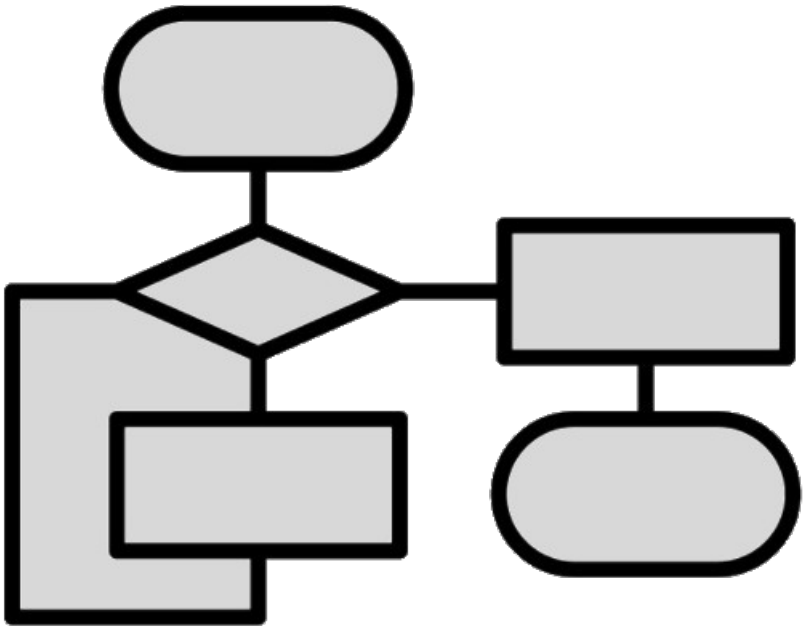


Algoritmos e Programas

Índice

ALGORITMOS E PROGRAMAS

1. Introdución: Os ordenadores dixitais

Un ordenador basicamente consta dos seguintes elementos:

- Memoria RAM ou volátil
- O bus de datos e de direccións
- CPU

A Memoria RAM dun ordenador non é máis que un conxunto de circuítos electrónicos capaces de manter valores de tensión que representan os díxitos "1" e "0" (***bits***) mentres reciba subministro eléctrico. Todos os datos que procesa un ordenador deben ser cargados previamente na memoria RAM.
Para codificar numericamente a información, se agrupan as celas de memoria en grupos de 8 bits denominados **bytes** aos que se lles asigna unha ***dirección de memoria.***

> Cada byte de memoria ten unha dirección que a identifica

O bus de direccións son liñas (condutores eléctricos) polas que se envían as direccións de memoria onde queremos ler ou escribir. Por cada liña eléctrica podemos enviar un "1" ou un "0" variando as tensións eléctricas, deste xeito, si temos 2 liñas no bus de direccións poderíamos utilizar unicamente 4 bytes de memoria: 00, 01, 10 e 11 (direccións 0,1,2 e 3)

> O tamaño do bus de dirección determina a cantidade máxima de memoria que podemos utilizar

O habitual nos ordenadores actuais son *buses de direccións de 64 bits*, cos que podemos indicar 2^{64} direccións distintas.
Os dispositivos tamén poden ter memoria RAM ademais da memoria instalada na placa do sistema (por exemplo as tarxetas gráficas). A esta memoria se accede xunto co resto formando parte do espazo de direccións do sistema que se selecciona mediante o bus de direccións.
O bus de datos son as liñas polas que se despraza a información dun elemento hardware a outro (por exemplo, da memoria RAM á CPU). Un ordenador desprazará tanta información de unha única vez como lle permita o seu bus de datos (aínda que soamente necesitemos acceder a 1 byte). O número de liñas do bus de datos coincide habitualmente co tamaño da información que é capaz de xestionar a CPU e se denomina ***tamaño de palabra*** e nos permite falar de "*ordenadores de 8 bits*", "*ordenadores de 32 bits*" ou "*ordenadores de 64 bits*".

> O tamaño do bus de datos determina a cantidade de información que se transfire en unha única operación

Un sistema operativo de 32 bits xestiona a información como si as transferencias foran sempre de 32 bits (aínda que si o ordenador é de 64 bits, se transfiran 64 bits de cada vez)
As CPU dos ordenadores están construídas de xeito que son capaces de levar a cabo operacións moi básicas de xeito moi rápido coa información que se atopa no seu interior. Para iso dispón dunhas "pequenas" memorias (habitualmente do mesmo tamaño que o bus de datos) denominadas **rexistros**. Cando se necesita traballar cunha información presente na RAM é necesario primeiro movela a eses rexistros, realizar a operación e movela de novo dende o rexistro que almacena o resultado á memoria RAM.
A información que se carga dende a RAM a CPU sempre é unha información binaria que representa unha operación que debe levar a cabo a CPU ou un dato. As operacións se cargan da área da RAM onde se atopan as instrucións do programa, mentres que os datos se cargarán da zona que indique o programa.
As CPU están construídas de xeito que responde a unha serie de códigos numéricos que indican que se debe realizar unha acción concreta moi simple como pode ser *sumar, restar, multiplicar, saltar a outra dirección de memoria, comparar valores, almacenar ou cargar valores en unha dirección de memoria....* etc.
Estes códigos se denominan ***códigos de operación*** coñecéndose o conxunto de todos eles como o **xogo de instrucións do procesador.**
Cada modelo ou fabricante de CPU incorpora o seu propio xogo de instrucións. Existe un estándar que estableceu Intel que propón un conxunto compatible de códigos chamado ***x86*** que é o que utilizan a maior parte de ordenadores domésticos baseados en Intel ou AMD. Outros tipos de CPU con xogos de instrucións distintos son ARM, o M1 de Apple ... etc.

> Un ordenador non é máis que un conxunto de circuítos dixitais capaces de almacenar e procesar información en formato binario, representando os díxitos binarios "1" e "0" mediante tensións eléctricas.

1.1 - Algoritmos e Programas

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

As características fundamentais que debe cumprir todo algoritmo son:

- Un algoritmo debe ser **preciso** e indicar a orde de realización de cada paso.
- Un algoritmo debe estar **definido**: Si seguimos o algoritmo para a mesma entrada varias veces os resultados obtidos sempre deben ser os mesmos.
- Un algoritmo debe ser **finito**: O algoritmo debe contar cun número finito de pasos

Na definición dun algoritmo debemos distinguir tres partes:

- **A entrada** de datos: A entrada de datos permite facilitar a información que queremos procesar mediante o algoritmo.
- **O procesamento** dos datos: Consiste en levar a cabo todos os pasos especificados polo algoritmo de xeito que se transforma a información de entrada producindo novos datos que chamaremos "resultado".
- **A saída**, ou devolución do resultado. Consiste en facilitar a información obtida como consecuencia transformar a información de entrada mediante os pasos indicados no algoritmo (o procesamento).

Na nosa vida diaria facemos uso de algoritmos continuamente para unha multitude de cousas, dende conducir a operacións matemáticas de uso común. Gran parte da nosa aprendizaxe a realizamos mediante a memorización dos algoritmos que nos permiten resolver tipos de problemas concretos (sumar, restar, multiplicar, dividir, resolver ecuacións, elaborar unha tortilla de patacas, etc).
Si nos fixamos ben, en todos estes casos aprendemos unha serie de pasos que aplicamos mecanicamente para chegar ao resultado. Actuamos como un ordenador que recibe unha información de entrada (os datos do problema a resolver, ou os ingredientes dunha tortilla), seguimos unha serie de pasos manipulando eses datos de entrada (procedemos a aplicar a receita da tortilla) e obtemos uns resultados (ao final, teremos unha tortilla).

> Debemos ter en conta que os pasos indicados nun algoritmo se levan a cabo de xeito literal, sen levar a cabo ningún proceso de "interpretación do que queren dicir", polo que deben ser moi claros e precisos.

Un "***Programa***" ou "Aplicación" e máis complexo que un algoritmo. Mentres que un algoritmo e a secuencia de pasos para resolver un problema concreto, un Programa é o conxunto de algoritmos necesarios para conseguir unha ferramenta cunha certa utilidade, como pode ser un editor de textos, un programa de contabilidade ou un videoxogo.

> A realización dun programa implica a resolución de moitos algoritmos.

> A información que utilizan e producen os programas se almacena na memoria RAM en áreas de memoria concretas. A esas áreas de memoria se lles da un nome (identificador) que se pode utilizar para almacenar información (asignación) ou recuperar información. Ese nome se denomina ***variable***.

1.2 – As linguaxes de Programación

Os algoritmos que queremos que leve a cabo un ordenador debemos especificalos facendo uso do *xogo de instrucións da CPU*, que como vimos no punto anterior son códigos numéricos que indican unha das operacións que é capaz de realizar a CPU. Deste xeito *un algoritmo non sería máis que un conxunto de números que indican as distintas operacións a realizar*. Cando expresamos os algoritmos deste xeito dicimos que está en ***código máquina***.

> "Código Máquina" fai referencia a un algoritmo expresado nos valores numéricos correspondentes co xogo de instrucións da CPU

Obviamente para unha persoa expresar un algoritmo ou un programa que consta de moitos algoritmos empregando código máquina non é viable de xeito xeral polo laborioso que resulta, polo que se desenvolveron as ***linguaxes de programación.*** As linguaxes de programación permiten expresar os algoritmos dun xeito máis próximo a linguaxe humana que unha secuencia de números, pero as CPU unicamente son capaces de executar código máquina, polo que se fai necesaria unha "tradución" a partir do algoritmo expresado polo programador (***código fonte***) xerando o código máquina executable polo ordenador (***código obxecto***).
O código obxecto consiste nos algoritmos do programa expresados en código máquina, pero é necesario indicarlle ao sistema, entre outras cousas, o algoritmo da aplicación polo que debe comezar a execución. Este proceso se denomina "enlazamento" ou "Link" e produce o ***código executable*** a partir dos códigos obxecto.

