

Técnicas de Programação

Documentação de Código – Código Auto-Explicativo

Profa. Elaine Venson

elainevenson@unb.br

Conceitos

- Documentação externa x documentação no nível de código
- Nível de código
 - O código como documentação
 - Comentários no código

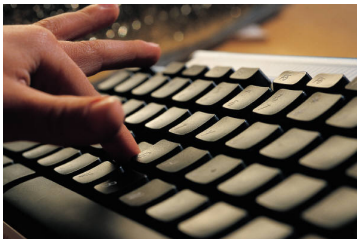
O Código como documentação

- Um bom **estilo de programação** é a forma mais eficiente de documentação em nível de código
 - Boa estrutura
 - Abordagens diretas e compreensíveis
 - Nomes de variáveis e rotinas significativos
 - Uso de constantes em vez de literais
 - Layout limpo
 - Minimização da complexidade das estruturas de dados e fluxos de controle

Frase

“Don’t document bad code - rewrite it”

(Kernighan e Plauger, 1978)



Motivações

- O código é criado para **comunicar** um conjunto claro de instruções:
 - Para o computador
 - Para as pessoas que irão corrigir ou evoluir o código
- O produto de software tem uma vida útil, ao longo da qual pode sofrer **constantes mudanças**
- Código tende a ser **mais difícil de ler** do que de escrever
- Um código claro tem mais **qualidade**, erros são menos prováveis e é mais barato para manter

Exemplo 1

- O que este código faz?

```
int fval(int i)
{
    int ret=2;
    for (int n1=1, n2=1, i2=i-3; i2>=0; --i2)
    {
        n1=n2; n2=ret; ret=n2+n1;
    }
    return (i<2) ? 1 : ret;
}
```

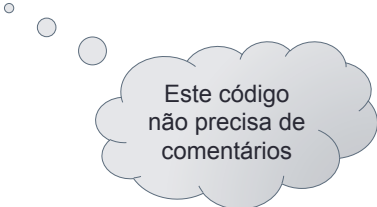
Exemplo 1 (cont)

- Versão do código auto-explicativa:

```
int fibonacci(int position)
{
    if (position < 2)
    {
        return 1;
    }
    int previousButOne = 1;
    int previous = 1;
    int answer = 2;
```

Exemplo 1 (cont)

```
for (int n = 2; n < position; ++n)
{
    previousButOne = previous;
    previous = answer;
    answer = previous + previousButOne;
}
return answer;
}
```



Este código
não precisa de
comentários

Técnicas para código auto-explicativo

- Código simples com boa apresentação
- Escolher nomes significativos
- Decomposição em funções atômicas
- Tipos descritivos
- Utilização de constantes
- Ênfase para código importante
- Agrupamento de informações relacionadas
- Utilização de Cabeçalho de Arquivo

Técnicas para código auto-explicativo

- Tratamento de erros apropriado
- Utilização de comentários efetivos

Código simples com boa apresentação

- O caminho “normal” deve ser óbvio no código
 - Fluxos de erro não devem confundir o fluxo normal
 - If-then-else na ordem natural
- Evitar muitas estruturas aninhadas
 - Equilibrar com **returns** no meio do código
 - SESE code (*Single Entry x Single Exit*)
- Cautela com otimização de código
 - Otimizar apenas quando for um gargalo para o programa
 - Incluir comentários sobre o caso

Escolher nomes significativos

- Definir nomes significativos para todas as **variáveis**, **tipos**, **arquivos** e **funções**
- O nome deve descrever o que ele **representa**
- Uma variável deve ser utilizada **apenas para o propósito** estabelecido pelo seu nome
- Bons nomes minimizam necessidade de **comentários** no código
- É o que mais aproxima o código da **linguagem natural**

Decomposição em funções atômicas

- Regra básica: **uma função, uma ação**
- Evitar **efeitos colaterais** de funções
 - Se for imprescindível, incluir comentários
- Manter as funções **curtas**
- **Quebrar** algoritmos complexos em pequenas funções com nomes descritivos

Tipos descritivos

- Utilizar recursos das linguagens para melhor descrever os tipos:
 - Para valores que nunca mudam, declarar como constante (**const** em C/C++)
 - Para valores que não podem ser negativos, declarar como **unsigned**
 - Usar **enumerações** para descrever conjunto de valores relacionados ou códigos

Utilização de constantes

- Evitar números mágicos (literais), utilizar constantes

```
const int efetuar_deposito = 7;
```

```
...
```

```
if(opcao == efetuar_deposito)
```

```
{
```

```
    // realizar depósito
```

```
}
```

Ênfase para código importante

- O que for mais importante deve se destacar no código
- Exemplos:
 - **Ordenar declarações** nas classes: declarações públicas devem vir antes das privadas
 - Quando possível **esconder** informações não essenciais
 - **Não esconder** código importante
 - Escrever uma instrução por linha
 - É possível escrever loops for inteligentes colocando toda lógica em uma única linha, mas fica difícil de ler

