

UFMG  
UNIVERSIDADE FEDERAL  
DE MINAS GERAIS

## Programação e Desenvolvimento de Software 2

Boas práticas de desenvolvimento

Prof. Douglas G. Macharet  
douglas.macharet@dcc.ufmg.br

DCC  
DEPARTAMENTO DE  
CIÊNCIA DA COMPUTAÇÃO

## Introdução

- Código afeta diversas etapas
  - Implementação, teste, manutenção, evolução, ...
- Fácil de entender, não de escrever!
  - Várias pessoas vão usar o código (até você!)
- Boas práticas de programação
  - Regras gerais para melhorar a qualidade
- Escrever código que funciona é diferente de escrever um bom código!

DCC UFMG

PDS 2 - Boas práticas de desenvolvimento 2

## Introdução

- Características de um bom código
  - Compreensível
  - Confiável
  - Eficiente
  - Extensível
- Se o código não é confiável/fácil de manter, você pode gastar muito tempo e recurso

DCC UFMG

PDS 2 - Boas práticas de desenvolvimento 3

## Introdução

- Dificuldades de se ter um código limpo
  - Prazos apertados
  - Cronograma extenso

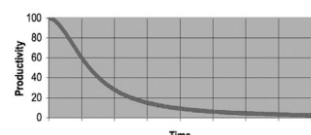


Figure 1-1  
Productivity vs. time

DCC UFMG

PDS 2 - Boas práticas de desenvolvimento 4

## Introdução

“One difference between a smart programmer and a professional programmer is that the professional understands that clarity is king. Professionals use their powers for good and write code that others can understand.”

— Robert C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship

DCC UFMG

PDS 2 - Boas práticas de desenvolvimento 5

## Regras gerais

- Mais tempo lendo código que escrevendo
- Utilização de certas regras
  - Melhora a legibilidade, facilita compreensão
- Nomes, organização, declarações, ...
- Gerais
  - Modele antes de implementar (CRC, UML)
  - Se possível, utilize Padrões de Projeto

DCC UFMG

PDS 2 - Boas práticas de desenvolvimento 6

## Regras gerais

### The Boy Scout Rule

The Boy Scouts of America have a simple rule that we can apply to our profession.

*Leave the campground cleaner than you found it.*

- Escrever certo e manter certo!

DCC 111

PDS 2 - Boas práticas de desenvolvimento

7

## Regras gerais

- Evitar o aninhamento de vários “ifs”
  - Reduzir a complexidade ciclomática
  - Por que?
- Definir tipos específicos para o problema
  - Tornam os programas mais legíveis
- Tratar as situações excepcionais
  - Robustez
  - Tolerância à falhas

DCC 111

PDS 2 - Boas práticas de desenvolvimento

8

## Regras gerais

### Complexidade ciclomática

```
bool exemplo(ifstream& arq, string arq_cam) {
    string linha;
    if (valido(arq_cam)) {
        if (arq.is_open()) {
            if (getline(arq, linha)) {
                if (linha.find("importante")) {
                    return true;
                } else {
                    return false;
                }
            } else {
                return false;
            }
        } else {
            return false;
        }
    } else {
        return false;
    }
}
```

DCC 111

PDS 2 - Boas práticas de desenvolvimento

9

## Regras gerais

### Complexidade ciclomática

```
bool exemplo(ifstream& arq, string arq_cam) {
    string linha;
    if (!valido(arq_cam))
        return false;

    if (!arq.is_open())
        return false;

    bool encontrou = false;
    if (getline(arq, linha)) {
        if (linha.find("importante")) {
            encontrou = true;
        }
    }

    return encontrou;
}
```

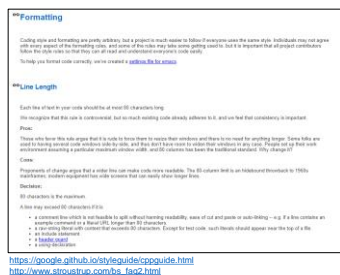
DCC 111

PDS 2 - Boas práticas de desenvolvimento

10

## Regras gerais

### Convenções



DCC 111

PDS 2 - Boas práticas de desenvolvimento

11

## Indentação

- Realça a estrutura lógica
- Torna o código mais legível
- Escolha um padrão e use
  - Configure seu editor cedo para isto
  - Escolha entre Tabs e Espaço
  - Indente com 2 ou 4 espaços
  - Se usar Tab configure seu editor para que uma tab apareça como 2 ou 4 espaços

