

Capítulo 1

Qualidade do software

O principal objetivo da Engenharia de Software é contribuir para a produção de programas de qualidade. Esta qualidade, porém, não é uma idéia simples; mas sim como um conjunto de noções e fatores.

É desejável que os programas produzidos sejam rápidos, confiáveis, modulares, estruturados, modularizados, etc. Esses qualificadores descrevem dois tipos de qualidade:

De um lado, considera-se aspectos como eficiência, facilidade de uso, extensibilidade, etc. Estes elementos podem ser detectados pelos usuários do sistema. A esses fatores atribui-se a qualidade externa do software.

Por outro lado, existe um conjunto de fatores do programa que só profissionais de computação podem detectar. Por exemplo, legibilidade, modularidade, etc. A esses fatores atribui-se a qualidade interna do software.

Na realidade, qualidade externa do programa em questão é que importa quando de sua utilização. No entanto, os elementos de qualidade interna são a chave para a conquista da qualidade externa. Alguns fatores de qualidade externa são apresentados na próxima seção. A seção posterior trata da qualidade interna, analisando como a qualidade externa pode ser atingida a partir desta.

Fatores de qualidade externa

Corretude

É a condição do programa de produzir respostas adequadas e corretas cumprindo rigorosamente suas tarefas de acordo com sua especificação.

Obviamente, é uma qualidade primordial. Se um sistema não faz o que ele foi proposto a fazer, então qualquer outra questão torna-se irrelevante.

Robustez

É a capacidade do programa de funcionar mesmo sob condições anormais.

A robustez do programa diz respeito ao que acontece quando do aparecimento de situações anômalas. Isto é diferente de corretude, que define como o programa se comporta dentro de sua especificação. Na robustez, o programa deve saber encarar situações que não foram previstas sem efeitos colaterais catastróficos. Neste sentido, o termo confiabilidade é muito utilizado para robustez; porém denota um conceito mais amplo que é melhor interpretado como que englobando os conceitos de corretude e robustez.

Extensibilidade

É definida como a facilidade com que o programa pode ser adaptado para mudanças em sua especificação.

Neste aspecto, dois princípios são essenciais:

Simplicidade de design: uma arquitetura simples sempre é mais simples de ser adaptada ou modificada

Descentralização: quanto mais autônomos são os módulos de uma arquitetura, maior será a probabilidade de que uma alteração implicará na manutenção de um ou poucos módulos.

Capacidade de reuso

É a capacidade do programa de ser reutilizado, totalmente ou em partes, para novas aplicações.

A necessidade de reutilização vem da observação de que muitos elementos dos sistemas seguem padrões específicos. Neste sentido, é possível explorar este aspecto e evitar que a cada sistema produzido os programadores fiquem “reinventando soluções” de problemas já resolvidos anteriormente.

Compatibilidade

É a facilidade com que o programa pode ser combinado com outros.

Esta é uma qualidade importante pois os programas não são desenvolvidos *stand-alone*. Normalmente, existe uma interação entre diversos programas onde o resultado de um determinado sistema é utilizado como entrada para outro.

Outros aspectos

Os aspectos resumidos acima são aqueles que podem ser beneficiados com a boa utilização das técnicas de orientação por objetos. No entanto, existem outros aspectos que não podem ser esquecidos:

Eficiência: É o bom aproveitamento dos recursos computacionais como processadores, memória, dispositivos de comunicação, etc. Apesar de não explicitamente citado anteriormente, este é um requisito essencial para qualquer produto. A linguagem C++ em particular, por ser tão eficiente quanto C, permite o desenvolvimento de sistemas eficientes.

Portabilidade: É a facilidade com que um programa pode ser transferido de uma plataforma (*hardware*, sistema operacional, etc.). Este fator depende muito da base sobre a qual o sistema é desenvolvido. A nível de linguagem, C++ satisfaz este fator por ser uma linguagem padronizada com implementações nas mais diversas plataformas.

