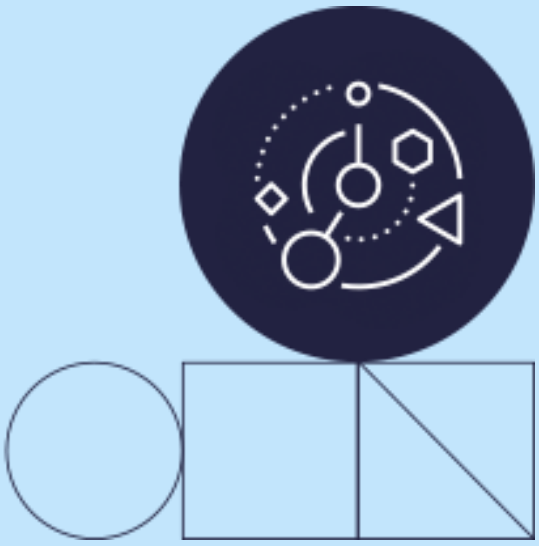
IBM SkillsBuild

IBM

Noções básicas de programação

Segmentação da programação modular, recursividade, global



Índice

Introdução

Ao desenvolver um programa, podemos nos fazer perguntas como: nosso programa segue uma programação modular? Ele tem uma estrutura básica ou é caótico? Os procedimentos e funções estão bem desenvolvidos? Nosso objetivo de estudo neste tópico será descobrir como é a programação modular, sua estrutura básica e como devem ser as funções e procedimentos dos programas. É historicamente apresentada como uma evolução da programação estruturada para resolver problemas de programação maiores e mais complexos do que a programação estruturada pode resolver.

A principal razão para usar um computador é para resolver problemas (no sentido mais geral da palavra), ou em outras palavras, para processar informações para obter um resultado a partir de dados de entrada.

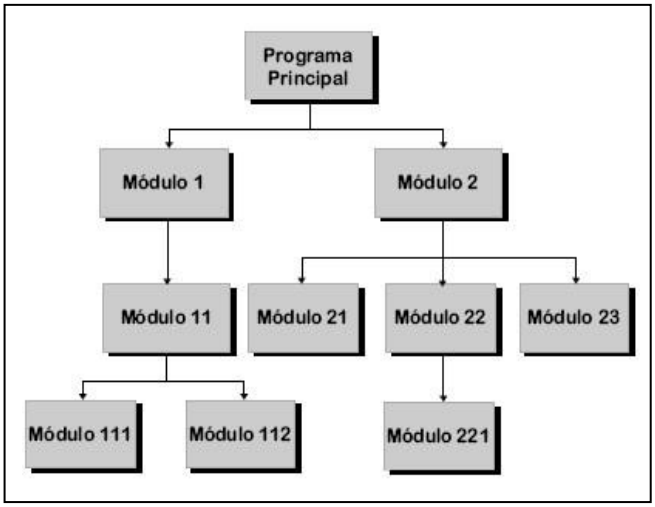
Durante a curta história dos computadores, a forma como os computadores são programados sofreu grandes mudanças. No início, a programação era uma arte (essencialmente uma questão de inspiração); mais tarde, a pesquisa teórica levou a um conjunto de princípios gerais que formam o conhecimento central de uma metodologia de programação. Isto consiste em obter "programas de qualidade". Isto pode ser avaliado por meio de diferentes características, não necessariamente em ordem de importância, que são apresentadas a seguir:

- **A correção** do programa, que é obviamente o critério indispensável, no sentido de que queremos obter programas corretos que resolvam o(s) problema(s) para o(s) qual(is) ele(s) é(são) projetado(s).
- **Compreensibilidade**, que inclui legibilidade e boa documentação, características que permitem maior facilidade e conveniência na manutenção dos programas.
- **Eficiência**, que expressa os requisitos de memória e tempo de execução do programa.
- **Flexibilidade**, ou a capacidade do programa de se adaptar às variações do problema inicial, o que permite que o programa seja usado por um período de tempo mais longo.
- **Transportabilidade**, que é a possibilidade de usar o mesmo programa em sistemas diferentes sem fazer mudanças significativas em sua estrutura.

Tendo em mente que um programa, ao longo de sua vida, é escrito apenas uma vez, mas lido, analisado e modificado muitas mais vezes, é de grande importância adquirir técnicas apropriadas de projeto e desenvolvimento, tais como a programação modular apresentada abaixo, a fim de obter programas com as características acima mencionadas.

Programação modular

A programação modular é a técnica de programação baseada na filosofia de projeto top-down, que consiste em dividir o problema original em vários sub-problemas (e estes, por sua vez, em sub-problemas menores, obtendo uma estrutura hierárquica ou em árvore) que podem ser resolvidos separadamente, e depois recompor os resultados para obter a solução para o problema. Um subproblema é chamado de módulo e é uma parte do problema que pode ser resolvido independentemente..



Um módulo é uma coleção estática de declarações definidas em um escopo particular de visibilidade e escondidas do resto do programa com o qual ele se comunica através de uma seção de interface onde a lista de exportações está incluída. Usando módulos, são construídas as unidades nas quais qualquer programa minimamente importante deve ser decomposto. Os módulos são conectados uns aos outros dando origem a uma estrutura modular em árvore que permite a solução do problema de programação.

Um módulo atua como uma caixa preta com a qual o resto do programa interage através de uma seção de interface. A interface (ou visão pública) é um conjunto de declarações de constantes, tipos, variáveis, procedimentos, funções, e assim por diante. A outra seção principal de um módulo é a implementação (ou visão privada) que inclui o código dos procedimentos e outros elementos constituintes da parte executável do módulo.