1.2.1 – A linguaxe Ensamblador

A linguaxe de programación máis próxima ao código máquina é a ***linguaxe Ensamblador***, que consiste unicamente en unha serie de códigos alfanuméricos que indican a instrución do *xogo de instrucións* da CPU que queremos utilizar.
Por exemplo, en lugar de poñer ***0xAD232332*** (un número expresado en hexadecimal) poderíamos indicar ***ADD AX,3.***
Obviamente non é posible executar un programa escrito utiliza ndo estes códigos, si non que é necesario convertelo a código máquina. Este proceso se denomina ***ensamblado***.

> A linguaxe Ensamblador se emprega na programación directa do hardware para crear *drivers* de dispositivos ou cando precisamos optimizar a velocidade de execución ao máximo. Non é unha linguaxe que se empregue habitualmente no desenvolvemento de aplicacións.

1.2.2 – As linguaxes de alto nivel

As linguaxes de alto nivel pretenden acercar máis a linguaxe humana os algoritmos de ordenador. Estas linguaxes permiten expresar operacións complexas no código fonte en lugar de limitarse as operacións básicas da CPU. Estas operacións son logo son "traducidas" a código máquina mediante unha aplicación que pode ser un **compilador** ou un **intérprete.**

1.2.2.1 – As linguaxes interpretadas

O proceso de tradución pode facerse analizando o texto do código fonte, traducindo e executando sobre a marcha sin producir ningún ficheiro con código executable. Este tipo de programa se chama "intérprete", e da lugar as ***linguaxes interpretadas***.
As principais vantaxes da interpretación é unha maior facilidade para a modificación, unha máis rápida notificación dos erros sintácticos e en xeral maior rapidez de desenvolvemento en aplicacións sinxelas, pero sobre todo a posibilidade de executar o mesmo programa (código fonte) en calquera sistema para o que teñamos o intérprete axeitado conseguindo así unha *portabilidade máxima*.
Pola contra, o maior inconveniente é a relativa lentitude da execución en comparación coas aplicacións compiladas aínda que hoxe en día se utilicen técnicas como a compilación JIT (Just In Time) que cachea o código máquina producido durante a interpretación para acelerar a execución.
Exemplos de linguaxes interpretadas son: PHP, Python, etc. e os que comunmente se coñecen como *linguaxes de scripting* utilizadas moito na administración de sistemas como *bash*, *Perl* ou *Python*

1.2.2.2 – As linguaxes compiladas

Outra posibilidade é traducir o **código fonte** creando un novo ficheiro en código máquina (**código obxecto**) mediante un proceso denominado "*compilación*". Posteriormente, mediante o enlazado se crea un ficheiro co **código executable**, que lle engade ao código obxecto da aplicación a información sobre que instrución debe dar comezo ao algoritmo da aplicación.
O programa encargado de realizar a tradución a código máquina se denomina ***compilador***, mentres que o que crea o código executable se chama enlazador ou ***linker***. As linguaxes que empregan este proceso se chaman ***linguaxes compiladas.***
As aplicacións compiladas son moito máis rápidas que as equivalentes interpretadas e ademais non precisan de nada adicional para a súa execución. Pola contra, a aplicación unicamente funcionará no sistema para o que foi compilada (ou un sistema compatible). Exemplos de linguaxes compiladas son C, C++, Pascal ....

1.2.2.3 – As linguaxes de máquina virtual (p-code)

Este sistema pretende acadar a vantaxe das linguaxes compiladas sen renunciar a portabilidade das linguaxes interpretadas.
Para conseguir isto, se deseña un "procesador virtual" que é capaz de executar un código máquina propio, compilando o código fonte a ese código máquina "virtual". A aplicación pode executarse en calquera sistema para o que dispoñamos dunha programa que facilite ese procesador virtual (máquina virtual ou *VM*).
Cada vez son mais comúns compiladores de *p-code* para linguaxes que antes eran interpretados, como Perl, Python ou PHP, pero a linguaxe de este tipo máis salientable e ***Java***.

2. Paradigmas de Programación

> Denominamos “paradigma de programación” ao xeito en que se indican os algoritmos que debe seguir o ordenador, e dicir, o xeito en que deseñamos os algoritmos e as aplicacións.

2.1 - Programación Imperativa e Declarativa

Na **programación declarativa** se describe a lóxica de computación necesaria para resolver un problema, sen seguir unha secuencia de ordes fixas. Na programación declarativa non é necesario definir algoritmos xa que o que se detalla é a solución ao problema, e non como se chega a esa solución. A solución se alcanza mediante mecanismos internos de control pero non se especifica exactamente como chegar a ela.

> No paradigma declarativo descríbese o problema sen fixarnos no algoritmo necesario para conseguir a solución.

Un exemplo de un uso declarativo dunha linguaxe son as instrucións de consulta, inserción, modificación e borrado mediante ***SQL***. Se indica o que se quere facer, sen entrar nos detalles de como se fai. “di o que queres, pero non os pasos a seguir para obter o queres”.
A diferencia da programación declarativa, na **programación imperativa** se describen un conxunto de sentencias que van alterando o estado (os datos) dun programa. A programación imperativa é a programación natural para as CPU, xa que é o modo no que traballan de xeito básico.

> Neste paradigma as solucións aos problemas se expresan indicando unha secuencia de accións a realizar para chegar á solución, que se coñece como algoritmo.

Dentro de esta categoría se engloban a *programación estruturada*, e a *programación orientada a obxectos*, sendo os paradigmas de uso máis xeral e estendido na actualidade pola súa gran versatilidade, e son utilizados polas linguaxes más habituais como JavaScript, PHP, C/C++, Python, C#, Java ... etc.,

> Cada vez é máis común que as linguaxes incorporen características de varios paradigmas que os converten en linguaxes multiparadigma, como poden ser as expresións lambda[1] .

2.1.1 - Programación Declarativa

Dentro de esta categoría se engloban a programación funcional con linguaxes coma ***Haskell*** ou a programación lóxica, con ***Prolog*** coma linguaxe máis representativo

2.1.1.1 – Programación Lóxica

A programación lóxica utiliza a lóxica (cálculo de predicados) para o análise dos problemas e o control das regras de dedución necesarias para alcanzar a solución.
Na Programación Lóxica, se traballa de xeito descritivo, establecendo relacións entre entidades indicando o que facer e non como facelo.
Se intentan resolver problemas da seguinte natureza de xeito automático: “Dado un problema S, saber si a afirmación A é solución ou non do problema ou en que casos o é”.

> A programación lóxica constrúe a base de coñecementos mediante regras e feitos que permiten chegar á solución. Pola súa natureza, este paradigma encaixa moi ben coa solución de problemas mediante intelixencia artificial a minaría de datos ou a robótica.

2.1.1.2 – Programación Funcional

Nas linguaxes funcionais puras como ***Haskell*** todas as funcións son puras, e dicir, non teñen efectos fora da propia función e os cambios de estado (datos almacenados polo programa) se representan mediante funcións que se encargan de transformar o estado da aplicación. Aínda que non son linguaxes imperativas, en xeral proporcionan mecanismos para describir as funcións coma unha serie de pasos. Na programación funcional non existen variables (ou ben en realidade son inmutables), xa que as funcións so necesitan procesar os datos e non se precisa almacenar o estado de nada. Cada función opera sobre os seus propios datos sen relación coas demais funcións.

1. As expresións lambda basicamente son funcións sen nome que se definen no mesmo lugar no que son chamadas. Tamén se coñecen como funcións anónimas.

Exemplo de programación lóxica:
Este exemplo é significativo do que é a programación lóxica. Indicamos ao sistema os nosos coñecementos en forma de cláusulas, e esas cláusulas nos permiten inferir resultados.
As cláusulas serían as seguintes:

chove(X):-
nubrado(X),
frío(X).

frío(X):-
Temperatura(X,baixa).

nubrado(X):-
sen_sol(X).

As afirmacións ou feitos serían:
temperatura(xoves,baixa).
sen_sol(xoves).

- Na primeira cláusula se establece que consideramos que o día X chove SI o día X está nubrado e o día X fai frío.
- A segunda cláusula se establece que o día X fai frío SI ese día a temperatura "baixa".
- A terceira cláusula establece que o día X está nubrado SI o día é un dia sin_sol.

- Na primeira afirmación se indica que o xoves a temperatura baixa
- Na segunda afirmación se indica que o xoves é un día sen sol

O sistema será capaz de responder á pregunta obxectivo "chove(xoves)" a partir de estas regras.

2.1.2 - Programación Imperativa

Podemos distinguir dúas categorías dentro da programación imperativa: A programación estruturada tradicional e a programación orientada a obxectos.

2.1.2.1 – Programación Estruturada

A programación estruturada recibe o seu nome do "***Teorema da estrutura***", que establece:

> Toda función resoluble cun programa de ordenador pode ser escrita nunha linguaxe de programación que combine unicamente tres estruturas lóxicas básicas: ***Secuencia***, ***Selección***, ***Iteración***.

- **Estrutura de Secuencia**: Consiste na execución dunha sentencia detrás de outra. *As sentencias poden ser asignacións, expresións aritméticas, lóxicas ou chamadas a funcións.*
- **Estrutura de Selección**: Execución dun grupo ou outro de instrucións dependendo dun valor lóxico (verdadeiro ou falso), que normalmente se obtén avaliando unha expresión lóxica. Utilizando pseudocódigo se pode representar así:

```
Si (condición) entón
        secuencia de accións
se non
        secuencia alternativa de accións
Fin-Si
```

- **Iteración:** Execución dun grupo de instrucións mentres ou ata que unha variable tome o valor certo. Este tipo de estrutura tamén se coñece como ***ciclo*** ou ***bucle***. **Utilizando pseudocódigo se pode representar:**

```
Mentres(condición)
    secuencia de accións
Fin Mentres
```

Este teorema elimina o uso de certas instrucións de uso común en linguaxes de moi baixo nivel como son as instrucións de salto (***goto***) aínda que a veces se poidan empregar por motivos de optimización de velocidade. Cando na nosa aplicación detectamos unha secuencia de accións para realizar unha acción concreta (*algoritmo*) que se repite varias veces ou que prevemos pode ser útil no futuro podemos darlle un nome e reutilizala cando a precisemos. E dicir, podemos construír unha ***función***

As linguaxes de programación imperativas proporcionan habitualmente multitude de funcións con accións comúns que se coñecen como ***libraría de funcións do sistema.*** Exemplos comúns de funcións son ***sqrt*** para calcular raíces cadradas ou ***print*** para visualizar información na pantalla.