Facilidade de uso: É a facilidade de aprendizagem de como o programa funciona, sua operação, etc.

Modularidade - qualidade interna

Alguns fatores apresentados na seção anterior podem ser beneficiados com o uso de orientação por objetos. São eles: corretude, robustez, extensibilidade, reuso e compatibilidade. Estes refletem as mais sérias questões para o desenvolvimento de programas. A medida que se olha para estes cinco aspectos, dois subgrupos surgem:

Extensibilidade, reuso e compatibilidade demandam arquiteturas que sejam flexíveis, design descentralizado e a construção de módulos coerentes conectados por interfaces bem definidas.

Corretude e robustez são favorecidos por técnicas que suportam o desenvolvimento de sistemas baseados em uma especificação precisa de requisitos e limitações.

Da necessidade da construção de arquiteturas de programas que sejam flexíveis, surge a questão de tornar os programas mais modulares.

Uma definição precisa para modularidade é dificilmente encontrada. No entanto, esta técnica não traz benefícios se os módulos não forem autônomos, coerentes e organizados em uma estrutura robusta. Será utilizado no decorrer deste texto um conjunto de cinco critérios e cinco princípios.

Critérios para modularidade

Seguem abaixo cinco critérios que ajudam a avaliar/construir a modularidade do *design*:

Decomposição

O critério de decomposição modular é alcançado quando o modelo de design ajuda a decomposição do problema em diversos outros subproblemas cujas soluções podem ser atingidas separadamente.

O método deve ajudar a reduzir a aparente complexidade de problema inicial pela sua decomposição em um conjunto de problemas menores conectados por uma estrutura simples. De modo geral, o processo é repetitivo: os subproblemas são também divididos em problemas menores e assim sucessivamente.

Uma exemplificação deste tipo de *design* é o chamado método *top-down*. Este método dirige os desenvolvedores a começar com uma visão mais abstrata do funcionamento do sistema. Esta visão abstrata vai sendo refinada como um conjunto de passos sucessivos menores e assim por diante até que seus elementos estejam em um nível que permita sua implementação. Este processo pode ser modelado como uma árvore.

Composição

O método que satisfaz o critério de composição favorece a produção de elementos de programas que podem ser livremente combinados com os outros de forma a produzir novos sistemas, possivelmente em um ambiente bem diferente daquele em que cada um destes elementos foi criado.

Enquanto que a decomposição se concentra na divisão do software em elementos menores a partir da especificação, a composição se estabelece no sentido oposto: agregando elementos de programas que podem ser aplicados para construção de novos sistemas.

A composição é diretamente relacionada com a questão da reutilização: o objetivo é achar maneiras de desenvolver pedaços de programas que executam tarefas bem definidas e utilizáveis em outros contextos. Este contexto reflete um sonho antigo: transformar o processo de design de programas como elementos independentes onde programas são construídos pela combinação de elementos existentes.

Um exemplo deste tipo de abordagem é a construção de bibliotecas como conjuntos de elementos que podem ser utilizados em diversos programas (pacotes gráficos, bibliotecas numéricas, etc.).

Entendimento

Um método que satisfaz o critério de entendimento ajuda a produção de módulos que podem ser separadamente compreendidos pelos desenvolvedores; no pior caso, o leitor deve ter atenção sobre poucos módulos vizinhos. Este critério é especialmente relevante quando se tem em vista o aspecto da manutenção.

Um contra-exemplo deste tipo de método é quando há dependência sequencial onde um conjunto de módulos é elaborado de modo que a execução dos mesmos seja feita em uma ordem determinada. Desta forma, os módulos não são entendidos de forma individual mas em conjunto com seus vizinhos.

Continuidade

Um método de design satisfaz a continuidade se uma pequena mudança na especificação do problema resulta em alterações em um único ou poucos módulos. Tal alteração não tem reflexos na arquitetura geral do sistema; isto é, no relacionamento inter-modular.

