Modularização

Introdução à Modularização

Exemplos de modularização, i.e., sistemas que são compostos por módulos com funções bem definidas e tão independentes quanto possível, são bem conhecidos. Por exemplo, a maior parte dos sistemas de alta fidelidade para audiófilos são compostos por módulos: o amplificador, o equalizador, o leitor de CD, o sintonizador, o leitor de cassetes, etc.

A divisão dum sistema em módulos tem várias vantagens. Para o fabricante, por um lado, a modularização tem a vantagem de reduzir a complexidade do problema, dividindo-o em subproblemas mais simples, que podem inclusivamente ser resolvidos por equipas independentes. Até sob o ponto de vista do fabrico é mais simples alterar a composição de um módulo, por exemplo porque se desenvolveram melhores circuitos para o amplificador, do que alterar a composição de um sistema integrado. Por outro lado, é mais fácil detectar problemas e resolvê-los, pois os módulos são, em princípio, razoavelmente independentes. Claro que os módulos muitas vezes não são totalmente independentes.

Por exemplo, o sistema de controlo à distância duma aparelhagem implica interação com todos os módulos simultaneamente. A arte da modularização está em identificar claramente que módulos devem existir no sistema, de modo a garantir que as ligações entre os módulos são minimizadas e que a sua coesão interna é máxima. Isto significa que, no caso dum bom sistema de alta fidelidade, os cabos entre os módulos são simplificados ao máximo e que os módulos contêm apenas os circuitos que garantem que o módulo faz a sua função. A coesão tem portanto a ver com as ligações internas a um módulo, que idealmente devem ser maximizadas. Normalmente, um módulo é coeso se tiver uma única função, bem definida.

Para um utilizador, por outro lado, a modularização tem como vantagem principal permitir a alteração de um único módulo sem ter de comprar um sistema novo. Claro que para isso acontecer o novo módulo tem de (1) ter a mesma função do módulo substituído e (2) possuir uma interface idêntica (os mesmo tipo de cabos com o mesmo tipo de sinal eléctrico).

Isto é, os módulos, do ponto de vista do utilizador, funcionam como "caixas pretas" com uma função bem definida e com interfaces bem conhecidas. Para o utilizador o interior de um módulo é irrelevante.

Mas a modularização tem outras vantagens: o amplificador pode no futuro ser reutilizado num sistema de vídeo, por exemplo, evitando a duplicação de circuitos com a mesma função (se nunca pensou nisso, lembre-se que a televisão tem o seu próprio amplificador, normalmente de fraca qualidade, servindo apenas para a encarecer). Aliás, o amplificador era já utilizado para amplificar o sinal de vários outros módulos (e.g., o leitor de CD ou o sintonizador).

As vantagens da modularização são muitas, como se viu. A modularização é um dos métodos usados em engenharia da programação para desenvolvimento de programas de grande escala. Mas a modularização é útil mesmo para pequenos programas, quanto mais não seja pelo treino que proporciona. Estes assuntos serão estudados com mais profundidade em Engenharia da Programação (IGE) e Concepção e Desenvolvimento de Sistemas de Informação (ETI).

As vantagens da modularização para a programação são pelo menos as seguintes:

1. Facilita a detecção de erros, pois é em princípio simples verificar qual é o módulo responsável pelo erro.

2. É mais fácil testar os módulos individualmente do que o programa completo.

3. É mais fácil fazer a manutenção (correção de erros, melhoramentos, etc.) módulo por módulo do que no programa total. Além disso, a modularização aumenta a probabilidade dessa manutenção não ter consequências nefastas nos outros módulos do programa.

4. Permite o desenvolvimento independente dos módulos. Isto simplifica o trabalho em equipa, pois cada elemento, ou cada subequipe, tem a seu cargo apenas alguns módulos do programa.

5. Porventura a mais evidente vantagem da modularização em programas de pequena escala, mas também nos de grande escala, é a possibilidade de reutilização do código* desenvolvido.

Um programador assume, ao longo do desenvolvimento dum programa, os dois papeis descritos acima: por um lado é fabricante, pois é sua responsabilidade desenvolver módulos; por outro é utilizador, pois fará com certeza uso de outros módulos, desenvolvidos por outrem ou por ele próprio no passado. Esta é uma noção muito importante.