2.1.2.2 - Programación Orientada a Obxectos

Mentres que na programación estruturada clásica os programas se enfocan como unha serie de pasos a executar para resolver un problema, na programación orientada a obxectos a aproximación é distinta. Os problemas se interpretan como unha serie de obxectos cunhas características concretas e capaces de realizar un conxunto de accións que crean novos obxectos e utilizan as súas funcionalidades.

> O conxunto de características (***atributos***) e de funcionalidade (***métodos***) das que dispón un obxecto se coñece como ***Clase***. Na programación orientada a obxectos un programa está composto de ***obxectos*** que pertencen a ***clases.***

Por exemplo, cando dicimos "Pintura ao Óleo" todos temos claro que características definen o que é unha pintura ao óleo. E unha **clase**. Sin embargo "o Guernica" é **un obxecto concreto** da clase "Pintura ao Óleo" (e dicir, ten todas as características que entendemos debe ter unha pintura ao óleo)
Do mesmo xeito que as linguaxes estruturadas clásicas teñen librarías de funcións, as linguaxes orientadas a obxectos dispoñen de ***librarías de clases***, que nos proporcionan clases con funcionalidades de uso común nas aplicacións.

3. Análise e Deseño: O ciclo de vida

A principal razón para que as persoas aprendan a programar e o obxectivo xeral das linguaxes de programación, é poder facer uso da computadora como unha ferramenta para a resolución de problemas. Axudado por unha computadora, a resolución dun problema pódese dividir en varias fases importantes:

- Análise do problema e desenvolvemento de algoritmos.
- Codificación do algoritmo na computadora.
- Probas de funcionalidade
- Implantación e Explotación

> O ***ciclo de vida*** dunha aplicación informática comeza cando se empeza a analizar o problema e remata canda a aplicación deixa de ser útil.

Existen varios modelos de ciclo de vida segundo o xeito de tratar cada unha de estas fases e dos momentos en que se levan a cabo.

3.1 – Análise do Problema e Deseño dos Algoritmos

O propósito da análise dun problema é comprender a natureza do problema. O problema debe estar ben definido si se desexa chegar a unha solución satisfactoria. A análise do problema esixe unha lectura previa co fin de obter unha idea xeral do que se solicita. A segunda lectura deberá servir para responder ás preguntas:

- ¿Que información debe proporcionar a resolución dun problema dado? : **Requisitos de Saída**
- ¿Que datos se necesitan para resolve-lo? : **Requisitos de Entrada**

> O Análise do problema é a fase máis delicada e importante da creación dun programa, xa que calquera erro ou malentendido significaría refacer absolutamente todo o traballo.

Unha vez comprendido o problema e necesario detectar que algoritmos será necesario deseñar e como facelos. Na programación imperativa se utilizan normalmente dous métodos: O Análise descendente ou "***top-down***", e o ***análise orientado a obxectos***.

3.1 – Análise descendente ou "Top-Down"

A complexidade dos algoritmos é moi variable e as veces a súa solución difícil de afrontar. E xeral, cando ocorre isto, e porque o problema a resolver en realidade está composto de outros problemas máis simples.
Tendo en conta que podemos agrupar os algoritmos que resolven unha funcionalidade concreta nunha función, o deseño descendente consiste en resolver o problema sin reparar nos detalles. Eses detalles se resolverán mediante "funcións" que conteñen o algoritmo para resolver ese tipo de sub-problema.

> Os algoritmos deben ser **completos e precisos**, é dicir, deben expresar todas as accións que se deben realizar para resolver o problema. Si algo non se expresa, non se leva a cabo, non existen as "suposicións" nin os sobreentendidos.

Por exemplo, para facer unha tortilla o algoritmo sería bastante complexo, xa que si dicimos "*batemos os ovos*" o resultado sería unha mistura de cáscara e ovos, porque non especificamos expresamente que deberíamos cascar o ovo, verter o contido nun cunco e tirar a cáscara....Si temos que reparar en todos estes detalles o algoritmo sería enorme, sería fácil cometer erros (por acción ou omisión) e a corrección sería difícil.

Si en cambio entendemos que “batemos os ovos” representa unha acción complexa (un algoritmo), que recibe ovos como entrada e nos devolve ovos batidos como saída, podemos empregar “batemos os ovos” no noso algoritmo. Unha vez rematado o algoritmo para crear a tortilla describiremos o algoritmo exacto para bater os ovos sen preocuparnos da tortilla.
Como beneficio adicional, poderíamos reutilizar ese algoritmo para calquera outra receita que precise bater ovos.

O ***deseño descendente*** consiste en abordar o problema complexo sin entrar nos detalles, para logo ir resolvendo cada un de eses detalles pendentes. Para resolver cada un destes detalles se aplica o mesmo sistema de xeito recursivo.

Como xa comentamos, cada sub-problema dará lugar a unha función (ou non, si cando o examinamos vemos que e algo moi simple e non precisa moitos pasos). As vantaxes máis importantes que nos proporciona esta técnica son:

- **Maior claridade dos algoritmos**: O código da solución é mais comprensible xa que non se perde nos detalles.
- **A modificación e corrección é máis simple**: Xa que precisamos modificar unicamente as funcións afectadas polos erros ou que necesitan ser modificadas.
- **A verificación é mais simple**: Podemos verificar o correcto funcionamento de cada función por separado, o que resulta máis simple e efectivo.
- **O Código é reutilizable**: Moitas funcións poden ser utilizadas en múltiples aplicacións
- **E máis fácil repartir o traballo**: Cada programador pode encargarse da solución dun sub-problema distinto.

3.2 - Análise orientado a Obxectos

A orientación a obxectos xira en torno a dous conceptos básicos:

- ***Obxecto:*** Un obxecto é un elemento dun programa que dispón dunha serie de características denominados “***atributos***” que indican información sobre o obxecto e de unha serie de funcionalidades ou “***métodos***” que son as “cousas que é capaz de facer” o obxecto.
- ***Clase:*** Unha clase é o “tipo” ao que pertence un obxecto. Por exemplo, o meu vehículo con matrícula 7654JTM pertence a clase “Vehículo” ou, expresado de outro modo, ***é un vehículo***. E podemos dicir eso porque o obxecto que ten matrícula 754JTM ten as características e métodos que teñen os obxectos que consideramos que son vehículos.

A definición dunha clase consiste en indicar que atributos e que métodos terán os obxectos desa clase

O valor que teñen os atributos de un obxecto nun instante determinado se denomina ***estado do obxecto***.

O análise orientado a obxectos supón un paso máis na dirección indicada polo deseño descendente. En lugar de enfocar o problema como unha serie de pasos a seguir para obter a solución, se pensa no programa como una interacción entre unha serie de obxectos que consiste en:

- ***Creación de obxectos***: Consiste na creación de novos obxectos necesarios para a solución do problema. Este proceso se coñece co nome de *creación de instancias* e se realiza chamando a un método especial da clase denominado ***construtor***
- ***Uso de funcionalidades ofrecida polos obxectos:*** Consiste na invocación de métodos de obxectos que proporcionan un resultado e/ou alterar o estado do obxecto de xeito que axudan a acadar a solución do problema. Este proceso se denomina *invocación*.

Para identificar os obxectos que compoñen a solución a un problema se utiliza habitualmente a seguinte técnica:

1. Se identifican **os substantivos** do enunciado do problema. **Son potenciais obxectos ou atributos**.
2. Se identifican **os verbos** do enunciado do problema. **Son potenciais métodos das clases.**
3. Se definen as Clases as que pertencen os distintos obxectos identificados utilizando os atributos e métodos detectados no enunciado.

A orientación a obxectos proporciona funcionalidades de gran utilidade a hora de desenvolver aplicacións:

- ***Protección de Acceso / Encapsulación***: A *encapsulación* consiste en facer que os obxectos funcionen como “caixas negras” inmunes a calquera acción externa. O comportamento dun obxecto debe ser o previsto sin importar o que se faga fora do mesmo. A *protección de acceso* regula como e dende onde se pode acceder aos atributos e funcionalidade da clase co obxecto de facilitar o logro da encapsulación.
- ***Herdanza:*** *A herdanza* consiste en deseñar novas clases partindo de clases existentes herdando de xeito automático todos os seus atributos e funcionalidades. A nova clase se coñece como ***clase filla*** ou herdada

e a clase de partida ***clase base*** ou ***clase pai***. Por exemplo, a clase "Motocicleta" podería definirse empregando da clase "Vehículo" definida previamente, xa que os obxectos "Motocicleta" deben dispoñer obrigatoriamente de todos os atributos e métodos que teñen os obxectos "Vehículo" dado que os obxectos "Motocicleta" SON "Vehículos".

- ***Sobrecarga:*** *A sobrecarga* consiste en definir novas versións de métodos (sobrecarga de métodos) ou de operadores (sobrecarga de operadores) que se comportan de xeito distinto en función do número e tipo de datos recibidos. A sobrecarga facilita a lexibilidade e mantemento das aplicacións.
- ***Superposición:*** *A superposición* consiste en modificar o comportamento dun método da clase base na clase filla. A acción que realizarían sería a mesma, pero o xeito de levala a cabo (o algoritmo) pode cambiar.
- ***Polimorfismo:*** *O polimorfismo* garante que cando invocamos un método sobre un obxecto, se invoca o método sobreposto correcto. Por exemplo, si teño un obxecto Vehículo que en realidade é un Triciclo, si chamo ao seu método "acelera" o polimorfismo garante que se leve a cabo o método "acelera" definido na clase Triciclo en lugar do definido na clase Vehículo.

