



1 Desenvolvimento de software

Sumario

1	Desenvolvemento de software.....	1
1.1	Convencións empregadas.....	4
1.2	Introdución.....	5
1.3	Código fonte, código obxecto e código executable: máquinas virtuais.....	8
1.3.1	Máquina virtual.....	9
1.4	Clasificación de software.....	10
1.5	Desenvolvemento de software.....	11
1.5.1	Paradigma de ciclo de vida clásico ou modelo en cascada.....	12
1.5.1.1	Análise.....	12
1.5.1.2	Deseño.....	13
1.5.1.3	Codificación.....	13
1.5.1.4	Probas.....	13
1.5.1.5	Instalación.....	14
1.5.1.6	Mantemento.....	15
1.5.1.7	Documentación.....	15
1.5.2	Modelo en espiral.....	16
1.5.3	Programación eXtrema.....	17
1.5.3.1	Planificación.....	17
1.5.3.2	Deseño.....	17
1.5.3.3	Codificación e probas.....	18
1.5.4	Scrum.....	18
1.5.5	Kanban.....	19
1.5.6	Métrica v.3.....	20
1.5.6.1	Planificación.....	20
1.5.6.2	Desenvolvemento.....	21
1.5.6.3	Mantemento.....	22
1.6	Tipos de linguaxes de programación.....	22
1.6.1	Linguaxes de marcas.....	22
1.6.2	Linguaxes de especificación.....	23
1.6.3	Linguaxes de consultas.....	23
1.6.4	Linguaxes de transformación.....	24
1.6.5	Linguaxes de programación.....	25
1.7	Clasificación das linguaxes de programación.....	26
1.7.1	Linguaxes de baixo nivel.....	26
1.7.2	Linguaxes de alto nivel.....	26
1.8	Clasificación das linguaxes de programación por xeracións.....	26
1.8.1	Características das linguaxes de primeira e segunda xeración.....	28
1.8.1.1	Compiladores.....	29
1.8.1.2	Os intérpretes.....	29
1.9	Características das linguaxes de terceira, cuarta e quinta xeración.....	30
1.10	Clasificación polo paradigma da programación.....	31
1.10.1	Paradigma imperativo.....	31
1.10.2	Paradigma declarativo.....	31
1.11	Paradigmas de programación.....	32
1.12	Características das linguaxes máis estendidas.....	35
1.12.1	Características da programación estruturada.....	35
1.12.1.1	Claridade.....	35
1.12.1.2	Teorema da estrutura.....	35

1.12.1.3	Deseño descendente.....	36
1.12.1.4	Programación modular.....	36
1.12.2	Tipos de datos abstractos (TAD).....	36
1.13	Características da programación orientada a obxectos.....	37
1.13.1	Abstracción.....	37
1.13.1.1	Encapsulamento.....	38
1.13.1.2	Modularidade.....	38
1.13.1.3	Xerarquía.....	38
1.13.1.4	Polimorfismo.....	38
1.14	Proceso de xeración de código.....	39
1.14.1	Edición.....	39
1.14.2	Compilación.....	39
1.14.2.1	Análise.....	39
1.14.2.2	Síntese.....	40
1.14.3	Enlace.....	40
1.14.4	Execución.....	40
1.15	Algoritmos e Programas.....	40
1.15.1	Ferramentas básicas para a creación de algoritmos.....	45
1.16	Actividades.....	53

1.1 Convencións empregadas

	Esta icona fai referencia a notas de introdución
	Esta icona indica aclaración
	Esta icona fai referencia a arquivos de configuración, de rexistro...
	Esta icona indica casos de uso
	Esta icona fai referencia a avisos o advertencias
	Esta icona indica incidencias
	Esta icona fai referencia a sección que inclúen instrucións paso a paso
	Esta icona fai referencia a sección que inclúen capturas de pantalla
	Esta icona fai referencia a actividades
	Esta icona fai referencia a documento esencial (licenza: http://www.ohmyicons.com)
	Esta icona fai referencia a enlace recomendado (licenza: http://iconleak.com)

1.2 Introducción

Na nosa vida diaria pódense atopar máis e máis electrónicos, comunicacións e ordenadores. Imos a un caixeiro automático dunha caixa ou banco e teremos que interactuar cunha interface táctil para comezar a executar a operación que nos interesa. Percorremos determinadas cidades onde indican en letreiros o estado do tráfico en tempo real ou incluso se hai espazos libres para aparcas na propia rúa. E non é necesario falar, se nos referimos a aplicacións informáticas, de todos os dispositivos electrónicos que usamos diariamente:

- Ordenadores de sobremesa
- Portátiles
- Teléfonos móbiles
- Axendas electrónicas
- Libros electrónicos
- Terminais de punto de venda
- Impresoras e fotocopiadoras
- Consolas de videoxogos
- Televisores

Pero estes son só un pequeno exemplo de todas as posibilidades que temos hoxe en día para usar dispositivos que leven un código de programación integrado, que teñen interfaces que permiten a interacción cos seus usuarios.

Antes de que ningún destes produtos chegue ao mercado, debe haber un proceso, moitas veces moi longo, que implica facer moitos tipos de pequenos traballos entre varias persoas. Unha destas tarefas, unha parte importante da creación dos produtos, será a programación, o desenvolvemento do código que se integrará no produto e que permitirá mostrar estas interfaces, interactuar cos usuarios e procesar a información e os datos ligados.

Pero para iso será necesario que previamente alguén deseñase estes procesos automatizados e se desenvolvera. É neste momento cando os contornos de desenvolvemento de software cobran importancia.

Os ambientes de desenvolvemento son ferramentas que ofrecen aos programadores moita facilidade á hora de crear unha aplicación informática. O módulo "Entornos de desenvolvemento" mostra o proceso de desenvolvemento dunha aplicación informática, indicando as características máis importantes das ferramentas que axudan a este procedemento.



A Real Academia Galega define a informática como a "Ciencia do tratamento automático da información por medio de máquinas electrónicas". Este tratamento necesita de:

- Soporte físico ou hardware formado por todos os compoñentes electrónicos tanxibles involucrados no tratamento. Segundo a Real Academia Galega é máis aconsellable dicir soporte físico que hardware e defíneo como a maquinaria ou elementos que compoñen un ordenador.
- Soporte lóxico ou software que fai que o hardware funcione e está formado por todos os compoñentes intanxibles involucrados no tratamento: programas, datos e documentación. . Segundo a Real Academia Galega é máis aconsellable dicir

soporte lóxico que software e defíneo como o conxunto de ordes e programas que permiten utilizar un ordenador.

- Equipamento humano ou persoal informático que manexa o equipamento físico e o lóxico para que todo o tratamento se leve a cabo.

O concepto de programa informático está moi ligado ao concepto de algoritmo. Un algoritmo é un conxunto de instrucións ou regras ben definidas ordenadas y finitas que permite resolver un problema.

	<p>Consultar: Que é un algoritmo?</p> <p>https://youtu.be/U3CGMyjzlvM</p>	
---	--	---

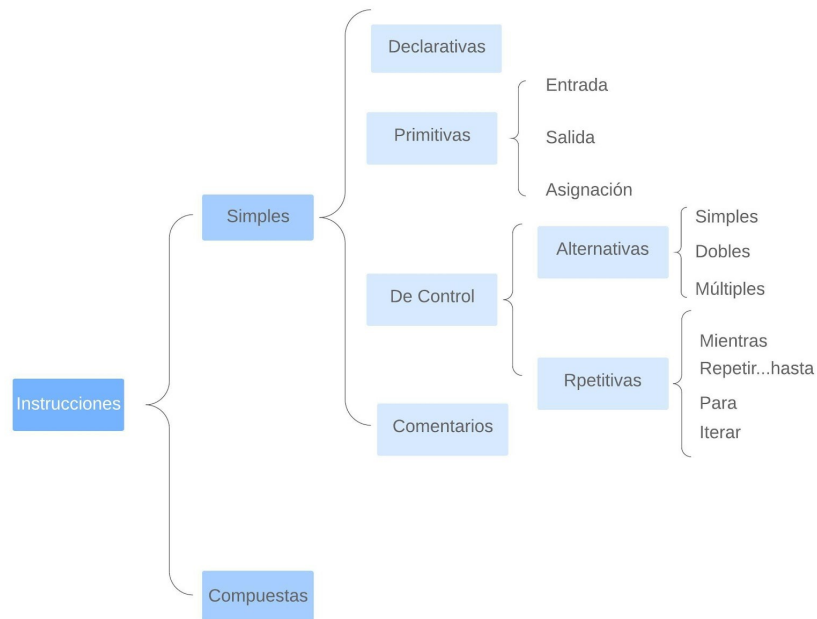
Calquera algoritmo debe ter as seguintes características:

- **Precisión** - A descrición de cada paso non debe levar a ambigüidades, de tal forma que os pasos son absolutamente explícitos e non inducen a erro.
- **Finitude** - O número de pasos debe ser finito, de forma que o algoritmo pódase executar nun tempo finito (tempo limitado).
- **Determinismo** – O algoritmo, dado un conxunto de datos de entrada idéntico, sempre debe devolver os mesmos resultados.

Para a elaboración dun algoritmo con estas características precisaremos dunha serie de elementos en base aos cales traballe. Principalmente falaremos de tipos de instrucións, de datos, e de operandos, os cales variarán en maior ou menor medida en función da linguaxe de programación elixida e especialmente en función do nivel do mesmo.

Tipos de instrucións

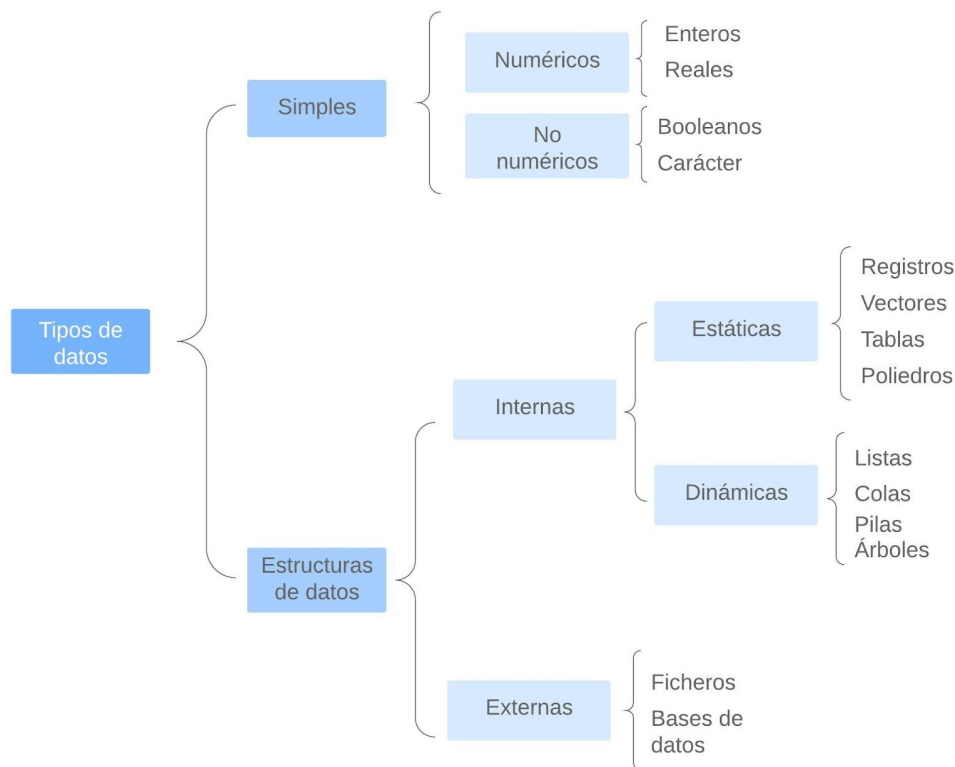
Por tanto, á hora de levar un algoritmo a un computador, este estará definido como un programa informático codificado en instrucións dunha linguaxe de programación determinada. De forma xeral podemos clasificar os tipos de instrucións en base ao seguinte esquema:



Datos e estruturas

Os datos son a información (valores ou referentes) que recibe o computador a través de distintos medios, e que é manipulada mediante o procesamento dos algoritmos de programación. O seu contido pode ser practicamente calquera: estadísticas, números, caracteres...

É de vital importancia coñecer qué tipos e estruturas de datos existen para poder seleccionar os que máis se adecuen aos nosos algoritmos.