É de toda a conveniência que um programador possa ser um mero utilizador dos módulos já desenvolvidos, sem se preocupar com o seu funcionamento interno: ele sabe qual a interface do módulo e qual a sua função, e usa-o. Isto permite reduzir substancialmente a complexidade da informação que o programador tem de ter presente na sua memória, conduzindo por isso a substanciais ganhos de produtividade e a uma menor taxa de erros.

Modularização

Modularização pode ser entendida como um conceito computacional que é empregado para dividir o seu programa em partes funcionais, partes essas que conversam umas com as outras. Observando bem, toda a nossa solução de software é, no final das contas, sequencial, isto é, o software é executado linha por linha, uma após a outra.

Porém, às vezes, uma linha manda o seu programa para outra parte dele e depois volta, isso é o que aprendemos com as Funções. Ainda assim, ao entrar na Função, a execução continua linha a linha.

Mas note que a Função é uma parte do seu programa que está em uma região diferente e é acessada por meio de uma chamada a ela e, quando a sua execução termina, volta para quem a chamou.

Dito isto, por que devemos pensar em nossa solução de software de forma modular, como se fossem blocos a serem unidos de alguma forma, como em um quebra cabeça, ou até mesmo como peças de Lego? Lembram-se que eu disse, lá no começo, que o objetivo de uma Função é fazer algo muito específico? Pois bem, o fazer algo específico é a parte modularizada, é o bloquinho que será juntado a outras partes.

Fazendo uma analogia com uma colônia de formigas: cada formiga tem uma "função" particular dentro da colônia e faz a sua parte corretamente, de forma que a "sociedade das formigas" funciona perfeitamente bem nesse regime! Se uma das formigas não cumprir com o seu papel, as consequências serão gravíssimas para a colônia. Em computação devemos manter isso em mente sempre.

A dificuldade da maioria das pessoas está em definir o "algo específico" para uma Função. Muitas vezes, principalmente quando estamos começando a programar, acabamos pondo muito código dentro de uma Função, código esse que provavelmente deve estar em outra parte do programa, outra Função, até mesmo uma Biblioteca.

Como alcançar esse nível de abstração? Como saber o que é o realmente necessário codificar em uma Função, ou até mesmo em uma Biblioteca? Bom, não existe uma fórmula mágica, mas o segredo é praticar.

Quanto mais você pratica, mais você refina o seu conhecimento de programação, melhora seu poder de observação, de abstração, síntese e começa a perceber que aquela Função que tem 20 linhas pode ser dividida em três funções, ou até mesmo quatro.

Aí você começa a perceber que muito desse código é repetitivo, isto é, muitas linhas você repete em várias partes do seu programa, então você começa a se perguntar: precisa sempre repetir tanto código?

E se eu fizesse uma Função que tem esse código e apenas chamá-la quando precisar dela em qualquer parte do meu programa? E se eu colocar esse código repetitivo em uma biblioteca e sempre utilizá-la quando precisar, em qualquer parte de quaisquer programas que eu estiver codificando?

Modularizar é isto basicamente, o que acaba resultando em Reutilização de código, algo extremamente importante atualmente em nossa área. Se a roda já existe, por que reinventá-la?

Se já existem inúmeras bibliotecas disponíveis por aí, para resolver diversos tipos de problemas, você deve utilizá-las, para poupar tempo de desenvolvimento de software. Para as empresas, o tempo é algo crucial e não deve nunca ser menosprezado, pois tempo é dinheiro e ninguém quer ter prejuízo.

Suas aplicações ficarão mais elegantes e limpas se conseguir modularizar e reutilizar código de forma adequada e isso será muito bom para o seu currículo, para a sua carreira e para a empresa que te contratar.

Funções, procedimentos e bibliotecas são recursos computacionais importantes para organizar, modularizar e reutilizar código em suas aplicações, não somente em linguagem C, mas em qualquer outra linguagem de programação.

Minha sugestão para você que pretende usar programação em qualquer projeto seu, seja ele uma solução de Internet das Coisas, Sistemas Embarcados, Eletrônica Digital, Aplicação Desktop ou Web, é: Antes de sair codificando, analise o problema, analise o contexto, arquitete sua solução, faça um desenho dos módulos da sua aplicação, faça um fluxograma.