3.2.1 - Ferramentas de deseño: UML

Cando analizamos un problema mediante orientación a obxectos é habitual facer uso de **UML (Unified Model Language)**. UML define unha serie de diagramas e documentos que se deben desenvolver na fase de análise e que se utilizarán para realizar a codificación. Os máis importantes son:

- **diagrama de casos de uso** que describe como se debe comportar a aplicación ante as accións do usuario do programa (actor).
- ***diagrama de clases*** que especifica todas as clases dos obxectos que interveñen no problema e a súa relación entre elas.
- ***Diagrama de secuencia*** que describe como interactúan os obxectos entre si para levar a cabo os distintos casos de uso.

Existen ferramentas automatizadas que a partir de estes diagramas son capaces de xerar de xeito automático o código da aplicación, sendo preciso unicamente a codificación dos algoritmos necesarios para resolver os métodos definidos nas clases que non están contemplados no diagrama de secuencia.

3.2.2 – Os Test de Unidade

A verificación de funcionalidades e a corrección de erros e unha das fases máis importantes do desenvolvemento de software. No caso da orientación a obxectos, gracias a encapsulación, esta verificación é bastante simple en comparación coa programación estruturada clásica, xa que cada obxecto debe ofrecer unha funcionalidade ben definida e completamente independente do funcionamento do resto do programa de xeito que nada do que se poda facer fora dun obxecto debe poder afectar ao seu funcionamento.
Si cumprimos estes requisitos, para validar unha aplicación unicamente é necesario:

- ***Test de Unidade***: Verificar o correcto funcionamento de todas as funcionalidades ofrecidas por cada clase.
- ***Test de Integración***: Verificar o correcto funcionamento unha vez unidos todos os obxectos ou cada vez que se engade / modifica un de eles.

Para o desenvolvemento dos test de unidade existen ferramentas automatizadas que nos permiten verificar as clases no instante que desexemos elaborando unha ***batería de probas***. A calidade do test de unidade dependerá da calidade da batería de probas empregada.

> A mellor aproximación é elaborar a batería de probas de xeito previo á codificación dos algoritmos.

As probas se deseñan a partir dos *requisitos de entrada*, elixindo os datos de entrada e calculando a man os resultados que se deben obter. Existen diversas técnicas para seleccionar os mellores valores para empregar como datos de proba de xeito que podamos ter probas rigorosas cun número de casos de entrada non demasiado alto.
O ***TDD*** (Test Driven Development) ou desenvolvemento orientado a probas é unha técnica de desenvolvemento de clases moi efectiva. Esta técnica consiste en:

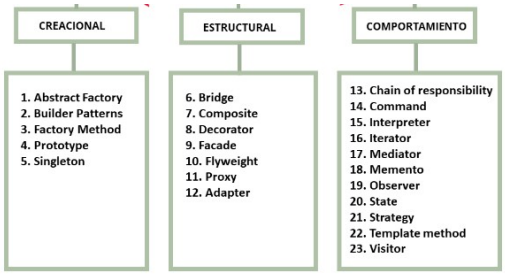
1. Se crea a clase pero a funcionalidade se deixa a mínimos sin importar que se leve a cabo correctamente ou non.
2. Para cada funcionalidade da clase (método) mentres non se cumpra toda a funcionalidade:
    1. Se engaden probas a batería de probas que verifica a funcionalidade do método
    2. Se executan os test
    3. Se corrixen os erros devoltos polo test e se volve ao punto 2

Ao remate, teremos unha clase que cumpre correctamente cos requisitos exixidos, e dicir, unha clase correcta.

Para a realización das baterías de test existen linguaxes especializados como ***JUnit*** (unha libraría de Java) ou ***PHPUnit*** (o equivalente en PHP).

3.3 – Os patróns de deseño

Un ***patrón de deseño*** é unha solución xeral reutilizable e aplicable a diferentes problemas de deseño de software.



Nesta lista están os nomes dos 23 patróns de deseño máis comúns. Podes consultar en Internet o obxecto de cada patrón e como se poden crear dun xeito simple.

4. Elementos de un programa

4.1 - Identificadores: Os espazos de nomes

Un identificador é un nome que da nome (identifica) a algo. Usamos identificadores para practicamente todos os elementos dos programas: O nome do programa, os obxectos, as clases, os métodos, os atributos .... etc.
Os caracteres que se poden utilizar para crear os identificadores dependen da linguaxe empregada, pero deberían seguirse as seguintes convencións:

- ***Deben ser significativos***: O nome empregado debe ofrecer unha idea de que é o que se está identificando.
- ***Non deben ser excesivamente longos***: Identificadores demasiado longos dificultan a lexibilidade do código.

A medida que reutilizamos código previamente deseñado, e sobre todo si empregamos código desenvolto por terceiros (librarías) aumenta a posibilidade que un identificador que estamos a utilizar coincida cun identificador utilizado no código "externo" provocando un conflito (por exemplo, dúas clases que se chamen igual). Para solucionar isto as linguaxes de programación utilizan habitualmente **espazos de nomes.**

Un **espazo de nomes** é un prefixo que se antepón implicitamente aos identificadores definidos nese ámbito co obxectivo de facelos únicos.

E común utilizar URI como espazos de nomes xa que os URI garanten que son únicos. En algúns linguaxes (como Java) os espazos de nomes implican tamén unha estrutura de almacenamento concreta. En Java o espazo de nomes ao que pertence un identificador indica tamén a carpeta na que se debe almacenar o código.

Existen algúns identificadores que non poden ser utilizados, xa que forman parte da propia linguaxe de programación. Se coñecen como ***palabras reservadas***.

4.2 – Literais, Constantes y Variables

Un **literal** é a representación escrita dun valor. Por exemplo *true, 34, 15.234, 'x', ou "Ola"*

Unha **variable** é un nome que identifica un espazo na memoria RAM onde podemos almacenar ou ler información

Unha **constante** é un nome que representa un valor

As vantaxes entre utilizar constantes sobre utilizar directamente o valor representado son:

- **Facilidade na lectura do programa:** O mesmo valor pode significar distintas cousas nun programa, o valor 3.14 pode ser o valor PI en unha liña do programa ou a nota dun alumno en outra liña.
- **Facilidade de modificación:** Si utilizamos o valor directamente e desexamos modificalo (por exemplo mellorando a precisión do valor de PI de 3.14 a 3.14159) debemos ir a todas as liñas do código onde aparece o valor 3.14, asegurarnos que ese valor 3.14 se refire realmente a PI e modificalo. Si en cambio en lugar de poñer 3.14 utilizando a constante definida PI basta con cambiar a definición de PI.

A diferencia máis importante entre unha constante e unha variable e que unha variable e o nome dun sitio que pode almacenar un valor, mentres que unha constante é un nome que lle damos a un valor.

As variables definidas teñen un ***ámbito de existencia*** ou ***ámbito de vida*** que fai referencia o área de código onde a variable está dispoñible. Normalmente as variables definidas nun bloque de código teñen existencia unicamente nese bloque. Un bloque de código é un grupo de instrucións "agrupadas". O xeito en que se definen os bloques varía coas linguaxes, pero en xeral se definen pechando o código entre **{** e **}.**

> É **moi importante** ter en conta que **o ámbito de existencia de unha variable é sempre o bloque de instrucións onde foi definida** (na maioría das linguaxes de programación)

4.3 - Operadores: Expresións Aritméticas e Lóxicas

> Unha ***expresión aritmética*** é unha expresión que se avalía a un valor numérico.
>
> Unha ***expresión lóxica*** é unha expresión que se avalía a un valor lóxico, e dicir, a ***verdadeiro*** ou a ***falso***.

Os operadores son símbolos que nos permiten formar expresións. Podemos distinguir dous tipos de operadores:

- **Operadores Aritméticos:** Son os operadores que nos permiten formar expresións aritméticas
- **Operadores Lóxicos:** Son os operadores que nos permiten formar expresións lóxicas.

> Normalmente se utiliza o operador = como operador de ***asignación***, non de comparación. Para a comparación é habitual utilizar o operador ==

Os operadores existentes poden variar coa linguaxe pero en xeral podemos distinguir os seguintes:

Expresión aritmética: z=2*y+5*x
Asignación (=) á variable z do valor producido polo cálculo de 2 polo valor almacenado en y mais 5 polo valor almacenado en x

Expresión lóxica: (a>5) && (b<3)
Si o valor almacenado en a é maior a 5 e o valor almacenado en b e menor que 3, o resultado é true, se non é así o resultado é false.

4.4 - Sentencias de Control: Selección e Iteración

> Mentres que as "***expresións***" nos permiten avaliar operacións aritméticas e lóxicas e as variables nos permiten almacenar información, as *sentencias de control* nos permiten alterar a orde natural de avaliación das sentencias (secuencia).

As sentencias de control son as típicas que define a programación estruturada: ***Selección*** e ***Iteración***. As sentencias de selección nos permiten levar a cabo un grupo de instrucións dependendo do resultado da avaliación dunha expresión lóxica, mentres que as de iteración nos permiten repetir un grupo de instrucións mentres unha expresión lóxica se cumpra.

4.4.1 – Selección

```
Si (condición)
  <Sentencias>
Se Non
  <Sentencias>
Fin Si
```

```
Si (condición)
  <Sentencias>
Fin Si
```

```
Segundo o valor de  (variable)
   Valor1: <sentencias>
   Valor2: <sentencias>
   ....
   En outro caso: <sentencias>
Fin Segundo
```

```
Segundo o valor de  (variable)
   Valor1: <sentencias>
   Valor2: <sentencias>
   ....
Fin Segundo
```

> As sentencias de selección permiten levar a cabo un grupo de instrucións dependendo do resultado da avaliación dunha expresión condicional.

4.4.2 – Iteración

```
Mentres (condición)
  <Sentencias>
Fin Mentres
```

```
Facer
  <Sentencias>
Mentres (condición)
```

```
Para(<inicialización>; <condición>; <post-condición>)
  <Sentencias>
Fin Para
```

> As sentencias de iteración permiten repetir a execución dun conxunto de instrucións **mentres** o resultado da avaliación dunha expresión condicional sexa certa.