Este critério reflete o problema de extensibilidade do sistema. A continuidade significa que eventuais mudanças devem afetar os módulos individualmente da estrutura do sistema e não a estrutura em si.

Um exemplo simples deste tipo de critério é a utilização de constantes representadas por nomes simbólicos definidos em um único local. Se o valor deve ser alterado, apenas a definição deve ser alterada.

Proteção

Um método de design satisfaz a proteção se este provê a arquitetura de isolamento quando da ocorrência de condições anômalas em tempo de execução. Ao aparecimento de situações anormais, seus efeitos ficam restritos àquele módulo ou pelo menos se propagar a poucos módulos vizinhos.

Os erros considerados neste critério são somente aqueles ocorridos em tempo de execução como falta de espaço em disco, falhas de hardware, etc. Não se considera, neste caso, a correção de erros, mas um aspecto importante para a modularidade: sua propagação.

Princípios de modularidade

Estabelecidos os critérios de modularidade, alguns princípios surgem e devem ser observados cuidadosamente para se obter modularidade. O primeiro princípio se relaciona com a notação e os outros se baseiam no modo de comunicação entre os módulos.

Seguem abaixo estes princípios:

Linguística modular

Este princípio expressa que o formalismo utilizado para expressar o design, programas, etc. deve suportar uma visão modular; isto é:

Módulos devem corresponder às unidades sintáticas da linguagem utilizada.

Onde a linguagem utilizada pode ser qualquer linguagem de programação, de design de sistemas, de especificação, etc.

Este princípio segue de diversos critérios mencionados anteriormente:

Decomposição: Se pretende-se dividir o desenvolvimento do sistema em tarefas separadas, cada uma destas deve resultar em uma unidade sintática bem delimitada. (compiláveis separadamente no caso de linguagens de programação)

Composição: Só pode-se combinar coisas como unidades bem delimitadas.

Proteção: Somente pode haver controle do efeito de erros se os módulos estão bem delimitados.

Poucas interfaces

Este princípio restringe o número de canais de comunicação entre os módulos na arquitetura do sistema. Isto quer dizer que:

Cada módulo deve se comunicar o mínimo possível com outros.

A comunicação pode ocorrer das mais diversas formas. Módulos podem chamar funções de outros, compartilhar estruturas de dados, etc. Este princípio limita o número destes tipos de conexão.

Pequenas interfaces

Este princípio relaciona o tamanho das interfaces e não suas quantidades. Isto quer dizer que:

Se dois módulos possuem canal de comunicação, estes devem trocar o mínimo de informação possível; isto é, os canais da comunicação inter-modular devem ser limitados.

Interfaces explícitas

Este é um princípio que vai mais adiante do que “poucas interfaces” e “pequenas interfaces”. Além de impor limitações no número de módulos que se comunicam e na quantidade de informações trocadas, há a imposição de que se explicita claramente esta comunicação. Isto é:

Quando da comunicação de dois módulos A e B, isto deve ser explícito no texto de A, B ou ambos.

Este princípio segue de diversos critérios mencionados anteriormente:

Decomposição e composição: Se um elemento é formado pela composição ou decomposição de outros, as conexões devem ser bem claras entre eles.

Entendimento: Como entender o funcionamento de um módulo A se seu comportamento é influenciado por outro módulo B de maneira não clara?

Ocultação de informação (information hiding)

A aplicação deste princípio assume que todo módulo é conhecido através de uma descrição oficial e explícita; ou melhor sua interface. Pela definição, isto requer que haja uma visão restrita do módulo.

Toda informação sobre um módulo deve ser oculta (privada) para outro a não ser que seja especificamente declarada pública.

A razão fundamental para este princípio é o critério de continuidade. Se um módulo precisa ser alterado, mas de maneira que só haja manutenção de sua parte não pública e não a interface; então os módulos que utilizam seus recursos (clientes) não precisam ser alterados. Além, quanto menor a interface, maiores as chances de que isto ocorra.

