, caso típico) con los puntos de interacción de esta clasificación con otras partes del sistema.
- **Diagramas de comportamiento**: Conducta en el tiempo de ejecución del sistema, tanto desde el punto de vista del sistema completo como de las instancias u objetos que lo integran.

  De prioridad **ALTA**

    - ***Diagramas de actividad**: Orden en el que se van realizando las tareas dentro de un sistema. Procesos de alto nivel de la organización. Incluye flujo de datos o modelo de la lógica compleja dentro del sistema.
    - **Diagramas de interacción- De secuencia**: Ordenación temporal en el paso de mensajes. Modela la secuencia lógica a través del tiempo de los mensajes entre instancias.

  De prioridad **MEDIA**

    - ***Diagramas de casos de uso**: Acciones a realizar en el sistema DESDE EL PUNTO DE VISTA DE LOS USUARIOS SIN FORMACIÓN EN DESARROLLO. Acciones, usuarios y relaciones entre ellos. Especifica la funcionalidad y el comportamiento mediante su interacción con usuarios y/o sistemas.
    - **Diagramas de máquinas de estado**: (Diagrama de estado, diagramas de estados y transiciones, diagrama de cambio de estados) Comportamiento de un sistema dirigido por eventos. Estados que puede tener un objeto o interacción y transiciones entre estados.

  De prioridad **BAJA**

  Diagramas de interacción
    - ***De comunicación/colaboración (UML 2.0)**: Organización estructural de los objetos que se reciben y envían mensajes. Instancias de clases, interrelaciones, flujo de mensajes entre ellas.
    - ***De interacción**: Conjunto de objetos y relaciones junto con los mensajes que se envían entre ellos. Es variante del diagrama de actividad que permite mostrar el flujo de control dentro de un sistema o proceso organizativo. Cada nodo de actividad puede representar otro diagrama de interacción.
    - ***De tiempo**: Cambio de un estado o una condición de una instancia o un rol a través del tiempo. Se usa para exhibir el cambio en el estado de un objeto en el tiempo, en respuesta a eventos externos.

3. Diagramas UML estructurales -> Diagramas de clases

- El más importante por representar el modelo estático del sistema, atributos, comportamientos y relaciones entre ellos. A partir de él se generarán las clases que formarán el programa informático que solucionará el problema.
- Contiene las clases de **dominio del problema** (dominio: área de conocimiento caracterizada por un conjunto de conceptos y terminología comprendida por los practicantes del dominio; dominio del problema: dominio sobre el que se define el problema a resolver por el sistema que se va a generar)
- Se componen de:
    - Clases:
    - Relaciones: Que pueden ser de asociación, agregación y herencia. Ej.: Clases "Persona" y "Coche" se establece relación CONDUCE.
    - Notas: Cuadro con comentarios para entender conceptos mejor
    - Elementos de agrupación: Se modela sistema grande agrupando clases y relaciones en paquetes que se relacionan entre sí

Clase

Un rectángulo dividido en tres filas: Primero el nombre de la clase; Segundo sus atributos con visibilidad; Tercero sus métodos con visibilidad. (-) privados; (+) públicos. Solo sería obligatorio definir el nombre, no los métodos, ni los atributos.

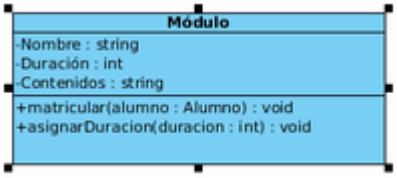
Atributos

Los atributos tienen que tener **nombre**, **tipo**, **visibilidad**. Puede indicarse un valor inicial.

Métodos

Necesario **nombre**, **parámetros**, **tipo de retorno**, **visibilidad**. Puede indicarse una descripción del método que aparecerá en la documentación.

El **método constructor** no devuelve nada y tiene el mismo nombre de la clase. Se usa para ejecutar acciones necesarias al instanciar la clase. Para destruir el objeto (según el lenguaje) también podrá existir un **método destructor** (En Java no).



Relaciones entre clases

Relación: Conexión entre dos clases que se incluye en el diagrama cuando hay algún tipo de relación entre ellas en el dominio del problema.

Estas relaciones no siempre es necesario que aparezcan especificadas en la lista de requisitos del problema. Es posible que no aparezcan específicamente pero se infieran (como las relaciones de herencia)

Las relaciones se representan como una línea continua. Los mensajes se envían entre objetos de clases relacionadas (en varias direcciones o quizás solo en una).

Las relaciones se caracterizan por la **cardinalidad** o **multiplicidad de clases** que representa cuántos objetos de una clase pueden verse involucrados en la relación.

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

Aquí por ejemplo:

- un alumno puede MATRICULARSE de 1 o más módulos (miro a la derecha de la relación)
- un módulo puede TENER MATRICULADOS 0 o más alumnos (miro a la izquierda de la relación)

Relación de herencia

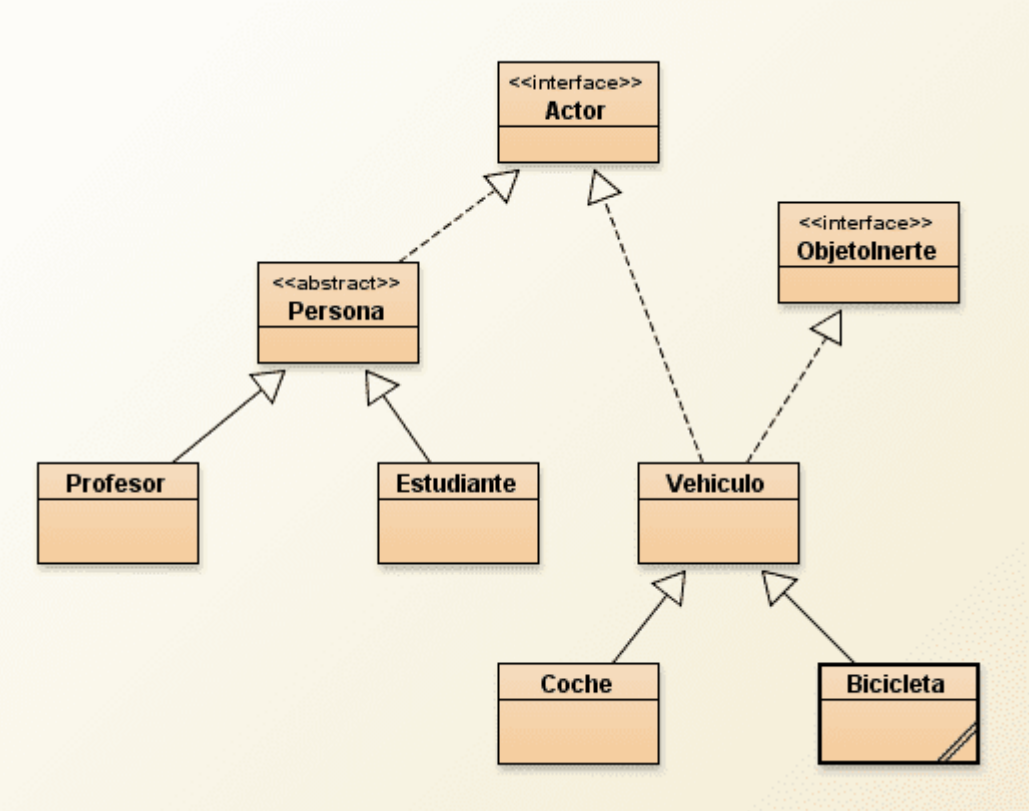
Herencia: Capacidad de un objeto para usar estructuras de datos y métodos presentes en sus antepasados. Permitiendo la reutilización de código. Pueden añadir su propio código especial y datos e incluso cambiar los que necesiten que sean diferentes.
Puede haber:

- Herencia simple: Un ascendente

- Herencia múltiple: Varios ascendentes (No permitido en Java)

Los atributos métodos y relaciones se muestran en el nivel más alto de la jerarquía en el que son aplicables.
La relación de herencia se representa en el diagrama como un triangulo en el extremo de la clase base (clase de la que hereda)

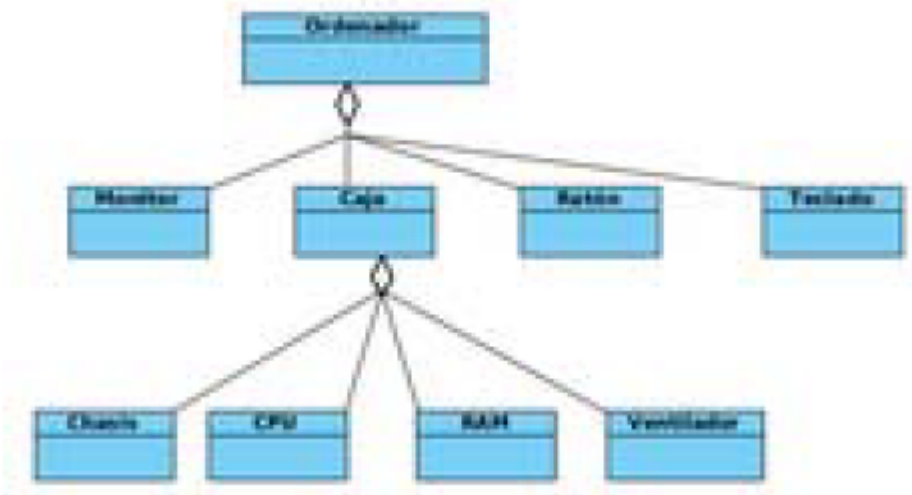
Cuando se **implementan** interfaces (caso de Java) se representa también con un triángulo en el extremo de la interfaz y la línea de la relación se pone punteada.