Tipos de operadores

Os operadores son símbolos que indican a operación a realizar cos datos do noso algoritmo. Podemos dividilos en:

- **Operadores Aritméticos:** para a execución de operacións aritméticas (+, -, *, /, etc.)
- **Operadores Lóxicos:** para operacións relacionadas coa álgebra de Boole (AND, OR, NOT, NAND, NOR Y XOR).
- **Operadores Relacionais:** Evalúan a relación entre dúas variables, constantes ou expresións. Indicarán igualdade, inferioridade, superioridade, etc.

Eficiencia dos algoritmos

Outra característica desexable dos algoritmos é a **eficiencia**. Cando falamos da eficiencia dun algoritmo informático referímonos ao maior ou menos uso que este fai dos recursos dun ordenador. Fundamentalmente evaluaremos o uso de memoria principal: **eficiencia en memoria** e o uso da CPU: **eficiencia computacional**. O primeiro vai directamente ligado coa utilización dos datos e estruturas durante a execución do noso algoritmo; mentres que o segundo dependerá da complexidade do mesmo, isto é, do número de operacións e por consiguiente, do tempo que requerirá o algoritmo para executarse.

Habitualmente o tempo de execución dun algoritmo aumenta según aumenta o tamaño dos datos de entrada, dependendo de cómo se comporte ese aumento diremos que a súa orde de complexidade será: constante - $O(1)$, logarítmica - $O(\log n)$, lineal - $O(n)$, cuadrática - $O(n^2)$, etc. en base á función matemática que mellor represente esta relación. É dicir, a orde de complexidade darános unha idea clara da escalabilidade do noso algoritmo.

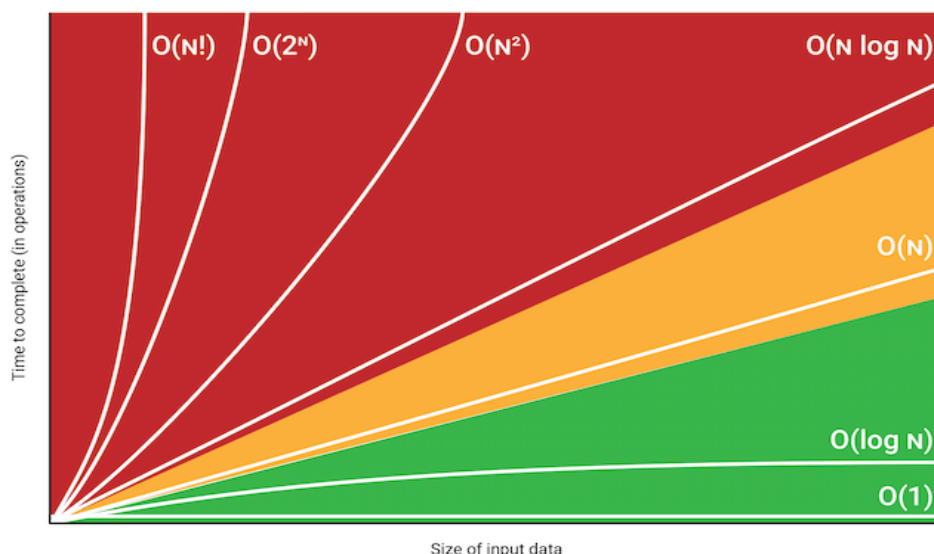
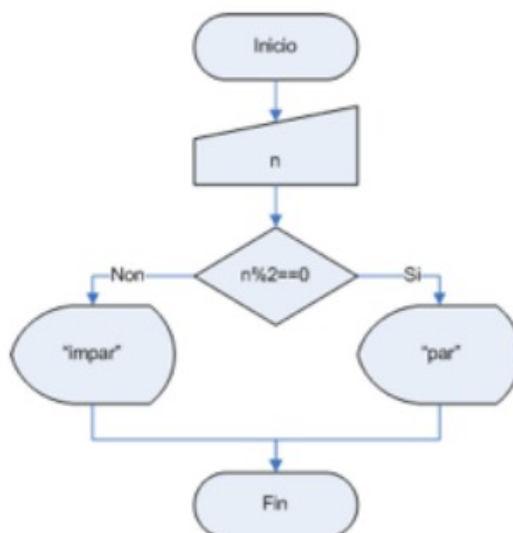


Gráfico – Notación Big-O

Os algoritmos son independentes da sintaxe de cada linguaxe de programación en particular, sendo evidente que o algoritmo que leve á solución dun determinado problema pode ser expresado utilizando distintas linguaxes de programación.

Exemplo de algoritmo representado mediante un diagrama de fluxo que permite ver se un número tecleado é par ou impar e que considera o 0 como número par:



O algoritmo pode escribirse nunha linguaxe de programación utilizando algunha ferramenta de edición, dando lugar a un código que deberá de gravarse nunha memoria

externa non volátil para que perdure no tempo. Exemplo de código escrito en linguaxe Java que se corresponde co anterior algoritmo:

```
package parimpar;

import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        short n;
        Scanner teclado = new Scanner(System.in);
        System.out.printf("Teclee o número enteiro entre %d e %d:",
            Short.MIN_VALUE, Short.MAX_VALUE);
        n = teclado.nextShort();
        if (n%2==0) {
            System.out.printf("%d é par\n", n);
        }
        else {
            System.out.printf("%d é impar\n", n);
        }
    }
}
```

O código terá que sufrir algunhas transformacións para que finalmente se obteña un programa que poida ser executado nun ordenador. Para a execución é preciso que o programa estea almacenando na memoria interna e volátil do ordenador; así o procesador pode ir collendo cada orde que forma o programa, resolvéndoa e controlando a súa execución. Se é preciso, accede á memoria interna para manipular variables, ou aos periféricos de entrada, saída ou almacenamento, fai cálculos ou resolve expresións lóxicas ata que finaliza con éxito a última instrución ou se atopa cun erro irresoluble.

Na execución do programa correspondente ao código exemplo anterior, o procesador necesitaría un número enteiro subministrado a través do teclado (dispositivo ou periférico de entrada), obtería o resultado dunha operación aritmética (resto da división enteira de n entre 2), resolvería unha expresión lóxica (descubrir se o resto anterior é 0), e executaría a instrución alternativa para que no caso de que o resto sexa 0 saia pola pantalla (dispositivo ou periférico de saída) que n é par ou que é impar. Exemplo de execución do código anterior e aspecto do resultado da execución na pantalla de texto cando se teclea o número 11:

Os programas informáticos son imprescindibles para que os ordenadores funcionen e son conxuntos de instrucións que unha vez executadas realizarán unha ou varias tarefas. O software é un conxunto de programas e neste concepto inclúense tamén os datos que se manexan e a documentación deses programas.

1.3 Código fonte, código obxecto e código executable: máquinas virtuais



Editores de texto sinxelos

Un editor de texto sinxelo é un que permite escribir só texto sinxelo. Exemplos son Notepad (Windows), Gedit ou Emacs (Unix)

Para crear un programa o que se fará é crear un ficheiro e escribir nun ficheiro a serie de instrucións que desexa que execute o ordenador. Estas instrucións deben seguir certas pautas en función da linguaxe de programación escollido. Ademais, deberían seguir unha determinada orde que dará sentido ao programa escrito. Para comezar, abondará cun simple editor de texto.

Unha vez que o programa rematou de escribir, dise que o conxunto de ficheiros de texto, onde se atopan as instrucións, contén o código fonte. Este código fonte pode ser dende un nivel moi alto, moi próximo á linguaxe humana, ata un nivel inferior, máis próximo ao código das máquinas, como o código ensamblador.

A tendencia actual é facer uso de linguaxes de alto nivel, é dicir, próximas á linguaxe humana. Pero isto causa un problema e é que os ficheiros de código fonte non conteñen a linguaxe da máquina que o ordenador entenderá. Polo tanto, son incomprensibles para o procesador. Para poder xerar código máquina, é necesario levar a cabo un proceso de tradución dende as mnemotécnicas contidas en cada ficheiro ata as secuencias binarias que o procesador entende.



O código obxecto das instrucións ten o seguinte aspecto:
10101001000001101100110 10100101011100001110111

O proceso chamado compilación é a tradución do código fonte dos ficheiros do programa a ficheiros de formato binario que conteñen instrucións nun formato que o procesador pode entender. O contido destes ficheiros chámase código obxecto . O programa que fai este proceso chámase compilador.

O código obxecto é o código fonte traducido (polo compilador) ao código máquina, pero este código aínda non o pode executar a computadora.

O código executable é a tradución completa a código máquina, realizada polo ligador (en inglés, ligador). O código executable é interpretado directamente pola computadora.

O ligador encárgase de inserir no código obxecto as funcións das bibliotecas necesarias para o programa e de levar a cabo o proceso de montaxe xerando un ficheiro executable.

Unha biblioteca (biblioteca en inglés) é unha colección de código predefinido que facilita a tarefa do programador cando codifica un programa.

O código fonte desenvolvido polos programadores converterase en código obxecto coa axuda do compilador. Isto axudará a localizar erros de sintaxe ou compilación atopados no código fonte. Co enlazador, que recollerá o código obxecto e as bibliotecas, xerárase o código executable.



1.1 Crea un esquema que recolla o proceso de transformación dun código fonte nun código executable

Solución

1.3.1 Máquina virtual

O concepto de máquina virtual xorde co obxectivo de facilitar o desenvolvemento de compiladores que xeran código para diferentes procesadores.

A compilación consta de dúas fases:

- A primeira parte do código fonte a unha linguaxe intermedia obtendo un programa equivalente cun nivel de abstracción inferior ao orixinal e que non se pode executar directamente.
- A segunda fase traduce a linguaxe intermedia a unha linguaxe comprensible por máquina.

Neste punto poderíase facer a pregunta: por que dividir a compilación en dúas fases? O obxectivo é que o código da primeira fase, o código intermedio, sexa común a calquera procesador e que o código xerado na segunda fase sexa específico de cada procesador. De feito, a tradución da linguaxe intermedia á linguaxe da máquina normalmente non se fai por compilación senón por un intérprete.

1.4 Clasificación de software

Pódese clasificar o software en dous tipos:

- Software de sistema. Controla e xestiona o hardware facendo que funcione, ocultando ao persoal informático os procesos internos deste funcionamento. Exemplos deste tipo de software son:
 - Sistemas operativos
 - Controladores de dispositivos
 - Ferramentas de diagnóstico
 - Servidores (correo, web, DNS,...)
 - Utilidades do sistema (compresión de información, rastreo de zoas defectuosas en soportes,...)

- Software de aplicación. Permite levar a cabo unha ou varias tarefas específicas en calquera campo de actividade dende que está instalado o software básico de sistema. Exemplos deste tipo de software son:
 - Aplicacións para control de sistemas e automatización industrial
 - Aplicacións ofimáticas
 - Software educativo
 - Software empresarial: contabilidade, nóminas, almacén,...
 - Bases de datos
 - Videoxogos
 - Software de comunicacións: navegadores web, clientes de correo,...
 - Software médico
 - Software de cálculo numérico
 - Software de deseño asistido por ordenador
 - Software de programación que é o conxunto de ferramentas que permiten ao programador desenvolver programas informáticos como por exemplo:
 - Editores de texto
 - Compiladores
 - Intérpretes
 - Enlazadores
 - Depuradores
 - Contornos de desenvolvemento integrado (IDE) que agrupa as anteriores ferramentas nun mesmo contorno para facilitar ao programador a tarefa de desenvolver programas informáticos e que teñen unha avanzada interface gráfica de usuario (GUI).

Dentro do software de aplicación pódese facer unha división entre:

- Software horizontal ou xenérico que se poden utilizar en diversos contornos como por exemplo as aplicacións ofimáticas.
- Software vertical ou a medida que só se poden utilizar nun contorno específico como por exemplo a contabilidade feita a medida para unha empresa en concreto.



1.2 Buscar en internet nomes comerciais de software de sistema e software de aplicación

1.5 Desenvolvemento de software

O proceso de desenvolvemento de software é moi diferente dependendo da complexidade do software. Por exemplo, para desenvolver un sistema operativo é necesario un equipo disciplinado, recursos, ferramentas, un proxecto a seguir e alguén que xestione todo. No extremo oposto, para facer un programa que visualice se un número enteiro que se teclea é par ou impar, só é necesario un programador ou un afeccionado á programación e un algoritmo de resolución.

Fritz Bauer nunha reunión do Comité Científico da OTAN en 1969 propuxo a primeira definición de Enxeñaría de software como o establecemento e uso de principios de enxeñería robustos, orientados a obter economicamente software que sexa fiable e funcione eficientemente sobre máquinas reais. Posteriormente, moitas personalidades destacadas do mundo do software matizaron esta definición que se normalizou na norma IEEE 1993 que define a Enxeñaría de software como a aplicación dun enfoque sistemático, disciplinado e medible ao desenvolvemento, funcionamento e mantemento do software.