- **Mentres**: Permite repetir a execución do grupo de sentencias mentres o resultado da avaliación da expresión condicional sexa certa. Si a condición non se cumpre na primeira avaliación, as sentencias non se levan a cabo xa que a condición **se avalía antes** de cada execución do grupo de sentencias.

- **Facer Mentres:** Permite repetir a execución do grupo de sentencias mentres o resultado da avaliación da expresión condicional sexa certa. As sentencias se levan a cabo como mínimo a primeira vez, xa que a condición **se avalía despois** de cada execución do grupo de sentencias.
- **Para:** E unha versión abreviada de Mentres pensada para "contar" dende un valor inicial ata un valor final nos pasos que se estime (de un en un, de dous en dous) e na dirección desexada (de menor a maior ou viceversa). Non obstante, se pode empregar como se estime. No paréntese se indica separado por ; o seguinte:
  - **Inicialización**: Sentencia ou grupo de sentencias separadas por comas que **se levan a cabo unha única vez** ao inicio da iteración.
  - **Condición:** Expresión condicional. Si esta expresión se avalía a falso (false), se rompe a iteración.
  - **Post-condición:** Sentencia ou grupo de sentencias separadas por comas que se levan a cabo ao final de cada execución do grupo de sentencias que desexamos repetir e antes de avaliar a expresión condicional

O seguinte algoritmo visualiza 10 veces "Ola" indicando o número de repetición:

```
idx=1
Mentres(idx<=10)
   Visualiza idx "–  Ola"
   idx=idx+1
Fin Mentres
```

E perfectamente posible realizar calquera programa utilizando unicamente unha destas tres estruturas, sendo ***Mentres*** a estrutura de uso máis común.

Algunhas linguaxes de programación utilizan (como PASCAL) a lóxica inversa en algunhas estruturas, repetindo un grupo de instrucións **ata** que unha expresión condicional sexa certa.

4.5 - Tipos de datos: Tipoloxía forte e débil, enum e Arrays

Un ***tipo de datos*** indica o tipo de información almacenada no área de memoria identificado pola variable. Con "tipo de información" nos referimos a dúas cousas concretas:

- ***A cantidade de espazo de memoria utilizado*** para almacenar a información
- ***O xeito de interpretar a información almacenada*** en ese espazo (a memoria so almacena valores binarios, bits).

Deste xeito, cando definimos por exemplo algo así: ***int x;***

Estamos creando un espazo en memoria co número de bytes que corresponde co tipo "int". Cando lemos a información almacenada nese espazo de memoria a linguaxe interpreta os bits almacenados como un número enteiro con signo expresado en binario.

Si o que definimos é : ***float f;***

Estamos creando un espazo en memoria co número de bytes que corresponde co tipo "float". Cando lemos a información almacenada nese espazo de memoria a linguaxe interpreta os bits almacenados como un número en coma flotante probablemente empregando un estándar como ***IEEE 754***.

Nas linguaxes de ***tipoloxía débil*** non é necesario indicar o tipo de datos das variables. O tipo de datos se establece cando se almacena a información.

Nas linguaxes de **tipoloxía forte** se debe indicar o tipo de datos no instante en que se define a variable, e non será posible almacenar datos de tipos "non compatibles".

Exemplos de linguaxes de tipoloxía débil son JavaScript, PHP ou Python mentres que do lado da tipoloxía forte podemos distinguir a C/C++ ou Java. Cando usamos *tipoloxía forte* o intento de almacenar información dun tipo distinto ó declarado supón un erro de sintaxe no programa, excepto que o formato de datos sexa compatible. Nese caso, si a "conversión" non supón perda de información (de un tipo con menor rango de valores a un tipo con un rango maior) se pode facer habitualmente de xeito transparente. Pola contra, ***si a conversión pode supoñer perda de información*** é necesario "forzar" a conversión mediante unha operación denominada ***"casting"***.

4.5.1 - O espazo en memoria e a capacidade

boolean : Unicamente necesita 1 bit de memoria, que se interpretará como o falso (0) ou verdadeiro (1). O que ocupa realmente depende do sistema.
byte: Utiliza 8 bits de memoria que se interpretan como un número enteiro con signo. Polo tanto pode representar os valores comprendidos entre -128 e 127
char: O seu tamaño depende do sistema de codificación empregado. A información que almacena se interpreta como un código asignado a un carácter alfabético. Si a codificación utilizada para os caracteres é UTF-8 utiliza 16 bits (2 bytes).
short: A información que almacena se interpreta como un número enteiro con signo. Polo tanto pode representar os valores comprendidos entre -32768 e 32767
int: O seu tamaño pode variar entre 2 e 4 bytes segundo a linguaxe e sistema operativo empregado. Na maioría das linguaxes en sistemas modernos o tamaño e de 32 bits (4 bytes). O seu contido se interpreta como un número enteiro con signo, e polo tanto pode representar valores comprendidos entre 2^{32} e 2^{32-1}
long: O seu tamaño pode variar entre 4 e 8 bytes segundo a linguaxe e sistema operativo empregado. Na maioría das linguaxes en sistemas modernos o tamaño de de 64bis (8 bytes). O seu contido se interpreta como un número enteiro con signo, e polo tanto pode representar valores comprendidos entre 2^{64} e 2^{64-1}
float: Emprega 4 bytes. O seu contido se interpreta como un número real segundo un formato de de codificación de coma flotante. O habitual e o emprego de o estándar IEEE 754.
double: Emprega 8 bytes. O seu contido se interpreta como un número real segundo un formato de codificación de coma flotante. O habitual é o emprego de o estándar IEEE 754
referencia (dirección de memoria): O xeito de que unha variable almacena unha referencia varía moito dunha linguaxe a outra. O seu tamaño está ligado ao espazo de direccións do sistema e ao tipo de sistemas operativos, pero en sistemas modernos o habitual é o emprego de 8 bytes (sistemas de 64 bits). O seu contido se interpreta como unha posición de memoria (un número que indica un byte concreto da memoria RAM).

Existen linguaxes que distinguen entre tipos con signo e sen signo (por exemplo, ***unsigned int***). Si prescindimos do signo nas variables numéricas podemos representar o dobre de números positivos.

> E importante recordar que a transferencia mínima de información realizada polos circuítos dun ordenador en un so paso está vinculada ao tamaño do seu bus de datos e dos rexistros de memoria da CPU. Os ordenadores modernos son habitualmente sistemas de 64 bits, polo que a transferencia mínima de información é de 64 bits aínda que so necesitemos acceder a un dato *boolean* (1 bit).

Como xa comentamos, é posible asignar valores de un tipo a outro mentres o formato de información sexa compatible, sendo necesario "forzar" a conversión si pode supoñer unha perda de información. Por exemplo, dadas dúas variables ***int x=21*** e ***long y=123*** a asignación **y=x** (asignamos o valor almacenado en **x** (*un número enteiro de 32 bits*) á variable **y** (*un número enteiro de 64 bits*) se pode realizar de xeito transparente, mentres que o inverso, ***x=(int)y*** precisa que o forcemos mediante un "*casting*".

> Para diferenciar un valor *char* dun identificador, normalmente se utilizan as comiñas simples. Deste xeito ***x*** é un identificador mentres que ***'x'*** é o carácter x.

4.5.2 – String, Arrays e estruturas de datos avanzadas

Os tipos de datos anteriores son **tipos simples**. E dicir, almacenan valores binarios utilizando unha codificación determinada (codificación binaria dun número, UTF-8, IEEE 754 ...) pero unicamente almacenan un valor do tipo indicado. En todas as linguaxes de programación tipos de datos estruturados que permiten almacenar información de natureza máis complexa, sendo os máis simples e común a *Táboas* ou ***Array*** e os ***String***.

> ***Unha Táboa ou Array*** é un tipo de dato que permite almacenar unha cantidade concreta de datos do mesmo tipo e acceder a cada un de eles mediante un ou varios índices. O número de índices necesarios para ler ou almacenar información nunha táboa se coñece como dimensión da táboa. Deste xeito podemos distinguir entre táboas de unha dimensión ou vectores, táboas de dúas dimensións ou matrices, táboas de tres dimensións, de catro, de cinco.... etc.
>
> Un dato de tipo ***String*** representa un conxunto de caracteres que forman un texto que se coñece como "cadea de caracteres".

Nas **linguaxes orientadas a obxectos** os *String* son habitualmente **obxectos que pertencen a clase *String***, e polo tanto teñen unha serie de atributos (como un tamaño ou número de caracteres) e métodos (como retornar a lonxitude do texto).
Nas **linguaxes imperativas tradicionais**, os String son **táboas de unha dimensión** que almacenan caracteres (***char***) indicando de xeito interno o seu tamaño. Algunhas linguaxes como ***Pascal*** utilizan o primeiro char do array para almacenar indicar o tamaño do String, mentres que outras como ***C*** almacenan no último elemento do array unha "marca" indicando o final do String.

> Para diferenciar un **String** dun identificador se utilizan habitualmente as comiñas dobres. Deste xeito, *Ola* sería un identificador mentres que *"Ola"* sería un String.

4.5.2.1 – Táboas *(Arrays)* de unha dimensión ou Vectores

Como xa se comentou anteriormente unha táboa de unha dimensión é un conxunto de valores do mesmo tipo, que permite acceder a cada un de eles utilizando un ou varios índices. O número de índices necesario para o acceso se denomina *dimensión* da táboa

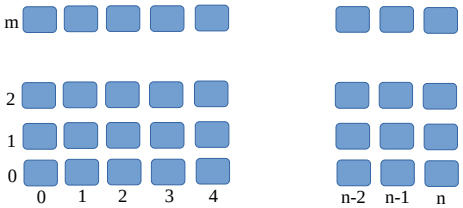
No caso das táboas de unha dimensión se precisa un único índice para acceder aos datos. Podemos representar graficamente unha táboa de unha dimensión de ***n+1*** elementos do seguinte xeito:

As táboas de unha dimensión tamén se coñecen co nome de "vectores", e son o tipo de táboa de uso máis común. A definición de táboas se realiza normalmente empregando **[]** indicando o tipo de datos e o número de elementos desexado. Por exemplo ***int e[20];*** definiría unha táboa (array) de unha dimensión identificado por ***e*** e que contén 20 elementos de tipo ***int***. Para acceder aos distintos elementos utilizaríamos os índices dende o ***0*** a o ***19***. Por exemplo ***int y=e[9];*** asignaría a variable de tipo ***int*** identificada por ***y***, o valor almacenado na posición ***9*** do array ***e***. A forma exacta de definir este tipo de datos depende da linguaxe empregada.

