**UF-1**

1. **¿Encuentras alguna diferencia entre el freeware y el software libre? Explícalas.**

Ambos se distribuyen sin coste, la diferencia está en que el software libre permite la modificación, mejora y redistribución del código fuente mientras que el freeware no, puede incluso no incluir el código fuente del programa.

1. **¿Cuáles son las etapas del ciclo de vida del software?**

En total son 10.

Definición de objetivos

Análisis de los requisitos y su viabilidad

Diseño general

Diseño en detalle  
Programación   
Prueba de unidad  
Integración  
Prueba beta   
Documentación  
Implementación   
  
DADDPPIPDI

1. Técnicas de comunicación con el cliente

* **Entrevistas.** Técnica tradicional en la que hablamos con el cliente.
* **Desarrollo conjunto de aplicaciones (JAD).** Entrevista de dinámica de

grupo en la que cada uno tiene un rol (usuarios, administradores,

desarrolladores, analistas, etc).

* **Planificación conjunta de requisitos** (JRP). Subconjunto de JAD, y se

dirigen a productos de alto nivel.

* **Brainstorming.** Reuniones en las que se intentan crear ideas desde

distintos puntos de vista. Adecuada para el comienzo del proyecto.

* **Prototipos:** Versión inicial del sistema en el que se puede ver el

problema y sus posibles soluciones. Se puede desechar o usar para

añadir más cosas.

* **Casos de uso.** Técnica definida por UML, y se basa en la

representación de lo que queremos que haga el sistema.

Representan requisitos funcionales del mismo sistema. Describe qué

hace el sistema, pero no cómo lo hace.

1. Modelos de flujo, descripción, inconvenientes y ventajas.

* **Modelo en Cascada:** las etapas del ciclo de vida siguen un orden, de tal forma que para cambiar de fase se tiene que haber finalizado la anterior.

**Ventajas:**

* Fácil de comprender, planificar y seguir
* La calidad del producto resultante es alta.
* Permite trabajar con con personal poco cualificado

**Inconvenientes:**

* Necesidad de tener todos los requisitos definidos desde el principio.
* Difícil volver atrás si se cometen fallos en una etapa.
* El producto no está disponible para su uso hasta que no está completo.

**Se recomienda cuando**

* El proyecto a realizar es parecido a alguno realizado anteriormente.
* Los requisitos son estables y están bien comprendidos.
* Los clientes no necesitan versiones intermedias.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

* **Modelo iterativo incremental:** se compone de varios ciclos de vida en cascada realimentados, en el que cada vez que se realiza el ciclo completo se crea un incremento.

**Ventajas:**

* No es necesario conocer los requisitos al comienzo.
* Permite la entrega de partes del proyecto operativas al cliente.
* Las entregas facilitan la realimentación de los próximos entregables.

**Inconvenientes:**

* Es difícil estimar el esfuerzo y el coste final necesario.
* Se tiene el riesgo de no acabar nunca.
* No recomendable con desarrollar sistemas en tiempo real, con altos riesgos, de alto nivel de seguridad o de procesamiento distribuido.

**Se recomienda cuando**:

* Los requisitos no están definidos y es posible que haya grandes cambios.
* Se están probando o introduciendo nuevas tecnologías.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

* **Modelo espiral:** se combina el modelo en cascada con el modelo iterativo. Cada ciclo de la espiral está formado por 4 fases (determinar objetivos, identificar y resolver riesgos, desarrollar y probar, planificación), y cuando se termina el ciclo, se produce una versión como en el modelo iterativo.

**Ventajas:**

* No requiere una definición completa de los requisitos para empezar a funcionar.
* Análisis de riesgo en todas las etapas
* Reduce riesgos del proyecto.
* Incorpora objetivos de calidad.

**Inconvenientes:**

* Es difícil evaluar los riesgos.
* El costo aumenta a medida que la espiral aumenta el número de iteraciones.
* Depende mucho de la fase de análisis de riesgos.

**Se recomienda cuando**:

* Proyectos de gran tamaño y que necesitan constantes cambios.
* Proyectos donde sea importante el factor riesgo.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

* **Modelos evolutivos:** se van a ir realizando iteraciones sobre el ciclo de vida del software, permitiendo entregar versiones en cada una de estas iteraciones, mientras se van añadiendo mejoras.

**Ventajas:**

* La especificación se puede desarrollar de forma creciente.
* Los usuarios y desarrolladores logran un mejor entendimiento.
* Es más efectivo que el modelo de cascada.

**Inconvenientes:**

* Proceso no visible.
* Sistema pobremente estructurado.
* Se requieren técnicas y herramientas.

1. **Tipos de requisitos, características**

• **Requisitos funcionales.** Nos describe con detalle la función que realiza

el sistema, la reacción ante determinadas entradas y cómo se

comporta en distintas situaciones, etc.

• **Requisitos no funcionales**. Va a tratar sobre todas las características

del sistema.

1. **¿El lenguaje Java es un lenguaje compilado o interpretado? Justifica tu respuesta.**

El **lenguaje Java** es **compilado** e **interpretado**. Los dos procesos se realizan en fases distintas: El compilador (javac) sólo tiene que estar en la plataforma de desarrollo, y el intérprete (**java**) tiene que estar en todos los clientes que quieran ejecutar el applet.

1. **¿Qué es un entorno de desarrollo?**

Es una aplicación informática que proporciona servicios integrales para facilitarle al desarrollador o programador el desarrollo de software.

Normalmente, un IDE consiste de un editor de código fuente, herramientas de construcción automáticas y un depurador. La mayoría de los IDE tienen auto-completado inteligente de código (*IntelliSense*). Algunos IDE contienen un compilador, un intérprete, o ambos, tales como NetBeans y Eclipse

1. **¿Qué es una licencia de software? Explica los distintos tipos de software según su licencia, trabajo y distribución.**

**Una licencia de software** es un contrato entre el licenciante (autor/titular de los derechos de explotación/distribuidor) y el licenciatario del programa informático (usuario consumidor /usuario profesional o empresa), para utilizar el software cumpliendo una serie de términos y condiciones establecidas dentro de sus cláusulas.

**Según el tipo de trabajo,** de sistema, de aplicación y de programación.

**Según el método de distribución:** Shareware , freeware, adware, multimedia y de uso específico.

**Según su Licencia**: Libre,propietario, de dominio público.

**Adware** programa que automáticamente ofrece publicidad no deseada

**El Propietario,** no tiene limitaciones, cuesta dinero, no se puede distribuir, no da el código fuente ni se puede modificar de manera comercial o sin ánimo de lucro.

**Shareware o de evaluación:** esta limitado por tiempo o no tiene el 100% de las características, es gratis, se puede distribuir (hacer copias), , no da el código fuente ni se puede modificar de manera comercial o sin ánimo de lucro.

**Freeware:** sin limitaciones, gratis, se puede distribuir, no da el código fuente ni se puede modificar de manera comercial o sin ánimo de lucro.

**Semilibre:** sin limitaciones, gratis, se pueden hacer copias, se puede modificar el código fuente solo si no es para fines comerciales.