Durante décadas os enxeñeiros e enxeñeiras de software foron desenvolvendo e mellorando paradigmas (métodos, ferramentas e procedementos para describir un modelo) que foran sistemáticos, predicibles e repetibles e así mellorar a produtividade e calidade do software.

A Enxeñaría de software é imprescindible en grandes proxectos de software, debe de ser aplicada en proxectos de tamaño medio e recoméndase aplicar algún dos seus procesos en proxectos pequenos.

Existen moitos modelos a seguir para desenvolver software. O máis clásico é o modelo en cascada. Destacan entre todos o modelo en espiral baseado en prototipos, a programación extrema como representativo dos métodos de programación áxil, e a Métrica 3 como modelo a aplicar en grandes proxectos relacionados coas institucións públicas.

1.5.1 Paradigma de ciclo de vida clásico ou modelo en cascada

O paradigma de ciclo de vida clásico do software, tamén chamado modelo en cascada consta das fases: análise, deseño, codificación, probas, instalación e mantemento e a documentación é un aspecto implícito en todas as fases. Algúns autores nomean estas fases con nomes lixeiramente diferentes, ou poden agrupar algunhas para crear unha nova fase ou nomear unha fase con máis detalle, pero en esencia as fases son as mesmas. Algunhas destas fases tamén se utilizan noutros modelos con lixeiras variantes.

As fases vanse realizando de forma secuencial utilizando a información obtida ao finalizar unha fase para empezar a fase seguinte. Deste xeito o cliente non pode ver o software

funcionando ata as últimas fases e daría moito traballo corrixir un erro que se detecta nas últimas fases pero que afecta ás primeiras.

1.5.1.1 *Análise*

Nesta fase o analista captura, analiza e especifica os requisitos que debe cumprir o software. Debe obter a información do cliente ou dos usuarios do software mediante entrevistas planificadas e habilmente estruturadas nunha linguaxe comprensible para o usuario. O resultado desta captura de información depende basicamente da habilidade e experiencia do analista aínda que poida utilizar guías ou software específico.

Ao finalizar esta fase debe existir o documento de especificación de requisitos do software (ERS), no que estarán detallados os requisitos que ten que cumprir o software, debe valorarse o custo do proxecto e planificarse a duración do mesmo. Toda esta información ten que comunicarse ao cliente para a súa aceptación.

A linguaxe utilizada para describir os ERP pode ser descritiva ou máis formal e rigorosa utilizando casos de usos na linguaxe de modelado UML. O estándar IEEE 830-1998 recolle unha norma de prácticas recomendadas para a especificación de requisitos de software.

1.5.1.2 *Deseño*

Nesta fase o deseñador deberá de descompoñer e organizar todo o sistema software en partes que podan elaborarse por separado para así aproveitar as vantaxes do desenvolvemento de software en equipo.

O resultado desta fase plásmase no documento de deseño de software (SDD) que contén a estrutura global do sistema, a especificación do que debe facer cada unha das partes e a maneira de combinarse entre elas e é a guía que os programadores e probadores de software deberán ler, entender e seguir. Este documento incluírá o deseño lóxico de datos, o deseño da arquitectura e estrutura, o deseño dos procedementos, a organización do código fonte e a compilación, e o da interface entre o home e o software. Exemplos de diagramas que indican como se construírá o software poden ser: modelo E/R para os datos, diagramas UML de clases, diagramas UML de despregue e diagramas UML de secuencia.

Nesta fase debe tratarase a seguridade do proxecto mediante unha análise de riscos (recompilación de recursos que deben ser protexidos, identificación de actores e roles posibles, recompilación de requisitos legais e de negocio como encriptacións ou certificacións a cumprir, etcétera) e a relación de actividades que mitigan eses riscos.

1.5.1.3 *Codificación*

Esta fase tamén se chama fase de programación ou implementación. Nela o programador transforma o deseño lóxico da fase anterior a código na linguaxe de programación elixida,

de tal forma que os programas resultantes cumpran os requisitos da análise e poidan ser executado nunha máquina.

Mentres dura esta fase, poden realizarse tarefas de depuración do código ou revisión inicial do mesmo para detectar erros sintácticos, semánticos e de lóxica.

1.5.1.4 Probas

Esta fase permite aplicar métodos ou técnicas ao código para determinar que tódalas sentencias foron probadas e funcionan correctamente.

As probas teñen que planificarse, deseñarse, executarse e avaliar os resultados. Considérase que unha proba é boa se detecta erros non detectados anteriormente. As probas realizadas inmediatamente despois da codificación poden ser:

- Probas unitarias cando permiten realizar tests a anacos pequenos de código cunha funcionalidade específica.
- Probas de integración cando permiten realizar tests a un grupo de anacos de código que superaron con éxito os correspondentes tests unitarios e que interactúan entre eles.

Outras probas sobre o sistema total poden ser:

- Probas de validación ou aceptación para comprobar que o sistema cumpre os requisitos do software especificados no ERS.
- Probas de recuperación para comprobar como reacciona o sistema fronte a un fallo xeral e como se recupera do mesmo.
- Probas de seguridade para comprobar que os datos están protexidos fronte a agresións externas.
- Probas de resistencia para comprobar como responde o sistema a intentos de bloqueo ou colapso.
- Probas de rendemento para someter ao sistema a demandas do usuario extremas e comprobar o tempo de resposta do sistema.

As probas deberán de ser realizadas en primeiro lugar polos creadores do software pero é recomendable que tamén sexan realizadas por especialistas que non participaron na creación e finalmente sexan realizadas por usuarios.

O software pode poñerse a disposición dos usuarios cando aínda no está acabado e entón noméase co nome comercial e un texto que indica o nivel de acabado. Ese texto pode ser:

- Versión Alfa. Versión inestable, á que aínda se lle poden engadir novas características.
- Versión Beta. Versión inestable á que non se lle van a engadir novas características pero que pode ter erros.

- Versión RC (Release Candidate) é case a versión final pero aínda poden aparecer erros.
- Versión RTM (Release To Manufacturing). Versión estable para comercializar.



1.3 Buscar en internet aplicacións software dispoñibles para o usuario en versión alfa, beta, RC ou RTM

Solución

1.5.1.5 Instalación

Esta fase tamén se denomina despregue ou implantación e consiste en transferir o software do sistema ao ordenador destino e configuralo para que poida ser utilizados polo usuario final. Esta fase pode consistir nunha sinxela copia de arquivos ou ser máis complexo como por exemplo: copia de programas e de datos que están comprimidos a localizacións específicas do disco duro, creación de accesos directos no escritorio, creación de bases de datos en localizacións específicas, etcétera.

Existen ferramentas software que permiten automatizar este proceso que se chaman instaladores. No caso dunha instalación simple, pode ocorrer que o instalador xere uns arquivos que permitan ao usuario final facer de forma automática unha instalación guiada e sinxela; noutro caso a instalación ten que ser feita por persoal especializado.

O software pode pasar a produción despois de resolver o proceso de instalación, é dicir, pode ser utilizado e explotado polo cliente.

1.5.1.6 Mantemento

Esta fase permite mellorar e optimizar o software que está en produción. O mantemento permitirá realizar cambios no código para corrixir erros encontrados, para facer o código máis perfecto (optimizar rendemento e eficacia, reestruturar código, perfeccionar documentación, etcétera), para que evolucione (agregar, modificar ou eliminar funcionalidades segundo necesidades do mercado, etcétera), ou para que se adapte (cambios no hardware utilizado, cambios no software do sistema xestor de bases de datos, cambios no sistema de comunicacións, etcétera).

Ao redor do 2/3 partes do tempo invertido en Enxeñería de software está dedicado a tarefas de mantemento e ás veces estas tarefas son tantas e tan complexas que é menos custoso volver a deseñar o código.

As versións de software resultantes do mantemento noméanse de diferente maneira dependendo do fabricante. Por exemplo: Debian 7.6, NetBeans IDE 6.5, NetBeans IDE 7.0.1, NetBeans 8.0, Java SE 8u20 (versión 8 update 20). Algúns fabricantes subministran aplicación que son parches que melloran o software instalado como por exemplo Service Pack 1 (SP1) para Windows Server 2008 R2, ou Service Pack 2 (SP2) para Windows Server 2008 R2.



1.4 Buscar en internet as últimas versións de Debian, NetBeans, Java, Windows.

Solución

1.5.1.7 Documentación

A creación de documentación está asociada a tódalas fases anteriores e en especial ás fases de codificación, probas e instalación. Para estas últimas fases pódese distinguir entre:

- Documentación interna. É a que aparece no código dos programas para que os programadores encargados de mantelo poidan ter información extra sen moito esforzo e para que de forma automática poida extraerse fóra do código esa información nun formato lexible, como por exemplo HTML, PDF, CHM, etcétera.

Exemplo de código Java con comentarios Javadoc que aportan información extra e permiten que algunhas aplicacións xeren automaticamente unha páxina web coa información que extraen dos comentarios:

- Documentación externa que é a que se adxunta aos programas e pode estar dirixida:
 - Aos programadores. Formada por exemplo por: código fonte, explicación de algoritmos complicados, especificación de datos e formatos de entrada/saída, listado de arquivos que utiliza ou xera a aplicación, linguaxe de programación, descrición de condicións ou valores por defecto que utiliza, diagramas de deseño.
 - Aos usuario. Formada por exemplo por: requisitos do sistema (tipo de ordenador no que funciona, sistema operativo que require, recursos hardware necesarios, etcétera), detalles sobre a instalación, explicacións para utilizar o software de forma óptima, descrición de posibles erros de funcionamento ou instalación e a forma de corrixilos.
- Autodocumentación que é documentación á que se accede durante a execución dos programas e pode conter: índice de contidos, guías sobre o manexo do programa, asistentes, axuda contextual, autores dos programas, versións dos mesmos, direccións de contacto, enlaces de referencia..

1.5.2 Modelo en espiral

Este modelo baséase na creación dun prototipo do proxecto que se vai perfeccionando en sucesivas iteracións a medida que se engaden novos requisitos, pasando en cada iteración polo proceso de análise, deseño, codificación e probas descritos no modelo en cascada. Ao final de cada iteración o equipo que desenvolve o software e o cliente

analizaran o prototipo conseguido e acordarán se inician unha nova iteración. Sempre se traballa sobre un prototipo polo que no momento que se decida non realizar novas iteracións e acabar o produto, haberá que refinar o prototipo para conseguir a versión final acabada, estable e robusta.

1.5.3 Programación eXtrema

A programación eXtrema ou eXtreme Programming é un método de desenvolvemento áxil de software baseado en iteracións sobre as fases de planificación, deseño, codificación e probas.

1.5.3.1 Planificación

Cada reunión de planificación é coordinada polo xestor do proxecto e nela:

- O cliente indica mediante frases curtas as prestacións que debe ter o software sen utilizar ningún tipo de tecnicismos nin ferramenta especial.
- Os desenvolvedores de software converterán cada prestación en tarefas que duren como máximo tres días ideais de programación de tal xeito que a prestación completa non requira máis de 3 semanas. De non conseguir estas cifras, revisariáanse as prestación e as tarefas de acordo co desexo do cliente.
- Entre todos decídese o número de prestacións que formarán parte de cada iteración que se denomina velocidade do proxecto.
- Ao final de cada iteración farase unha reunión de planificación para que o cliente valore o resultado; se non o acepta, haberá que engadir as prestacións non aceptadas á seguinte iteración e o cliente deberá de reorganizar as prestacións que faltan para que se respecte a velocidade do proxecto.

É importante a mobilidade das persoas, é dicir, que en cada iteración os desenvolvedores traballen sobre partes distintas do proxecto, de forma que cada dúas ou tres iteracións, os desenvolvedores haxan traballado en todas as partes do sistema.

1.5.3.2 Deseño

Á diferenza do modelo en cascada, nesta fase utilízase unha tarxeta manual tipo CRC (class, responsibilities, collaboration) por cada obxecto do sistema, na que aparece o nome da clase, nome da superclase, nome das subclases, responsabilidades da clase, e obxectos cos que colabora. As tarxetas vanse colocando riba dunha superficie formando unha estrutura que reflicta as dependencias entre elas. As tarxetas vanse completando e recolocando de forma manual a medida que avanza o proxecto. Os desenvolvedores reuniranse periodicamente e terán unha visión do conxunto e de detalle mediante as tarxetas.