Para um bom projeto modular, os algoritmos a serem desenvolvidos devem ser concebidos como uma hierarquia de módulos intercomunicadores onde cada módulo tem uma função clara e distinta e onde nenhum módulo acessa diretamente outros módulos, mas sempre usa os mecanismos de interface.

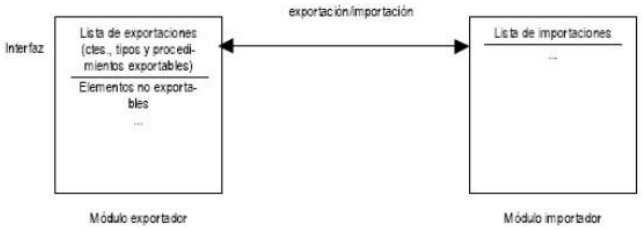
Vantagens

- Facilita o projeto de cima para baixo
- Diminui a complexidade do algoritmo
- Diminui o tamanho geral do programa
- Reutilização: economia de tempo de programação
- Divisão da programação entre uma equipe de programadores redução do tempo de desenvolvimento
- Facilidade de depuração: testes individuais dos módulos
- Programas mais fáceis de modificar
- Estruturação em bibliotecas específicas (biblioteca modular)

Interface

Um módulo pode fornecer ao módulo "comunidade" seus próprios recursos, tipos e procedimentos dentro do que é conhecido como a seção de interface do módulo. Normalmente é possível exportar cinco tipos de elementos: constantes, tipos, variáveis, procedimentos e funções.

Importação e exportação são ações simétricas. Para que um módulo possa importar um tipo, procedimento ou outro elemento, é necessário que outro módulo o exporte. Naturalmente, é possível que o mesmo módulo se comporte tanto como exportador quanto como importador.



Implementação

A parte de implementação detalha as definições e o projeto de todos os elementos contidos no módulo. A implementação pode também incluir a lista de importação para os elementos externos utilizados pelo módulo.

Como vimos, quando um projeto por decomposição modular é realizado, muitas vezes acontece que alguns módulos se referem à interface de outros módulos. Isto pode acontecer quando um módulo usa um tipo declarado em outro lugar ou chama um procedimento ou função que tenha sido definido fora do próprio módulo.

Para utilizar um tipo ou procedimento fora de um módulo, é obrigatório especificar o local onde estes elementos foram originalmente definidos. Além disso, é desejável que a declaração de objetos fora do módulo, mas usados no módulo, seja feita em uma seção separada. Para este fim, algumas versões da chamada lista de importação podem ser usadas.

Cada Em C, as importações podem ser incluídas explicitamente declarando os procedimentos e funções no mesmo módulo ou usando um arquivo de cabeçalho com a sintaxe: # include "archivoDeExportaciones.h" onde archivoDeExportaciones corresponde ao nome do arquivo específico a ser usado.

Tipos de módulos

De acordo com sua função dentro do programa:

- Programa o módulo principal.
- Módulos o módulos secunda.

Dependendo de seu **uso:**

- Funções: devolver um valor (avaliação da função).
- Procedimentos: realizar tarefas, mas não devolver um valor diretamente.

Dependendo dos mecanismos de ativação:

- Invocado por referência.
- Provocado por interrupção (em ambientes de tempo real).

De acordo com o caminho de controle (descreve a forma como é executado internamente):

- Módulos convencionais: ter uma única entrada e uma única saída e executar sequencialmente uma tarefa de cada vez.
- Módulos reentrantes: concebidos de tal forma que não podem, de forma alguma, modificar-se a si mesmos ou os endereços que referenciam localmente. Assim, o módulo pode ser usado para mais de uma tarefa ao mesmo tempo.

Dentro de uma estrutura de programa:

- Um **módulo seqüencial** que é referenciado e executado sem interrupção aparente pelo software aplicativo.
- Um **módulo incremental** que pode ser interrompido, antes de terminar, pelo software de aplicação e posteriormente retomado no ponto em que foi interrompido. Este tipo de módulo é freqüentemente referido como uma corrotina.
- Um **módulo paralelo** que executa ao mesmo tempo que outro módulo, em ambientes de multiprocessadores simultâneos. Um nome usado para este tipo é conrrutina.

Refinamento e modularidade

Aperfeiçoamento sucessivo

Proposta por Niklaus Wirth em 1971, foi uma das primeiras estratégias de projeto de cima para baixo. Nele, a arquitetura de um programa é desenvolvida em níveis sucessivos de refinamento dos detalhes processuais. Uma hierarquia é desenvolvida pela decomposição de uma declaração macroscópica de uma função sucessivamente, até que as declarações da linguagem de programação sejam alcançadas.

O refinamento é realmente um processo de elaboração. Ela começa com uma declaração de função (ou uma descrição das informações). Ou seja, a declaração descreve a função ou informação conceitualmente, mas não fornece informações sobre o funcionamento interno da função ou a estrutura interna da informação. O refinamento faz com que o designer expanda a declaração original, dando cada vez mais detalhes à medida que refinamentos sucessivos (elaborações) ocorrem.

O conceito de modularidade

Refere-se ao fato de que o software é dividido em componentes com nomes e localizações particulares, que são chamados de "módulos" e que são integrados para satisfazer as exigências do problema.

Para ilustrar este ponto, considere a seguinte discussão, baseada em observações da solução de problemas humanos.

- Que C(x) seja uma função que defina a complexidade de um problema x e E(x) seja uma função que defina o esforço (em tempo) necessário para resolver um problema x. Para dois problemas, p1 e p2, se C(p1) > C(p2) segue-se que E(p1) > E(p2).

Para um caso geral, este resultado é intuitivamente óbvio. A solução de um problema difícil leva mais tempo.