Relación de agregación y composición

Agregación: **Elementos pueden existir sin el elemento contenedor**. Asociación binaria todo-parte (pertenece a, tiene un, es parte de). El tiempo de vida de los objetos no tiene por qué coincidir. Ej.: Ordenador compuesto de piezas sueltas con entidad por sí mismas.

Se representa con un rombo en el extremo de la entidad contenedora. En agregación es blanco.

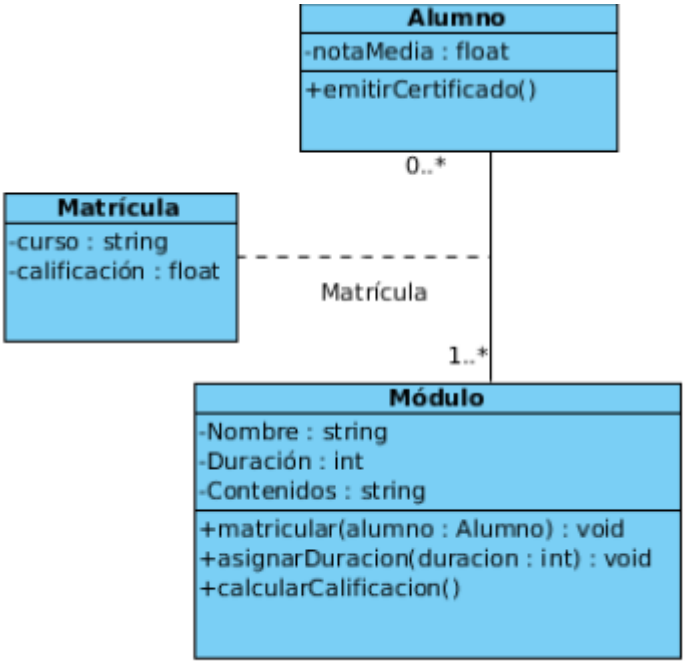


Composición: **Unos no pueden existir sin otros. Cuando el objeto "todo" es eliminado, también se eliminan los objetos "parte"**. Agregación fuerte en la que una "parte" está relacionada como máximo con un "todo" en un momento dado. Ej.: Rectángulo y vértices. Centro comercial y tiendas.

Se representa con un rombo en el extremo de la entidad contenedora. En composición es negro.

Atributos de enlace

Para completar información de alguna forma pueden añadirse atributos a la relación. Ej.: Alumno ---- Curso y una relación Matrícula con atributos



Paso de requisitos del sistema al diagrama de clases

OBTENCIÓN DE CLASES

Partiendo de los requisitos del sistema, situar en una tabla:

- Nombres (sinónimos) que se refieran a clases
- Entidades externas: Que producen o consumen información (sistemas, dispositivos, personas) que usará la aplicación
- Ocurrencias o sucesos que ocurren en una operación del sistema
- Papeles o roles de personas que interactúan con el sistema
- Lugares que establecen el contexto del problema y la función del sistema
- Estructuras que definen una clase de objetos o clases relacionadas de objetos

No incluir en la lista cosas que no sean objetos (como operaciones de otro objeto que darían lugar a un método).

Después, estudiar cada objeto de la tabla para ver si pasará al diagrama de clases, cosa que se hará si cumple todos o muchos de estos criterios:
1.-La información del objeto es necesaria para que el sistema funcione
2.-El objeto tiene un conjunto de atributos que pueden encontrarse en cualquier ocurrencia del objeto. Si solo es un atributo normalmente será añadido como atributo de otro objeto.
3.-El objeto tiene un conjunto de operaciones identificables para cambiar el valor de sus atributos y son comunes a cualquier ocurrencia del objeto
4.-Es una entidad externa que consume o produce información para la producción de cualquier solución en el problema

Además de estos, aquellos adicionales que completen el modelo. También ver que diferentes descripciones del problema pueden implicar diferentes decisiones en la creación de objetos y atributos.

OBTENCIÓN DE ATRIBUTOS; OPERACIONES; RELACIONES

Los atributos deben contestar a la pregunta ¿Qué elementos definen completamente al objeto en el contexto del problema?
Las operaciones describen el comportamiento y modifican características manipulando datos, realizando algún cálculo, monitorizando el objeto frente a la ocurrencia de un suceso de control.
Para las relaciones... buscar mensajes que se pasen entre objetos y relaciones de composición y agregación. Las relaciones de herencia se suelen encontrar al ver objetos con atributos y métodos comunes.

REVISIÓN FINAL

Finalmente revisar el diagrama obtenido y ver si cumple las especificaciones. Refinar el diagrama completando aspectos que no aparezcan en la descripción recurriendo a entrevistas con el cliente o a conocimiento en la materia.

Generación de código y documentación a partir del diagrama de clases

En algunos casos las herramientas CASE, como Visual Paradigm, podrán generar (o actualizar) las clases en código fuente a partir del diagrama de clases UML.

- Con el SDE integrado de VP-UML en Netbeans. Diferenciar entre "Sincronizar con el código" (Código fuente eliminado no se recupera, solo se actualiza el existente). "Forzar sincronizado a código" (Se actualiza todo el código que pueda partir del modelo, incluido el del código eliminado de Netbeans).
- Desde VP-UML puede darse a Herramientas >> Generación instantánea >> Java configurando idioma, clases a generar y otras características.

Es "Specification" o "Comentarios" se podrían añadir comentarios y luego con estos generar informes (HTML, PDF, doc) con la documentación de los diagramas seleccionados.
Indicar qué diagramas se quiere que intervengan, qué elementos se añadirán, propiedades de página, marca de agua...

4. Diagramas UML de comportamiento

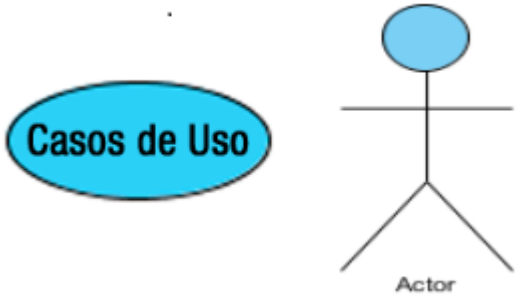
COMPORTAMIENTO DINÁMICO DE LOS OBJETOS : como la creación-destrucción de objetos, paso de mensajes entre objetos, orden en que deben hacerse, funcionalidad que el usuario espera realizar, cómo influyen elementos externos en el sistema...

4.1. Diagramas de casos de uso

- **Resuelve la falta de comunicación entre el EQUIPO DE DESARROLLO y el EQUIPO QUE DEMANDA UNA SOLUCIÓN SOFTWARE**. Son **fáciles de interpretar**, facilitan comunicación con el cliente.
- Determina **QUÉ puede hacer** cada tipo de usuario con el sistema, de una forma que **las personas ajenas al desarrollo puedan entenderlo**.
- Los casos de uso son visualización gráfica de los **REQUISITOS FUNCIONALES DEL SISTEMA**, representan las funciones que un sistema puede ejecutar.
- Su **función** es dirigir el proceso de creación de software definiendo qué se espera de él
- De los diagramas de casos de uso se desprenden otros que describirán la estructura y el comportamiento del sistema, influyen directamente en la implementación del sistema y en su arquitectura final. También se usa en la fase de pruebas para verificar que el sistema cumple con los requisitos funcionales, creándose muchos de los casos de prueba de caja negra a partir de los casos de uso.
- Normalmente los diagramas de casos de uso se hacen incluso antes del diagrama de clases

Los casos de uso se representan como elipses
Los actores que interactúan con ellos se representan como monigotes

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.



Actores

Los actores representan un tipo de usuario del sistema (cualquier cosa externa que interactúa con el sistema sea ser humano, otro sistema informático, una empresa, una entidad para la autenticación de claves...)

Debe diferenciarse el actor de la forma en que se interactúa con el sistema. Un usuario puede interpretar diferentes roles según la operación que ejecuta. Cada rol es un actor diferente.
Los **actores** no representan a individuos particulares, sino a roles que estos pueden jugar.

Pueden ser 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; precisos para alcanzar un objetivo), iniciadores (no interactúan con el sistema, pero desencadenan el trabajo de otro actor). Si no son iniciados por ningún usuario o elemento software, puede crearse un actor llamado "Tiempo" o "Sistema"

Casos de uso

Especifica una secuencia de acciones, incluyendo variantes, que el sistema puede llevar a cabo y que produce un resultado observable de valor concreto.
El conjunto de los casos de uso forma el **comportamiento requerido** del sistema.