1.5.3.3 Codificación e probas

Unha diferenza desta fase con relación á fase de codificación do modelo en cascada é que os desenvolvedores teñen que acordar uns estándares de codificación (nomes de

variables, sangrías e aliñamentos, etcétera), e cumprilos xa que todos van a traballar sobre todo o proxecto.

Outra diferenza é que se aconsella crear os test unitarios antes que o propio código a probar xa que entón se ten unha idea máis clara do que se debe codificar.

Unha última diferenza é que se aconsella que os programadores desenvolvan o seu traballo por parellas (pair programming) xa que está demostrado que dous programadores traballando conxuntamente fronte ao mesmo monitor pasándose o teclado cada certo tempo, fano ao mesmo ritmo que cada un polo seu lado pero o resultado final é de moita máis calidade xa que mentres un está concentrado no método que está codificando, o outro pensa en como ese método afecta ao resto de obxectos e as dúbidas e propostas que xorden reducen considerablemente o número de erros e os problemas de integración posteriores.

1.5.4 Scrum

Este método está especialmente deseñado para proxectos de alto nivel de incerteza, carga laboral e prazos reducidos. O secreto do seu éxito está nos sprints: intervalos de tempo nos que se desenvolven miniproxectos dentro do proxecto principal.

Scrum é unha metodoloxía áxil para o desenvolvemento de proxectos que require maior rapidez e adaptabilidade nos seus resultados.

Planificación do sprint

Cada sprint ten un obxectivo particular.

Na primeira reunión do equipo defínense aspectos como a funcionalidade, obxectivos, riscos do sprint, prazos de entrega, entre outros.

Desenvolvemento

Cando o traballo do sprint está en curso, os encargados deben garantir que non se xeren cambios de último momento que podan afectar aos obxectivos do mesmo. Ademais, asegúrase o cumprimento dos prazos establecidos para o seu remate.

Revisión do sprint

Ao finalizar o desenvolvemento do miniproxecto, é posible analizar e avaliar os resultados. En caso de ser necesario, todo o equipo colaborará para saber os aspectos que necesitan ser cambiados. Nesta fase foméntase a colaboración e retroalimentación entre todos.

Retroalimentación

Os resultados poden entregarse para recibir unha realimentación non só por parte dos profesionais dentro do proxecto, senón tamén das persoas que utilizarán directamente o que se desexa lograr, é dicir, os clientes potenciais. As leccións aprendidas durante esta etapa permitirán que o seguinte sprint poda ser máis efectivo e áxil.

A metodoloxía scrum non se utiliza en todos os casos. Emprégase cando a empresa ten recursos, madurez e experiencia, unha estrutura organizacional áxil e innovadora, entre outros factores. Contar cun profesional que asegure estes principios será o primeiro paso.

1.5.5 Kanban

Kanban é unha palabra xaponesa que significa algo así como “tarxetas visuais” (kan significa visual e ban tarxeta). Esta técnica creouse en Toyota e utilízase para controlar o avance do traballo, no contexto dunha liña de produción.

Kanban non é unha técnica específica de desenvolvemento de software, o seu obxectivo é xestionar de maneira xeral como se van completando as tarefas, pero nos últimos anos tense utilizada na xestión de proxectos de desenvolvemento de software, a miúdo con Scrum.

As principais regras de Kanban son as tres seguintes:

- Visualizar o traballo e as fases do ciclo de produción ou fluxo de traballo
- Determinar o límite do traballo en curso
- Medir o tempo en completar unha tarefa

Ao igual que Scrum, Kanban baséase no desenvolvemento incremental, dividindo o traballo en partes. Unha das principais aportacións é que utiliza técnicas visuais para ver a situación de cada tarefa.

O traballo divídese en partes. Normalmente, cada unha destas partes escríbese nun post-it e pégase nun muro ou taboleiro. Os post-it soen ter información variada, pero, ademais da descrición, deberían ter unha estimación da duración da tarefas.

O muro ten tantas columnas como estados polos que pode pasar unha tarefa (exemplo, en espera de ser desenvolvida, en análise, en deseño, etc.

O obxectivo desta visualización é que quede claro o traballo a realizar, en que está traballando cada persoa, que todo o equipo teña algo que facer e ter claras as prioridades das tarefas. As fases do ciclo de produción ou fluxo de traballo decídense segundo o caso, non hai nada acoutado

Quizais unha das principais ideas de Kanban é que o traballo en curso debería estar limitado, édicir, que o número máximo de tarefas que se poden realizar en cada fase debe ser algo coñecido.

En Kanban debe definirse cantas tarefas como máximo poden realizarse en cada fase do ciclo de traballo (exemplo, como máximo 4 tarefas en desenvolvemento, como máximo 1 en probas, etc). A este número de tarefas chámase límite do “work in progress”. A isto engádeselle outra idea razoable como que para comezar cunha nova tarefa algunha outra tarefa previa debe ter finalizado.

1.5.6 Métrica v.3

Métrica versión 3 é unha metodoloxía de planificación, desenvolvemento e mantemento de sistemas de información promovido pola Secretaría de Estado de Administraciones Públicas do Ministerio de Hacienda y Administraciones Públicas e que cubre o desenvolvemento estruturado e o orientado a obxectos .

Esta metodoloxía ten como referencia o Modelo de Ciclo de Vida de Desenvolvemento proposto na norma ISO 12.207 "Information technology- Software live cycle processes".

Consta de tres procesos principais: planificación, desenvolvemento e mantemento. Cada proceso divídese en actividades non necesariamente de execución secuencial e cada actividade en tarefas.

No portal de administración electrónica (<http://administracionelectronica.gob.es/>) pódese acceder aos documentos pdf no que está detallada toda a metodoloxía e o persoal informático que intervéen en cada actividade.

1.5.6.1 Planificación

O proceso de planificación de sistemas de información (PSI) ten como obxectivo a obtención dun marco de referencia para o desenvolvemento de sistemas de información que respondan a obxectivos estratéxicos da organización e é fundamental a participación da alta dirección da organización. Consta das actividades:

- Descrición da situación actual.
- Un conxunto de modelos coa arquitectura da información.
- Unha proposta de proxectos a desenvolver nos próximos anos e a prioridade de cada un.
- Unha proposta de calendario para a execución dos proxectos.
- A avaliación dos recursos necesarios para desenvolver os proxectos do próximo ano.
- Un plan de seguimento e cumprimento de todo o proposto.

1.5.6.2 Desenvolvemento

Cada proxecto descrito na planificación ten que pasar polo proceso de desenvolvemento de sistemas de información que consta das actividades:

- Estudio de viabilidade do sistema (EVS) no que se analizan os aspectos económicos, técnicos, legais e operativos do proceso e se decide continuar co proceso ou abandonalo. No primeiro caso haberá que describir a solución encontrada: descrición, custo, beneficio, riscos, planificación da solución. A solución pode ser desenvolver software a medida, utilizar software estándar de mercado, solución manual ou unha combinación delas.

- Análise do sistema de información (ASI) para obter a especificación de requisitos software que conterá as funcións que proporcionará o sistema e as restricións ás que estará sometido, para analizar os casos de usos, as clases e interaccións entre elas, para especificar a interface de usuario, e para elaborar o plan de probas.
- Deseño do sistema de información (DSI) no que se fai o deseño de comportamento do sistema para cada caso de uso, o deseño da interface de usuario, o deseño de clase, o deseño físico de datos (se é necesario tamén se fai o deseño da migración e carga inicial de datos), a especificación técnica do plan de probas e o establecemento dos requisitos de implantación (implantación do sistema, formación de usuarios finais, infraestruturas, etcétera).
- Construción do sistema de información (CSI) no que se prepara a base de datos física, se prepara o entorno de construción, xérase o código, execútanse as probas unitarias, as de integración e as de sistema, elabóranse os manuais de usuario, defínese a formación dos usuarios finais e constrúense os compoñentes e procedementos da migración e carga inicial de datos.
- Implantación e aceptación do sistema (IAS) que ten como obxectivo a entrega e aceptación do sistema total e a realización de todas as actividades necesarias para o paso a produción. Para iso séguense os pasos: formar ao equipo de implantación, formar aos usuarios finais, realizar a instalación, facer a migración e carga inicial de datos, facer as probas de implantación (comprobar que o sistema funcione no entorno de operación), facer as probas de aceptación do sistema (comprobar que o sistema cumpre os requisitos iniciais do sistema), establecer o nivel de mantemento e servizo para cando o produto estea en produción. O último paso é o paso a produción para o que se analizarán os compoñentes necesarios para incorporar o sistema ao entorno de produción, de acordo ás características e condicións do entorno no que se fixeron as probas e se realiza a instalación dos compoñentes necesarios valorando a necesidade de facer unha nova carga de datos, unha inicialización ou unha restauración, fíxase a data de activación do sistema e a eliminación do antigo.

1.5.6.3 *Mantemento*

O obxectivo deste proceso é a obtención dunha nova versión do sistema de información desenvolvido con Métrica 3, a partir das peticións de mantemento que os usuarios realizan con motivo de problemas detectados no sistema ou pola necesidade de mellora do mesmo.



1.5 Buscar en internet nomes de metodoloxías áxiles de desenvolvemento de software.

Solución

1.6 Tipos de linguaxes de programación

Non existe unha clasificación das linguaxes informáticas adoptada pola maioría dos autores se non que hai grandes diferenzas entre eles. Unha clasificación posible é: linguaxes de marcas, especificación, consulta, transformación e programación.

1.6.1 Linguaxes de marcas

Permiten colocar distintivos ou sinais no texto que serán interpretadas por aplicacións ou procesos. Exemplos de linguaxes de marcas:

- A linguaxe XML (eXtensible Markup Language) que é unha metalinguaxe extensible pensada para a transmisión de información estruturada e que pode ser validada.
- As linguaxes HTML (Hiper Text Markup Language) ou XHTML (eXtensible Hiper Text Markup Language) =XML+HTML que serven ambas para publicar hipertexto na Word Wide Web.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
    Exemplo básico de XML
-->
<alumnos>
  <alumno>
    <nome>Pepe</nome>
    <apelidos>Ruiz Arias</apelidos>
  </alumno>
  <alumno>
    <nome>María Dolores</nome>
    <apelidos>González Paz</apelidos>
  </alumno>
</alumnos>
```

Exemplo de código XHTML editado en Notepad++:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<!-- Exemplo moi básico de XHTML con estilo externo -->
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Exemplo básico de xhtml</title>
    <meta content="text/html; charset=utf-8"
          http-equiv="Content-Type" />
    <link rel="stylesheet" href="exemplo_css.css"
          type="text/css" />
  </head>
  <body>
    <h1>Cabeceira principal</h1>
    <p class="principal">Este párrafo contén texto e un enlace a
      <a href="http://www.realacademiagalega.org/">Real
Academia Galega
    </a>.
    </p>
  </body>
</html>
```


1.6.2 Linguaxes de especificación

Describen algo de forma precisa. Por exemplo CSS (Cascading Style Sheets) é unha linguaxe formal que especifica a presentación ou estilo dun documento HTML, XML ou XHTML. Exemplo de código CSS que se podería aplicar ao exemplo XHTML anterior e editado en Notepad++:

```
body{
    font-family: "MS Sans Serif", Geneva, sans-serif;
    font-size: 15px;
    color: Black;
    border:black 2px double;
    padding: 40px;
    margin: 20px;}

h1 {
    font: 40px "Times New Roman", Times, serif;
    font-weight: bolder;
    word-spacing: 25px;}

p.principal{
    text-align: center;
    font: 10px "MS Serif", "New York", serif;}
```