Outra propriedade interessante que foi encontrada a partir da experimentação na solução de problemas humanos é a seguinte C(pl+p2) > C(p1) + C(p2) que indica que a complexidade de um problema composto de p1 e p2 é maior que a complexidade total quando cada problema é considerado separadamente. Pode ser deduzido que

E(pl+p2) > E(p1) + E(p2)

Isto indica que é mais fácil resolver um problema complexo quando ele é dividido em pedaços mais manejáveis. Da desigualdade acima, pode-se concluir que, se dividíssemos o software indefinidamente, o esforço necessário para desenvolvê-lo seria negligenciavelmente pequeno. Entretanto, à medida que o número de módulos cresce, o esforço (custo) associado às interfaces entre os módulos também cresce. Portanto, tanto a sobre-modularização quanto a sub-modularização devem ser evitadas.

O projeto de cima para baixo ou de cima para baixo (Top Down)

Ela utiliza os conceitos de refinamento e modularidade descritos acima. Ela consiste em uma série de decomposições sucessivas do problema inicial, que descrevem o refinamento progressivo do conjunto de instruções que farão parte do projeto.

O uso desta técnica de projeto tem os seguintes objetivos básicos:

- Simplificación - Simplificação do problema e dos blocos resultantes de cada decomposição.
- As diferentes partes do problema podem ser projetadas/desenvolvidas independentemente e até mesmo por pessoas diferentes.
- O projeto final é estruturado na forma de blocos ou módulos, o que facilita a implementação e a manutenção.

A principal vantagem do projeto de cima para baixo é que ele reduz a dificuldade de resolver e manter os problemas de projeto. A desvantagem associada é que, como o problema é dividido em sub-problemas e o número de módulos cresce, há um aumento no número de interfaces entre eles com a conseqüente complexidade associada.

Ocultação de informações

O **princípio de ocultação de informações** proposto pela Parnaso sugere que os módulos devem ser "caracterizados por decisões de projeto que os escondem uns dos outros". Em outras palavras, os módulos devem ser especificados e projetados de tal forma que as informações (procedimentos e dados) contidos em um módulo sejam inacessíveis a outros módulos que não necessitem de tais informações.

A ocultação implica que, para alcançar uma modularidade eficaz, um conjunto de módulos independentes deve ser definido, que se comunicam entre si apenas através das informações necessárias para desempenhar a função do software. A abstração ajuda a definir as entidades processuais (ou de informação) que compõem o software. A ocultação estabelece e impõe restrições ao acesso aos detalhes de procedimento interno de um módulo e a quaisquer estruturas de dados usadas localmente no módulo.

O uso da ocultação de informações como critério de projeto para sistemas modulares revela seus maiores benefícios quando se torna necessário fazer modificações durante os testes e, posteriormente, a manutenção do software. Como a maioria dos dados e procedimentos será escondida de outras partes do software, os erros inadvertidamente introduzidos durante a modificação são menos propensos a se propagar em outro lugar no software.

Independência funcional e qualidade de software

O conceito de independência funcional é uma derivação direta do conceito de modularidade e dos conceitos de abstração e ocultação de informações.

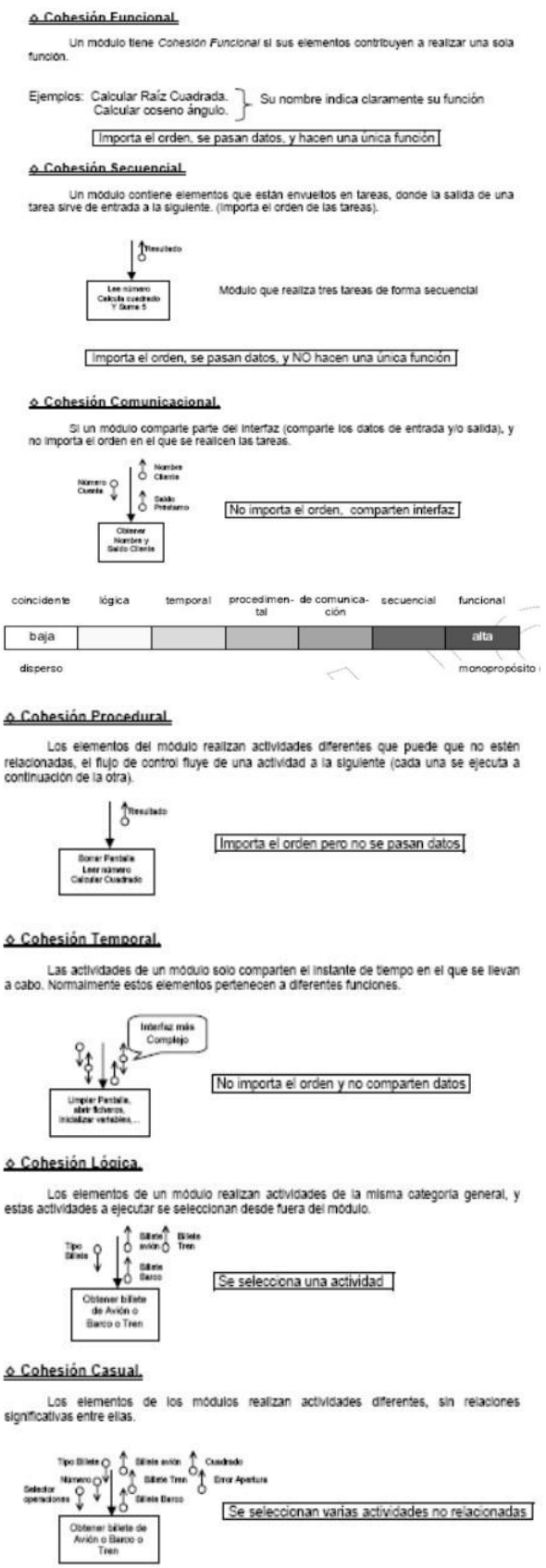
A independência funcional é alcançada através do desenvolvimento de módulos com "uma função clara" e uma "aversão" à interação excessiva com outros módulos. Em outras palavras, trata-se de projetar software de tal forma que cada módulo se concentre em uma subfunção específica dos requisitos e tenha uma interface simples, quando visto de outras partes da estrutura do software.