Lo importante no es crear el diagrama, sino la descripción de cada caso porque esto ayudará al equipo de desarrollo a crear el sistema. Para esto se usan otros diagramas que describen la dinámica del caso de uso (como el diagrama de secuencia) y una descripción textual que debe incluir:

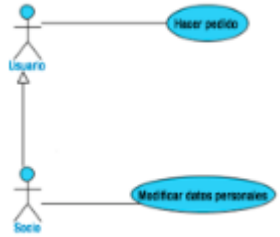
- **Nombre**: Del caso de uso
- **Actores**: Que interactúan con el sistema en el caso de uso
- **Propósito**: Lo que se espera que haga
- **Precondiciones**: Lo que debe cumplirse para que se lleve a cabo el caso de uso
- **Flujo normal**: Flujo normal de eventos que deben cumplirse para ejecutar el caso de uso exitosamente desde el punto de vista del actor y del sistema
- **Flujo alternativo**: Flujo de eventos que se llevan a cabo con casos inesperados o poco frecuentes. No se deben incluir errores como tipos de datos incorrectos o parámetros necesarios omitidos.
- **Postcondiciones**: Se cumplen una vez realizado el caso de uso

Relaciones

Las relaciones representan qué actores realizan tareas de los casos de uso (quiénes las inician) pero también hay relaciones más complejas:

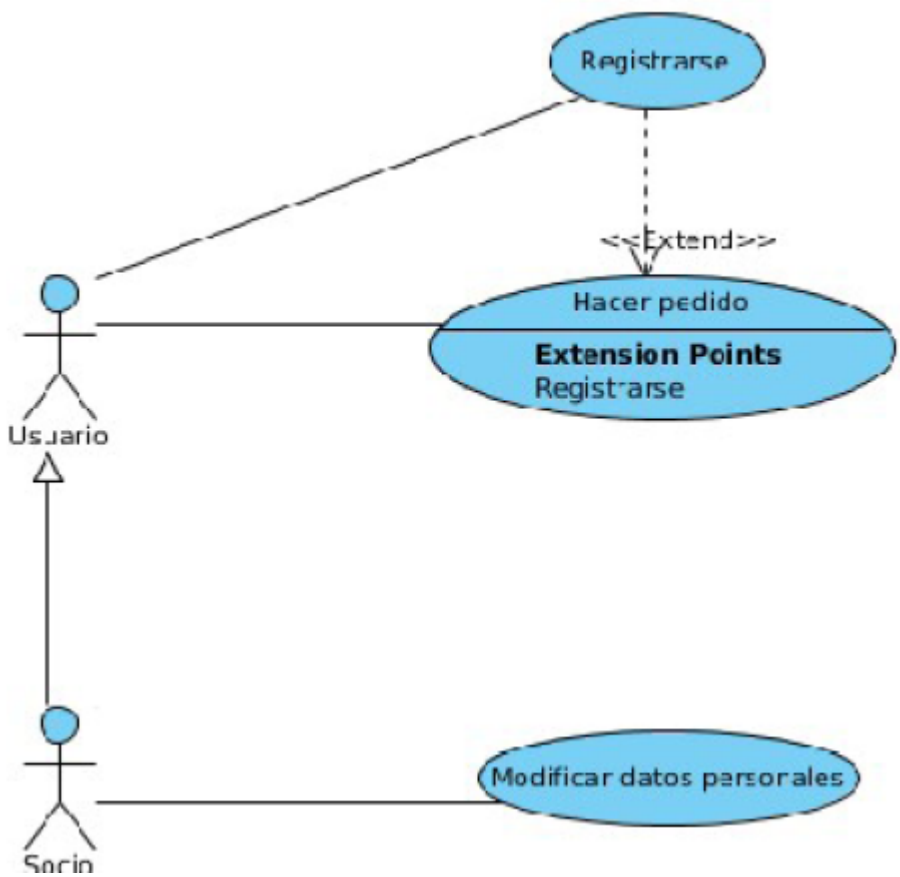
- **Asociación**: Relación entre el actor que interactúa con el sistema para iniciarlo. Línea continua une a un actor con un caso de uso. Ej.: "Usuario" "hacer pedido"
- **Generalización**: Es posible que unos casos de uso tengan comportamientos a otros que los modifican o completan de alguna forma. Se puede definir un caso base de forma abstracta y unos hijos que heredan sus características. (Suele usarse poco en los diagramas de casos de uso). Ej.: La persona puede hacer un pedido y, si quiere, darse de alta. Habría un rol "Usuario"

"hacer pedido" y un rol "Socio" que hereda de "Usuario" y puede "Modificar datos personales" (y además, "hacer pedido")



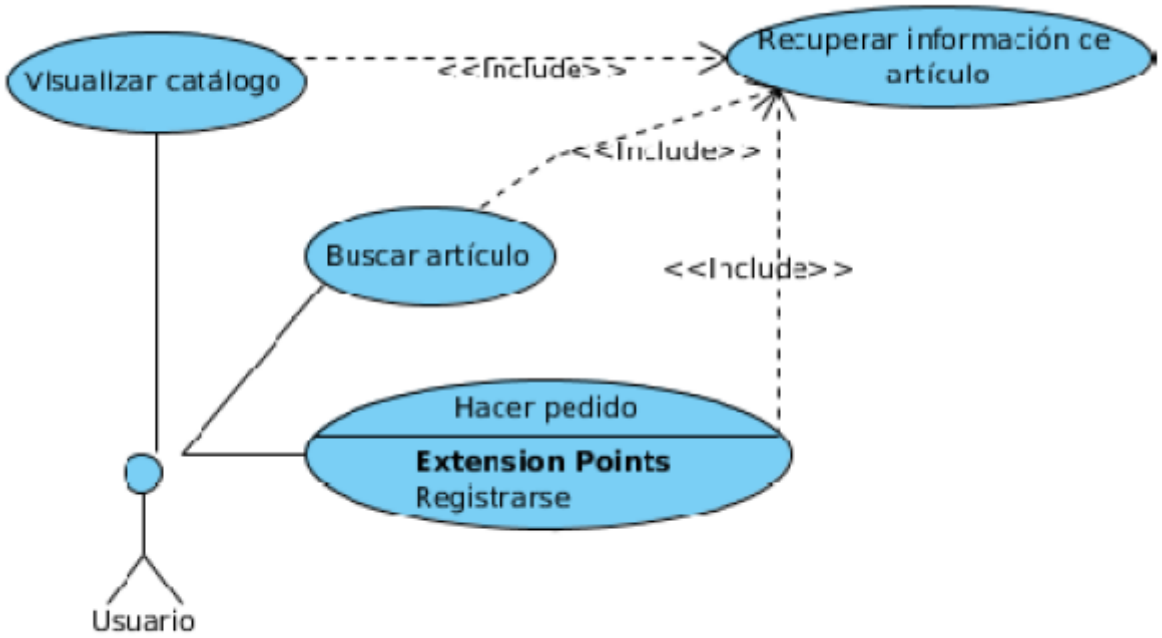
A la hora de usar las siguientes ("extends" e "include") debe tenerse en cuenta que los casos de uso extendidos o incluidos deben representar un flujo de actividad completo desde el punto de vista de lo que el actor espera que el sistema haga por él (caso de uso). No deben usarse estas herramientas para descomponer un caso de uso de envergadura en otros más pequeños.

- **Extensión**: Se puede usar relación "extends" para indicar que el comportamiento del caso de uso es diferente según alguna circunstancia. Así se simplifica el flujo de casos de uso complejos. Cuando un caso de uso extendido se ejecuta se indica en la especificación del caso de uso como un punto de extensión. Ej.: Cuando el usuario hace un pedido, si no es socio, se le da la posibilidad de darse alta en el sistema en ese momento, aunque puede realizar el pedido aunque no lo sea.



- **Inclusión**: Cuando la ejecución del caso de uso incluido se da en la rutina normal del caso que lo incluye se usa "include". Es útil cuando se desea especificar un comportamiento en común con dos o más casos de uso. Aunque es un error frecuente usar esta técnica para hacer subdivisión de funciones (debe tenerse cuidado). Ej.: Cuando se hace un pedido, se busca información de los artículos para obtener el precio. Pero esto también forma parte de otros como visualizar el catálogo de productos y buscar un artículo concreto. Al tener entidad por sí solo se separa del resto de casos de uso y se incluye en los otros tres. Ej2.: Empleado "Servir pedido" y "Consultar stock" (Consultar stock es "extends" porque tiene entidad suficiente como para ser caso de uso por sí mismo, que podrá usarse en otros casos y además siempre debe

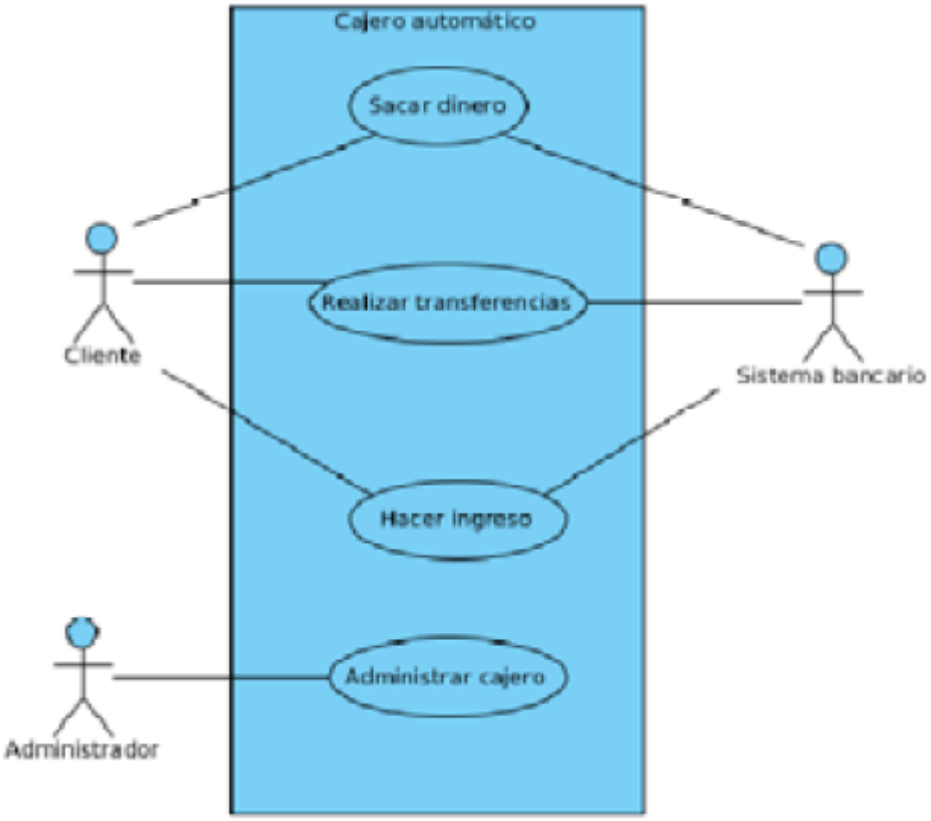
hacerse).
Ventajas: Descripciones de casos de uso cortas y se entienden mejor. Identificación de funcionalidad común puede identificar reutilización de componentes ya existentes.
Inconvenientes: Los diagramas son más difíciles de leer por los clientes.