Apesar de não haver uma regra absoluta sobre o que deve ser mantido público ou não, a idéia geral é simples: a interface deve ser uma descrição do funcionamento do módulo. Tudo que for relacionado com sua implementação não deve ser público.

É importante ressaltar que, neste caso, este princípio não implica em proteção no sentido de restrições de segurança. Embora seja invisível, pode haver acesso às informações internas do módulo.

Calculadora RPN em C

Os primeiros exemplos serão feitos a partir de um programa escrito em C. Várias modificações serão feitas até que este se torne um programa C++. O programa é uma calculadora RPN (notação polonesa reversa), apresentada nesta seção.

Este tipo de calculadora utiliza uma pilha para armazenar os seus dados. Esta pilha está implementada em um módulo à parte. O header file está listado abaixo:

```
#ifndef stack_h
#define stack_h

#define MAX 50

struct Stack {
    int top;
    int elems[MAX];
};

void push(struct Stack* s, int i);
int pop(struct Stack* s);
int empty(struct Stack* s);
struct Stack* createStack(void);

#endif
```

A implementação destas funções está no arquivo stack-c.c:

```
#include <stdlib.h>
#include "stack-c.h"

void push(struct Stack*s, int i) { s->elems[s->top++] = i; }
int pop(struct Stack*s)          { return s->elems[--(s->top)]; }
int empty(struct Stack*s)        { return s->top == 0; }

struct Stack* createStack(void)
{
    struct Stack* s = (struct Stack*)malloc(sizeof(struct Stack));
    s->top = 0;
    return s;
}
```

A calculadora propriamente dita utiliza estes arquivos:

```
#include <stdlib.h>
#include <stdio.h>
#include "stack-c.h"

/* dada uma pilha, esta função põe nos
   parâmetros n1 e n2 os valores do topo
   da pilha. Caso a pilha tenha menos de dois
   valores na pilha, um erro é retornado */
int getop(struct Stack* s, int* n1, int* n2)
{
    if (empty(s))
    {
        printf("empty stack!\n");
        return 0;
    }
    *n2 = pop(s);
    if (empty(s))
    {
        push(s, *n2);
        printf("two operands needed!\n");
        return 0;
    }
    *n1 = pop(s);
    return 1;
}
```

```

/* a função main fica em um loop
   lendo do teclado os comandos da calculadora.
   Se for um número, apenas empilha.
   Se for um operador, faz o calculo.
   Se for o caracter 'q', termina a execução.
   Após cada passo a pilha é mostrada. */
int main(void)
{
    struct Stack* s = createStack();
    while (1)
    {
        char str[31];
        int i;
        printf("> ");
        gets(str);
        if (sscanf(str, "%d", &i)==1) push(s, i);
        else
        {
            int n1, n2;
            char c;
            sscanf(str, "%c", &c);
            switch(c)
            {
                case '+':
                    if (getop(s, &n1, &n2)) push(s, n1+n2);
                    break;
                case '-':
                    if (getop(s, &n1, &n2)) push(s, n1-n2);
                    break;
                case '/':
                    if (getop(s, &n1, &n2)) push(s, n1/n2);
                    break;
                case '*':
                    if (getop(s, &n1, &n2)) push(s, n1*n2);
                    break;
                case 'q':
                    return 0;
                default:
                    printf("error\n");
            }
        }
    }
    {
        int i;
        for (i=0; i<s->top; i++)
            printf("%d:%6d\n", i, s->elems[i]);
    }
}

```

Tipos abstratos de dados

O conceito de tipo foi um passo importante no sentido de atingir uma linguagem capaz de suportar programação estruturada. Porém, até agora, não é possível usar uma metodologia na qual os programas sejam desenvolvidos por meio de decomposições de problemas baseadas no reconhecimento de abstrações. Para a abstração de dados adequada a este propósito não basta meramente classificar os objetos quanto a suas estruturas de representação; em vez disso, os objetos devem ser classificados de acordo com o seu comportamento esperado. Esse comportamento é expressado em termos das operações que fazem sentido sobre esses dados; estas operações são o único meio de criar, modificar e se ter acesso aos objetos.