> Sempre que necesitemos almacenar un conxunto de datos do mesmo tipo podemos utilizar un dato de tipo array

4.5.2.2 – Táboas *(Arrays)* Multidimensionais: As matrices

Unha táboa multidimensional utiliza máis de un índice para ler ou almacenar a información. O caso máis típico son as táboas de *dúas dimensións* ou ***matrices***, que utilizan dous índices para o acceso aos datos. Podemos representar graficamente unha táboa de dúas dimensións do seguinte xeito:



Unha matriz é como unha estrutura formada por filas e columnas, onde cada fila de ***m*** elementos é un array de unha soa dimensión de ***n*** elementos. A fila se indica polo primeiro índice, mentres que a columna se indica mediante o segundo. A definición de unha matriz de 12 "filas" e 4 "columnas" podería ser algo así: ***int m[12][4];*** e accederiamos con operacións como ***int y=m[0][5];*** , que asignaría a variable identificada por **y** o valor int almacenado na posición ***[0][5]*** do array **m**.

> O uso de arrays de máis de 3 dimensións e pouco habitual. Os arrays de máis de dúas dimensións son difíciles de representar de xeito gráfico e a idea de "fila" e "columna" deixa de ter sentido.

4.5.2.3 - Estruturas dinámicas: Pilas, Colas, Listas e Árbores

Os String e Arrays son estruturas de datos relativamente simples e non ofrecen características máis avanzadas que o almacenamento de un **número máximo fixo** de elementos do mesmo tipo.

> Os ***String*** e ***Array*** na maioría das linguaxes de programación forman parte da propia sintaxe da linguaxe, mentres que as estruturas dinámicas complexas están incluídas na libraría de funcións ou clases do sistema, ou é necesario crealas dende 0.

Existen outras estruturas de datos máis avanzadas que permiten o almacenamento de un conxunto de datos sen máis límites que a memoria dispoñible. Estas estruturas van utilizando memoria segundo se vai precisando, en lugar de "marcala" como reservada na definición como o caso dos Array e String, polo que se coñecen como **estruturas dinámicas de almacenamento**. As estruturas avanzadas máis importantes son:

- **Pilas (Stack):** Unha pila é unha estrutura de tipo ***LIFO*** (Last Input First Output). Pode ser estática (si se crea utilizando un Array) ou dinámica (si se crea utilizando unha estrutura dinámica como unha Lista). Sobre este tipo de estrutura so se poden realizar dúas operacións: ***push*** que coloca un dato "enriba" dos datos existentes, e ***pop***, que retira e devolve o dato que está enriba de todo (o ultimo dato posto con ***push***). Un símil sería unha pila de pratos. So podemos poñer e quitar pratos da cima. Para acceder a un prato necesitamos antes quitar os pratos de enriba.

- **Colas (Queue)**: É unha estrutura similar ás Pilas, pero é ***FIFO*** (First Input First Output). A operación básica dunha cola consiste en poñer un dato na cola, e quitar o dato da cola. Para entender o funcionamento desta estrutura podemos facer un símil con unha cola do cine. O primeiro en pasar e o primeiro en chegar a cola.
- **Listas (List):** As listas son unha estrutura dinámica complexa. Unha lista está composta de "nodos" que almacenan os datos e a dirección do seguinte nodo (lista simplemente enlazada) ou dos nodos veciños seguinte e anterior (lista dobremente enlazada). Deste xeito podemos percorrer a lista, acceder a un elemento ou borrar un elemento.
- **Arbores (Trees)**: As árbores son listas nas que cada nodo pode ter máis de un "descendente" (podemos pensar nunha lista coma un árbore no que cada nodo so ten un descendente). Son especialmente útiles para manter ordenados os datos almacenados e se empregan habitualmente para crear índices de bases de datos. O tipo de Tree máis común é a arbore binaria, na que cada nodo pode ter dous descendentes.

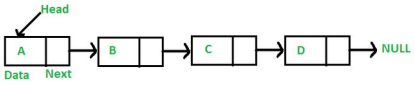


Figura 2: *Lista Enlazada*

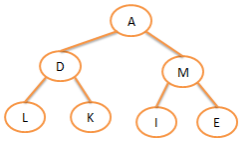


Figura 1: *Árbore binaria*

4.5.3 – Os Enum

O propósito dos **enum** é restrinxir o conxunto de datos que se poden asignar a unha variable. Case todas as linguaxes modernas os soportan. Para utilizar un enum é necesario indicar en primeiro lugar un identificador para o enum e cales son os valores que permite.
Unha vez definido o enum, será posible definir variables de ese novo tipo, restrinxindo así o conxunto de valores que pode almacenar. Unha posible sintaxe podería ser :

```
enum Dias { LUNS, MARTES, MÉRCORES, XOVES, VENRES, SÁBADO, DOMINGO }
Dias diasemana;
```

En primeiro lugar se define un conxunto de valores ou ***enum*** que se chamará "*Dias*". Iso nos permite definir variables restrinxidas a ese conxunto de valores como facemos na seguinte liña que define un espazo na memoria identificado por ***diasemana*** no que so podemos almacenar un calquera dos valores definidos no *enum Días.*

> A definición dun ***enum*** especifica o conxunto de valores que formarán o enum, o que crea un novo tipo de datos co que podemos definir variables.

4.6 – Funcións.

> Unha ***función*** é un algoritmo que realiza unha tarefa concreta de utilidade para a realización dun algoritmo.

Cando dende un algoritmo chamamos a unha función lle pasamos a información que precisa, e normalmente recibimos o resultado do procesamento da mesma.

1. A función debe recibir a información que precisa para levar a cabo o algoritmo en unhas variables especiais denominadas **parámetros** e levará a cabo as instrucións que compoñen o algoritmo para producir un **resultado** que pode ser retornado.

> Os parámetros son **variables de ámbito local** a función e se crean no instante de invocar a función desaparecendo cando a función remata.

Normalmente nunha linguaxe de programación unha función se define indicando en primeiro lugar o tipo de información que vai retornar, seguido dun identificador que da nome a función mais unha lista de definición de variables entre paréntese que recibirán os valores de entrada para que a función poda realizar o seu traballo.
Por exemplo, unha función que calcule un factorial de un número podería definirse así :

```
funcion factorial(int num) que retorna un long
        f=1;
        Mentres (num>1)
                f=f*num;
                num--;
        Fin Mentres;
        retornar f;
```

4.6.1- Paso de parámetros por valor e por referencia

Como xa se comentou os parámetros son variables locais a función, e polo tanto a modificación do seu valor non ten efecto fora da función. Simplemente o valor que pasamos cando chamamos a función se copia nas variables especificadas nos parámetros. É, polo tanto, un **paso de parámetro por valor.**
Sin embargo é posible pasar en lugar de o valor co que queremos traballar, a referencia (dirección de memoria) onde se atopa o valor. E dicir, o parámetro non almacenará o valor, se non a dirección de memoria onde está o valor, sendo posible polo tanto que a función altere datos accesibles dende fora da función. Esta técnica se coñece como **paso de parámetro por referencia**.

```
x=23
y=incrementa(x)
visualiza("Y="+y)
visualiza("X="+x)
```

```
funcion incrementa(x)
  x=x+1
  retorna x
```

Si o paso de parámetros é por valor, o algoritmo visualizaría ***Y=24*** e ***X=23***, xa que a variable **x** definida na función NON é a mesma que a **x** definida no algoritmo principal se non que é unha variable local á función "***incrementa"*** que se crea cando comeza a execución da función e se elimina a súa finalización.
Si o paso de parámetros é por *referencia* o algoritmo visualizaría ***Y=24***, ***x=24***, xa que a variable **x** definida na función NON é a mesma que a **x** definida no algoritmo principal, se non que é unha variable local á función ***"incrementa"*** que se crea cando comeza a execución da función e se elimina a súa finalización. Pero esa variable non contén o valor "23" no inicio da función xa que ***o que se copia é a referencia (dirección) da variable x*** definida no algoritmo principal e as operacións se realizan sobre o contido desa referencia, ou sexa, altera o valor da variable x do algoritmo principal.
Algunhas linguaxes utilizan operadores especiais para indicar que se está a traballar cunha referencia, outras non.

4.6.2 – A Recursión

A recursión consiste en unha función que se chama a sí mesma. Existen numerosos problemas na que a definición é de natureza recursiva (a solución está baseada na propia definición), polo que é simple resolvelos empregando recursión.
Os problemas recursivos habitualmente constan de dous casos ou estados:

- Caso básico: E o estado para o que coñecemos a solución.
- Caso recursivo: E un estado do que non sabemos a solución, pero a podemos calcular chamando de novo ao mesmo método.

Exemplo, Factorial de un número: O factorial de 0 é 1. O factorial dos outros números se calculan multiplicando o número polo factorial do número anterior.
Como vemos, o caso básico é cando o número é 0, e os casos recursivos son o resto. Isto nos permite solucionar o factorial do seguinte xeito:

```
funcion factorial(int num) que retorna un long
        if (num == 0) return 1;
        else          return num*factorial(num-1);
```

> Debemos ter coidado en que a recursión chegue a un punto en que se da unha solución en lugar de continuar chamando á función, en caso contrario caeremos nunha ***recursión infinita*** que romperá o programa ou deixará o equipo "bloqueado". Tamén pode resultar problemática un exceso de chamadas recursivas que poden agotar a memoria na pila do programa.