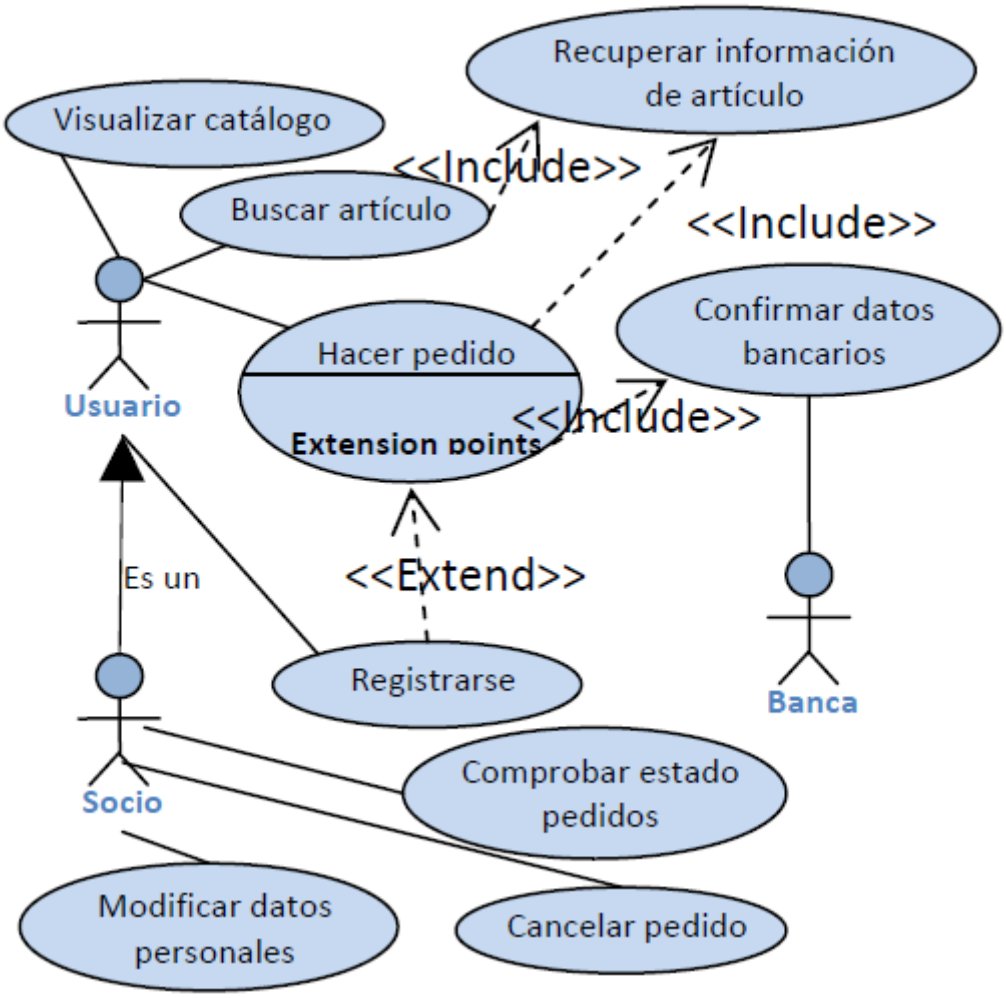


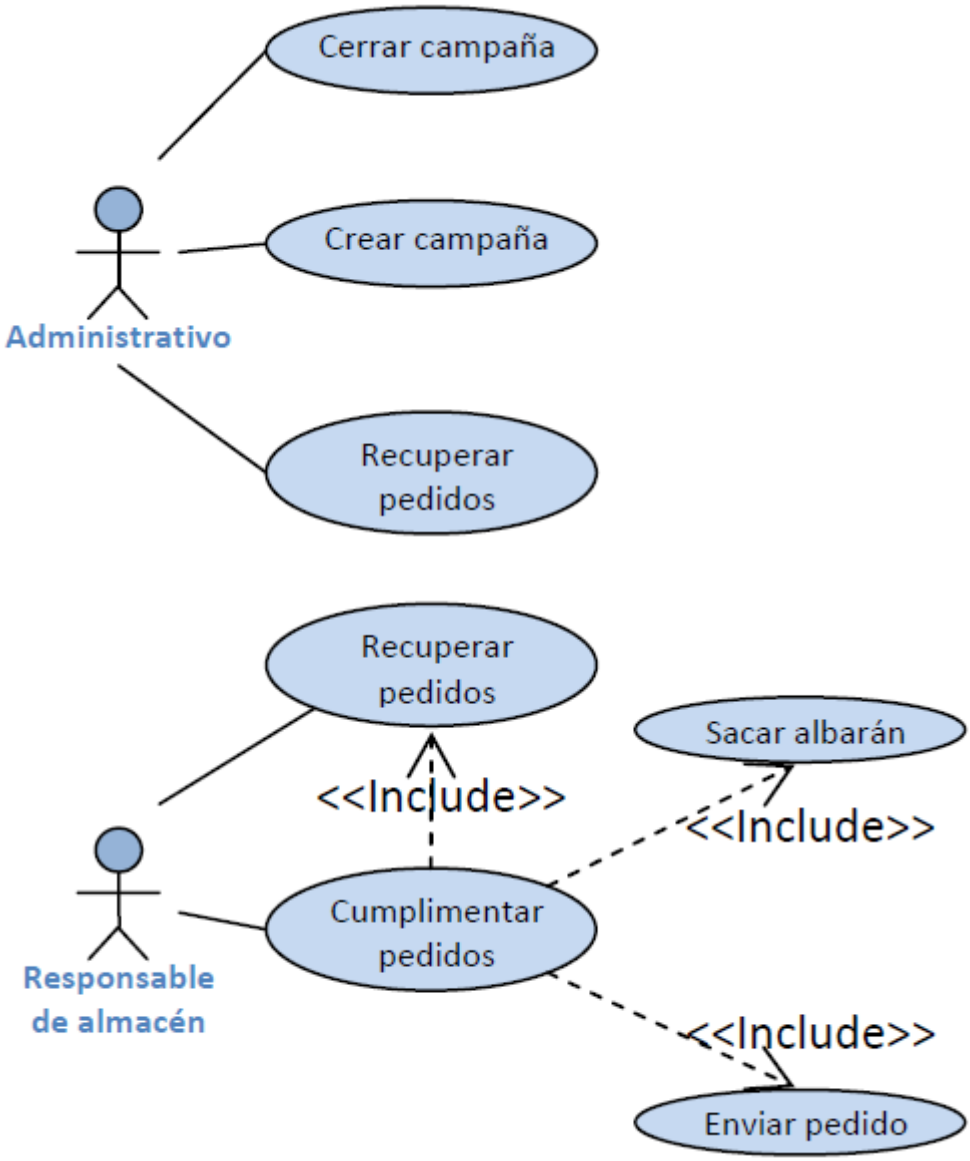
Elaboración de casos de uso

Identificar actores
Identificar funcionalidades
Funcionalidad de cada uno

Se parte de una descripción lo más detallada posible del problema, se detecta quién interactúa con el sistema para obtener los actores, se busca qué tareas realizan esos actores para descubrir casos de uso más genéricos. Se refina el diagrama añadiendo casos de uso más generarles para detectar casos por inclusión (aparecen en dos o más generales), exclusión y generalización. Al diagrama generado se le llama **diagrama frontera** (diagrama que incluye los casos de uso genéricos del sistema, que podrán ser desglosados en nuevos diagramas que los describan si es necesario. Se especifica enmarcando los casos de uso en un recuadro que deja a los actores fuera)


Cajero automático
Sacar dinero
Realizar transferencias
Hacer ingreso
Administrar cajero
Cliente
Sistema bancario
Administrador


Recuperar información de artículo
Visualizar catálogo
<<Include>>
Buscar artículo
<<Include>>
Confirmar datos bancarios
Hacer pedido
Extension points
<<Include>>
Usuario
Es un
<<Extend>>
Registrarse
Banca
Socio
Comprobar estado pedidos
Modificar datos personales
Cancelar pedido



Luego cada caso de uso debe documentarse (es lo más importante). Por ejemplo el caso de uso "Hacer pedido" (y sus posibles variantes):

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".

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.

Escenario

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

El caso de uso debe especificar un comportamiento deseado pero no imponer cómo se llevará a cabo (debe decir QUÉ pero no CÓMO).