1.6.3 Linguaxes de consultas

Permiten sacar ou manipular información de un grupo de información. Por exemplo, a linguaxe de consultas SQL (Standard Query Language) permite buscar e manipular información en bases de datos relacionais e a linguaxe XQuery permite buscar e manipular información en bases de datos XML nativas. Exemplo de script SQL:

```
CREATE DATABASE `empresa` ;

USE `empresa`;

CREATE TABLE `centros` (
    `cen_num` int(11) NOT NULL default '0',
    `cen_nom` char(30) default NULL,
    `cen_dir` char(30) default NULL,
    UNIQUE KEY `numcen` (`cen_num`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;

INSERT INTO `centros` VALUES
    (10,'SEDE CENTRAL','C/ ALCALA, 820-MADRID'),
    (20,'RELACION CON CLIENTES','C/ ATOCHA, 405-MADRID');

CREATE TABLE `deptos` (
    `dep_num` int(11) NOT NULL default '0',
    `dep_cen` int(11) NOT NULL default '0',
    `dep_dire` int(11) NOT NULL default '0',
    `dep_tipodir` char(1) default NULL,
    `dep_presu` decimal(9,2) default NULL,
    `dep_depen` int(11) default NULL,
    `dep_nom` char(20) default NULL,
    UNIQUE KEY `numdep` (`dep_num`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;

INSERT INTO `deptos` VALUES
    (122,10,350,'F','60000.00',120,'PROCESO DE DATOS'),
    (121,10,110,'P','200000.00',120,'PERSONAL'),
    (120,10,150,'P','30000.00',100,'ORGANIZACION'),
    (112,20,270,'F','90000.00',110,'SECTOR SERVICIOS'),
```

```
(111,20,400,'P','111000.00',110,'SECTOR INDUSTRIAL'),
(110,20,180,'P','15000.00',100,'DIRECCION COMERCIAL'),
(130,10,310,'P','20000.00',100,'FINANZAS'),
(200,20,600,'F','80000.00',100,'TRANSPORTES'),
(100,10,260,'P','120000.00',NULL,'DIRECCION GENERAL');
```

Exemplo de expresión FLOWR (for, let, order by, where, return) en XQuery:

```
for $libro in doc("libros.xml")/bib/libro
let $editorial := $libro/editorial
where $editorial="MCGRAW/HILL" or contains($editorial, "Toxosoutos")
return $libro
```

1.6.4 Linguaxes de transformación

Actúan sobre unha información inicial par obter outra nova. Por exemplo, a linguaxe XSLT (eXtensible Stylesheet Language Transformations) permite describir as transformacións que se van realizar sobre un documento XML para obter outro arquivo. Exemplo de transformación XSL sinxela editada en NetBeans, que cando actúa sobre o exemplo XML anterior obtén unha páxina HTML cunha lista dos alumnos:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Exemplo sinxelo de transformación XSL -->
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">
    <xsl:output method="html"/>
    <xsl:template match="alumnos">
        <html>
            <head>
                <title>fundamentos.xml</title>
            </head>
            <body>
                <h1>Alumnos</h1>
                <ul>
                    <xsl:for-each select="alumno">
                        <li>
                            <xsl:value-of select="apelidos"/>,
                            <xsl:value-of select="nome"/>
                        </li>
                    </xsl:for-each>
                </ul>
            </body>
        </html>
    </xsl:template>
</xsl:stylesheet>
```

1.6.5 Linguaxes de programación

Permiten comunicarse cos dispositivos hardware e así poder realizar un determinado proceso e para iso poden manexar estruturas de datos almacenados en memoria interna ou externa, e utilizar estruturas de control. Dispoñen dun léxico, e teñen que cumprir regras sintácticas e semánticas.

O léxico é o conxunto de símbolos que se poden usar e poden ser: identificadores (nomes de variables, tipos de datos, nomes de métodos, etcétera), constantes, variables, operadores, instrucións e comentarios.

A regras de sintaxe especifican a secuencia de símbolos que forman unha frase ben escrita nesa linguaxe, é dicir, para que non teña faltas de ortografía.

As regras de semántica definen como teñen que ser as construcións sintácticas e as expresións e tipos de datos utilizadas.

Exemplo de código Java sinxelo editado en NetBeans:

```
package parimpar;

import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        short n;
        Scanner teclado = new Scanner(System.in);
        System.out.printf("Teclee o número enteiro entre %d e %d:",
            Short.MIN_VALUE, Short.MAX_VALUE);
        n = teclado.nextShort();
        if (n%2==0) {
            System.out.printf("%d é par\n", n);
        }
        else {
            System.out.printf("%d é impar\n", n);
        }
    }
}
```

1.7 Clasificación das linguaxes de programación

As linguaxes de programación poden clasificarse segundo o afastadas ou próximas que estean do hardware, por xeracións, polo paradigma da programación, pola forma de traducirse a linguaxe máquina e de executarse, e na arquitectura cliente servidor.

Clasificación segundo a distancia ao hardware

Poden clasificarse en linguaxes de baixo e alto nivel.

1.7.1 Linguaxes de baixo nivel

Están baseados directamente nos circuitos electrónicos da máquina polo que un programa escrito para unha máquina non poderá ser utilizada noutra diferente. Poden ser linguaxe máquina ou linguaxe ensambladora.

A linguaxe máquina é código binario (ceros e uns) ou hexadecimal (números de 0 a 9 e letras de A a F) que actúa directamente sobre o hardware. É a única linguaxe que non necesita tradución xa que o procesador recoñece as instrucións directamente.

A codificación en linguaxe ensambladora é mnemotécnica, é dicir, utiliza etiquetas para describir certas operacións. É necesario traducir o código ensamblador a linguaxe máquina para que o procesador recoñeza as instrucións.

1.7.2 Linguaxes de alto nivel

Tentan acercarse á linguaxe humana e sepáranse do coñecemento interno da máquina e por iso necesitan traducirse a linguaxe máquina. Esta tradución fai que a execución sexa

máis lenta que nas linguaxes de baixo nivel pero ao non depender do procesador poden executarse en diferentes ordenadores.

1.8 Clasificación das linguaxes de programación por xeracións

Unha vez establecido o concepto de programa de ordenador e os conceptos de código fonte, código obxecto e código executable (así como o de máquina virtual), agora é necesario establecer as diferenzas entre os distintos tipos de código fonte existente, a través dos cales é posible obter un programa informático.

Unha linguaxe de programación é unha linguaxe que permite establecer unha comunicación entre o home e a máquina. A linguaxe de programación identificará o código fonte, que o programador desenvolverá para indicarlle á máquina, unha vez que ese código se converta en código executable, que pasos debe seguir.

Nos últimos anos houbo unha evolución constante nas linguaxes de programación. Estableceuse unha evolución crecente na que se incorporan elementos que permiten a creación de programas cada vez máis sólidos e eficientes. Isto facilita moito a tarefa do programador para o desenvolvemento, mantemento e adaptación de software. Hoxe en día, incluso hai linguaxes de programación que permiten a creación de aplicacións informáticas para persoas sen coñecementos técnicos de informática, debido á existencia dunha creación practicamente automática do código a partir dalgunhas preguntas.

Os diferentes tipos de linguaxes son:

- Linguaxe de primeira xeración ou linguaxe de máquina.
- Linguaxes de segunda xeración ou linguaxes ensambladores.
- Linguaxes de terceira xeración ou linguaxes de alto nivel.
- Linguaxes de cuarta xeración ou linguaxes de propósito específico.
- Linguaxes de quinta xeración.

O primeiro tipo de linguaxe que se desenvolveu é a chamada linguaxe de primeira xeración ou linguaxe de máquina. É a única linguaxe que o ordenador entende directamente.

A súa estrutura está totalmente adaptada aos circuitos impresos de ordenadores ou procesadores electrónicos e moi lonxe da forma de expresión e análise de problemas humanos (as instrucións exprésanse en código binario). Isto fai que a programación nesta linguaxe sexa tediosa e complicada, xa que é necesario un profundo coñecemento da arquitectura física do ordenador. Ademais, debe apreciarse que o código da máquina fai que o programador poida usar todos os recursos do hardware, de xeito que se poden obter programas moi eficientes.

Actualmente, debido á complexidade de desenvolver este tipo de linguaxe, está practicamente obsoleto. Só se usará en procesadores moi específicos ou para funcionalidades moi específicas.

O segundo tipo de linguaxe de programación é a linguaxe da segunda xeración ou linguaxe ensambladora. Esta é a primeira linguaxe de programación que usa códigos mnemotécnicos para indicar á máquina as operacións a realizar. Estas operacións, moi básicas, deseñáronse a partir do coñecemento da estrutura interna da propia máquina.

Cada instrución de linguaxe ensambladora corresponde a unha instrución de linguaxe máquina. Este tipo de linguaxes dependen enteiramente do procesador usado pola máquina, por iso se di que están orientadas á máquina.

A partir do código escrito en linguaxe ensamblador, o programa tradutor (ensamblador) convérteo en código de primeira xeración, que será interpretado pola máquina. Este tipo de linguaxe úsase xeralmente para programar controladores ou aplicacións en tempo real, xa que require un uso moi eficiente da velocidade e da memoria.

1.8.1 Características das linguaxes de primeira e segunda xeración

Pódense establecer vantaxes das linguaxes de primeira e segunda xeración:

- Permiten escribir programas altamente optimizados que aproveiten ao máximo o hardware dispoñible.
- Permiten ao programador especificar exactamente que instrucións quere executar.

As desvantaxes son as seguintes:

- Os programas escritos en linguaxes de baixo nivel están completamente ligados ao hardware onde se executarán e non se poden transferir facilmente a outros sistemas con hardware diferente.
- Debe coñecer en profundidade o sistema e a arquitectura do procesador para escribir bos programas.
- Non permiten expresar de forma directa conceptos habituais a nivel de algoritmo.
- Son difíciles de codificar, documentar e manter.

O seguinte grupo de linguaxes coñécese como linguaxes de terceira xeración ou linguaxes de alto nivel . Estas linguaxes máis evolucionadas empregan palabras e frases relativamente fáciles de entender e tamén ofrecen facilidades para expresar alteracións no fluxo de control dun xeito bastante sinxelo e intuitivo.

As linguaxes de terceira xeración ou de alto nivel úsanse cando se quere desenvolver aplicacións grandes e complexas, onde a prioridade é facilitar e comprender como facer as cousas (linguaxe humana) sobre o rendemento do software ou o seu uso de a memoria.

Os esforzos por facer a tarefa de programación independente da máquina onde se executará resultaron na aparición de linguaxes de programación de alto nivel.

As linguaxes de alto nivel adoitan ser fáciles de aprender porque están compostas por elementos de linguaxes naturais, como o inglés. O seguinte é un exemplo dun algoritmo implementado nunha linguaxe de alto nivel, concretamente Basic. Este algoritmo calcula a factorial dun número dado á función como parámetro. Pódese ver con que facilidade se entende cun coñecemento mínimo de inglés.

Como resultado desta distancia da máquina e do achegamento ás persoas, os programas escritos en linguaxes de programación de terceira xeración non poden ser interpretados directamente polo ordenador, pero é necesario levar a cabo a súa tradución á linguaxe da máquina de antemán. . Hai dous tipos de tradutores: compiladores e intérpretes.

1.8.1.1 Compiladores

Son programas que traducen o programa escrito cunha linguaxe de alto nivel á linguaxe máquina. O compilador detectará posibles erros do programa de orixe para obter un programa executable depurado.

Alguns exemplos de códigos de programación que terán que pasar por un compilador son: Pascal, C, C ++, .NET, ...

O procedemento que debe seguir un programador é o seguinte:

- Crea o código fonte.
- Crea código executable usando compiladores e ligadores.
- O código executable depende de cada sistema operativo. Para cada sistema hai un compilador, é dicir, se queres executar o código con outro sistema operativo tes que recompilar o código fonte.
- O programa resultante execútase directamente desde o sistema operativo.

1.8.1.2 Os intérpretes

O intérprete tamén é un programa que traduce código de alto nivel á linguaxe da máquina, pero a diferenza do compilador, faino en tempo de execución. É dicir, non se realiza un proceso previo de tradución de todo o programa fonte a código de bytes, senón que se traduce e execútase instrución por instrución.

Alguns exemplos de códigos de programación que terán que pasar por un intérprete son: JavaScript, PHP, ASP ...

Algunhas características das linguaxes interpretadas son:

- O código interpretado non o executa directamente o sistema operativo, senón que fai uso dun intérprete.
- Cada sistema ten o seu propio intérprete.

O intérprete é sensiblemente máis lento que o compilador, xa que realiza a tradución ao mesmo tempo que a execución. Ademais, esta tradución faise sempre que se executa o programa, mentres que o compilador só a realiza unha vez. A vantaxe dos intérpretes é que fan que os programas sexan máis portátiles. Deste xeito, un programa compilado nun ordenador con Windows non funcionará nun Macintosh nin nun ordenador con Linux, a non ser que o programa de orixe se recompila no novo sistema.

1.9 Características das linguaxes de terceira, cuarta e quinta xeración

As linguaxes de terceira xeración son aquelas que son capaces de conter e executar, nunha soa instrución, o equivalente a varias instrucións dunha lingua de segunda xeración.

As vantaxes das linguaxes de terceira xeración son:

- O código dos programas é moito máis sinxelo e comprensible.
- Son independentes do hardware (non fan referencia a el). Por esta razón é posible "transportar" o programa entre diferentes ordenadores / arquitecturas / sistemas operativos (sempre que haxa un compilador para esta linguaxe de alto nivel no sistema de destino).
- É máis fácil e rápido escribir programas e máis fácil de manter.

As desvantaxes das linguaxes de terceira xeración son:

- A súa execución nun ordenador pode ser máis lenta que o propio programa escrito en linguaxe de baixo nivel, aínda que isto depende moito da calidade do compilador que realiza a tradución.

Exemplos de linguaxes de programación de terceira xeración: C, C ++, Java, Pascal ...

A lingua das linguaxes de cuarta xeración ou propósito específico . Proporcionan un nivel moi alto de abstracción na programación, o que permite o desenvolvemento de aplicacións sofisticadas nun curto espazo de tempo, moi inferior ao necesario para as linguaxes de 3a xeración.

Algúns aspectos que antes se tiñan que facer a man están automatizados. Inclúen ferramentas orientadas ao desenvolvemento de aplicacións (IDE) que permiten definir e xestionar bases de datos, realizar informes (por exemplo, informes Oracle), consultas (por exemplo, informix 4GL), ... módulos, escribindo moi poucas liñas de código ou ningunha. Permiten crear prototipos dunha aplicación rapidamente. Os prototipos ofrécenche unha idea de como é a aplicación e de como funciona antes de rematar o código. Isto facilita a obtención dun programa que satisfaga as necesidades e expectativas do cliente.

Algúns dos aspectos positivos deste tipo de linguaxe de programación son:

- Maior abstracción.
- Menos esforzo de programación.
- Menor custo de desenvolvemento de software.
- Baseado na xeración de código a partir de especificacións de moi alto nivel.
- As solicitudes pódense realizar sen ser un experto en linguaxes.
- Normalmente teñen un conxunto limitado de instrucións.
- Son específicos do produto que lles ofrece.

Estas linguaxes de programación de cuarta xeración están orientadas basicamente a aplicacións empresariais e xestión de bases de datos.

Algúns exemplos de linguaxes de cuarta xeración son Visual Basic, Visual Basic .NET, SAP ABAP, FileMaker, PHP, ASP, 4D ...

As linguaxes de quinta xeración son linguaxes específicas para o tratamento de problemas relacionados coa intelixencia artificial e sistemas expertos.

En lugar de executar só un conxunto de comandos, o obxectivo destes sistemas é "pensar" e anticiparse ás necesidades dos usuarios. Estes sistemas aínda están en desenvolvemento. Sería o paradigma lóxico.

Algúns exemplos de linguaxes de quinta xeración son Lisp ou Prolog.

1.10 Clasificación polo paradigma da programación

Un paradigma de programación é unha metodoloxía ou filosofía de programación a seguir cun núcleo central incuestionable, é dicir, as linguaxes que utilizan o mesmo paradigma de programación utilizarán os mesmos conceptos básicos para programar. A evolución das metodoloxías de programación e a das linguaxes van parellas ao longo do tempo. Poden clasificarse en dous grandes grupos: o grupo que segue o paradigma imperativo e o que segue o paradigma declarativo.

1.10.1 Paradigma imperativo

O código está formado por unha serie de pasos ou instrucións para realizar unha tarefa organizando ou cambiando valores en memoria. As instrucións execútanse de forma secuencial, é dicir, hasta que non se executa unha non se pasa a executar a seguinte. Por exemplo, Java, C, C++, PHP, JavaScript, e Visual Basic.

Dentro das linguaxes imperativas distínguese entre que as que seguen a metodoloxía estruturada e as que seguen a metodoloxía orientada a obxectos.

A finais dos anos 60 naceu a metodoloxía estruturada que permitía tres tipos de estruturas no código: a secuencial, a alternativa (baseada nunha decisión) e a repetitiva (bucles). Os programas están formados por datos e esas estruturas.

A metodoloxía estruturada evolucionou engadindo o módulo como compoñente básico dando lugar á programación estruturada e modular. Os programas estaban formados por módulos ou procesos por un lado e datos por outro. Os módulos necesitaban duns datos de entrada e obtían outros de saída que á súa vez podían ser utilizados por outros módulos. Por exemplo, C é unha linguaxe estruturada e modular.

Nos anos 80 apareceu a metodoloxía orientada a obxectos que considera o obxecto como elemento básico. Esta metodoloxía popularizouse a principios dos 90 e actualmente é a máis utilizada. Por exemplo, C++ e Java son linguaxes orientadas a obxectos. Cada obxecto segue o patrón especificado nunha clase. Os programas conteñen obxectos que se relacionan ou colaboran con outros obxectos da súa mesma clase ou doutras clases. A clase está composta de atributos (datos) e métodos (procesos) e pode ter as propiedades:

- Herdanza. Poden declararse clases filla que herdán as propiedades e métodos da clase nai. Os métodos herdados poden sobrecargarse, é dicir, dentro da mesma clase pode aparecer definido con distinto comportamento o mesmo método con diferente firma (número de parámetros e tipo dos mesmos).
- Polimorfismo. Propiedade que permite que un método se comporte de diferente maneira dependendo do obxecto sobre o que se aplica.
- Encapsulamento. Permite ocultar detalles internos dunha clase.

1.10.2 Paradigma declarativo

O código indica que é o que se quere obter e non como se ten que obter, é dicir, é un conxunto de condicións, proposicións, afirmacións, restricións, ecuacións ou transformacións que describen o problema e detallan a súa solución. O programador ten que pensar na lóxica do algoritmo e non na secuencia de ordes para que se leve a cabo. Por exemplo, Lisp e Prolog son linguaxes declarativas.

1.11 Paradigmas de programación

É difícil establecer unha clasificación xeral das linguaxes de programación, xa que hai un gran número de linguaxes e ás veces versións diferentes dunha mesma linguaxe. Isto fará que en calquera clasificación que se faga a mesma lingua pertenza a máis dun dos grupos establecidos. Unha clasificación moi estendida, tendo en conta o funcionamento dos programas e a filosofía coa que foron concibidos, é a seguinte:

- Paradigma imperativo / estruturado.
- Paradigma dos obxectos.
- Paradigma funcional.
- Paradigma lóxico.

O paradigma imperativo / estruturado debe o seu nome ao papel dominante que xogan as oracións imperativas, é dicir, as que indican realizar unha determinada operación que modifica os datos almacenados na memoria.

Algunhas das linguaxes imperativas son C, Basic, Pascal, Cobol ...

A técnica seguida na programación imperativa é a programación estruturada. A idea é que calquera programa, por complexo e grande que sexa, pode estar representado por tres tipos de estruturas de control:

- Secuencia.
- Selección.
- Iteración.

Por outra banda, tamén se propón desenvolver o programa coa técnica de deseño de arriba abaixo . É dicir, modular o programa creando porcións máis pequenas de programas con tarefas específicas, que se subdividen noutros subprogramas cada vez máis pequenos. A idea é que estes subprogramas normalmente chamados funcións ou procedementos deben resolver un único obxectivo ou tarefa.

Imaxinemos que temos que facer unha solicitude que rexistre os datos básicos do persoal dun colexio, datos como o nome, o DNI e que calcule o salario dos profesores así como o dos administrativos, onde o salario dos administrativos é o salario. base (SOL_BASE) * 10 mentres que o salario dos profesores é salario base (SOL_BASE) + número de horas impartidas (numHoras) * 12.