**Libre:** Sin limitaciones, gratis, se puede distribuir, y se puede modificar el código fuente para fines sin ánimo de lucro o comerciales.

1. **Explica con tus palabras el funcionamiento de un compilador.**

Un compilador traduce el código fuente en código objeto, también se encarga de detectar errores en el código fuente, pero no los corrige solo los detecta para que los corrijamos y poder crear el código objeto.

El compilador analiza el código fuente de 6 maneras diferentes,

-**Análisis léxico**: Encargado de identificar las palabras reservadas.

-**Análisis sintácticas**: Comprueba que se cumplan las reglas sintácticas.

-**Análisis semántico**: Comprueba las declaraciones del código fuente sean correctas.

-**Generación de código intermedio**: analiza el código al completo para facilitar la traducción a código objeto.

-**Optimización de código**: mejora del código generado.

-**Generación de código**: se crea el código objeto.

1. **Explica las ventajas y desventajas de la máquina virtual de java, tipos de máquinas virtuales**

Hay dos tiposde **máquinas virtuales, de sistema** permite virtualizar maquinas con distintos sistemas operativos en cada una.

**Máquinas virtuales del** proceso, se ejecuta como un proceso normal del SO y solo soporta un proceso.

Las ventajas y desventajas de la máquina virtual de java son las siguientes.

**Ventajas:**

Portabilidad. Los lenguajes de programación ejecutados en una máquina virtual son independientes de la plataforma, por lo que podemos ejecutar estos programas en cualquier sistema operativo. Permite realizar aplicaciones multiplataforma.

**Desventajas:**

- Son más lentos que los lenguajes compilados.

1. **¿Cuáles son los elementos de los lenguajes de programación?**

Consta de tres elementos; **alfabeto, sintaxis y semántica.**

- El alfabeto es el conjunto de símbolos permitidos.

- La sintaxis son las reglas para realizar las construcciones con los símbolos.

- La semántica son las reglas que determinan el significado de las construcciones realizadas.

1. **Realiza la clasificación de los siguientes lenguajes: C, Java, C#, JavaScript, PHP, Prolog.**



1. **¿Qué es un intérprete?**

Programa con la misma funcionalidad que un compilador, con la diferencia del procesamiento que realiza en el código fuente, pero la traducción se realiza instrucción a instrucción, según lo va leyendo.

1. **Realiza un resumen/esquema de las fases del desarrollo de software.**

El ciclo de vida del software se compone de una serie de fases, tratando cada uno de los aspectos del desarrollo de una aplicación.

**Fase 1: Análisis**

Es la tarea más importante del proceso, puesto que en esta fase se recopilan los requisitos necesarios que la aplicación debe solucionar. Para la obtención de estos requisitos podemos emplear varias técnicas: entrevistas, prototipos, brainstorming, casos de uso..

**Fase 2: Diseño**

Es la fase en la que se traducen los requisitos recopilados durante el análisis. Se piensa el modo en que se llevará a cabo la resolución del problema, por ejemplo, los diagramas de flujo, de cajas, tablas de decisión o el pseudocódigo.

**Fase 3: Codificación**

En esta fase se transforman los requisitos en el lenguaje de programación anterior. Es conveniente seguir algunas pautas para un mejor mantenimiento posteriormente, como es el uso de comentarios, la nomenclatura de los diferentes elementos de programación y la organización de los ficheros que forman parte del proyecto.

**Fase 4: Pruebas**

Es la fase en la que se detectan los errores, tanto de la parte de codificación como en la especificación del diseño.

**Fase 5: Documentación**

Durante el desarrollo es necesario llevar a cabo el registro de toda la información necesaria del proceso y del producto. También es necesario realizar el manual de usuario, que explica cómo utilizar este producto.

**Fase 6: Explotación**

En esta fase es necesario llevar a cabo la instalación de la aplicación creada, de esta forma, el cliente ya puede utilizar el producto.

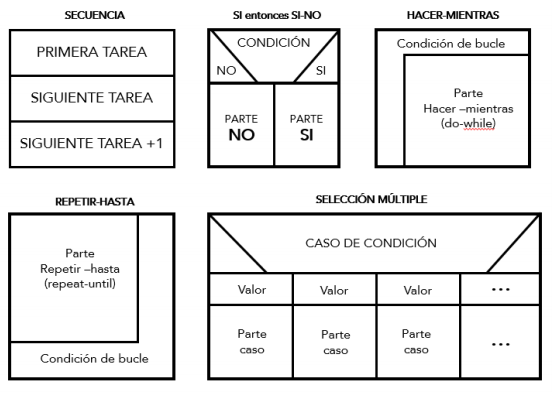
**Fase 7: Mantenimiento**

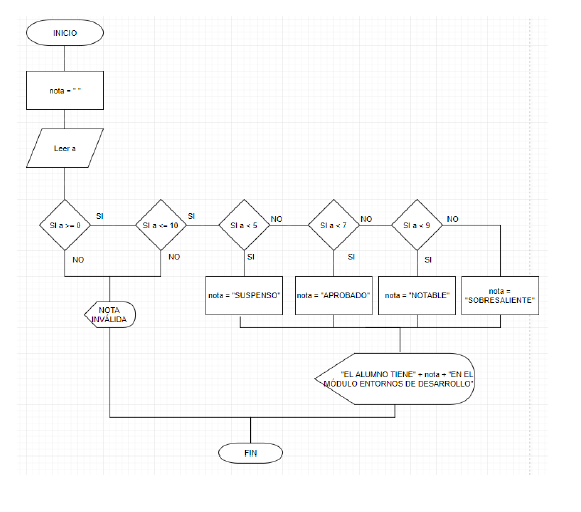
Durante esta fase se lleva a cabo la modificación del producto para corregir los fallos. Se deben tener en cuenta los distintos tipos de mantenimiento que existen; adaptativo, correctivo, perfectivo y preventivo.

1. **Realiza el pseudocódigo y el diagrama de flujo para que, dada una nota entre 0 y 10, el programa muestre su nota en modo texto.**

Simbología de los diagramas de flujo.

|  |  |
| --- | --- |
| Procesos |  |
| Flujos de datos |  |
| Almacén de datos |  |
| Entidades externas |  |
| Terminador: Inicio y final de un programa. |  |
| Símbolo de Entrada/Salida, representa entrada y salida de datos. |  |
| Pantalla: se utiliza para representar salida por pantalla. |  |
| Impresora: Se usa como símbolo de entrada (documento de entrada) y salida (impresora) |  |
| Teclado se utiliza como símbolo de entrada por teclado. |  |
| Conector, se utiliza para unir una parte del diagrama con otra. |  |
| Llamada a subrutina o procedimiento |  |
| Disco magnético, representa una función de entrada/salida para soporte en un disco magnético. |  |
|  |  |



****

1. **¿Qué es un IDE?**

Un IDE es una aplicación informática cuya función es simplificar la tarea del programador mediante un conjunto de herramientas.

1. **Enumera los componentes de un IDE.**

Editor de texto. Parte en la que escribimos el código fuente.