Agrupamento de informações relacionadas

- Procurar manter informações relacionadas em um **único local**
- A API de um **componente** deve ficar em um único arquivo
- Se for informação demais para um único arquivo, **questionar** o projeto (design)
- Utilizar recursos das linguagens para deixar os **agrupamentos explícitos**:
 - Packages em JAVA, Namespace em C++, Enumeradores

Utilização de cabeçalho de arquivo

- Incluir um **bloco de comentários** no início de cada arquivo para descrever seu **conteúdo e projeto** a que pertence
- **Padrão** na maioria das empresas
- Exemplo:

```
/*  
 * File: Foo.java  
 * Purpose: Foo class implementation  
 * Notice: (c) 1066 Foo industries. All rights reserved.  
 */
```

Tratamento de erros apropriado

- Tratar erros nos **níveis adequados**:
 - Problemas de I/O de disco devem ser tratados nos pontos em que o código acessa o disco
- Em cada nível do programa os erros devem ter uma descrição precisa de acordo com o **contexto**
- Não apresentar erros que não tenham sentido na interface com o usuário
- Em código auto-explicativo, erros tratados nos níveis apropriados ajudam o leitor a entender a **origem** do erro, o que **significa** e qual seu **impacto** naquele ponto

Utilização de comentários efetivos

- Após aplicar todas as técnicas anteriores, comentários no código ainda podem ser **necessários**
- Qual a quantidade necessária?
 - Apenas adicionar comentários se a **clareza** do código não pode ser melhorada de outra forma
- Analisar o código:
 - O comentário pode ser evitado com alguma mudança de nome de variável ou criação de uma subrotina?

Ferramentas de Documentação

- Geração de documentação a partir do **código fonte** com base em blocos de comentários com **formato específico**
- Tornaram-se mais populares desde que a Sun lançou o **Javadoc**
- Toda a documentação de **API** do Java é gerada pelo Javadoc

Javadoc – estrutura da documentação

```
/**
 * Esta é a documentação da classe Widget.
 * A ferramenta sabe isso porque o comentário
 * começou com os caracteres especiais '/**'.
 * @author Nome do autor aqui
 * @version Número da versão aqui
 */
class Widget
{
    public:
        /**
         * Esta é a documentação para um método.
         */
        void method();
};
```

Ferramentas de documentação

- A ferramenta de documentação interpreta cada arquivo fonte, extrai a documentação, constrói uma base com referências cruzadas de tudo o que encontrar e gera uma documentação formatada como saída
- É possível documentar:
 - Informações de copyright
 - Data de criação
 - Informação de referências cruzadas
 - Marcar código obsoleto como “deprecated
 - Apresentar descrição para cada parâmetro de função

Ferramentas de documentação

- Existem várias ferramentas disponíveis, além do Javadoc
- Exemplos: NDoc para C# e Doxygen
- Benefícios:
 - Encoraja escrever documentação e mantê-la atualizada
 - Não exige etapa adicional para gerar código compilável
 - Baixa curva de aprendizado
 - As ferramentas oferecem recursos de busca e referência cruzada e

Ferramentas de documentação

- Consequências:

- É útil para documentação de APIs (não para código interno)
- Os comentários podem ficar imersos no código e difíceis de ler para ter uma visão geral

- Boas práticas:

- Para itens públicos escrever uma ou duas sentenças
- Documentar variáveis e parâmetros se não forem auto-explicativos
- Não incluir comentários para tudo, apenas para o que for necessário
- Documentar parâmetros de entrada/saída
- Precondições, pós-condições, exceções, efeitos colaterais

Exercício

1) Escrever um código melhorado e auto-explicativo para a função bubblesort:

```
void bsrt(int a[], int n)
{
    for (int i = 0; i < n-1; i++)
        for (int j = n-1; j > i; j--)
            if (a[j-1] > a[j])
            {
                int tmp = a[j-1];
                a[j-1] = a[j];
                a[j] = tmp;
            }
```

Exercício

- Análise do código:
 - Nome da função não está claro
 - Nome dos parâmetros não é significativo
 - Nomes das variáveis não é significativo
 - O Código que troca os elementos ficaria mais claro em uma função separada

Resultado possível (em C)

```
void swap(int *first, int *second)
{
    int temp = *first;
    *first = *second;
    *second = temp;
}

void bubblesort(int items[], int size)
{
    for (int pos1 = 0; pos1 < size-1; pos1++)
        for (int pos2 = size-1; pos2 > pos1; pos2--)
            if (items[pos2-1] > items[pos2])
                swap(&items[pos2-1], &items[pos2]);
}
```