A independência funcional é importante no desenvolvimento de aplicações de software porque software com modularidade efetiva, ou seja, com módulos independentes, é fácil de desenvolver porque sua função pode ser dividida e as interfaces são simplificadas (considere as implicações quando o desenvolvimento é feito por uma equipe). Os módulos independentes são mais fáceis de manter (e testar) porque limitam os efeitos colaterais das modificações de design/código, reduzem a propagação de bugs e incentivam a reutilização dos módulos. Em resumo, a independência funcional é a chave para um bom design e o design é a chave para a qualidade do software.

A independência é medida por dois critérios qualitativos: coesão e acoplamento. A coesão é uma medida da força funcional relativa de um módulo. O acoplamento é uma medida da interdependência relativa entre os módulos.

Coesão

Ele mede o grau de conexão funcional entre os elementos (instruções, definição de dados, chamadas de módulo) do mesmo módulo. Quanto mais forte for a coesão, melhor será a manutenção do módulo.



Acoplamento

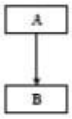
O grau de interdependência entre os módulos de um sistema. Este critério deve ser minimizado, para que o ruído do sistema seja atenuado (erros em um módulo não se propagam para outros módulos) e a manutenção seja realizada sem olhar dentro de outros módulos.

Acoplamiento Normal.

La diferencia entre los tres tipos de acoplamiento normal radica en el tipo de información que se intercambian.

Dos módulos A y B están **acoplados normalmente** si se cumplen las condiciones:

1.- A invoca a B.
2.- B realiza su función retomando el control a A.
3.- Toda información que comparten o se pasan, es por medio de los parámetros presentes en la llamada.

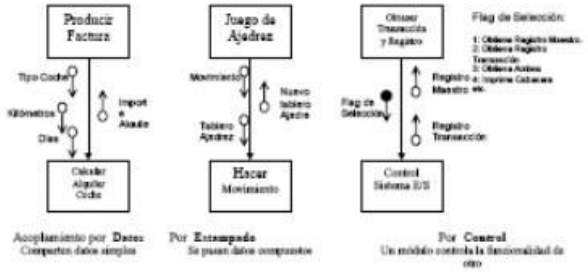


Acoplamiento (Normal) por Datos.
Dos módulos A y B están acoplados por datos si están acoplados normalmente y todos los datos que se intercambian son elementales.

Acoplamiento (Normal) por Estampado.
Dos módulos A y B acoplados normalmente están acoplados por estampado si uno le pasa a otro datos compuestos como vectores y registros. (Los datos compuestos provocan indirección, ya que se debe consultar su estructura en algún sitio).

Acoplamiento (Normal) por Control.
Dos módulos A y B acoplados normalmente están acoplados por control si uno le pasa a otro datos con la intención de controlar su lógica interna.

Ejemplos Acoplamiento Normal:



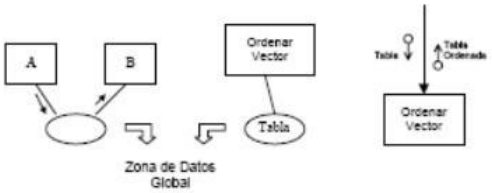
A un módulo se le debe enviar únicamente la información que necesite. Síntoma de una mala organización de los módulos es la presencia de **datos vagabundos**: datos que viajan por gran parte del D.E. sin ser utilizados por la mayoría de los módulos por los que pasan.

Acoplamiento Común (o global).

Dos módulos A y B están acoplados globalmente si se refieren a una misma zona de datos o variable global

Las variables globales no son aconsejables porque:

- Un error en un módulo puede aparecer en otro que comparte la variable.
- Es difícil saber que módulo modifica los datos
- Poca reutilización de módulos y difícil mantenimiento.



Acoplamiento por contenido.

Dos módulos están acoplados por contenido si uno se refiere al interior de otro de una de las siguientes maneras: modificando o leyendo sus datos internos o saltando al interior de su código (GOTO).

• Comparación de los distintos tipos de Acoplamiento.

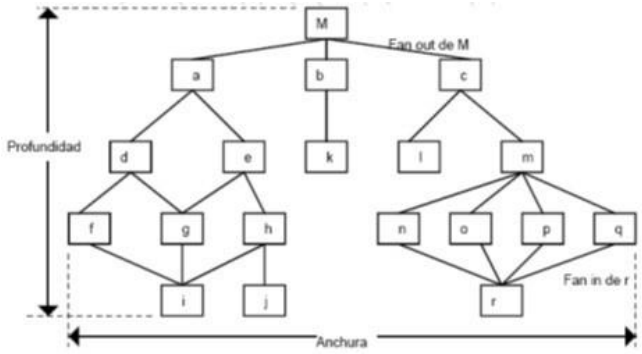
Tipo de Acoplamiento	Modificabilidad	Legibilidad	Reusabilidad Módulos
Por Datos	Buena	Buena	Buena
Por Estampado	Buena	Media	Media
Por Control	Pobre	Pobre	Pobre
Global	Media	Mala	Pobre
Contenido	Mala	Mala	Mala

Visión pesimista: Dos módulos pueden presentar varios tipos de acoplamiento, en este caso se considera que tienen el peor de los acoplamientos que presentan.