```
const float SOL_BASE = 1.000;

Struct Administrativo
{
    string nom;
    string DNI;
    float salario;
}

Struct Profesor
{
    string nom;
    string DNI;
    int numHoras;
    float salario;
}

void AsignarsalarioAdministrativo (Administrativo Administrativo1)
{
    Administrativo1. salario = SOL_BASE * 10;
}

void AsignarsalarioProfesor (Profesor Profesor1)
{
    Profesor1. salario = SOL_BASE + (numHoras * 12);
}
```

O paradigma de obxectos, normalmente coñecido como Object Oriented Programming (OOP, ou OOP), é un paradigma de creación de programas baseado nunha abstracción do mundo real. Nun programa orientado a obxectos, a abstracción non son

procedementos ou funcións senón obxectos. Estes obxectos son unha representación directa de algo no mundo real, como un libro, unha persoa, un pedido, un empregado ...

Algunhas das linguaxes de programación orientadas a obxectos son C ++, Java, C # ...

Un obxecto é unha combinación de datos (chamados atributos) e métodos (funcións e procedementos) que nos permiten interactuar con el. Neste tipo de programación, polo tanto, os programas son conxuntos de obxectos que interactúan entre si a través de mensaxes (chamadas a métodos).

A programación orientada a obxectos baséase na integración de 5 conceptos: abstracción, encapsulación, modularidade, xerarquía e polimorfismo, que cómpre entender e seguir dun xeito absolutamente rigoroso. Non seguilos sistematicamente, omitilos puntualmente ás prásas ou por outras razóns fainos perder todo o valor e os beneficios que nos proporciona a orientación ao obxecto.

```
class Traballador {
    private:
        string nom;
        string DNI;
    protected:
        static const float SOL_BASE = 1.000;
    public:
        string GetNom() {return this.nom;}
        void SetNom (string n) {this.nom = n;}
        string GetDNI() {return this.DNI;}
        void SetDNI (string dni) {this.DNI = dni;}
        float salario() = 0;
}

class Administrativo: public Traballador {
    public:
        float Salario() {return SOL_BASE * 10;}
}

class Profesor: public Traballador {
    private:
        int numHoras;
    public:
        float Salario() {return SOL_BASE + (numHoras * 15);}
}
```

O paradigma funcional baséase nun modelo matemático. A idea é que o resultado dun cálculo sexa a entrada do seguinte, e así sucesivamente ata que unha composición produza o resultado desexado.

Os creadores das primeiras linguaxes funcionais pretendían convertelas en linguaxes de uso universal para o procesamento de datos en todo tipo de aplicacións, pero co paso do tempo foise empregando principalmente nos campos da investigación científica e das aplicacións matemáticas. .

Unha das linguaxes máis típicas do paradigma funcional é Lisp. Vexa un exemplo de programación factorial con esta linguaxe:

```
> (defun factorial (n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
FACTORIAL
> (factorial 3)
6
```

O paradigma lóxico ten como principal característica a aplicación das regras da lóxica para inferir conclusións a partir de datos.

Un programa lóxico contén unha base de coñecemento sobre a que se fan consultas. A base de coñecemento está composta por feitos, que representan a información do sistema expresada como relacións entre datos e regras lóxicas que permiten deducir consecuencias a partir de combinacións entre feitos e, en xeral, outras regras.

Unha das linguaxes máis típicas do paradigma lóxico é Prolog.

O paradigma úsase amplamente en aplicacións relacionadas coa intelixencia artificial, particularmente no campo dos sistemas expertos e no procesamento da linguaxe humana. Un sistema experto é un programa que imita o comportamento dun experto humano. Polo tanto, contén información (é dicir, unha base de coñecemento) e unha ferramenta para comprender as preguntas e atopar a resposta correcta examinando a base de datos (un motor de inferencias).

Tamén é útil en problemas combinatorios ou require unha gran cantidade ou gama de solucións alternativas, segundo a natureza do mecanismo de retroceso (backtracking).

1.12 Características das linguaxes máis estendidas

Hai moitas linguaxes de programación diferentes, ata o punto de que moitas tecnoloxías teñen a súa propia linguaxe. Cada unha destas linguaxes ten unha serie de peculiaridades que a fan diferente do resto.

As linguaxes de programación máis comúns son as máis empregadas en cada unha das distintas áreas da informática. No campo da educación, por exemplo, unha linguaxe de programación moi utilizada é MODULA-2 que se usa en moitas universidades ou centros educativos para ensinar aos principiantes na programación.

As linguaxes de programación máis estendidas correspondentes a diferentes campos, tecnoloxías diferentes ou diferentes tipos de programación teñen unha serie de características en común que son as que marcan as similitudes entre todos.

1.12.1 Características da programación estruturada

A programación estruturada foi desenvolvida polo holandés Edsger W. Dijkstra e baséase no chamado teorema da estrutura. Polo tanto usa só tres estruturas: secuencia, selección

e iteración, sendo innecesaria a utilización da instrución ou instrucións transferencia incondicionalmente (GOTO, EXIT FUNCTION, EXIT SUB ou múltiples RETURN).

Así, as características da programación estruturada son a claridade, o teorema da estrutura e o deseño de arriba abaixo.

1.12.1.1 Claridade

Debe haber suficiente información no código para que o programa poida ser comprendido e verificado: comentarios, nomes de variables comprensibles e procedementos comprensibles ... Calquera programa estruturado pode lerse de principio a fin sen interrupción na secuencia de lectura normal. .

1.12.1.2 Teorema da estrutura

Proba que calquera programa pode escribirse usando só as tres estruturas básicas de control:

Secuencia: instrucións executadas sucesivamente, unha tras outra.

Selección: a instrución condicional con dobre alternativa, da forma "se condición, entón OraciónA, pero OraciónB"

Iteración: o bucle condicional "while condition, do SentenceA", que executa as instrucións repetidamente mentres se cumpre a condición. Na figura .9 pódese ver unha figura que ilustra a estrutura básica de iteración.

1.12.1.3 Deseño descendente

O deseño descendente é unha técnica que se basea no concepto de "dividir e conquistar" co fin de resolver un problema no campo da programación. Trátase de resolver o problema ao longo de diferentes niveis de abstracción partindo dun nivel máis abstracto e rematando nun nivel de detalle.

A visión moderna da programación estruturada introduce as características da programación modular e dos tipos de datos abstractos (TAD).

1.12.1.4 Programación modular

Realizar un programa sen seguir unha técnica de programación modular adoita producir un enorme conxunto de frases cuxa execución é complexa de seguir e comprender, facendo case imposible depurar erros e a introdución de melloras. Pode que incluso teña que abandonar o código preexistente porque é máis sinxelo comezar de novo.

Cando falamos de programación modular, referímonos á división dun programa en partes máis manexables e independentes. Unha regra práctica para acadar este propósito é establecer que cada segmento do programa non supere, en lonxitude, un puñado de codificación.

Na maioría das linguaxes, os módulos tradúcense a:

Procedementos: son subprogramas que realizan unha tarefa dada e devolven 0 ou máis dun valor. Úsanse para estruturar un programa e mellorar a súa claridade.

Funcións: son subprogramas que realizan unha tarefa concreta e devolven un único resultado ou valor. Úsanse para crear novas operacións que o linguaxe non ofrece.

1.12.2 Tipos de datos abstractos (TAD)

Na programación, o tipo de datos dunha variable é o conxunto de valores que pode asumir a variable. Por exemplo, unha variable booleana só pode tomar dous valores posibles: verdadeiro, falso. Ademais, hai un conxunto limitado pero ben definido de operacións que teñen sentido sobre os valores dun tipo de datos; así, as operacións típicas no tipo booleano son AND ou OR.

As linguaxes de programación supoñen un certo número de tipos de datos, que poden variar dun linguaxe a outro; Así, Pascal ten o todo , o real , o booleano , os caracteres ... Este tipo de datos chámanse tipos de datos básicos no contexto das linguaxes de programación.

Ata hai uns anos, toda a programación baseábase neste concepto de tipo e non foron poucos os problemas que xurdiron, especialmente relacionados coa complexidade dos datos que había que definir. Xurdiu a posibilidade de poder definir tipos abstractos de datos , onde o programador pode definir un novo tipo de datos e as súas posibles operacións.

1.13 Características da programación orientada a obxectos

Un dos conceptos importantes introducidos pola programación estruturada é a abstracción de funcionalidades a través de funcións e procedementos. Esta abstracción permite que un programador use unha función ou procedemento sabendo só o que fai, pero sen saber o detalle de como o está a facer.

Non obstante, este feito ten varios inconvenientes:

- As funcións e os procedementos comparten os datos do programa, o que provoca que os cambios nun deles afecten ao resto.
- Ao deseñar unha aplicación é moi difícil predicir en detalle que funcións e procedementos necesitaremos.
- A reutilización do código é difícil e acaba copiando e pegando certos anacos de código e axustándoos. Isto é especialmente común cando o código non é modular.
- A orientación a obxectos, concibida nos anos 70 e 80, pero estendida a partir dos 90, permitiu superar estas limitacións.

A orientación do obxecto (en diante OO) é un paradigma para construír programas baseados nunha abstracción do mundo real.

Nun programa orientado a obxectos, a abstracción non son procedementos nin funcións, son obxectos. Estes obxectos son unha representación directa de algo no mundo real, como un libro, unha persoa, unha organización, unha orde, un empregado ...

Un obxecto é unha combinación de datos (chamados atributos) e métodos (funcións e procedementos) que nos permiten interactuar con el. En OO, entón, os programas son conxuntos de obxectos que interactúan entre si a través de mensaxes (chamadas a métodos).

As linguaxes OOP (programación orientada a obxectos) son aquelas que implementan máis ou menos fielmente o paradigma de OO. A programación orientada a obxectos baséase na integración de 5 conceptos: abstracción, encapsulación, modularidade, xerarquía e polimorfismo, que cómpre entender e seguir dun xeito absolutamente rigoroso. Non seguilos sistematicamente ou omitilos puntualmente, por présas ou por outras razóns, fai que perdas todo o valor e os beneficios que aporta a orientación ao obxecto.

1.13.1 Abstracción

É o proceso no que se separan as propiedades máis importantes dun obxecto das que non o son. É dicir, a abstracción define as características esenciais dun obxecto do mundo real, os atributos e comportamentos que o definen como tal e logo modélanos nun obxecto de software. No proceso de abstracción a implementación de cada método ou atributo non debe ser preocupante, só é preciso definilos.

Na tecnoloxía orientada a obxectos a principal ferramenta para apoiar a abstracción é a clase . Unha clase pódese definir como unha descrición xenérica dun grupo de obxectos que comparten características comúns, que se especifican nos seus atributos e comportamentos.

1.13.1.1 Encapsulamento

Permite aos obxectos escoller que información se publica e que información está oculta no resto de obxectos. É por iso que os obxectos normalmente presentan os seus métodos como interfaces públicas e os seus atributos como datos privados ou protexidos, sendo inaccesibles a outros obxectos. As características que se poden conceder son:

- Público: calquera clase pode acceder e usar calquera atributo ou método declarado público.
- Protexido: calquera clase herdada pode acceder e usar calquera atributo ou método declarado protexido na clase pai.
- Privado: ningunha clase pode acceder e usar un atributo ou método declarado privado.

1.13.1.2 Modularidade

Permite modificar as características de cada unha das clases que definen un obxecto, independentemente das outras clases da aplicación. Noutras palabras, se unha aplicación pódese dividir en módulos separados, normalmente clases, e estes módulos poden compilarse e modificarse sen afectar aos demais, entón a aplicación implementouse nunha linguaxe de programación que admita a modularidade.

1.13.1.3 Xerarquía

Permite a ordenación de abstraccións. As dúas xerarquías máis importantes dun sistema complexo son a herdanza e a agregación.

A herdanza tamén se pode ver como un xeito de compartir código, polo que ao usar a herdanza para definir unha nova clase só tes que engadir o que é diferente, é dicir, reutilizar métodos e variables, e especialízase no comportamento.

Por exemplo, pódese chamar a unha clase pai nomeada *treballador* dúas clases fillas, é dicir, dous subtipos de traballadores administrativos e profesores.

A agregación é un obxecto que está composto por unha combinación doutros obxectos ou compoñentes. Así, un ordenador está composto por unha CPU, unha pantalla, un teclado e un rato, e estes compoñentes non teñen sentido sen o ordenador.

1.13.1.4 Polimorfismo

É unha característica que permite dar diferentes formas a un método, xa sexa na definición ou na implementación.

O método de sobrecarga (overload) consiste en implementar o mesmo método varias veces pero con parámetros diferentes, de xeito que invocándoo, o compilador decide que métodos se deben executar, dependendo dos parámetros da chamada.

Un exemplo de método sobrecargado é o que calcula o salario dun traballador nunha empresa. Dependendo do posto que ocupe o traballador terá máis ou menos conceptos na súa nómina (máis ou menos incentivos, por exemplo).

O mesmo método pode chamarse *CalcularSalario* implementarse de forma diferente segundo calcule o salario dun traballador (con menos elementos na súa lista, o que significa que o método recibe menos variables) ou se calcula salario dun xerente.

Os métodos de sobrescritura (override) consisten en reimplementar un método herdado dunha superclase exactamente a mesma definición (incluído o nome do método, os parámetros e o valor de retorno).

Un exemplo de sobrecarga de método podería ser o de método *Area()*. A partir dunha clase *Figura* que contén o método *Area()*, existe unha clase derivada para algúns tipos de figuras (por exemplo, *Rectangulo* ou *Cadrado*).

A implementación do método *Area()* será diferente en cada unha das clases derivadas; estes pódense implementar de xeito diferente (dependendo de como se calcule a área da figura en cada caso) ou definirse de xeito diferente.

1.14 Proceso de xeración de código

A xeración de código consta dos procesos de edición, compilación, e enlace. Cando o código estea finalizado, poderase executar para producir resultados.

1.14.1 Edición

Esta fase consiste en escribir o algoritmo de resolución nunha linguaxe de programación mediante un editor de texto ou unha ferramenta de edición incluída nun contorno de desenvolvemento. O código resultante chámase código fonte e o arquivo correspondente chámase arquivo fonte.

1.14.2 Compilación

Consiste en analizar e sintetizar o código fonte mediante un compilador, para obter, se non se atopan erros, o código obxecto ou un código intermedio multiplataforma. As persoas non entende ese código e non se pode executar directamente.

Esta fase non se aplicará ás linguaxes interpretadas aínda que estas poden ter ferramentas que permitan facer un análise léxico e sintáctico antes de pasar a executarse.

1.14.2.1 Análise

As análises realizadas son:

- Análise léxico no que se comproba que os símbolos utilizados sexan correctos, incluídos os espazos en branco.
- Análise sintáctico no que se comproba que as agrupacións de símbolos cumpran as regras da linguaxe.
- Análise semántico no que se fan o resto de comprobacións, como por exemplo, que as variables utilizadas estean declaradas, ou a coherencia entre o tipo de dato dunha variable e o valor almacenado nela, ou a comparación do número e tipo de parámetros entre a definición e unha chamada a un método.

1.14.2.2 Síntese

A síntese permite:

- A xeración de código intermedio independente da máquina. Algunhas linguaxes compiladas como C pasan antes por unha fase de preprocesamento na que se levan a cabo operacións como substituír as constantes polo valor ou incluír arquivos de cabeceira. Outras linguaxes como Java xeran bytecode Java que xa poderá ser executado nunha JVM e outras como C# xera o código CIL que será executado no contorno CLR.
- A tradución do código intermedio anterior a código máquina para obter o código obxecto. Esta tradución leva consigo tamén unha optimización do código. Este novo código aínda non está listo para executarse directamente.

1.14.3 Enlace

Esta fase consiste en enlazar mediante un programa enlazador o arquivo obxecto obtido na compilación con módulos obxectos externos para obter, se non se atopan erros, o arquivo executable.

O arquivo obxecto obtido na compilación pode ter referencias a códigos obxecto externos que forman parte de bibliotecas externas estáticas ou dinámicas:

- Se a biblioteca é estática, o enlazador engade os códigos obxecto das bibliotecas ao arquivo obxecto, polo que o arquivo executable resultante aumenta de tamaño con relación ao arquivo obxecto, pero non necesita nada máis que o sistema operativo para executarse.
- Se a biblioteca é dinámica (Dynamic Link Library - DLL en Windows, Shared objects en Linux), o enlazador engade só referencias á biblioteca, polo que o arquivo executable resultante apenas aumenta de tamaño con relación ao arquivo obxecto,

pero a biblioteca dinámica ten que estar accesible cando o arquivo executable se execute.

1.14.4 Execución

A execución necesita de ferramentas diferentes dependendo de se a linguaxe é interpretada, compilada ou de máquina virtual ou execución administrada.

Se a linguaxe é interpretada, será necesario o arquivo fonte e o intérprete para que este baia traducindo cada instrución do arquivo fonte a linguaxe máquina (análise e síntese) e executándoa. Por Exemplo: Python.

Se a linguaxe é compilada será necesario o arquivo executable, e nalgúns casos tamén se necesitan bibliotecas dinámicas. Por exemplo: C.

Se a linguaxe precisa de máquina virtual, será necesario ter o código intermedio e a máquina virtual para que esta vaia traducindo o código intermedio a linguaxe máquina e executándoo. Por exemplo: Java ou os programas para a plataforma Android. Estes últimos utilizan Java (adaptado) e a máquina virtual Dalvik (DVM) ou a máquina ART (Android Runtime) que teñen unha estrutura distinta a JVM, para interpretar o código intermedio.

1.15 Algoritmos e Programas

Un algoritmo é unha secuencia ordenada de operacións tal que a súa realización resolve un determinado problema.

Os ordenadores basicamente non son máis que un conxunto de circuítos que son capaces levar a cabo operacións moi básicas (suma, resta, multiplicación, división, comparación, salto, copia de información) de modo moi rápido. A programación non é máis que o procedemento polo que indicamos as operacións a realizar para resolver un problema.

Estas operacións deben indicarse de modo moi preciso, xa que un ordenador levará a cabo as instrucións indicadas de xeito literal, sen ningunha interpretación pola súa parte, isto contrasta co xeito en que podemos indicar os algoritmos a outros seres humanos, na maior parte dos casos cunha descrición non demasiado precisa é suficiente.

Para poder levar a cabo as instrucións dos algoritmos nos ordenadores de uso común, estas deben estar codificadas en formato binario, xa que é moi simple construír sistemas electrónicos que realicen as operacións básicas neste formato. O formato binario ou base 2 emprega 2 símbolos para codificar información. Como contraste, para representar números habitualmente utilizamos o formato base 10 que emprega 10 símbolos.

Para poder representar información non numérica, é necesario establecer un código que faga corresponder certas combinacións binarias con símbolos concretos. No momento de descifrar a información é necesario saber si se trata de un número ou de un símbolo non numérico, para poder interpretalo do xeito correcto. O código para representar símbolos máis habitual historicamente é o código ASCII, aínda que hoxe se utiliza practicamente sempre o código UNICODE.

Os ordenadores modernos están construídos de xeito que dispoñen de:

- Unha memoria capaz de almacenar tanto os datos a procesar como as instrucións dos algoritmos.
- Unha serie de dispositivos que nos permiten introducir datos (que serán codificados a secuencias binarias).

- Unha serie de dispositivos que permiten visualizar os resultados (convertendo as secuencias binarias representado os resultados nos caracteres que esteamos habituados a ler).

Un programa implementa un ou varios algoritmos especificados mediante unha linguaxe de programación

Inicialmente, os algoritmos programábanse introducindo no ordenador directamente o código binario das instrucións que desexábamos levar a cabo (do xogo de instrucións que era capaz de executar o ordenador). Este código recibe o nome de **código máquina**, xa que é o código que executa realmente a máquina.

A programación directamente en código máquina era moi tediosa, longa e propensa a erros polo que se decidiu utilizar nemónicos para cada unha das instrucións das que dispoñía cada ordenador e logo escribir os programas utilizando eses nomes. Para que o ordenador puidera levar a cabo as instrucións era necesario traducir eses nomes ao código binario correspondente. O proceso de tradución denominouse “ensamblado” e a os nomes e as regras empregadas “Linguaxe Ensambladora”.

O ensamblador encárgase de traducir un ficheiro fonte escrito en linguaxe ensambladora a un ficheiro obxecto en código máquina. A linguaxe ensambladora é directamente traducible a código máquina. A linguaxe ensambladora depende do hardware.

Os nemónicos da linguaxe ensambladora almacénanse nunha táboa chamada táboa de símbolos que o programa ensamblador consulta para converter unha determinada instrución escrita en linguaxe ensambladora a código máquina.

Os algoritmos escritos en linguaxe ensambladora eran moi longos de escribir e difíciles de entender, xa que as instrucións empregadas eran demasiado simples, polo que se desenvolveron linguaxes que permitían empregar estruturas máis complexas como a execución condicional, a estrutura repetitiva, o uso de variables ou o agrupamento de instrucións en funcións. Estas instrucións tamén era necesario traducilas ao código binario específico do ordenador, mediante un proceso denominado “compilación” realizado por un programa chamado “Compilador”, específico da linguaxe que se utilizara.

Outro xeito de conseguir que o ordenador levara a cabo os programas realizados con este tipo de linguaxes era que en lugar de traducir todo o programa para producir un programa “binario”, se traducía e executaba no mesmo momento, “interpretando” a linguaxe ao código

máquina necesario sobre a marcha. Este proceso chámase “interpretación”, e o programa encargado de facer este traballo “Intérprete”.

Un concepto intermedio entre o compilador e o intérprete popularizado pola linguaxe de

programación Java é o da máquina virtual. Neste caso os programas escritos en linguaxe de alto nivel compílanse a un código intermedio en vez de a código máquina. Este código intermedio é executado por un intérprete coñecido como Máquina Virtual.

Unha aplicación é un conxunto de algoritmos codificados mediante unha linguaxe de programación, que, combinados entre si realizan unha actividade mais ampla. Como por exemplo un procesador de textos, un visor de vídeo.... etc. Estas aplicacións utilizan multitude de algoritmos necesarios para poder levar a cabo a súa actividade.

1.15.1 Ferramentas básicas para a creación de algoritmos

Calquera algoritmo se pode desenvolver empregando unicamente as seguintes estruturas:

- **Secuencia** - As instrucións se levan a cabo en secuencia, unha detrás de outra. Estas instrucións poden ser:
 - expresións aritmético-lóxicas que empregan operadores aritméticos e/ou operadores lóxicos. Entre os operadores aritméticos podemos destacar: suma (+), resta (-), multiplicación (*), división (/) e asignación (=). Entre os operadores lóxicos podemos destacar: igualdade (==), desigualdade (!=), maior que (>), menor que (<), maior ou igual que (>=), menor ou igual que (<=), e (&&), ou (||) e negación (!).
 - operacións de entrada ou saída de datos , como pode ser a solicitude de datos por teclado ou a visualización de información en pantalla.
- **Selección** - Permite levar a cabo un conxunto de instrucións ou outro dependendo do resultado da avaliación dunha expresión lóxica.
- **Iteración** - Permite repetir un conxunto de instrucións dependendo do resultado da avaliación dunha expresión lóxica.

O emprego de unicamente estas estruturas da lugar á programación estruturada.

Exemplo 1:

Realización de multiplicacións mediante sumas