5. Operacións de Entrada e Saída

Para escribir un programa de ordenador verdadeiramente útil é necesario que o usuario da aplicación poda introducir de algún xeito a información a procesar no sistema e tamén poda ver o resultado do procesamento.

Para a entrada de datos por parte do usuario se utilizan **interfaces de entrada**, como o teclado, o rato, unha unidade de disco, unha tarxeta de rede ... etc.

Para facilitar os resultados ao usuario se usan as **interfaces de saída**, como a pantalla, a impresora, unha unidade de disco, unha tarxeta de rede ... etc.

As interfaces que se poden utilizar tanto para a entrada como para a saída de datos se coñecen co nome de **interfaces de entrada-saída**

5.1 Interfaces de Entrada e de Saida: Entrada e Saída en Consola

As linguaxes de programación normalmente utilizan funcións/clases que forman parte da libraría de funcións estándar da linguaxe para a realización das operacións de entrada e saída de datos.
Existen numerosas familias de funcións dependendo o tipo de entrada ou saída que queiramos facer: Ler datos do teclado, visualizar na pantalla, obter datos dunha BBDD, almacenar datos nunha BBDD, imprimir os datos... etc.

A interface de entrada máis común é o teclado, e practicamente todas as linguaxes nos permiten ler información do teclado e almacenala nunha variable. A interface de saída máis común é a pantalla, e practicamente todas as linguaxes nos permiten visualizar información na pantalla.

Exemplo:
Visualizar "Introduce o teu nome"
Ler nome
Visualizar "Hola ," nome

6. Xestión de Erros: As Excepcións

"Tradicionalmente" para xestionar as condicións de erro cando se realiza unha operación nunha función ou método se empregan valores de retorno para indicar si a operación se levou a cabo correctamente ou se produciu algún fallo. Por exemplo, o seguinte algoritmo en Java retorna a posición dunha letra nun *String*, ou -1 para indicar que a letra non se atopa (condición de erro):

```
int charPos(char c, String str) {
        int len=str.length();
        int idx=0;
        while(idx!=len) {
                if (str.charAt(idx)==c) return idx;
        }
        return -1;
}
```

Si dentro dos valores a retornar calquera resultado pode ser válido é imposible retornar un valor que indique un erro. Por exemplo o seguinte algoritmo codificado en Java fai unha división enteira mediante restas, pero é imposible retornar un código que indique que a división non se pode facer:

```
int divide(int dividendo, int divisor) {
        int resultado=0;

// Non podo colocar ningún valor int que indique un erro, porque todos os int poden ser un resultado correcto.
        if (divisor==0) resultado=????????????;
        else
                while(dividendo >= divisor) {
                        dividendo=dividendo-divisor;
                        resultado=resultado+1;
                }
        return resultado;
}
```

Si dentro dun construtor se produce unha condición de erro, é imposible notificalo cun valor de retorno, xa que un construtor SEMPRE retorna un novo obxecto da clase.

Ademais é pesado comprobar sempre o valor de retorno a continuación dunha chamada a un método ou función, e ademais enche o código de sentencias "if" o que o fai longo e difícil de ler. A consecuencia de isto moitas veces se deixan de controlar as posibles condicións de erro, coas consecuencias que iso pode ter a longo prazo.

Para solucionar estes problemas se crearon as excepcións. (***Exception***).

Una "**excepción**" é unha condición de erro que se produciu nun punto do programa e que é necesario tratar.

As excepcións se "lanzan" no instante en que se detecta o erro, e poden ser "capturadas" ou "relanzadas" no código que chama ao método ou función onde se produciu o erro.

```
int divide(int dividendo, int divisor) throws DivideByZeroException {
        int resultado=0;

        if (divisor==0) throw new DivideByZeroExceoption();
        else  while(dividendo >= divisor) {
                        dividendo=dividendo-divisor;
                        resultado=resultado+1;
              }
        return resultado;
}
```

```
// Exemplo de chamada tratando excepción en Java

try {
        r=divide(30,25);
        System.out.println("30/25 = "+r);
} catch(DivideByZeroException err) {
        System.out.println("Non se pode dividir entre 0");
}
```

7. Almacenamento de Información

Cando almacenamos información nun dispositivo (saída de información) e moi importante como ***organizamos*** esa información, xa que isto determinará a velocidade de consulta e almacenamento e a facilidade de modificación e borrado. Sen embargo, as posibilidades de organización da información están limitadas polo ***tipo de soporte*** empregado.
O modo en que organicemos a información tamén determinará as posibilidades para ***acceder*** a mesma.

> A información almacenada nun soporte recibe o nome de ***ficheiro*** ou arquivo. O lugar exacto onde lemos ou escribimos no ficheiro ven dado pola *posición de traballo* ou ***punteiro do ficheiro*** que se vai actualizando automaticamente a medida que lemos ou escribimos.

7.1 – Soporte, Organización e Acceso

> Con ***soporte*** nos referimos ao sistema de almacenamento empregado (unidade de disco, cinta magnética, SSD ... etc).
>
> Con ***organización*** nos referimos ao xeito en que organizamos a información, e dicir, os criterios que empregamos para decidir onde almacenamos a información. As posibilidades de organización da información dependen do tipo de soporte.
>
> Con ***acceso*** nos referimos ao xeito empregado para acceder a información. Os modos en que podemos acceder a información dependen de como a teñamos organizada.

Podemos distinguir os seguintes tipos de soporte:

- ***Soportes de acceso secuencial***: Para acceder a información concreta no soporte é necesario pasar por toda a información gardada con anterioridade. Non é posible colocar o *punteiro do ficheiro* nunha posición concreta. Un soporte secuencial so permite organizar os datos secuencialmente, un detrás de outro.
- ***Soportes de acceso directo***: O soporte permite colocar o *punteiro do ficheiro* no lugar desexado.

Si a información está *organizada* de xeito secuencial, so é posible acceder na orden en que foi gardada xa que non é posible localizar o lugar onde se atopa a información buscada aínda que o soporte sexa de acceso directo.

> Cando almacenamos información nun dispositivo normalmente está organizada en ***rexistros***. Un ***rexistro*** é un conxunto de información relacionada entre si denominada ***campos***.

Un exemplo de rexistro pode ser un *Cliente*, que está formado polos campos *NIF*, *Nome*, *Dirección*, *Teléfono* e *e-mail*.

> Os rexistros poden ser de ***lonxitude fixa*** (todos os rexistros miden o mesmo número de bytes) ou de ***lonxitude variable*** (uns rexistros miden máis bytes que outros). Os rexistros de lonxitude fixa son moito máis simples de organizar que os de lonxitude variable

Podemos distinguir tres xeitos principais e comúns de organizar a información nos soportes: ***Organización Secuencial***, ***Organización Indexada*** e ***Organización Directa***.

7.2 – Organización Secuencial

A organización secuencial consiste en ir almacenando os datos un detrás de outros. Cando almacenamos unha información se almacenará sempre no último lugar, ao final dos datos existentes.
Este tipo de organización se pode levar a cabo en calquera tipo de soporte, é a máis sinxela e ineficiente e ten varios problemas importantes:

- Para acceder a unha información concreta é necesario ler as anteriores.
- Non é posible modificar a información dun xeito simple e rápido, xa que a nova información pode non caber no espazo da antiga, ou pode sobrar.
- Non é posible borrar información dun xeito simple e rápido.

Si o **soporte é de acceso secuencial** para eliminar ou modificar información é necesario crear un novo ficheiro que omita os datos a borrar ou garde os datos modificados, eliminando ao rematar o ficheiro orixinal e renomeando o novo.

Si o **soporte é de acceso directo** podemos utilizar dúas aproximacións para eliminar datos:

- Incluír na información un campo “especial” que indique si os datos son válidos ou están borrados de xeito que cando accedemos a información se ignoran os rexistros nos que ese campo indica que está

eliminado. Este xeito de organizar a información fai que o ficheiro creza cada vez mais a medida que se eliminan rexistros, creando “fragmentación”, o que fai necesario “des-fragmentar” de xeito periódico.

- Copiar o ficheiro nun ficheiro novo que non teña os datos a eliminar, eliminando o ficheiro orixinal unha vez se remata e renomeando o novo ficheiro. Este tamén e o modo de “des-fragmentar” un ficheiro para eliminar os rexistros marcados como eliminados de xeito que non ocupen espazo.

Con soportes de acceso directo se poden crear organizacións secuenciais máis elaboradas e efectivas, especialmente si os rexistros son de lonxitude fixa como reutilizar o espazo marcado como “borrado” para gardar os novos rexistros. Para modificar datos en soportes de acceso directo con existros de *lonxitude variable* o máis práctico e eliminar o rexistro vello e dar de alta o novo. Si o rexistro é de *lonxitude fixa*, podemos escribir a nova información enriba da vella (xa que o soporte nos permite recolocarnos na posición onde queremos escribir).

7.3 – Organización Indexada

A ***organización indexada*** so é posible si o soporte é de acceso directo, e consiste na creación dun ficheiro adicional denominado ***índice*** que nos indica a posición no ficheiro de datos do rexistro desexado segundo o valor dun campo do rexistro que denominamos ***chave.***

O ficheiro índice habitualmente ten organizacións moi complexas como árbores B+, que permiten percorrer en orde e localizar a posición onde se atopan os datos correspondentes a un valor de chave con gran velocidade. O ficheiro que almacena os datos en cambio non precisa de organizacións complexas e pode estar organizado secuencialmente.

Os sistemas xestores de bases de datos empregan comunmente para xestionar as táboas e as relacións entre elas ficheiros indexados.

7.4 – Organización Directa

A organización directa consiste en calcular mediante unha operación sobre a información a posición que lle corresponde dentro do arquivo de datos. Esta operación se denomina ***hash***.

Esta organización e a que máis rapidamente permite almacenar, consultar e modificar información pero presenta algúns problemas:

- O ficheiro é como mínimo tan grande como a posición máis alta asignada pola operación de ***hash***.
- Non existe ningún ***hash*** perfecto que asigne a información distinta posicións distintas en todos os casos. Cando para información diferente a operación de ***hash*** produce a mesma posición temos unha ***colisión***. As colisións é necesario tratalas dun xeito especial, normalmente cun ficheiro de colisións de organización secuencial.