** Compilador**. Se encarga de traducir el código fuente escrito en lenguaje

de alto nivel a un lenguaje de bajo nivel en el que la máquina sea capaz de

interpretarlo y ejecutarlo.

 **Intérprete o interpretador**. Realiza la traducción a medida que se ejecuta

la instrucción. Son más lentos que los compiladores, pero no dependen de

la máquina sino del propio intérprete.

 **Depurador (Debugger).** Depura y limpia los errores en el código fuente.

Permite detener el programa en cualquier punto de ruptura para

examinar la ejecución.

 **Constructor de interfaz gráfica.** Simplifica la creación de interfaces

gráficas de usuario permitiendo la colocación de controles usando un

editor WYSIWYG de arrastrar y soltar.

 **Control de versiones.** Controla los cambios realizados sobre las

aplicaciones, obteniendo así revisiones y versiones de las aplicaciones en

un momento dado.

1. **¿Qué es un plugin?**

Un plugin es un complemento software que añade funcionalidades al IDE.

1. **Investiga 4 IDEs diferentes y 2 plugin, y explica sus posibles funciones**

**IDE**

**- IntelliJ IDEA**: IDE multiplataforma, que permite trabajar con lenguajes como Java, JavaScript, HTML, XSS, XML o Ruby, por lo que permite crear con cualquier tipo de aplicaciones con estos lenguajes.

**- JDeveloper**: IDE multiplataforma, que permite trabajar con lenguajes como Java, XML, PL/SQL, JavaScript, PHP o UML, y permite crear cualquier tipo de aplicaciones con estos lenguajes.

**- NetBeans**: IDE multiplataforma, que permite trabajar con lenguajes como Java, C, C++, PHP, XML, HTML, y permite crear cualquier tipo de aplicaciones con estos lenguajes.

**- Microsoft Visual Studio**: IDE de Windows, que permite trabajar con lengaujes C#, C++, Java, .NET, Python, PHP o Visual Basic, y permite crear cualquier tipo de aplicaciones con estos lenguajes.

**Plugin**

**- AdobeFlashPlayer** -> para los navegadores web

- **WooCommerce ->** para WordPress

1. **Pasos para la instalación de Eclipse junto con el plugin WindowBuilder.**

Aceptamos la licencia,una vez instalado, lo lanzamos y seleccionamos la carpeta donde se guardarán los proyectos.Pantalla de bienvenida. Accedemos a la ventana principal pulsando en Workbench.

Ventana principal de trabajo.

Help->Install New Software.

Aparece una ventana donde añadimos el link de descarga http://download.eclipse.org/windowbuilder/WB/release/R201506241200-1/4.5/ en el campo work with, y pulsamos el botón Add.

Después seleccionamos todos los elementos de WindowBuilder y click en Next. Aceptamos los términos de la licencia y click en Finish.

Una vez instalado WindowBuilder creamos un nuevo proyecto java desde File->New>Java Proyect.

Después click botón derecho sobre el proyecto y seleccionamos New->Other.

En la ventana siguiente: WindowBuilder->Swing Designer->Application Window y next.

Finalmente elegimos el nombre de la aplicación, el package y finish.

**UF-2**

1. **¿Qué diferencias existen entre pruebas de caja blanca y pruebas de caja negra?**

Las pruebas de caja blanca o estructurales se centran en validar los requisitos funcionales especificados, comprobando la salida que muestra respecto a las diferentes entradas de datos posibles. Se encarga de ejecutar todos los caminos independientes que se encuentran en el programa.

Las pruebas de caja negra o de comportamiento se centran en el proceso, es decir, en el funcionamiento interno del programa. Su función es encontrar los diferentes errores que pueden surgir.

**```**

1. **Tipos de pruebas, estrategia de pruebas, descripción y errores que buscan evitar.**

Un caso de prueba es un conjunto de entradas, condiciones de ejecución y resultados esperados, desarrollado para conseguir un objetivo particular o condición de prueba.

Prueba llevar a cabo el diseño de casos de pruebas se utilizan dos técnicas que no son excluyentes ósea, se complementan.

**-Prueba de caja negra blanca:** Conocido como pruebas estructurales o de caja de cristal, se basan en el minucioso examen de los detalles procedimentales del código de la aplicación, fijándose en el funcionamiento interno del programa.

**-Prueba de caja negra:** Se llevan a cabo sobre la interfaz del software no hace falta conocer la estructura interna del programa ni su funcionamiento pues se centran más en las funciones, entradas y salidas del programa, se pretende obtener casos de prueba que demuestren que las funciones internas del software sean operativas.

**La estrategia de pruebas** de software se compone de un ciclo de cuatro tipos de pruebas: unidad, integración, validación y sistema. Se deben usar las 4 tipos de prueba en el orden:

**Las pruebas de unidad** se centran en comprobar la funcionalidad de cada uno de los módulos, con el objetivo de eliminar errores en la interfaz y en la lógica interna.

**Las pruebas de integración** se realizan cuando se juntan estos módulos, y se comprueba que no haya conflicto entre los módulos y que el diseño es correcto.

**Hay dos enfoques:**

Prueba de **integración no incremental:** También llamado big bang, prueba los módulos por separado y después todos juntos

En la **Incremental** el programa completo se va construyendo y probando en pequeños segmentos, en este caso es más fácil localizar los errores Hay dos estrategias **ascendente y descendente** que influye desde donde se comenzara a construir y probar los módulos, en el descendente desde el modulo principal y el ascendente desde el ultimo modulo.

**Las pruebas de validación** se encargan de comprobar que los requisitos del documento ERS coinciden con los desarrollados**.** Se hacen en el entorno real de trabajo.

**Hay dos técnicas:**

Prueba Alfa llevado a cabo por el cliente o usuario en el lugar de desarrollo, el desarrollador ira observando y registrando los errores y problemas de uso.

Prueba Beta llevado a cabo por los usuarios finales, el desarrollador no está presente, es el usuario quien informa de los problemas al desarrollador para que los arregle y saque una nueva versión.

**Las pruebas de sistema** comprenden diferentes tipos de pruebas, entre las que se encuentran la funcionalidad, el rendimiento, la recuperación y la seguridad entre otras, su función es ejercitar profundamente el software.

**Hay tres pruebas:**

**Recuperación:** Se fuerza el fallo y comprueba que se recupere correctamente.

**Seguridad:** Se comprueba que este protegido contra accesos ilegales.

**Resistencia:** Trata de enfrentar al sistema con situaciones que demandan gran cantidad de recursos.

**Pruebas de código** consiste en ejecutar el programa o parte de el en busca de errores

1. **¿Qué es documentar el código de un programa?**

Documentar el código de un programa es añadir suficiente información como para explicar lo que hace, punto por punto, de forma que no sólo los ordenadores sepan qué hacer, sino que además los humanos entiendan qué están haciendo y por qué.

Hay que añadir explicaciones a todo lo que nos evidente, no hay que explicar lo que se hace, si no explicar por qué se hace.

1. **Partes de un proyecto.**

**Refactorización:** nos permite optimizar el código , realizando cambios sin que altere el comportamiento final del producto

**Control de versiones:** Control de versiones es la capacidad de poder recordar todos los cambios que se han realizado tanto en la estructura de directorios como en el contenido de los archivos.