Cada módulo será responsável por algo que deve ser feito na sua aplicação, e deve solucionar de forma bem objetiva uma parte do problema, podendo ser implementados em forma de bibliotecas e, dentro das bibliotecas, como funções.

Essa prática de analisar, avaliar e desenhar blocos e como esses blocos conversam entre si é ótima para desenvolver sua capacidade de abstração e organização.

Acoplamento Entre Módulos e Coesão de Módulos

Coesão e Acoplamento são princípios de engenharia de software muito utilizados. Quando queremos ter uma arquitetura madura e sustentável, temos que levar em conta estes dois princípios, pois cada um deles tem um propósito específico que visa melhorar o design do software. O que acontece é que muitas pessoas não sabem a diferença entre eles e acabam não conseguindo obter os benefícios que colocá-los em prática na hora de se desenhar a arquitetura de um software.

Coesão está, na verdade, ligado ao princípio da responsabilidade única, que foi introduzido por Robert C. Martin no início dos anos 2000 e diz que uma classe deve ter apenas uma única responsabilidade e realizá-la de maneira satisfatória, ou seja, uma classe não deve assumir responsabilidades que não são suas. Uma vez sendo ignorado este princípio, passamos a ter problemas, como dificuldades de manutenção e de reuso. Observe o exemplo abaixo:

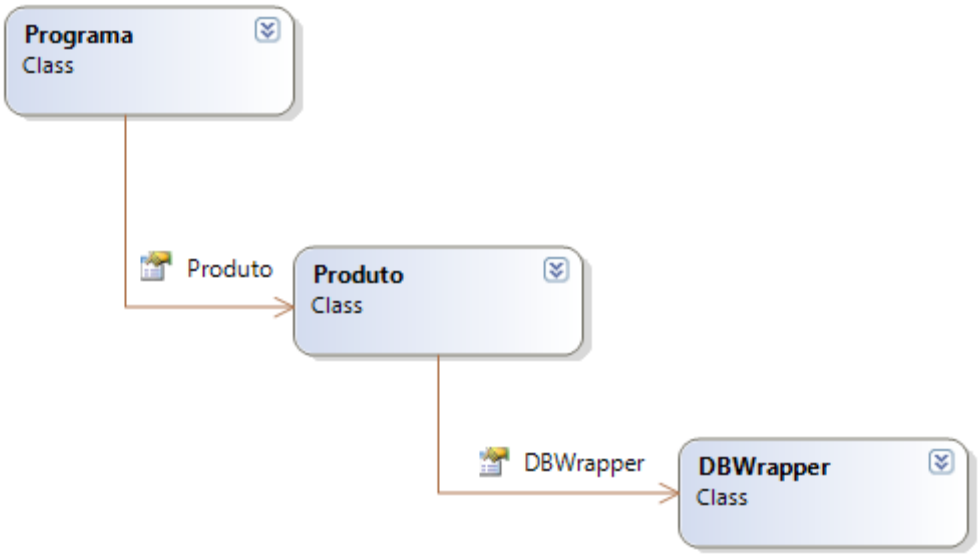
```
public class Programa
{
    public void ExibirFormulario()    {
        //implementação
    }

    public void ObterProduto()    {
        //implementação
    }

    public void gravarProdutoDB   {
        //implementação
```

```
    }
}
```

Como visto no exemplo acima, a classe Programa tem responsabilidades que não são suas, como obter um produto e gravá-lo no banco de dados. Então, dizemos que esta classe não está coesa, ou seja, ela tem responsabilidades demais, e o que é pior, responsabilidades que não são suas. Observe abaixo outro exemplo:

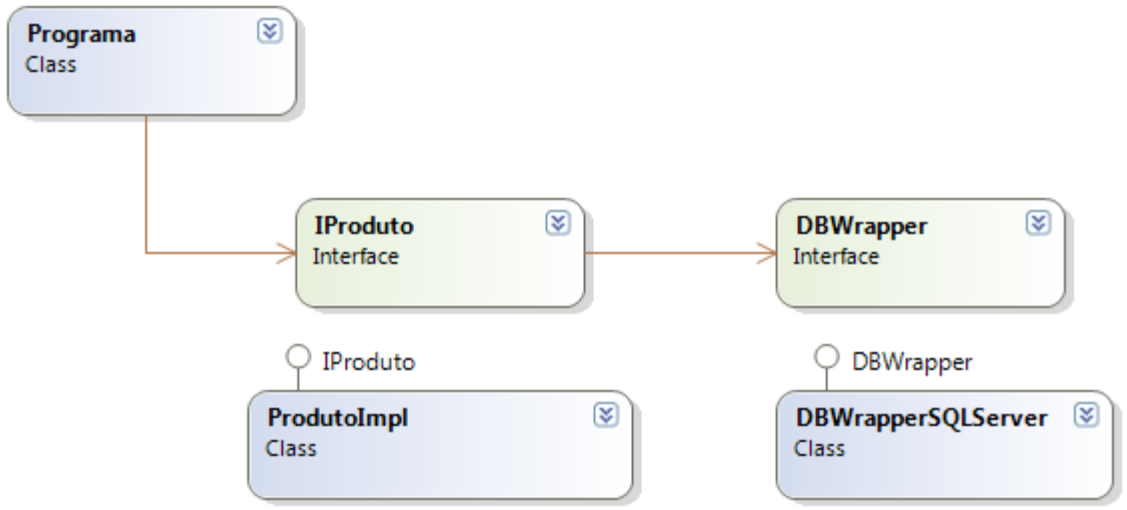
```
public class Programa
{
    public void MostrarFormulario()    {
      //Implementação
    }
   public void BotaoGravarProduto( ) {
      Produto.gravarProduto();
   }
}
```

Vemos no exemplo acima, uma clara separação de responsabilidades, o que contribui para um design desacoplado e organizado. O formulário não assume o papel de cadastrar o produto, ele pede a quem tem a responsabilidade para que faça tal tarefa. O que temos que ter em mente é que uma classe deve ser responsável por exercer uma única responsabilidade e fazer outras classes cooperarem quando necessário. Já o acoplamento significa o quanto uma classe depende da outra para funcionar. E quanto maior for esta dependência entre ambas, dizemos que estas classes elas estão fortemente acopladas. O forte acoplamento também nos traz muitos problemas, problemas até semelhantes aos que um cenário pouco coeso nos traz. Observe o diagrama abaixo:



Como podemos ver na cadeia de classes acima, o forte acoplamento na mesma, torna muito custoso a sua manutenção e o seu gerenciamento, pois qualquer mudança vai afetar toda a cadeia de classes. A saída para cenários como este, é o que chamamos de Inversão de Controle que foi abordado

em um post aqui. E uma maneira de mudar o quadro acima, seria inverter o controle e utilizar o padrão de injeção de dependência, para diminuir o acoplamento e evitar futuros problemas. Observe o diagrama abaixo com as devidas alterações:



Perceba como a dependência está na direção oposta, ou seja, não é mais de implementações concretas que estão baseados os relacionamentos entre as classes. Observe que utilizar abstrações, mantém um cenário que estará preparado para os impactos que as possíveis mudanças poderiam trazer.

Repare que a classe Produto se relaciona com uma abstração(interface) de DBWrapper, ou seja, tem uma classe chamada DBWrapperSqlServer que implementa esta interface, amanhã, o banco de dados passar a ser Oracle, basta adicionar uma classe DBWrapperOracle para atender a mudança de banco de dados. Trabalhando desta maneira passamos a não ter medo das mudanças, pois em se tratando de software, elas irão ocorrer com muita frequência, e não tivermos um cenário suscetível a elas encontraremos sérios problemas para evoluir o software.

Sabemos que no mundo “real” nem sempre podemos ter um cenário ideal, que é com um baixo acoplamento e uma alta coesão. Mas como arquitetos, temos que saber tomar a decisão correta, pois determinadas decisões não poderão ser revertidas, dependendo da fase em que estiver o projeto ou o tipo de decisão tomada. Ter classes com responsabilidades claras e um baixo acoplamento, embora não seja fácil de serem construídas, nos traz benefícios como baixo impacto em uma possível manutenção, gerenciamento e mudança no negócio facilitados. Também não podemos esquecer que as aplicações evoluem, mudam e, muitas das vezes, se transformam. Se ignorarmos as melhores práticas na hora de desenhar uma arquitetura, poderemos ter sérios problemas. E para nos auxiliar, temos o princípio da responsabilidade única e a inversão de controle, cujo objetivo é obter-se um cenário de responsabilidades claras entre nossas classes e um baixo acoplamento.