Estrutura do programa e hierarquia de controle

A hierarquia de controle, também chamada de estrutura do programa, representa a organização (muitas vezes hierárquica) dos componentes (módulos) do programa e implica uma hierarquia de controle. Não representa aspectos processuais do software, tais como a seqüência dos processos, a ocorrência ou ordem das decisões ou a repetição das operações.

Muitas notações diferentes são usadas para representar a hierarquia de controle. O mais comum é um diagrama em árvore chamado diagrama de estrutura.



- Fan out é uma medida do número de módulos que são diretamente controlados por outros módulos.
- Fan in indica quantos módulos controlam diretamente um determinado módulo.
- As relações de controle entre os módulos são expressas da seguinte forma: diz-se que um módulo que controla outro módulo é superior a ele, e inversamente, diz-se que um módulo controlado por outro módulo é um subordinado do controlador.
- A hierarquia de controle também representa duas características sutilmente diferentes da arquitetura de software: visibilidade e conectividade.
- **O escopo** indica o conjunto de componentes do programa que um determinado módulo pode invocar ou utilizar seus dados, mesmo que o faça indiretamente. É, portanto, um conceito em parte relacionado com a ocultação de informações. Assim, um dado é visível por um dado módulo quando seu valor pode ser usado ou variar do módulo, e um módulo é visível de outro módulo quando o último pode invocar a execução do primeiro.
- **A conectividade** indica o conjunto de componentes que são diretamente invocados ou cujos dados são utilizados em um determinado módulo. Por exemplo, um módulo que em um determinado momento aciona a execução de outro módulo é conectado a esse módulo.

Podemos concluir dizendo que a visibilidade é assim uma medida da conectividade potencial de um programa e que quanto maior for essa magnitudes, menor será o nível de ocultação de informações e, portanto, maiores serão as chances de aparecerem efeitos colaterais irritantes.

Projeto de funções

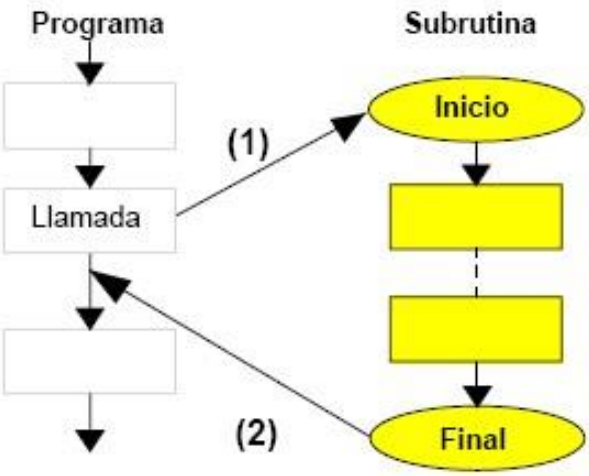
Um dos componentes que os idiomas estruturados incorporam é um tipo de seqüências algorítmicas individualizadas que podem ou não receber valores de entrada e podem ou não retornar valores de saída.

Isso envolveria a substituição de todo um conjunto de instruções que pode incluir qualquer combinação das chamadas estruturas básicas (seqüenciais, condicionais, iterativas) por um identificador que pode incorporar a declaração de valores. Posteriormente, o conjunto de instruções nomeado pode ser invocado em qualquer parte do programa usando o identificador. Após a execução, e dependendo do tipo de estrutura utilizada, os resultados podem ser devolvidos.

Procedimentos

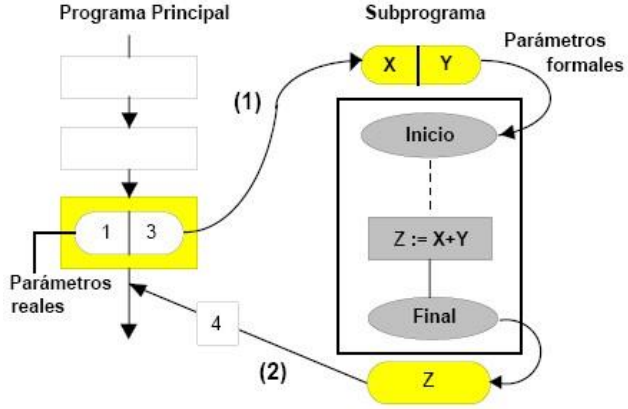
Eles são usados para definir partes de um programa, associando um identificador. Estas partes podem então ser ativadas usando declarações de chamada. Um procedimento é, portanto, um algoritmo projetado de tal forma que é suscetível de ser chamado por outros algoritmos que, por sua vez, podem ser procedimentos.

Você chama um procedimento digitando seu nome seguido das expressões em que deseja que ele funcione. Estes estão entre parênteses e na ordem em que foram especificados na chamada de procedimento.



Parâmetros

Os parâmetros são tanto as variáveis usadas na definição do procedimento como os valores usados na chamada. As primeiras são chamadas parâmetros formais e as últimas são chamadas parâmetros reais.



Os parâmetros reais, dependendo do caso, podem ser constantes, variáveis definidas no escopo do procedimento de chamada ou expressões (uma mistura de constantes, variáveis e operadores).

A lista de parâmetros formais {pf1, pf2,..., pfn} e a lista de parâmetros reais de qualquer chamada a ela {pr1, pr2,..., prm} devem satisfazer as seguintes condições:

- m = n
- Dado um par de parâmetros que ocupam a mesma posição em suas respectivas listas pfi e pri, seu tipo deve ser igual ou pelo menos compatível. Entretanto, seu nome não tem que ser o mesmo.