**Documentación:** Es el texto escrito que acompaña a los proyectos.

1. **Qué tipos de documentación podemos encontrar en un proyecto? Nómbralos y pon una breve descripción.**

**Son tres documentos y siguen este orden de creación:** plan de pruebas, especificaciones de pruebas y informes de prueba.

**Plan de pruebas** describe elalcance, enfoque, recursos y el calendario de las actividades de prueba, Identifica las características a probar, tareas que se van a realizar, personal responsable de cada tarea y riesgos asociados.

**Especificaciones de prueba:** cubierto por tres tipos de documento, especificación del diseño de prueba, especificación de diseño de prueba y la especificación de los procedimientos.

1. **Complejidad ciclomatica y numero de caminos.**

Se calcula, midiendo el número de regiones del grafo, restando el número de aristas con el número de nodos +2 y mirando el número de nodos predicados +1. El resultado de las 3 operaciones debe salir igual. Si el número resultante esta entre 1 y 10, no existe mucho riesgo.

El número de caminos es la cantidad de recorridos diferentes que puede hacer el programa y finalizar correctamente.

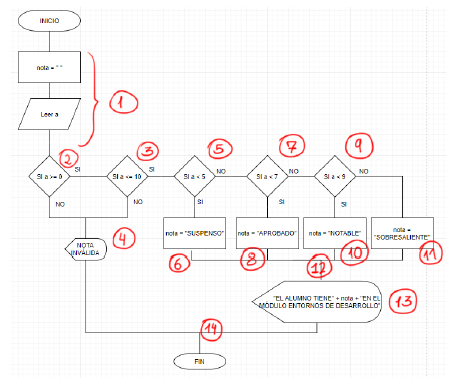
Un nodo es una instrucción.

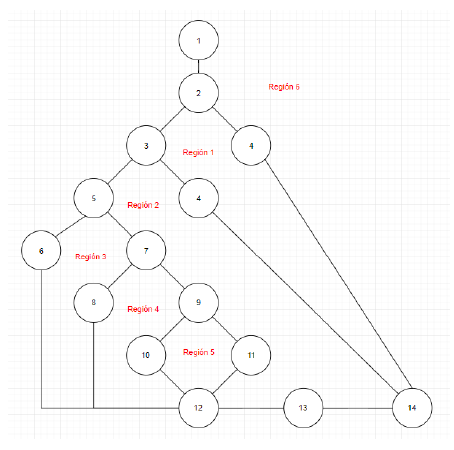
Los nodos predicados, son los nodos que tienen diferentes aristas para recorrer(diferentes opciones)

La arista es el recorrido entre un nodo y otro.

La región son los diferentes espacios que se crean entre las aristas y los nodos al terminar el grafo

**Grafo de un diagrama de flujo.**





**Número de nodos => 15**

**Número de aristas => 19**

**Número de regiones => 6**

**Número de nodos predicados => 5 (2,3,5,7,9)**

**Complejidad ciclomática**

**V(G) = Número de regiones del grafo = 6**

**V(G) = Número de aristas – Número de nodos + 2 = 19 – 15 + 2 = 6**

**V(G) = Número de nodos predicados + 1 = 5 + 1 = 6**

**Como la complejidad ciclomática está entre 1 y 10, no existe mucho riesgo, el programa es sencillo.**

**Caminos:**

**1. 1-2-4-14**

**2. 1-2-3-4-14**

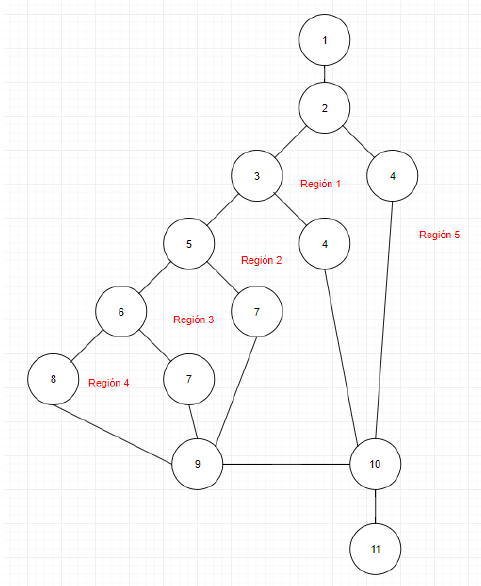
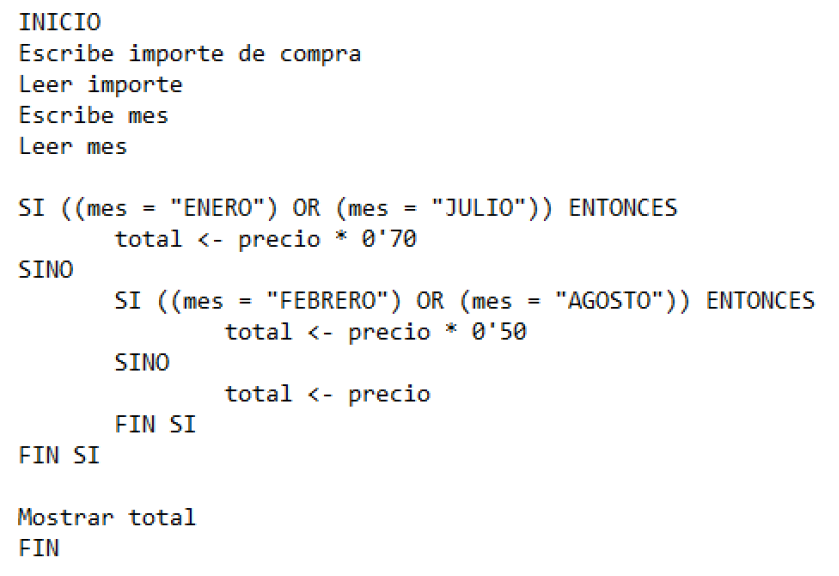
**3. 1-2-3-5-6-12-13-14**