Los escenarios se documentan en los diagramas de secuencia

4.2. Diagramas de secuencia

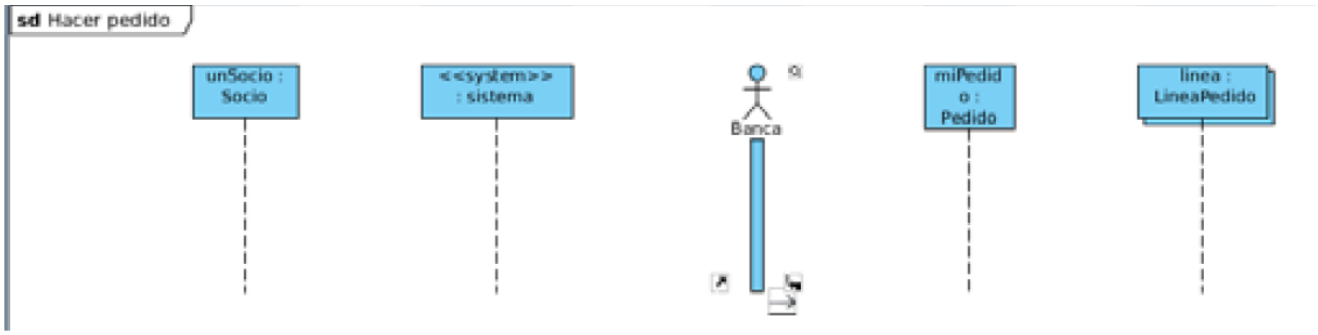
Los diagramas de secuencia formalizan la **descripción de un escenario representando mensajes que fluyen en el sistema, quién los envía y quién los recibe**.

(No se incluyen los eventos por ejemplo... aquí nos centramos en los mensajes)

Los **objetos y actores** del escenario se representan con **rectángulos distribuidos horizontalmente en la parte superior del diagrama** a los que se asocia una,**línea de vida**, línea punteada vertical que representa el paso del tiempo y que el objeto existe. De ella salen en orden los mensajes que se pasan entre ellos. (Así puede hacerse una idea el equipo de desarrollo de las diferentes operaciones que deben ocurrir al ejecutarse y el orden; de qué objetos participan en el caso de uso y cómo interaccionan a lo largo del tiempo)

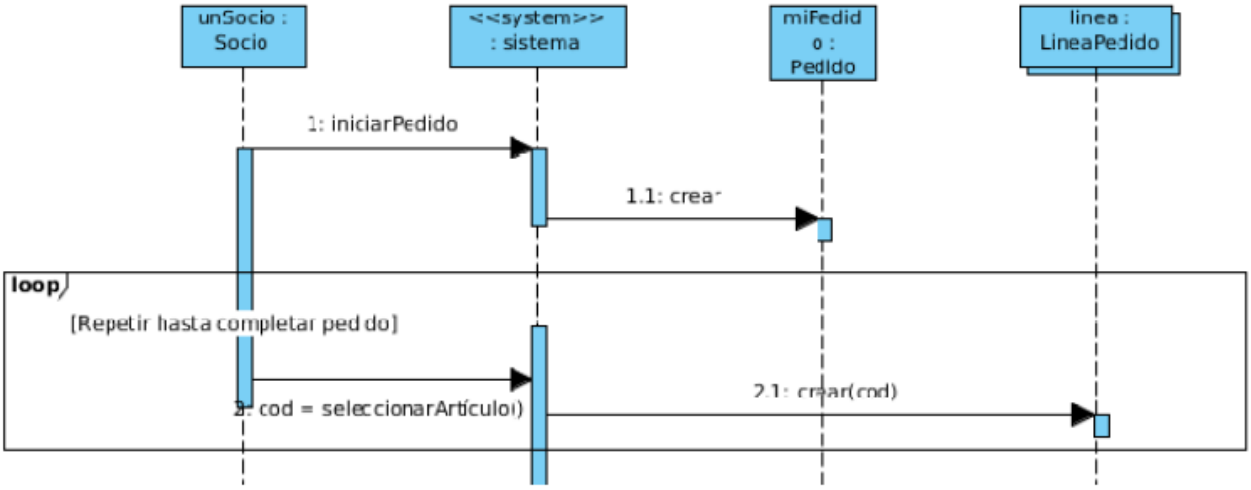
En el rectángulo de la línea de vida se anota el nombre del objeto (opcional), dos puntos y el nombre de la clase a la que pertenece.
Se usará el sistema para representar solicitudes al mismo como pulsar un botón para abrir una ventana o llamada a una subrutina.

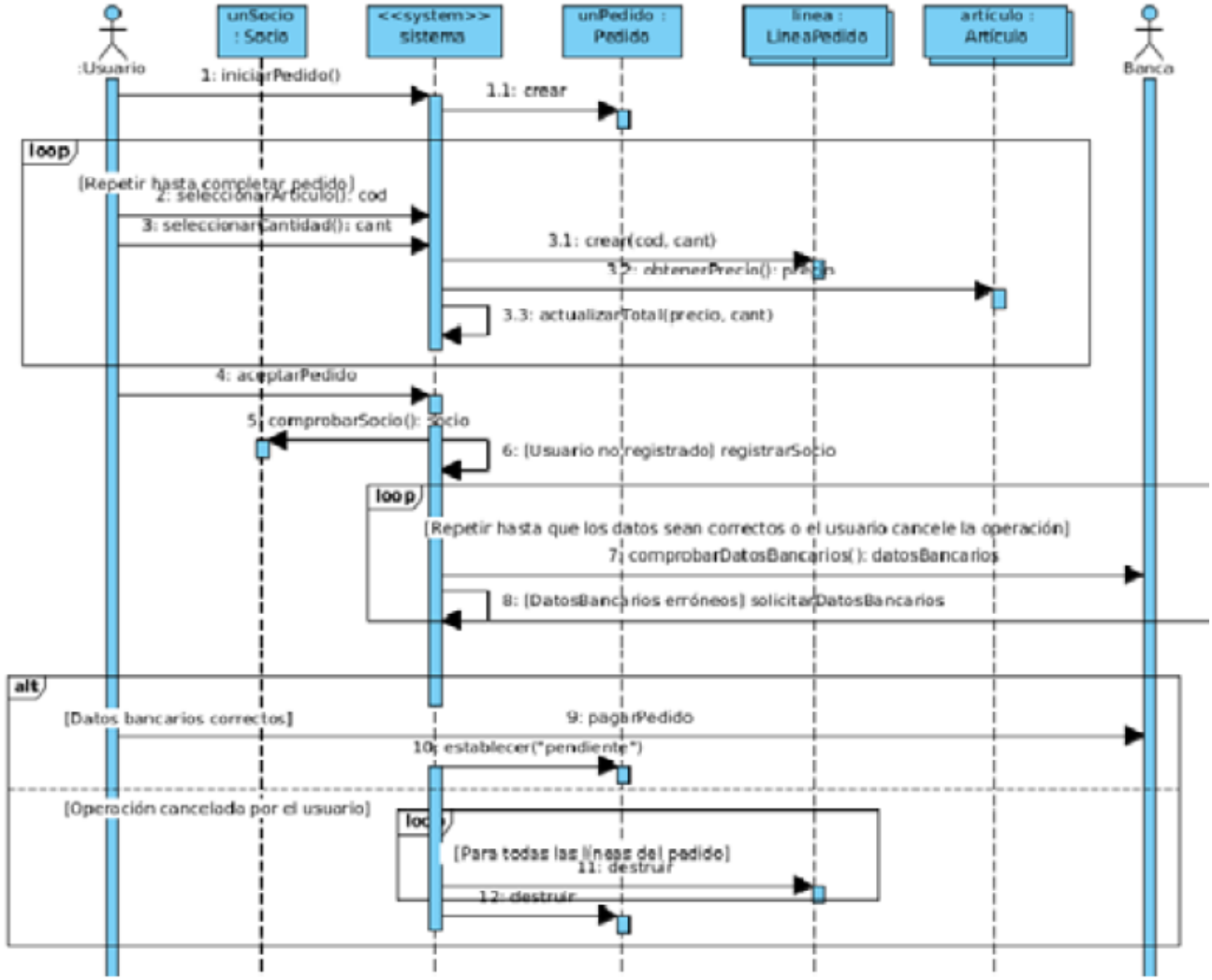
Si el objeto puede tener varias instancias aparecerá como un rectángulo sobre otro (como las líneas de pedido, que pueden ser varias)



La **invocación de métodos** (mensajes) son flechas horizontales que van de una línea de vida a otra, indicando la dirección del mensaje del que envía al que recibe. Ambos pueden ser el mismo objeto y su orden viene determinado por su posición vertical: mensaje debajo de otro indica que se envía después, no es necesario una secuencia.

Se pueden representar **iteraciones** usando marcos (normalmente se nombra el marco con el tipo de bucle a ejecutar y la condición de parada) y **condicionales** en función de un valor determinado,. También se pueden incluir etiquetas y notas en el margen izquierdo.