Dependendo de seu papel no procedimento, os parâmetros, formais ou reais, podem ser de três tipos:

- **Parâmetros de entrada**: são aqueles usados para fornecer dados ao procedimento. Se houver uma mudança no valor do parâmetro formal dentro do procedimento, o parâmetro real não será afetado. Os parâmetros reais, neste caso, podem ser constantes, variáveis ou expressões.
- **Parâmetros de saída**: são aqueles usados para exportar os dados do procedimento. Eles não fornecem um valor inicial, portanto são inicializados diretamente pelo procedimento que atribui valores a eles. Assim, mudanças no valor do parâmetro formal afetarão o parâmetro real, que deve ser uma variável.
- **Parâmetros de entrada/saída**: estes são aqueles cuja função inclui os dois parâmetros anteriores. Por um lado, eles fornecem valores e por outro lado são modificados pelo procedimento de exportação de valores. Os parâmetros reais também devem ser variáveis.

Parâmetro de passagem

- ***Por valor:*** únicamente estamos interessados apenas no valor, não nas modificações que ele possa ter dentro do subalgoritmo. Trabalhamos com uma cópia do valor passado. Estes são parâmetros unidirecionais, que passam informações do algoritmo para o subalgoritmo. Pode ser qualquer expressão que possa ser avaliada naquele momento.
- ***Por referência:*** uma referência é passada para o local da memória onde ela é encontrada. Eles são usados tanto para receber como para transmitir informações entre o algoritmo e o subalgoritmo. Deve ser uma variável.

Definiçãode procedimentos

Deve ser utilizada a sintaxe da linguagem de programação que está sendo utilizada.

Em geral, todos os idiomas geralmente dividem esta definição em duas partes:

- **Cabeçalho ou interface**: inclui o identificador do procedimento geralmente precedido por uma palavra reservada, como procedimento, e a lista formal de parâmetros com zero ou mais parâmetros. Esta lista indica o tipo dos parâmetros e sua classe. Para o tipo, algumas notações específicas devem ser utilizadas. Para fins de explicação, utilizaremos as seguintes palavras reservadas que precederão os parâmetros:
    - Input, eles são precedidos pela palavra reservada ent.
    - Saída, pela palavra sal.
    - Input/output, pela palavra entSal.
- **Corpo**: o corpo do procedimento é composto pelas declarações e instruções nas quais o próprio algoritmo do procedimento é executado.

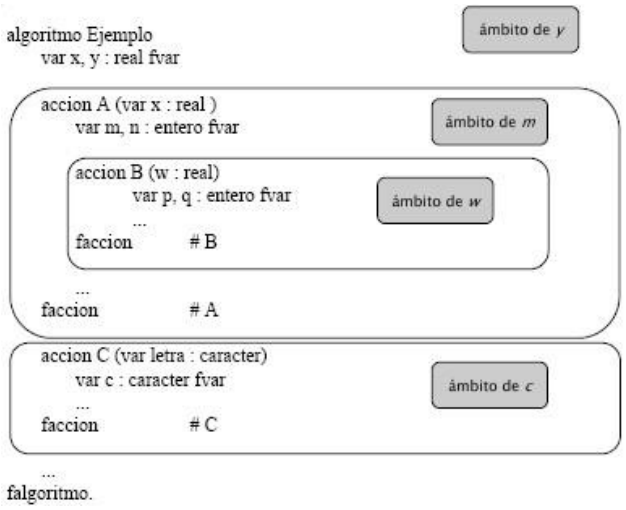
Funções

São procedimentos com características peculiares: com exceção de um parâmetro de saída, todos os outros são parâmetros de entrada. O parâmetro de saída serve para manter o valor retornado pela função.

Não é por acaso que as funções têm apenas um parâmetro de saída. De fato, as funções de saída fazem sentido como um tipo diferente de subprograma do que procedimentos, pois calculam um valor, um valor que é atribuído ao parâmetro de saída e que é acessado de fora da função usando o identificador da própria função usado na chamada. É por isso que na chamada a uma função o parâmetro de saída não deve aparecer na lista de parâmetros. Esta estrutura, além de facilitar a escrita de aplicações, evita efeitos colaterais indesejados que podem ocorrer ao utilizar parâmetros de E/S processuais típicos.

A instrução de chamada avalia a função e faz uma atribuição interna ao identificador que pode então ser atribuído, na mesma chamada, a uma variável. As chamadas de função também podem aparecer em expressões complexas que não sejam atribuição. A exigência que deve ser sempre atendida é que o tipo do valor que a função retorna seja compatível com o exigido na expressão. Este tipo do valor retornado é conhecido como o tipo da função.

A definição de funções é muito semelhante à definição de procedimentos. Entretanto, há uma série de diferenças:



- Como já dissemos, a função tem um tipo igual ao tipo do valor que ela retorna. Para especificar este tipo, na definição da função, vamos adicioná-lo no final do cabeçalho, após a lista de parâmetros e usando a sintaxe normal de especificação do tipo, ou seja, dois pontos, ':', seguido do tipo retornado.
- Como todos os parâmetros de entrada são parâmetros de entrada, a lista formal de parâmetros não utiliza a tag en.
- O valor retornado pela função é indicado utilizando a palavra reservada retorna seguido da expressão correspondente a esse valor. É costume adotar a convenção que esta instrução aparece apenas no final da função e é a única maneira de terminar a função.

Escopo do identificador

Um conjunto de declarações onde este identificador pode ser usado.

- **Variáveis locais:** variável declarada dentro de um subprograma e, portanto, disponível somente durante a operação do subprograma.
- **Variáveis globais:** variáveis declaradas no programa principal e, portanto, podem ser utilizadas pelo programa principal e por todos os seus subprogramas.
- ***Efeito lateral:*** efeito de um módulo em outro módulo que não faz parte da interface explicitamente definida entre eles.