**4. 1-2-3-5-7-8-12-13-14**

**5. 1-2-3-5-7-9-10-12-13-14**

**6. 1-2-3-5-7-9-11-12-13-14**

**Grafo de flujo de un pseudocodigo.**



**Número de nodos => 13**

**Número de aristas => 16**

**Número de regiones => 5**

**Número de nodos predicados => 4 (2,3,5,6)**

**Complejidad ciclomática**

**V(G) = Número de regiones del grafo = 5**

**V(G) = Número de aristas – Número de nodos + 2 = 16 – 13 + 2 = 5**

**V(G) = Número de nodos predicados + 1 = 4 + 1 = 5**

**Como la complejidad ciclomática está entre 1 y 10, no existe mucho riesgo, el programa es sencillo.**

**Caminos:**

**1. 1-2-4-10-11**

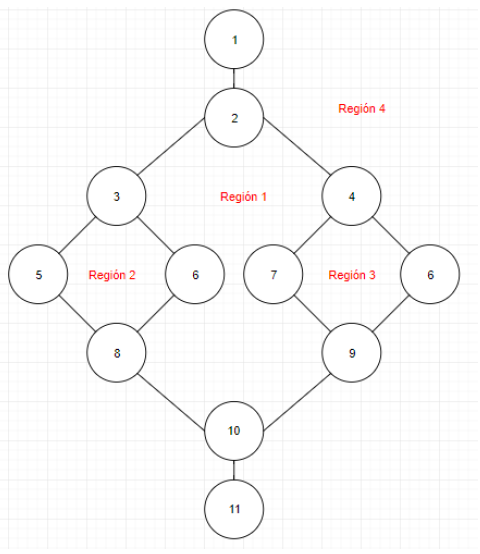
**2. 1-2-3-4-10-11**

**3. 1-2-3-5-7-9-10-11**

**4. 1-2-3-5-6-7-9-10-11**

**5. 1-2-3-5-6-8-9-10-11**

1. **Realiza el grafo de flujo de un programa que recibe tres números por teclado y muestra el menor de los números. Indica el número de nodos, aristas, regiones y nodos predicado. Calcula la complejidad ciclomática y muestra los caminos independientes.**



**Número de nodos => 12**

**Número de aristas => 14**

**Número de regiones => 4**

**Número de nodos predicados => 3 (2,3,4)**

**Complejidad ciclomática**

**V(G) = Número de regiones del grafo = 4**

**V(G) = Número de aristas – Número de nodos + 2 = 14 – 12 + 2 = 4**

**V(G) = Número de nodos predicados + 1 = 3 + 1 = 4**

**Como la complejidad ciclomática está entre 1 y 10, no existe mucho riesgo, el programa es sencillo.**

**Caminos:**

**1. 1-2-3-5-8-10-11**

**2. 1-2-3-6-8-10-11**

**3. 1-2-4-7-9-10-11**

**4. 1-2-4-6-9-10-11**

1. **¿Qué es la refactorización?**

Es una técnica de la ingeniería de software que permite la optimización de un código previamente escrito.

- Limpia el código, mejorando la consistencia y la claridad.

- Mantiene el código, no corrige errores ni añade funciones nuevas.

- Va a permitir facilitar la realización de cambios en el código.

- Se obtiene un código limpio y altamente modularizado.

1. **¿Cuándo consideras que es necesario refactorizar?**

- Cuando detectamos el mismo código en distintos lugares.

- Cuando los métodos son muy largos.

- Cuando una clase tiene muchos métodos, atributos e instancias.

- Cuando se pasan muchos parámetros.

- Cuando una clase es frecuentemente modificada.

- Cuando se deben realizar cambios para compatibilizar con otros cambios en otras clases.

- Cuando un método utiliza más elementos de otra clase que de la suya propia.

- Cuando tenemos clases que sólo tienen atributos y métodos de acceso a ellos.

-Cuando una subclase utiliza pocas características de sus superclases.

1. **¿Qué son los bad smells?**

Son los síntomas para refactorizar:

- Código duplicado.

- Métodos muy largos.

- Clases muy grandes.

- Lista de parámetros extensa.

- Cambio divergente.

- Cirugía a tiro pistola.

- Clase de solo datos.

- Legado rechazado.

1. **¿Cuáles son las partes de un software más conflictivas a la hora de refactorizar?**

Las bases de datos y las interfaces.

1. **Nombra los métodos de refactorización que podemos encontrar en Eclipse.**

- Rename

- Move

- Extract Constant

- Extract Local Variable

- Convert Local Variable to Field

- Extract Method

- Change Methos Signature

- Inline

- Member Type to Top Level

- Extract Interface

- Extract Superclass

- Convert Anonymous Class to Nested

1. **¿Qué es el control de versiones?**

La capacidad de recordar los cambios realizados tanto en la estructura de directorios como en el contenido de los archivos.

1. **Describe en qué consisten los siguientes términos del control de versiones:**

a. Repositorio: Lugar donde se almacenan todos los datos y los cambios.

b. Revisión: Es una versión concreta de los datos almacenados.

c. Etiquetar: Identificar puntos del proyecto fácilmente.

d. Trunk: Línea principal del desarrollo del proyecto.

e. Crear un Branch(rama): son copias del proyecto, por lo que se crea una bifurcación entre el proyecto y su copia, y se puede modificar cualquiera de las dos versiones.

f. Checkout: crear una copia de trabajo del proyecto, o de archivos y carpetas del repositorio en el equipo local.

g. Commit: se confirman los cambios realizados en local para integrarlos al repositorio.

h. Importación: subir carpetas y archivos del equipo local al repositorio.

i. Update: se integran los cambios realizados en el repositorio en la copia de trabajo local.

j. Merge: se unen los cambios de varias ramas a la vez.

k. Conflicto: Explicar claramente lo que hace el programa, de esta manera todo el equipo de desarrollo sabrá lo que se está haciendo y por qué.

1. **¿Qué es subversion? ¿Cómo funciona el ciclo de vida de esta herramienta?**

Es una herramienta multiplataforma de código abierto para el control de versiones.

El ciclo de vida de subversion parte del comienzo del software, se crea el tronco del proyecto (trunk) y después se van añadiendo funcionalidades a este desarrollo mediante las ramas, para que no dañen el proyecto. Cada vez que se tienen listas (sin bugs) estas funcionalidades se fusionan con la rama principal. Por último, cada vez que hay una nueva versión del software, se añade una etiqueta para poder identificar estas partes fácilmente.

1. **¿Qué tipo de documentación se realizan de un proyecto?**

- Documentación de las especificaciones.

- Documentación del diseño.

- Documentación del código fuente.

- Documentación de usuario final.

1. **¿Qué es JavaDoc? ¿Para qué se utiliza? Etiquetas**

Es una utilidad de Java que se utiliza para extraer y generar documentación directamente del código en formato HTML.Es una herramienta de Oracle

|  |  |  |  |
| --- | --- | --- | --- |
| **Tag** | **Descripción** | **Uso** | **Versión** |
| @author | Nombre del desarrollador. | nombre\_autor | 1.0 |
| @deprecated | Indica que el método o clase es antigua y que no se recomienda su uso porque posiblemente desaparecerá en versiones posteriores. | descripción | 1.0 |
| @param | Definición de un parámetro de un método, es requerido para todos los parámetros del método. | nombre\_parametro descripción | 1.0 |
| @return | Informa de lo que devuelve el método, no se puede usar en constructores o métodos "void". | descripción | 1.0 |
| @see | Asocia con otro método o clase. | referencia (#método(); clase#método(); paquete.clase; paquete.clase#método()). | 1.0 |
| @throws | Excepción lanzada por el método | nombre\_clase descripción | 1.2 |
| @version | Versión del método o clase. | versión | 1.0 |

**UF-3**

## Elaboración de diagramas de clases.

Para poder realizar estos diagramas, es necesario conocer el **lenguaje unificado de modelado (UML)**

**Diagramas de secuencia,colaboración,casos de uso y de clases son los 4 modelos estudiados.**

### Objetos

Un objeto es aquello que tiene

**Atributos** (propiedades más valores)

**Comportamiento** (acciones y reacciones a mensajes)

**Identidad (**propiedad que lo distingue de los demás objetos).

Una **clase** es un conjunto de objetos con estructura y comportamiento comunes.