Elaborar diagramas de secuencia

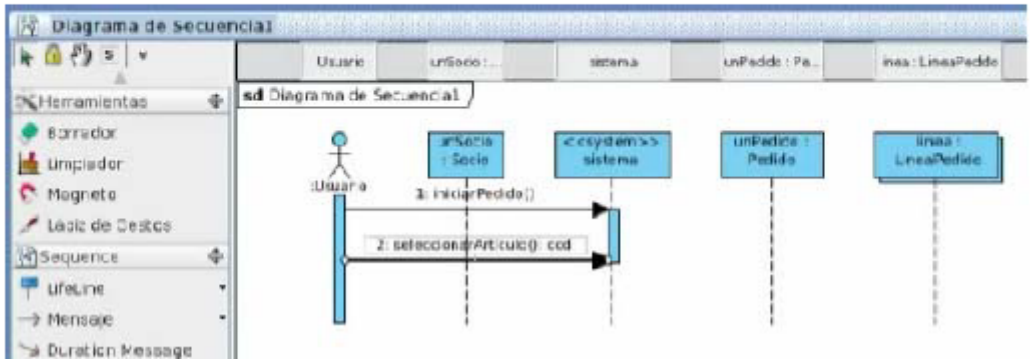
Incluir entidades participantes

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

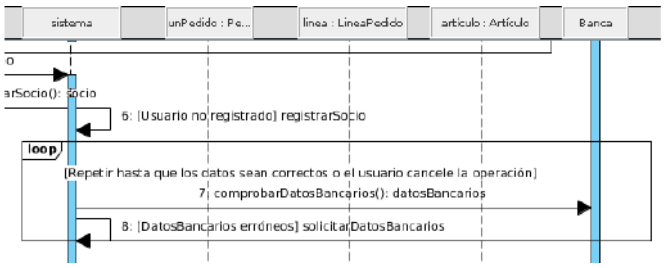
Objetos.
También necesitamos que intervengan cuatro objetos de nuestro sistema: un socio, un pedido con sus líneas de pedido, y los artículos a pedir.
Para **nombrar** estas entidades en la especificación indicamos su nombre y seleccionamos en el clasificador de base, que muestra la lista de clases del sistema la clase a la que pertenece, así aparce automáticamente su nombre en el formato adecuado.
Indicaremos que las líneas de pedido y los artículos son un conjunto de ojetos marcando en la especificación la casilla **"Multi-object"**, en la zona inferior del cuadro de diálogo.

Sistema.
Por último también interviene el sistema en si mismo, que utilizaremos para las operaciones relacionadas con la interfaz gráfica y las que se lanzan directamente para crear objetos, recuperar información del sistema como los datos de un artículo, o comunicarnos con entidades externas como la banca.
Se añade como una línea de vida más, a la que en su especificación indicamos que su nombre es sistema, y en la pestaña **Estereotipos** seleccionamos **System**.

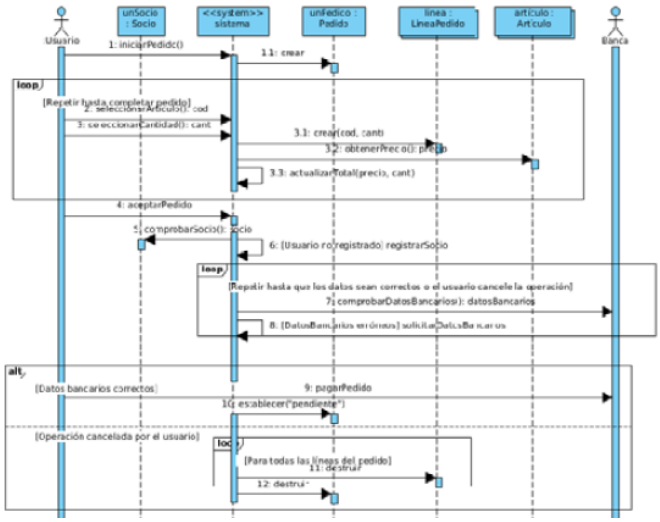
Los mensajes pueden devolver un valor, que podemos escribir en la especificación del mismo en la opción return value.



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.

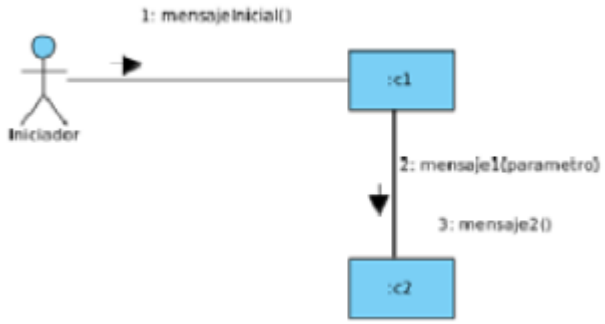


4.3. Diagramas de colaboración

Los diagramas de colaboración complementan a los de secuencia, mostrando las interacciones entre objetos del diagrama mediante el paso de mensajes entre ellos. Los nodos son objetos y las aristas enlaces entre objetos. Los objetos se conectan mediante enlaces y se comunican mediante los mensajes.

No se incluyen líneas de vida, sino que los mensajes se numeran para determinar su orden.

Por ejemplo: Actor iniciador manda mensaje a objeto c1 que inicia escenario; objeto c1 envía mensaje1 que lleva parámetro a objeto c2 y el mensaje2 al objeto c2.



En cierto modo, son semejantes a los de secuencia. (Representan misma información, representan entidades del sistema y los mensajes que circulan entre ellas y es fácil detectar la secuencialidad)

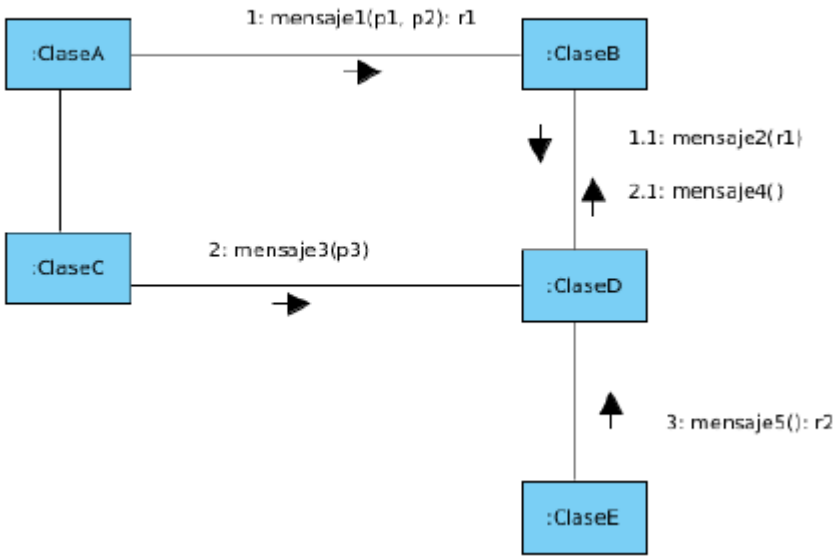
En los rectángulos aparece como "NombreClase" "NombreObjeto" ":nombreClase" (objeto genérico de la clase) "NombreObjeto:nombreClase"

Para que sea posible el paso de mensaje debe haber una asociación entre los objetos, quedando garantizada la navegación y la visibilidad entre ambos.

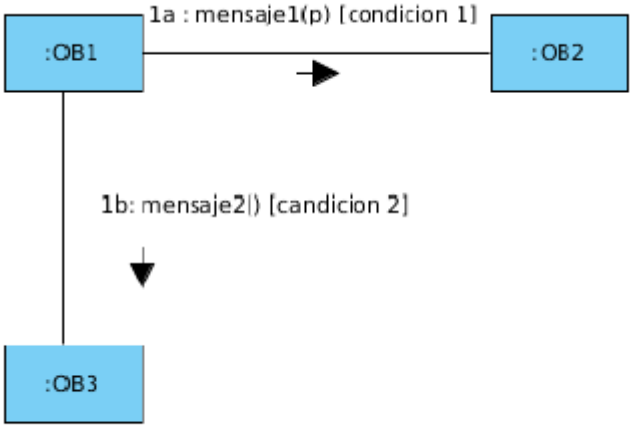
El mensaje es la especificación de una comunicación entre objetos que transmite información y desencadena una acción en el destinatario. La sintaxis es la siguiente:

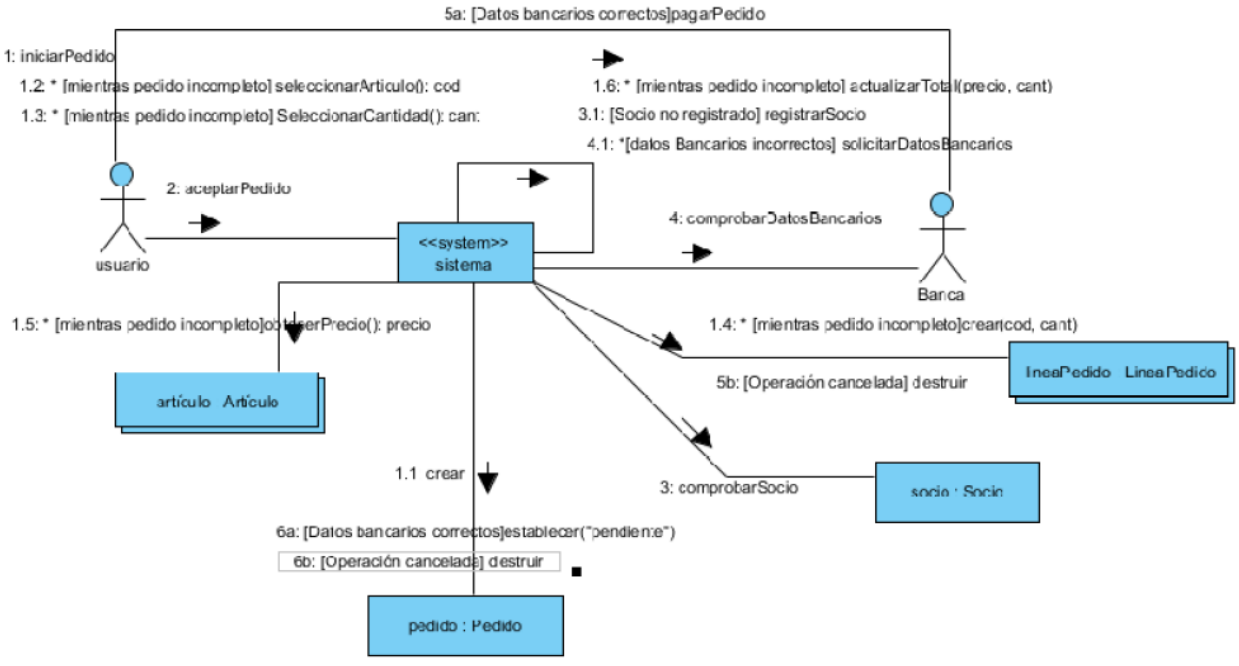
```
[secuencia][*][Condición de guarda]{valorDevuelto} : mensaje (argumentos)
```