```
Algoritmo multiplicacion_sumas
  Leer multiplicando
  Leer multiplicador
  resultado=0
  Mientras (multiplicador>0) Hacer
    resultado=resultado+multiplicando
    multiplicador=multiplicador-1
  Fin Mientras
  Escribir resultado
FinAlgoritmo
```

Exemplo 2

O seguinte algoritmo debería ler 50 números e imprimir só o promedio dos números positivos. Hai varios erros no código. Localiza os erros e suxire correccións para conseguir que o algoritmo funcione correctamente.

Estruturas iterativas

O bucle para é un bucle controlado por un contador e debería usarse cando se sabe o número de repeticións que se van realizar no bucle.

Por exemplo, ler as notas de 30 alumnos:

O bucle mentres debería usarse cando a condición do bucle se verifica antes de executar as iteracións do bucle e sabemos cal é a condición para que o bucle siga a executarse.

Por exemplo, se queremos ler só números positivos, os números válidanse antes de entrar no bucle e, mentres se introduzan números negativos, seguen a pedirse números válidos.

O bucle repetir debería usarse cando debe realizarse polo menos unha iteración do bucle e, tras cada iteración verificar se a condición de saída do bucle se cumpre ou non.

Por exemplo, ler números e calcular a súa suma ata que se introduce un valor determinado como pode ser o 0.

Exemplo 3

O seguinte algoritmo debería ler ata 20 números e parar no caso de que a suma dos números lidos sexa superior a 50. Finalmente, mostrar a suma final. Hai varios erros no código. Localiza os erros e suxire correccións para conseguir que o algoritmo funcione correctamente.

Exemplo 4

O seguinte algoritmo debería ler 500 números, xerar un cociente chamado k, mostrar cada valor de k e finalmente mostrar cantos números eran maiores que 10. Hai varios erros no código. Localiza os erros e suxire correccións para conseguir que o algoritmo funcione correctamente.

Exemplo 5

O seguinte pseudocódigo debería pedir ao usuario a temperatura en grados centígrados de todas as horas dun día e mostrar en grados Fahrenheit a temperatura máxima, a temperatura mínima e a temperatura promedio. O algoritmo ten varios erros que hai que detectar e corrixir.

Exemplo 6

Creouse un temporizador nun sitio web para darlle ao usuario 60 segundos para que introduza o seu código de acceso antes de que se lle bloquee automaticamente o acceso ao sitio. O algoritmo para cando teñen pasado 69 segundos. Cambiouse o deseño para incluír unha mensaxe de aviso cando se sobrepasen os 45 segundos. Implementa esta nova característica do algoritmo.

Funcións

As funcións utilízanse moito en programación. Empaquetan e separan do resto do programa unha parte de código que realiza algunha tarefa específica.

Son, por tanto, un conxunto de instrucións que executan unha tarefa determinada e que encapsulamos nun formato estándar para que nos sexa moi doado de manipular e volver usar.

No seguinte exemplo creamos en pseudocódigo unha función chamada mensaxe() que o que fai é mostrar unha mensaxe de saúdo cinco veces seguidas.

Para utilizar unha función debemos facer dúas cousas. Por un lado, declarar a función o que corresponde co bloque `Funcion Mensaxe()... Fin Funcion`. O nome da función é `Mensaxe`. Por outro lado, desde o algoritmo principal chamamos á función utilizando a instrución `Mensaxe()`. A esta función non se lle pasa información ningunha para que realice a súa función.

A unha función tamén podemos pasarlle información de tal maneira que, dependendo da información que se lle pase, poderá devolver resultados distintos.

No seguinte pseudocódigo preguntamos o nome de cinco persoas e o programa saúda a esas cinco persoas.

Neste caso declaramos a función como `Mensaxe(nom)`. Ao 'nom' dentro das parénteses chamámoslle parámetro dentro da función e o valor que lle pasamos neste caso é o nome da variable nome chámase argumento.

Unha función pode non ter parámetros, ter un parámetro ou ter varios parámetros. Non teñen que coincidir os nomes dos parámetros cos argumentos pero si debe haber o mesmo número.

As funcións tamén poden devolver resultados para os casos nos que se necesite o resultado da función para facer cálculos no resto do programa.

No seguinte exemplo a función Area calcula a área dun cadrado e no algoritmo principal faise a suma das áreas dos cinco cadrados.

As funcións permiten crear programas mellor estruturados e máis claros, evitando repeticións innecesarias e facilitando o seu mantemento.

1.16 Actividades



1.6 Considere o desenvolvemento dun sistema cuxo dominio de aplicación é coñecido, os seus obxectivos e requirimentos funcionais son estables e simples de comprender desde un principio, a tecnoloxía para utilizar xa esta predeterminada e é ben coñecida polo equipo de desenvolvemento. Que tipo de modelo de ciclo de vida elixiría para o desenvolvemento do devandito sistema?.

Solución



1.7 Considere agora o desenvolvemento dun sistema cuxo dominio de aplicación non é moi coñecido polo equipo de desenvolvemento. Neste caso, o cliente tampouco ten moi claro que é o que quere, de maneira que os obxectivos e requirimentos funcionais do sistema son inestables e difíciles de comprender. Ademais, o equipo de desenvolvemento vai utilizar unha tecnoloxía que lle resulta completamente nova. Discuta que modelo de ciclo de vida é máis apropiado e que etapas deberíanse utilizar para desenvolver este sistema.

Solución



1.8 A elección do ciclo de vida do software é fundamental para conseguir ter éxito nun proxecto de desenvolvemento.

No artigo:

<https://melsatar.blog/2012/03/21/choosing-the-right-software-development-life-cycle-model/>

danse varios criterios para a elección dun ciclo de vida ou outro en función das características do proxecto.

Na seguinte URL:

<https://thedigitalprojectmanager.com/es/agile-frente-a-waterfall/>

tamén se fai unha comparativa entre o ciclo de vida en cascada e as metodoloxías áxiles.

Xustifica a elección do ciclo de vida para os seguintes casos:

- 1. Proxecto de desenvolvemento de portal Web con pouca funcionalidade e moita carga de interface gráfica.**
- 2. Proxecto de desenvolvemento dunha aplicación de xestión cuns prazos de entrega bastante agresivos e que, polo tanto, esixen que se acurte o tempo de desenvolvemento.**
- 3. Proxecto de desenvolvemento de aplicación de xestión a integrar nun contorno heteroxéneo e para a cal se poden considerar diferentes alternativas en función das diferentes integracións co contorno.**

Solución



1.9 Responde ás seguintes cuestións:

- a) Definir software de sistema e software de aplicación. Citar un nome comercial de cada un deles.
- b) Explicar brevemente a fase de probas no modelo en cascada.
- c) Explicar brevemente a fase de codificación e probas na programación eXtrema.
- d) Definir brevemente o que é Métrica 3.
- e) Diferenzas e semellanzas entre as versións alfa e a beta dunha aplicación.

Solución