Las propiedades de los objetos son claves. Los principios del modelo OO son:

* + - **Abstracción**: donde se capturan sus movimientos. Se pretende conseguir una descripción formal.
    - **Encapsulación**: oculta los métodos y atributos a otros objetos pasando a ser privados.
    - **Modularidad**: es la capacidad de un sistema o aplicación para dividirse en pequeños módulos independientes.
    - **Jerarquía o herencia**: permite que algunas clases tengan propiedades y características de una clase superior.
    - **Polimorfismo**: de un modo más preciso: dos instancias u objetos, pertenecientes a distintas clases, pueden responder a la llamada a métodos del mismo nombre, cada uno de ellos con distinto comportamiento encapsulado, pero que responden a una interfaz común (marcada a través del mecanismo de la herencia).
    - **Tipificación:** definición precisa de un objeto de forma que no puedan ser intercambiados.
    - **Concurrencia:** propiedad de un objeto que está activo de otro que no lo está.
    - **Persistencia:** propiedad de un objeto a través de la cual su existencia trasciende en el tiempo y/o el espacio.

### Diagramas de clases. (Esto caerá en el examen seguro de forma practica y hay que comprender lo que se hace, pues puede poner un dibujo y decir en un tipo test que opción es la que representa lo dibujado)

**Introducción**

Un diagrama de clases sirve para visualizar las relaciones entre las clases que involucran el sistema, las cuales pueden ser asociativas, de herencia, de uso y de contenimiento.

Un diagrama de clases esta compuesto por los siguientes elementos:

* [Clase](https://users.dcc.uchile.cl/~psalinas/uml/modelo.html#clase): atributos, métodos y visibilidad.
* [Relaciones](https://users.dcc.uchile.cl/~psalinas/uml/modelo.html#relacion): Herencia, Composición, Agregación, Asociación y Uso.

**Elementos**

* **Clase**

Es la unidad básica que encapsula toda la información de un Objeto (un objeto es una instancia de una clase). A través de ella podemos modelar el entorno en estudio (una Casa, un Auto, una Cuenta Corriente, etc.).

En UML, una clase es representada por un rectángulo que posee tres divisiones:



En donde:

* + **Superior**: Contiene el nombre de la Clase
  + **Intermedio**: Contiene los atributos (o variables de instancia) que caracterizan a la Clase (pueden ser private, protected o public).
  + **Inferior**: Contiene los métodos u operaciones, los cuales son la forma como interactúa el objeto con su entorno (dependiendo de la visibilidad: private, protected o public).

Ejemplo:

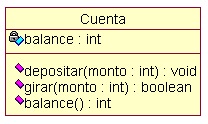
Una Cuenta Corriente que posee como característica:

* + Balance

Puede realizar las operaciones de:

* + Depositar
  + Girar
  + y Balance

El diseño asociado es:



Atributos y Métodos:

* + **Atributos:**

Los atributos o características de una Clase pueden ser de tres tipos, los que definen el grado de comunicación y visibilidad de ellos con el entorno, estos son:

* + - **public** (+,https://users.dcc.uchile.cl/~psalinas/uml/img/modelo/public.jpg): Indica que el atributo será visible tanto dentro como fuera de la clase, es decir, es accsesible desde todos lados.
    - **private** (-,https://users.dcc.uchile.cl/~psalinas/uml/img/modelo/private.jpg): Indica que el atributo sólo será accesible desde dentro de la clase (sólo sus métodos lo pueden accesar).
    - **protected** (#,https://users.dcc.uchile.cl/~psalinas/uml/img/modelo/protected.jpg): Indica que el atributo no será accesible desde fuera de la clase, pero si podrá ser accesado por métodos de la clase además de las subclases que se deriven (ver herencia).
  + **Métodos:**

Los métodos u operaciones de una clase son la forma en como ésta interactúa con su entorno, éstos pueden tener las características:

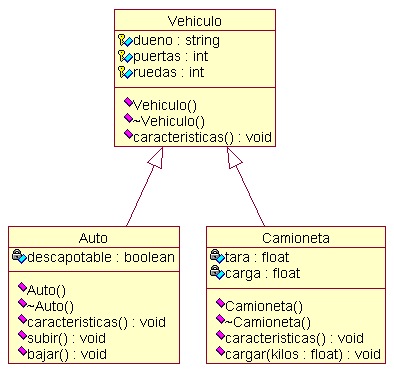
* + - **public** (+,https://users.dcc.uchile.cl/~psalinas/uml/img/modelo/public2.jpg): Indica que el método será visible tanto dentro como fuera de la clase, es decir, es accsesible desde todos lados.
    - **private** (-,https://users.dcc.uchile.cl/~psalinas/uml/img/modelo/private2.jpg): Indica que el método sólo será accesible desde dentro de la clase (sólo otros métodos de la clase lo pueden accesar).
    - **protected** (#,https://users.dcc.uchile.cl/~psalinas/uml/img/modelo/protected2.jpg): Indica que el método no será accesible desde fuera de la clase, pero si podrá ser accesado por métodos de la clase además de métodos de las subclases que se deriven (ver herencia).
* **Relaciones entre Clases:**

Ahora ya definido el concepto de Clase, es necesario explicar como se pueden interrelacionar dos o más clases (cada uno con características y objetivos diferentes).

Antes es necesario explicar el concepto de cardinalidad de relaciones: En UML, la cardinalidad de las relaciones indica el grado y nivel de dependencia, se anotan en cada extremo de la relación y éstas pueden ser:

* + **uno o muchos**: 1..\* (1..n)
  + **0 o muchos**: 0..\* (0..n)
  + **número fijo**: m (m denota el número).
  + **Herencia (Especialización/Generalización)**: https://users.dcc.uchile.cl/~psalinas/uml/img/modelo/herencia1.jpg

Indica que una subclase hereda los métodos y atributos especificados por una Super Clase, por ende la Subclase además de poseer sus propios métodos y atributos, poseerá las características y atributos visibles de la Super Clase (public y protected), ejemplo:



En la figura se especifica que Auto y Camión heredan de Vehículo, es decir, Auto posee las Características de Vehículo (Precio, VelMax, etc) además posee algo particular que es Descapotable, en cambio Camión también hereda las características de Vehiculo (Precio, VelMax, etc) pero posee como particularidad propia Acoplado, Tara y Carga.

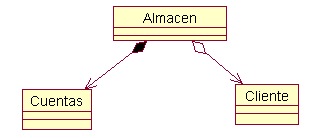
Cabe destacar que fuera de este entorno, lo único "visible" es el método Caracteristicas aplicable a instancias de Vehículo, Auto y Camión, pues tiene definición publica, en cambio atributos como Descapotable no son visibles por ser privados.