Secuencia: Nivel de anidamiento del envío del mensaje dentro de la interacción. Mensajes se numeran para indicar el orden en que se envían y si es necesario se puede anidar con subrangos
* Mensaje iterativo
Condición de guarda: Debe cumplirse para que el mensaje sea enviado
ValorDevuelto: Lista de valores devueltos por el mensaje
Mensaje: Nombre del mensaje
Argumentos: Parámetros que se pasan al mensaje



Cuando hay una condición se repite el número de secuenmcia y se añaden las condiciones necesarias; enviándose uno u otro pero no ambos:





- Actividades que se repiten o pueden repetirse se marcan con asteriscos y su condición

- Condiciones de guarda se escriben en el mismo nombre que el mensaje
- Flujo alternativo de eventos si el usuario cancela el pedido o no, obliga a modificar la secuencia pasando a 4a 6a, 5b 6b.
- Objeto sistema con estereotipo system

4.4. Diagramas de actividad

Es una especialización del diagrama de estado, organizado respecto de las acciones. Se compone de actividades y representa cómo se pasa de una a otras. Las actividades se enlazan por transiciones automáticas (cuando termina una, se desencadena otra).
Es fundamental para el flujo de control entre actividades distinguiéndose cuáles se pueden llevar a cabo secuencialmente y cuáles conurrentemente. Define la lógica de control:

- En el modelado de los procesos del negocio
- Análisis de un caso de uso
- Compresión del flujo de trabajo a través de varios casos de uso
- Expresar aplicaciones multihilo.

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.

Elementos del diagrama

Estados

Estados de actividad: Elemento compuesto que se descompone en otros estados de actividad y de acción
Estados de acción: Estado que representa la ejecución de una acción atómica que no se puede descomponer ni interrumpir (invocación de operación). Se considera que la ejecución conlleva tiempo insignificante
Es posible definir **Estado inicial** y **Estado final**

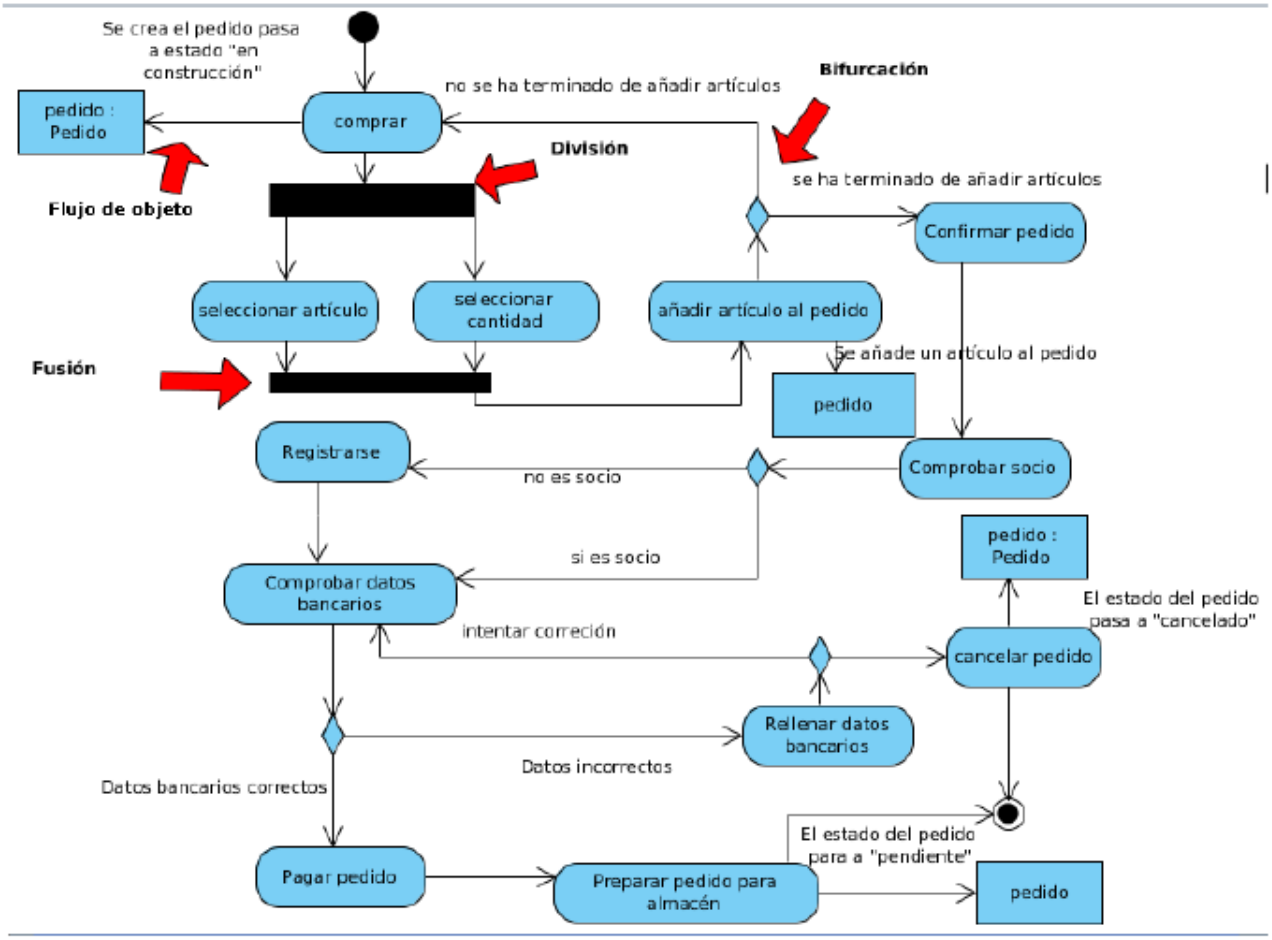
Transiciones

Relación entre dos estados que indica que un objeto en el primer estado hará ciertas acciones y pasará al segundo estado cuando ocurra un evento específico y satisfaga ciertas condiciones. Se representa por línea dirigida de un estado inicial al siguiente. Pueden encontrarse varias transiciones:

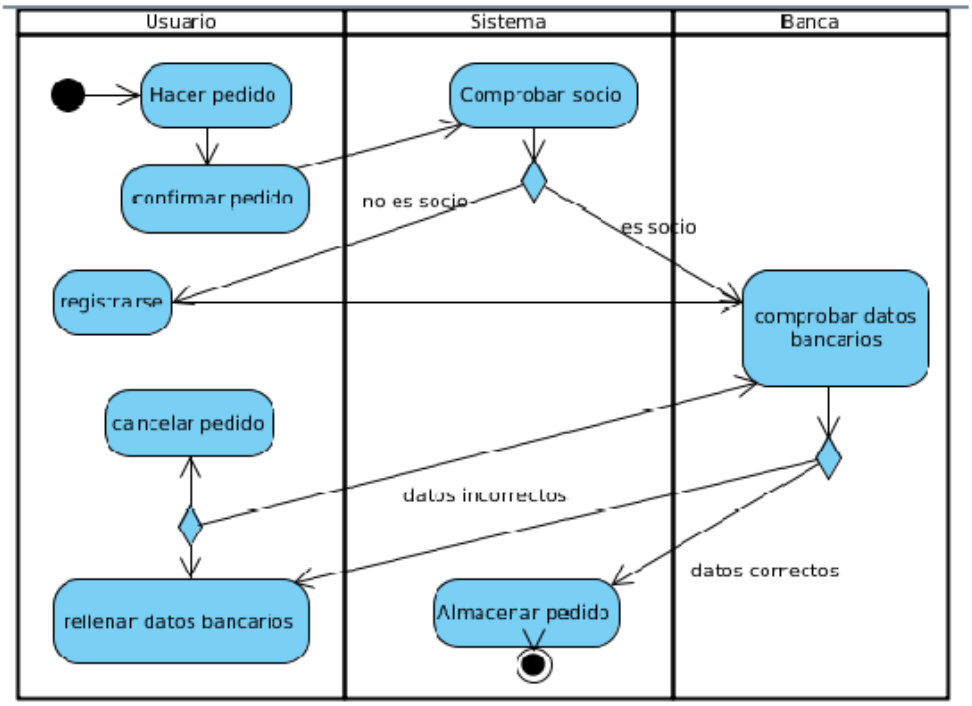
- **Secuencial** o sin disparadores: Al completar una acción se ejecuta la acción de salida y pasa al siguiente estado
- **Bifurcación** (Decision node). Caminos alternativos según el valor de una expresión booleana. Condiciones de salida no deben solaparse y deben cubrir todas las posibilidades (puede ejecutarse la palabra clave else)
- **Fusión** (Merge node): Redirigen varios flujos de entrada a un único flujo de salida. Sin espera ni sincronización.
- **División** (Fork node) Sincronización o ejecución paralela de actividades. Son concurrentes.
- **Unión** (Join node): Los flujos entrantes se sincronizan, es decir, cada uno espera hasta que han alcanzado la unión.

Objetos: Cuando interviene un objeto no se usan los flujos de eventos habituales sino flujos de objetos (representados con flechas) que permiten visualizar qué objetos están dentro del flujo de control asociados a un diagrama de actividades. Junto a ello puede indicarse cómo cambian los valores de sus atributos, su estado o sus roles.