Cando se busca un dato, en primeiro lugar se calcula a posición onde debería estar mediante a operación hash. Si o dato non está nesa posición, se procede a examinar o ficheiro de colisións de xeito secuencial.

8.- Programación Concorrente

Habitualmente as aplicacións comezan a súa execución pola primeira instrución do algoritmo e van levando a cabo as instrucións unha por unha ata que o algoritmo remata.
Sin embargo, existen situacións nas que precisamos que a aplicación realice varias cousas ao mesmo tempo. E dicir, que sexa capaz de levar a cabo ao mesmo tempo máis de un algoritmo.

A ***programación concorrente*** ou ***multiprogramación*** consiste no deseño de algoritmos que poden ser levados a cabo correctamente ao mesmo tempo.

Existen dúas técnicas principais de programación concorrente. A creación de fíos (***threads***) e a creación de procesos.

A ***multiprogramación mediante procesos*** consiste en lanzar novos procesos (*fillos*) que leven a cabo o algoritmo desexado ao mesmo tempo que o proceso orixinal (*pai*) mediante unha operación denominada ***fork.***

A ***multiprogramación mediante fíos*** consiste en considerar que a aplicación que lanzamos é o “fío principal ou fío pai”, que para levar a cabo outro algoritmo de xeito simultáneo lanza unha función nun “fío secundario” ou “fillo” que se vai levar a cabo de xeito simultáneo ao fío principal.

O uso de procesos é típico dos sistemas UNIX / Linux mentres que os fíos son comúns nos sistemas Windows. Cada un destes sistemas ten as súas vantaxes e inconvenientes.

- Cando lanzamos un proceso con ***fork*** o proceso creado é independente do proceso principal no sentido en que non comparten ningún tipo de información. Isto fai a programación máis simple no sentido de que non se van a producir efectos laterais porque os algoritmos accedan e modifiquen a mesma información, pero ao mesmo tempo dificulta a comunicación entre os dous procesos.
- Cando lanzamos unha función nun novo ***fío de execución***, o fío pode acceder e modificar a información do fío principal. Isto fai máis simple a comunicación entre o fío principal e os seus fillos, pero o acceso simultáneo a recursos compartidos pode provocar problemas facendo necesario o uso de bloqueos (***lock***) ou "semáforos" que regulen o acceso aos mesmos.
- Os ***fíos*** se lanzan máis rapidamente que os procesos, xa que non é necesario que o sistema cree un novo proceso.

9.- Comunicación Cliente / Servidor

A ***comunicación cliente/servidor*** fai referencia a un tipo de comunicación na que unha parte espera e atende peticións (o servidor) mentres que as outras partes se conectan e as realizan (os clientes)

A comunicación cliente/servidor é típica nas reces TCP/IP e en particular en Internet. Un exemplo de comunicación cliente/servidor moi utilizado é o servizo Web no que os clientes son os navegadores (f*irefox*, *chrome... etc*) que se conectan ao servizo web ofrecido por un ***servidor web*** (*apache, nginx...etc.*).
Cando programamos un cliente necesitamos conectarnos co servidor. En cambio, cando programamos un servidor necesitaremos esperar conexións por parte dos clientes. Si desexamos poder atender simultaneamente a varios clientes, necesitaremos tamén lanzar un proceso ou un fío para atender a cada cliente que se conecte. O algoritmo a seguir sería similar o seguinte:

Cliente:

```
Configurar a Conexión co Servidor
Conectar co servidor
Mentres non se queira finalizar
        Enviar Petición
        Ler resposta
        Procesar Resposta
Fin Mentres
Pechar a conexión
```

Servidor:

```
Configurar o Servidor
Mentres non se queira deixar de dar servizo
        Esperar conexión por parte dun cliente
        Lanzar un Fío ou un Proceso para atender as peticións
Fin Mentres
```

As linguaxes de programación non incorporan a funcionalidade necesaria para estas tarefas na propia linguaxe, se non que precisan de librarías de funcións/clases externas para facer este traballo que normalmente son proporcionadas pola libraría estándar da linguaxe.

10.- Programación Orientada a Eventos

Un ***evento*** ou suceso do sistema é unha acción ou circunstancia que se da no sistema nun instante determinado, como pode ser unha pulsación dunha tecla, o movemento ou un click do rato, ... etc.

Os programas orientados a eventos están a executar un bucle cun algoritmo similar a este:

```
Mentres non remate o programa:
   Recoller Eventos
   Si temos un evento pendente
         procesar evento
   Fin Si
Fin Mentres
```

O procesamento dos eventos consiste na execución dunha función encargada de xestionar o evento denominada ***xestor de eventos***. Normalmente o sistema establece uns xestores de eventos que teñen os algoritmos necesarios para as accións básicas do sistema.

Unha ***aplicación orientada a eventos*** consiste basicamente no deseño da interface e a programación dos *xestores de eventos* de interese para esa interface ou para a funcionalidade da aplicación.

Os métodos xestores de eventos non deberían requirir moito tempo de procesamento, xa que mentres se está procesando o evento, non se recollen máis eventos, polo que o sistema queda "conxelado". Si precisamos realizar tarefas que consuman un tempo significativo deberíamos lanzar un novo fío de execución / proceso que se encargue.

10.1 - Os elementos da GUI

Os elementos da GUI permiten ao usuario interactuar coa aplicación. Podemos distinguir os seguintes elementos de uso común:

- **Etiquetas**: Son textos que se amosan na interface
- **Imaxes**: Son debuxos que se amosan na interface
- **Campos de Entrada**: Son zonas onde o usuario pode introducir información. Existen entradas de distintos tipos segundo o tipo de información que se desexa introducir.
- **Botóns**: Son elementos que se poden pulsar e lanzan un evento que provoca a execución do seu xestor
- **Fiestras**: Son recadros onde se debuxan as interfaces de usuario. Normalmente teñen unha barra de título e se poden desprazar e cambiar de tamaño.
- **Menús**: Son listas de opcións que pode elexir o usuario facendo click co rato ou seleccionando co teclado.
- **Caixas de Selección**: Son despregables que permiten elexir entre varios elementos que aparecen
- **Caixas Combinadas**: Son despregables como as caixas de selección que ademais permiten introducir texto.
- **Botóns de Selección**: Son botóns que poden estar en estado “pulsado” ou “non pulsado”.
- **Selectores ou RadioButtons**: Son botóns de selección que forman parte dun grupo no que so un de eles pode estar en estado “pulsado”.

> Cada elemento da GUI “dispara” unha serie de eventos segundo cambia o seu estado que poden ser xestionados mediante os métodos xestores de eventos programados na aplicación.

10.2 - O deseño visual: MVC

O deseño visual de aplicacións emprega unha aplicación (deseñador) que permite colocar os elementos da GUI de xeito visual (arrastrando e soltando compoñentes nunha fiestra) e programar os algoritmos dos xestores de eventos que precise a aplicación. Unha vez deseñada a interface, é posible seguir dúas aproximacións distintas:

- Se xera de xeito automático o código necesario para construír a interface. Si desexamos modificar a interface necesitaríamos alterar ese código e recompilar o programa.
- Se xera un ficheiro que describe a composición da interface, normalmente en XML. Se fai uso de obxectos ou funcións da libraría para interpretar ese documento e construír a interface. É posible modificar o aspecto da interface sen necesidade de alterar a aplicación.

Os inconvenientes do uso dun ficheiro XML de descrición da interface son unha menor velocidade a hora de “pintar” o contorno e a necesidade de un ficheiro adicional ao código executable. Pola contra é máis flexible e permite a implantación de “skins” ou “peles” que alteren a visualización do programa.

Hoxe en día é moi común utilizar a aproximación *Modelo-Vista-Controlador* ou ***MVC*** nas aplicacións con contorno gráfico. Este modelo consiste en dividir a aplicación en tres partes completamente diferenciadas:

- **O Modelo**: O modelo consiste no código que leva a cabo a funcionalidade da aplicación. Recibe información da interface de usuario, os procesa e envía o resultado producido para a súa presentación.

> O **Modelo** debe ser completamente independente da interface de usuario, se limita a recibir información, procesala e retornar o resultado

- **A Vista**: A vista xestiona a interface de usuario. E a encargada de crear a interface coa que o usuario interactúa co sistema e de presentar os resultados devoltos polo modelo.

> A **Vista** non debería procesar a información que introduce o usuario. Debe limitarse a recollela, enviala para ser procesada e recoller e presentar na interface os resultados.

- O **Controlador:** O Controlador é a parte encargada de recibir os datos do interface e envialos a parte apropiada do modelo. Tamén recolle os resultados do modelo e os envía a zona da vista apropiada para ser visualizados.

> O **Controlador** se limita a recibir datos da vista e envialas ao modelo e viceversa. Na maior parte das ocasións simplemente envía os datos a un sitio ou a outro dependendo da acción solicitada polo usuario. Un exemplo de controlador é cando decidimos que facer (modelo) dependendo da elección do usuario nun menú (vista).

Outro aspecto que deberíamos ter en conta no deseño das interfaces gráficas de usuario é que o aspecto da interface se adapte de xeito automático aos cambios de tamaño da fiestra onde se pintan, ou ó tamaño da pantalla en que se amosen. Para conseguir isto é común utilizar uns elementos gráficos especiais denominados ***administradores de deseño***.

> Os ***administradores de deseño*** axudan a manter a posición e tamaño relativo dos elementos cando cambiamos o tamaño do área de visualización. Todos os compoñentes gráficos deberían colocarse dentro dun administrador de deseño que xestionará o seu tamaño relativo e a súa posición respecto o resto de elementos.

Existen varios tipos de administradores de deseño típicos como os Hbox, Vbox, GridLayout... etc. Tamén é posible colocar un administrador de deseño dentro de outro.

> Os elementos da GUI capaces de albergar no seu interior outros elementos como os administradores de deseño se coñecen como ***contedores***.