* + **Agregación**: https://users.dcc.uchile.cl/~psalinas/uml/img/modelo/agregacion1.jpg

Para modelar objetos complejos, n bastan los tipos de datos básicos que proveen los lenguajes: enteros, reales y secuencias de caracteres. Cuando se requiere componer objetos que son instancias de clases definidas por el desarrollador de la aplicación, tenemos dos posibilidades:

* + - **Por Valor**: Es un tipo de relación estática, en donde el tiempo de vida del objeto incluido esta condicionado por el tiempo de vida del que lo incluye. Este tipo de relación es comunmente llamada **Composición** (el Objeto base se contruye a partir del objeto incluido, es decir, es "parte/todo").
    - **Por Referencia**: Es un tipo de relación dinámica, en donde el tiempo de vida del objeto incluido es independiente del que lo incluye. Este tipo de relación es comunmente llamada **Agregación** (el objeto base utiliza al incluido para su funcionamiento).

Un Ejemplo es el siguiente:



En donde se destaca que:

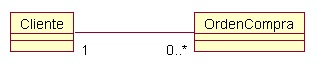
* + - Un Almacen posee Clientes y Cuentas (los rombos van en el objeto que posee las referencias).
    - Cuando se destruye el Objeto Almacen también son destruidos los objetos Cuenta asociados, en cambio no son afectados los objetos Cliente asociados.
    - La composición (por Valor) se destaca por un rombo relleno.
    - La agregación (por Referencia) se destaca por un rombo transparente.

La flecha en este tipo de relación indica la navegabilidad del objeto refereniado. Cuando no existe este tipo de particularidad la flecha se elimina.

* + **Asociación**: https://users.dcc.uchile.cl/~psalinas/uml/img/modelo/asociacion1.jpg

La relación entre clases conocida como Asociación, permite asociar objetos que colaboran entre si. Cabe destacar que no es una relación fuerte, es decir, el tiempo de vida de un objeto no depende del otro.

Ejemplo:



Un cliente puede tener asociadas muchas Ordenes de Compra, en cambio una orden de compra solo puede tener asociado un cliente.

* + **Dependencia o Instanciación (uso)**: https://users.dcc.uchile.cl/~psalinas/uml/img/modelo/dependencia1.jpg

Representa un tipo de relación muy particular, en la que una clase es instanciada (su instanciación es dependiente de otro objeto/clase). Se denota por una flecha punteada.

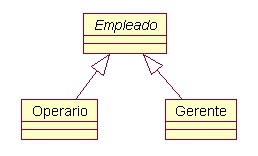
El uso más particular de este tipo de relación es para denotar la dependencia que tiene una clase de otra, como por ejemplo una aplicación grafica que instancia una ventana (la creación del Objeto Ventana esta condicionado a la instanciación proveniente desde el objeto Aplicacion):

https://users.dcc.uchile.cl/~psalinas/uml/img/modelo/edependencia.jpg

Otro ejemplo seria la relación entre un objeto Mascota un objeto Veterinario y un objeto intervención, intervención estaría condicionado la mascota y al veterinario.

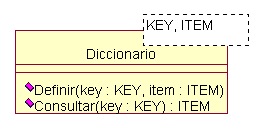
Cabe destacar que el objeto creado (en este caso la Ventana gráfica) no se almacena dentro del objeto que lo crea (en este caso la Aplicación).

* **Casos Particulares**:
  + **Clase Abstracta**:



Una clase abstracta se denota con el nombre de la clase y de los métodos con letra "itálica". Esto indica que la clase definida no puede ser instanciada pues posee métodos abstractos (aún no han sido definidos, es decir, sin implementación). La única forma de utilizarla es definiendo subclases, que implementan los métodos abstractos definidos.

* + **Clase parametrizada**:



Una clase parametrizada se denota con un subcuadro en el extremo superior de la clase, en donde se especifican los parámetros que deben ser pasados a la clase para que esta pueda ser instanciada. El ejemplo más típico es el caso de un Diccionario en donde una llave o palabra tiene asociado un significado, pero en este caso las llaves y elementos pueden ser genéricos. La genericidad puede venir dada de un Template (como en el caso de C++) o bien de alguna estructura predefinida (especialización a través de clases).

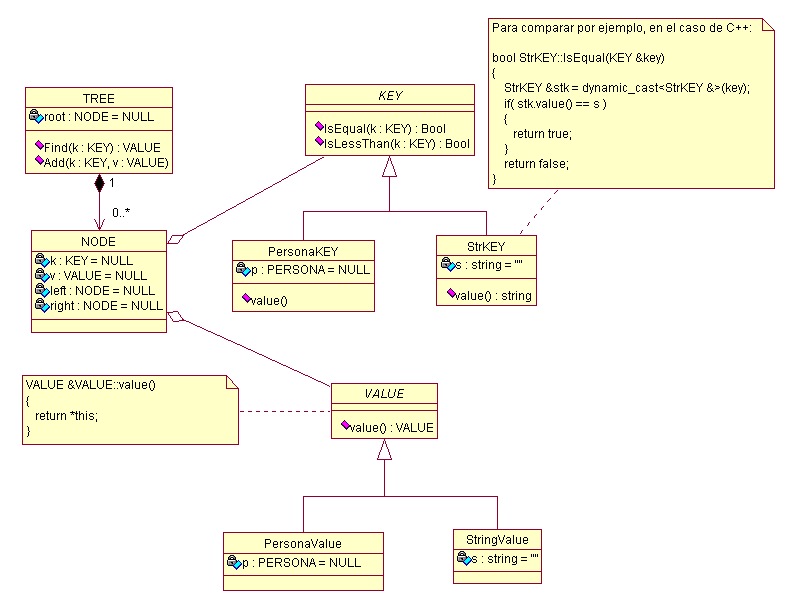
En el ejemplo no se especificaron los atributos del Diccionario, pues ellos dependerán exclusivamente de la implementación que se le quiera dar.

**Ejemplo**:

Supongamos que tenemos tenemos un el caso del Diccionario implementado mediante un árbol binario, en donde cada nodo posee:

* key: Variable por la cual se realiza la búsqueda, puede ser generica.
* item: Contenido a almacenar en el diccionario asociado a "key", cuyo tipo también puede ser genérico.

Para este caso particular hemos definido un Diccionario para almacenar String y Personas, las cuales pueden funcionar como llaves o como item, solo se mostrarán las relaciones para la implementación del Diccionario:



Para la creación de UML podemos usar estas alicaciones ArgoUML o usar Eclipse pero necesitaremos el plugin UML2.

1. **diagrama de casos de usos. (Esto caerá en el examen seguro de forma practica)**

#### **Elementos del diagrama de casos de uso**

Los elementos que tendrá un diagrama de casos de uso son:

* + - **Actores**: puede ser cualquier cosa que interactúa con el sistema, y es externo a él. No tiene que ser una persona. Se representa con un monigote y un nombre debajo.
    - **Casos de uso**: representa una unidad funcional del sistema que realizará una orden de algún agente externo, tanto de un agente como de otro caso de uso. Será iniciado por un actor, a partir de entonces otros actores podrán participar de él. Se represetan con un óvalo o eclipse y una descripción textual.
    - **Relaciones**: existen varios tipos. La más común es entre actores y casos de uso representada con una línea contínua.
    - También podemos tener un rectángulo que delimite el sistema.