Regras para o cálculo do escopo de um identificador:

- Um identificador declarado em um bloco é acessível somente a partir desse bloco e de todos os blocos nele incluídos (é considerado local para esse bloco). Um parâmetro formal também é considerado uma declaração local para o bloco de função.
- Os identificadores declarados fora de qualquer bloco são considerados globais e podem ser usados de qualquer parte do arquivo.
- Quando temos um bloco dentro de outro bloco e em ambos os blocos são declarados identificadores com o mesmo nome, o do bloco interno "esconde" o do bloco ext.

Recursividade

- Um objeto é recursivo se sua definição exigir a definição prévia do objeto em um caso mais simples.
- Uma função é recursiva se sua resolução requer a solução prévia da função para casos mais simples.
- **Um algoritmo** A que solucione um problema P é recursivo se for baseado **direta ou indiretamente** em si mesmo.

Algoritmos recursivos são particularmente apropriados se o problema em si ou o cálculo a ser realizado ou a própria estrutura com a qual o problema funciona aceita uma definição recursiva. Entretanto, a existência de tais definições não garante que a melhor maneira de resolver um problema seja usar um algoritmo recursivo.

Sempre que projetamos um algoritmo recursivo, devemos assegurar não apenas que o número de chamadas seja finito (a recursividade termina), mas também que seja pequeno, já que o espaço de memória (pilha de recursividade) é necessário para armazenar em cada chamada os objetos locais, os parâmetros da chamada e o estado do processo em andamento para recuperá-la quando a chamada atual terminar e a antiga tiver que ser retomada.

Tipos de recursividade

Recidiva linear:

Se cada chamada recursiva gera, no máximo, uma outra chamada recursiva.

- FINAL: se a chamada recursiva for a última operação a ser realizada, retornando como resultado o que quer que tenha sido obtido da chamada recursiva sem nenhuma modificação.
- NÃO-FINAL: O resultado obtido da chamada recursiva é combinado para dar o resultado da função de chamada.

Ejem: Función recursiva que calcule el factorial de un número entero positivo.

```
funcion Fact(n:entero) retorna entero
inicio
  si n=0 entonces
    devolver(1)
  si_no
     devolver(n * Fact(n-1))
  fin_si
fin_funcion
```

Recidiva múltipla:

Se qualquer chamada pode gerar mais de uma chamada adicional.

Ejem: Función recursiva que calcule el término n de la serie de Fibonacci. Siendo n entero positivo.

```
funcion Fibo(n:entero) retorna entero
inicio
  si n=0 o n=1 entonces
    devolver(1)
  si_no
     devolver(Fibo(n-1) + Fibo(n-2))
  fin_si
fin_funcion
```

Recidiva aninhada:

Há recorrência aninhada quando um dos argumentos da função recursiva é o resultado da chamada recursiva.

Ejem: Calcular la raiz n de un número entero x

```
Funcion raices (n,x: entero) retorna real
inicio
     según sea
     n=1: devolver(raiz2(x))
     n>1: devolver raices(n – (n div 2), raices((n div 2),x))
 fin_según
fin_funcion
```

Etapas de desenho recursivo

Um projeto recursivo consistirá das seguintes etapas:

- Definição do problema.
- Análise do caso. Identificação da função limitadora.
- Transcrição algorítmica e verificação de cada caso.
- Validação da indução: a função limitadora diminui estritamente nas chamadas.

A recursividade nos algoritmos existe quando um algoritmo se invoca a si mesmo ou é invocado em outro algoritmo anteriormente chamado algoritmo. A recursividade requer duas condições para seu correto funcionamento:

- As invocações sucessivas devem ser realizadas com versões cada vez menores do problema inicial.
- Sem esta condição de terminação, o algoritmo não poderia ser construído seguindo esta técnica e sua execução produziria um ciclo infinito.

A recorrência e a iteração são os dois mecanismos fornecidos pelas linguagens de programação para descrever cálculos que têm que ser repetidos um certo número de vezes.

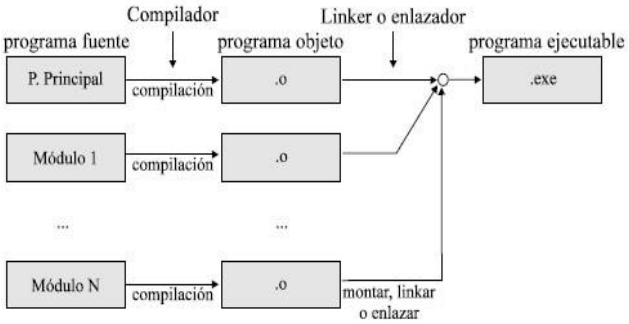
Bibliotecas

Uma biblioteca é *um conjunto documentado, testado e, quando apropriado, previamente compilado de procedimentos e funções que podem ser invocados a partir de outro programa. As bibliotecas são um exemplo claro de reutilização de software.*

Uma biblioteca básica deve fornecer uma coleção de estruturas de dados, funções e procedimentos independentes do tipo de aplicação onde eles serão utilizados. Esta coleção deve ser suficiente para cobrir as necessidades da maioria das aplicações nos idiomas que permitem seu uso. Além disso, uma biblioteca ideal deve ser:

- **Abrangente**: a biblioteca deve fornecer uma família de subprogramas, ligados por uma interface compartilhada, mas utilizando cada representação de forma diferente, para que os desenvolvedores possam selecionar aquelas mais apropriadas para a aplicação em questão.
- **Adaptável**: todos os aspectos específicos da plataforma devem ser claramente identificados e isolados, para que substituições e adaptações locais possam ser feitas (por exemplo, pelo uso de funções proprietárias que sejam intermediárias entre o código da aplicação e as invocações dos procedimentos da biblioteca).
- **Eficiente**: os componentes devem ser fáceis de incorporar ao próprio código (eficiência de recursos do compilador), usar quantidades razoáveis de memória e tempo de execução (eficiência de tempo de execução) e ser compreensíveis e seguros de usar (eficiência de recursos de desenvolvimento).
- **Seguro**: é um requisito fundamental que a biblioteca seja totalmente testada em todos os ambientes previsíveis. Um dos indicadores de tal robustez é o uso de exceções para identificar condições para as quais as pré-condições de um algoritmo são violadas. Quando estas exceções são geradas, o sistema deve ser capaz de manter a estabilidade sem reações anormais, quebras abruptas na seqüência de execução ou corrupções no espaço de endereçamento do programa.
- **Simples**: uma característica que está se tornando cada vez mais difícil de atender (as técnicas O.O. ajudam muito neste aspecto). É uma questão de fornecer à biblioteca uma organização clara e consistente que facilite a identificação e seleção das estruturas e procedimentos apropriados para o propósito requerido.
- **Extensível**: os desenvolvedores internos devem ser capazes de acrescentar funcionalidade à biblioteca sem alterar sua integridade arquitetônica original.
- **Independente da plataforma de execução final**: uma característica que está se tornando cada vez mais importante. O objetivo é tornar a biblioteca o mais independente possível do hardware e do sistema operacional onde a aplicação que está sendo desenvolvida será executada no final. Para este fim, são criadas bibliotecas abstratas que atuam como uma interface. Estas bibliotecas se conectam de forma transparente para o desenvolvedor com outras bibliotecas que dependem dos serviços da plataforma.

Esta conexão pode ocorrer tanto na compilação do código, o que significa que ele terá que ser recompilado para cada tipo de plataforma, ou dinamicamente em tempo de execução, como ocorre, por exemplo, com as bibliotecas de classe Java, o que requer o uso do que é conhecido como "máquinas virtuais".



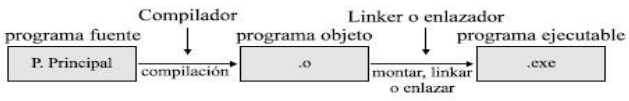
APIs

Uma API (Application Programming Interface) é um conjunto de bibliotecas de programação, desenvolvido e publicado pelos fabricantes de elementos como sistemas operacionais ou dispositivos de hardware, para permitir que os programadores de aplicações utilizem os serviços e possibilidades desses elementos. Por extensão, qualquer conjunto de funções que fazem parte de certas aplicações, mas que são utilizáveis a partir de outras aplicações, é chamado de API.

Quando dividimos um problema em módulos diferentes, cada módulo nem sempre tem que corresponder a um subprograma. Ele pode corresponder a todo um conjunto de subprogramas que serão agrupados em um arquivo separado. A principal vantagem de se dividir o programa em arquivos separados é que eles podem ser programados e compilados separadamente. Desta forma, não é necessário compilar o programa inteiro a cada vez.

Isto normalmente é feito para programas muito grandes ou simplesmente quando desejamos criar uma biblioteca de subprogramas.

- Compilação de um programa em um único arquivo:



- Compilação de um programa em vários arquivos:

Como observado acima, a maioria dos idiomas permite ao programador criar bibliotecas de funções que podem ser invocadas de dentro de seu programa e aparecer como se fossem incorporadas ao próprio idioma. Normalmente, em ambientes Windows, os módulos do programa contendo as funções são pré-compilados em arquivos de programas objeto (.obj) que podem ser agrupados em arquivos de biblioteca (.lib) usando um bibliotecário (um programa auxiliar ou parte de uma IDE de ambiente de desenvolvimento integrado).

Quando uma versão executável final de uma aplicação deve ser criada, um linker escaneia os arquivos objeto da aplicação em busca de referências a funções que não estão definidas no próprio programa, depois escaneia todos os arquivos da biblioteca que foram solicitados para uso, procurando por funções em falta. O linker extrai os módulos contendo as funções invocadas, inclui-os no arquivo executável e os vincula às chamadas do programa da aplicação.

Este processo é conhecido como **link estático**, já que todas as informações de endereçamento necessárias ao programa para acessar as funções da biblioteca são fixadas definitivamente quando o executável é criado e permanecem inalteradas no momento da execução. Tradicionalmente, os linkadores incluem os módulos inteiros ao ligá-los nos executáveis finais, embora as últimas versões IDE sejam agora capazes de extrair apenas o código correspondente à função referenciada.

A ligação estática produz um enorme desperdício de memória já que cada programa inclui sua própria cópia de cada biblioteca utilizada (por exemplo, imagine o gerenciamento de janelas no Windows).

Com o **link dinâmico**, os módulos do programa contendo as funções também são pré-compilados em arquivos de programa objeto (.obj), mas, em vez de agrupá-los em arquivos de biblioteca, eles são ligados em um formato especial de arquivo executável do Windows conhecido como DLL7, biblioteca de link dinâmico. Quando uma DLL é construída, o construtor especifica quais funções devem ser acessíveis a partir de outras aplicações em execução por meio da técnica, já estudada, chamada exportação.

Ao criar um arquivo executável para Windows, o linker analisa os arquivos de destino do aplicativo e lista todas as funções que ainda não estão incluídas no código do programa, juntamente com uma indicação das bibliotecas de links dinâmicos onde elas estão localizadas.

Ao executar uma aplicação com acesso a DLLs, cada vez que uma função localizada na biblioteca de links dinâmicos é invocada, o endereço real do link é calculado e a função é ligada dinamicamente à aplicação. Desta forma, embora haja uma única cópia na memória de cada DLL, os programas podem compartilhar as funções incluídas na biblioteca de DLLs.