Se usan **carriles o calles** para ver QUIENES son los responsables de hacer las distintas actividades (qué parte de la organización es responsable de una actividad): Cada calle tiene un nombre único. Puede ser implementada por una o varias clases. Las actividades de cada calle se considera independiente y se ejecutan concurrentemente a las de otras calles.



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:



4.5. Diagramas 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 puedan ocurrir.

Representan máquinas de estados (autómatas de estados finitos) para modelas 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.

Con el diagrama de estados se cumple que:

- Un objeto está en un estado concreto en un cierto momento determinado por los valores de sus atributos
- La transición de un estado a otro es momentánea y se produce con un 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 simultáneos se procesan como si se hubieran procesado en cualquier orden, sin pérdida de generalidad.

Estados

Estado: 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 E/S
- Actividad a realizar
- Subestados cuando el estado sea complejo y necesite diagrama
- Eventos diferidos
  Hay dos estados especiales el "estado inicial" (punto de partida 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 la instancia de clase y el "estado final" estado espacial 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.

Eventos

Un evento es un acontecimiento que ocupa un lugar en el tiempo y en el espacio y que funciona como un estímulo que dispara una transición en una máquina de estados. Existen eventos externos e internos, según quién los produzca.

Tipos de eventos:

- **Señales (Excepciones)**: La recepción de una señal (entidad a la que se le ha dado nombre, estereotipada, explícitamente prevista para la comunicación explícita), prevista para la comunicación explícita y asíncrona entre objetos. 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 receptor y puede disparar una transición a la máquina de estados
- **LLamadas:** Recepción de una petición para invocar una operación. Normalmente el evento de llamada es modelado como una operación del objeto receptor, manejado como un método del receptor e implementado como una acción o transición de la máquina de estados.

- **Paso del tiempo:** Representa el paso del tiempo (tiempo absoluto respecto de reloj real o virtual o paso de cantidad de tiempo dada desde que el objeto entra en el estado). Palabras clave: after(2 segundos); after 1 ms desde la salida del devInactivo...
- **Cambio de estado**: Representa el cambio en el estado o el cumplimiento de una condición. Palabras clave: when, seguida de una expresión booleana que puede ser de tiempo ode otra clase: when (hora = 11:30) when (altitud < 1000)

Transiciones

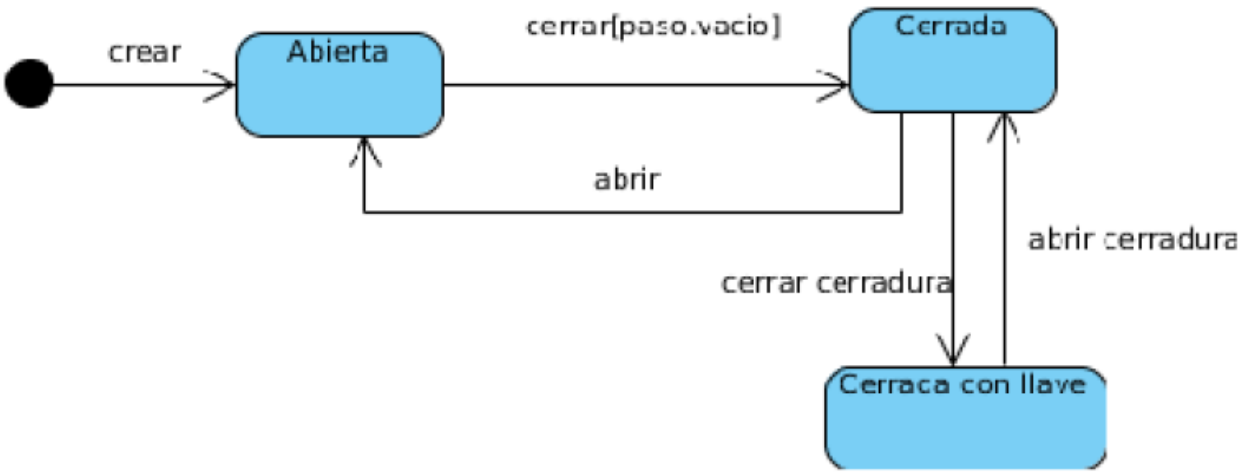
Una transición de un estado A a un estado B se produce cuando se origina un evento asociado y se satisface cierta condición especificada, en cuyo caso se ejecuta la acción de salida de A, la acción de entrada a B y la acción asociada en la transición.

La signatura de la transición es:

```
Evento(argumentos) [condicion] / accion
```

Elementos de una transición:

- **Estados origen y destino:** Se dispara la transición 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 eventos se vieron más arriba.
- **Condición de guarda:** Expresión booleana. Si es falsa, la transición no se dispara. Si no hay otra transición etiquetada con el mismo evento que pueda dispararse, este se pierde.
- **Acción**: computación atómica ejecutable. Puede incluir llamadas a operaciones del objeto que incluye la máquina de estados (o sobre otro visibles), creación o destrucción de objetos o envío de una señal a otro objeto.



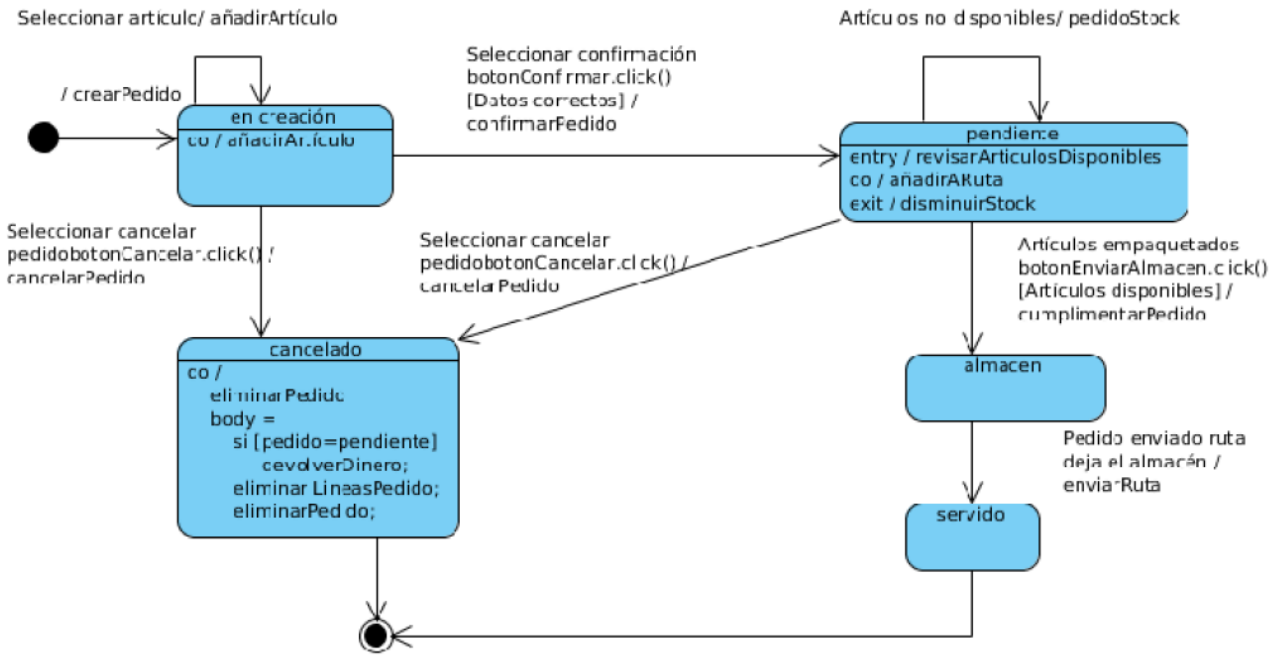
Aquí `cerrar[paso.vacio]` implica que para cerrar la puerta el paso debe estar vacío. No se podrá cerrar hasta que no lo esté.

Vamos a ver el diagrama de estados asociado a un objeto "pedido" ya que este tiene un comportamiento significativo en tiempo real, su situación física y en el sistema va evolucionando con el tiempo y participa en varios casos de uso.

Se definen como estados del pedido:

- En creación
- Pendiente
- En almacén
- Servido
- Cancelado

Las transiciones entre estados se producen por llamadas a procedimientos, no interviniendo ni cambios de estado o tiempo, ni señales:



5. Ingeniería inversa

Proceso de analizar código, documentación y comportamiento de una aplicación para identificar sus componentes actuales y dependencias para extraer y crear una abstracción del sistema e información del diseño. No se altera el sistema en estudio, solo se produce un conocimiento adicional del mismo.

Tipos de ingeniería inversa:

- **Ingeniería inversa de datos**: Sobre código de bases de datos (aplicación SQL) para obtener modelos relacionales o sobre el modelo relacional para obtener el diagrama E-R.
- **Ingeniería inversa de lógica o de proceso**: Sobre el código de un programa para averiguar su lógica (reingeniería) o sobre un documento de diseño para obtener documentos de análisis o de requisitos
- **Ingeniería inversa de interfaces de usuario**: Busca mantener la lógica interna del programa para obtener modelos y especificaciones que sirvieron de base para la construcción de la misma, para tomarlas como punto de partida en ingeniería directa para modificar la interfaz.

En herramientas como VP-UML se le daría a "Update UML Model" o a Herramienta >> CodigoI >> Reverse Code, indicándole dónde se localiza el código fuente. Obtiene clases y relaciones de herencia, el resto de relaciones deben establecerse a mano.