#### **Relaciones en un diagrama de casos de uso**

|  |  |  |
| --- | --- | --- |
| **RELACIÓN** | **FUNCIÓN** | **NOTACIÓN** |
| **Asociación** | Línea de comunicación entre actor y caso de uso |  |
| **Extensión <<extend>>** | Especifica que el comportamiento de un caso de uso es diferente según las circunstancias. Apuntará a la relación de caso de uso que extenderá | <<extend>> |
| **Generalización de casos** | Generalización entre clases. El caso |  |
| **de uso** | de uso hijo hereda el  comportamiento y significado del padre. |  |



<<include>>

Permite que un caso de uso base

incluya el comportamiento de otro caso de uso

**Inclusión <<include>> o**

**<<uses>>**

|  |  |  |
| --- | --- | --- |
| **Actor** | Representa el papel desempeñado por un usuario |  |

|  |  |  |
| --- | --- | --- |
| **Marco** | Dentro del marco se representaran las acciones que se llevan a cabo |  |
| **Acciones** | Representa la accion llevada a cabo |  |

**Os pongo un ejemplo de la PAC que hicimos:**

Realice un diagrama de casos de usos para la página web de la tienda de la clínica veterinaria.

• El administrador se encarga de modificar y eliminar los productos.

• Mientras que, los clientes pueden buscar productos tanto por ID como por nombre, y añadirlos al carrito.

• Además, los clientes pueden efectuar compras, para lo que deben indicar el método de pago.



1. **Tablas de caso de uso, como hacer una. (Esto caerá en el examen seguro de forma practica)**

Para documentar los casos de uso, podremos hacerlo a través de una plantilla que nos describa lo que hace el actor y lo que ocurre cuando se interactúa con dicho sistema. Una plantilla sencilla podrá ser:

**Nombre** del caso de uso.

**ID** del caso de uso.

**Pequeña descripción** de lo que se espera del caso de uso.

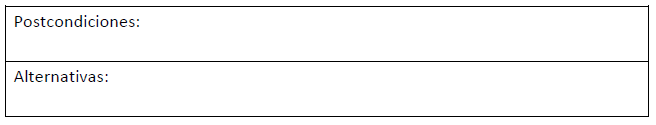
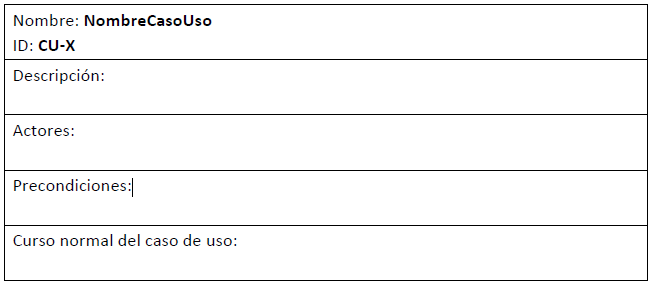
**Actores implicados**. Existen principales y secundarios. Los primeros activan el sistema y los segundos usan el caso de uso una vez iniciado.

**Precondiciones**. Serán las condiciones que se deberán de cumplir antes de que empieze el caso de uso.

**Curso normal**. Pasos del caso de uso para que finalice correctamente.

**Postcondiciones**. Las condiciones que se deberán de cumplir cuando finalice el caso de uso.

**Alternativas**. Errores o excepciones.



**Diagrama de secuencia**

El diagrama de secuencia nos mostrará gráficamente los eventos que fluyen de los actores del sistema

Los elementos principales serán:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **SÍMBOLO** | | **FUNCIÓN** | | **NOTACIÓN** | |
| **Marco** | | Da borde visual al diagrama de secuencia. A la izquierda del marco se escribe la etiqueta **sd** seguida de un nombre | |  | |
| **Línea de vida** | | Representa un participante durante la interacción | |  | |
| **Actor** | | Representa el papel desempeñado por un usuario | |  | |
| **Mensaje** | | **Mensaje síncrono** | |  | |
| **Mensaje asíncrono** | |  | |
| **Mensaje de retorno** | |  | |
| **Activación** | | Opcionales. Representa el tiempo durante el que se ejecuta una función. Se suele poner cuando está activo un método. | |  | |

Tendrá un nombre acompañado o no de parámetros.

* **Mensaje síncrono**: se comporta de la siguiente forma: cuando se envía un mensaje a un objeto, no se recibe el control hasta que el objeto receptor ha finalizado la ejecución.
* **Mensaje asíncrono**: representa flujo de control asíncrono. El que envía un mensaje asíncrono contínua con su trabajo después de ser enviado, no espera la finalización del mensaje en el receptor. Su utilización la podemos ver en sistemas multi-hilos donde se producen procesos concurrentes.
* **Mensaje de retorno**: representa mensaje de confirmación. De uso opcional.

#### **CREACIÓN Y DESTRUCCIÓN DE OBJETOS**

En el diagrama de secuencia podemos ver la creación y destrucción de objetos.

Para la creación se representa mediante un mensaje que termina en el objeto que será creado. Este mensaje puede llevar la identificación <<*create*>>.

La destrucción finalizará la línea de vida del objeto, y se representa mediante una X grande en su línea de vida. El mensaje puede llevar la identificación <<*destroy*>>.

La **autodelegación** o **mensaje reflexivo** es un mensaje que un objeto se envía a sí mismo regresando la flecha del mensaje de vuelta a la misma línea de vida.

**Diagramas de colaboración:** ste diagrama muestra los objetos junto con los mensajes que se envían entre ellos

El diagrama de colaboración posee los siguientes elementos:

|  |  |  |
| --- | --- | --- |
| **SÍMBOLO** | **FUNCIÓN** | **NOTACIÓN** |

|  |  |  |
| --- | --- | --- |
| **Objetos o roles** | Se representa con un rectángulo que contiene el nombre y la clase del objeto en el siguiente formato objeto:Clase. |  |
| **Enlaces** | Arcos del grafo que conecta ambos objetos. Se podrán mostrar muchos mensajes en un mismo enlace pero cada uno con un número de secuencia único |  |
| **Mensajes** | Se representa mediante una flecha dirigida con un nombre y un número de secuencia |  |
| **Número de secuencia** | Indica el orden de un mensaje dentro de la iteración. Comenzará con el 1 y se incrementará conforme se envíen mensajes. El primer mensaje no lleva número de secuencia. | |
| **Iteración** | Se representa colocando un \* después del número de secuencia y una condición encerrada entre corchetes. | |
| **Alternativa** | Se indican con condiciones entre corchetes. Los caminos alternativos tendrán el mismo número de secuencia, seguido del número de subsecuencia. | |
| **Anidamiento** | Se puede mostrar el anidamiento de mensajes con números de secuencia y subsecuencia. | |

Para numerar los mensajes se puede usar varios esquemas:

* Númeración simple: comienza en 1 y se incrementa una unidad y no hay nivel de anidamiento.
* Numeración decimal: tiene varios subíndices para indicar anidamiento de operaciones.