Classes em C++

Uma classe em C++ é o elemento básico sobre o qual toda orientação por objetos está apoiada. Em primeira instância, uma classe é uma extensão de uma estrutura, que passa a ter não apenas dados, mas também funções. A idéia é que tipos abstratos de dados não são definidos pela sua representação interna, e sim pelas operações sobre o tipo. Então não há nada mais natural do que incorporar estas operações no próprio tipo. Estas operações só fazem sentido quando associadas às suas representações.

No exemplo da calculadora, um candidato natural a se tornar uma classe é a pilha. Esta é uma estrutura bem definida; no seu header file estão tanto a sua representação (*struct Stack*) quanto as funções para a manipulação desta representação. Seguindo a idéia de classe, estas funções não deveriam ser globais, mas sim pertencerem à estrutura *Stack*. A função *empty* seria uma das que passariam para a estrutura. A implementação de *empty* pode ficar dentro da própria estrutura:

```
struct Stack {  
    // ...  
    int empty() { return top == 0; }  
};
```

ou fora:

```
struct Stack {  
    // ...  
    int empty();  
};  
  
int Stack::empty(void) { return top == 0; }
```

No segundo caso a declaração fica no header file e a implementação no .c. A diferença entre as duas opções será explicada na seção sobre funções inline.

Com esta declaração, as funções são chamadas diretamente sobre a variável que contém a representação:

```
void main(void)  
{  
    struct Stack s;  
    s.empty();  
}
```

Repare que a função deixou de ter como parâmetro uma pilha. Isto era necessário porque a função estava isolada da representação. Agora não, a função faz parte da estrutura. Automaticamente todos os campos da estrutura passam a ser visíveis dentro da implementação da função, sem necessidade se especificar de qual pilha o campo *top* deve ser testado (caso da função *empty*). Isto já foi dito na chamada da função.

O paradigma da orientação a objetos

Esta seção apresenta cinco componentes chave do paradigma de orientação por objetos. As componentes são objeto, mensagem, classe, instância e método. Eis as definições:

Objeto: é uma abstração encapsulada que tem um estado interno dado por uma lista de atributos cujos valores são únicos para o objeto. O objeto também conhece uma lista de mensagens que ele pode responder e sabe como responder cada uma. Encapsulamento e abstração são definidos mais à frente.

Mensagem: é representada por um identificador que implica em uma ação a ser tomada pelo objeto que a recebe. Mensagens podem ser simples ou podem incluir parâmetros que afetam como o objeto vai responder à mensagem. A resposta também é influenciada pelo estado interno do objeto.

Classe: é um modelo para a criação de um objeto. Inclui em sua descrição um nome para o tipo de objeto, uma lista de atributos (e seus tipos) e uma lista de mensagens com os métodos correspondentes que o objeto desta classe sabe responder.

Instância: é um objeto que tem suas propriedades definidas na descrição da classe. As propriedades que são únicas para as instâncias são os valores dos atributos.

Método: é uma lista de instruções que define como um objeto responde a uma mensagem em particular. Um método tipicamente consiste de expressões que enviam mais mensagens para objetos. Toda mensagem em uma classe tem um método correspondente.

Traduzindo estas definições para o C++, classes são estruturas, objetos são variáveis do tipo de alguma classe (instância de alguma classe), métodos são funções de classes e enviar uma mensagem para um objeto é chamar um método de um objeto.

Resumindo:

Objetos são instâncias de classes que respondem a mensagens de acordo com os métodos e atributos, descritos na classe.

Exercício 1 - Calculadora RPN com classes

Transformar a pilha utilizada pela calculadora em uma classe.