DCC 111

PDS 2 - Boas práticas de desenvolvimento

12

## Nomes

- Structs/Classes
  - Substantivos no singular
- Atributos
  - Substantivos no singular (ou plural se coleção)
- Métodos
  - Primeira palavra é um verbo no infinitivo

## Nomes

- Structs/Classes
  - Cliente, Esporte, Animal, ...
- Atributos
  - nome, numJogadores, especie, ...
- Métodos
  - enviarPagamento, fazerJogada, comer, ...

## Nomes

- Utilize nomes significativos/pronunciáveis

```
struct DtaRord {
    time_t cridmahms;
    time_t moddmahms;
    int pszqint = 102;
};
```



```
struct Cliente {
    time_t dataHoraCriacao;
    time_t dataHoraModificacao;
    int idRegistro = 102;
};
```



## Nomes

- Underline
  - `int num_clientes;`
  - `struct list *lista_alunos;`
- CamelCase
  - `int numClientes;`
  - `struct lista *listaAlunos;`
- Escolha um padrão e atenha-se a ele!

## Atributos

- Utilizar constantes quando necessário
  - Evitar “números mágicos” e Strings repetidas
  - Devem ter nomes significativos e não seu valor
  - Nomes em letras maiúsculas / precedidos por *k*
- Exemplo
  - Utilizar `COR_DA_FONTE` e não `AMARELO`
  - `kDiasDaSemana`

## Métodos

- Faça métodos pequenos
  - Aproximadamente 20 linhas
  - Facilita a leitura e alterações futuras
- Devem ser utilizados poucos argumentos
- Cada método possui uma funcionalidade
  - Muitos níveis → muitas responsabilidades
  - Extraia trechos em métodos privados

## Métodos

- Código deve ser lido de cima para baixo
  - Da mesma forma como é lido uma narrativa
  - Sujeitos, verbos e predicados
  - Frases em ordem coerente
- Todas as funções devem ser seguidas pelas que compõe o próximo nível de abstração

## Métodos

### Exemplo

- O que o código abaixo faz?

```
list<vector<int>> pegarValores(list<vector<int>> lista1) {
    list<vector<int>> lista2;
    for (vector<int> x : lista1)
        if (x[0] == 4)
            lista2.push_back(x);
    return lista2;
}
```

## Métodos

### Exemplo

- Por que é difícil dizer o que o código faz?
  - Que tipos de coisas estão em `lista1`?
  - Qual a importância do item zero em `lista1`?
  - Qual a importância do valor 4?
  - Como a lista retornada pode/dever ser usada?

## Métodos

### Exemplo

- Contexto
  - Use nomes que revelam seu propósito
- Campo Minado
  - Tabuleiro → Lista de células (`lista1`)
  - Cada posição (célula) armazena um vetor
    - Status, Bandeira, Bomba, Valor, Vazio, ...
  - Posição 0
    - Valor 4: célula marcada com bandeira
- Como melhorar o código anterior?

## Métodos

### Exemplo

```
const int kPosicaoStatus = 0;
const int kValorBandeira = 4;

list<vector<int>> pegarCelulasMarcadas(list<vector<int>> tabuleiro) {
    list<vector<int>> celulasMarcadas;
    for (vector<int> celula : tabuleiro)
        if (celula[kPosicaoStatus] == kValorBandeira)
            celulasMarcadas.push_back(celula);
    return celulasMarcadas;
}
```

- Ainda é possível melhorar?
  - Célula → TAD

## Métodos

### Exemplo

Valor ou Referência?

```
list<Celula> pegarCelulasMarcadas(list<Celula> tabuleiro) {
    list<Celula> celulasMarcadas;
    for (Celula celula : tabuleiro)
        if (celula.estaMarcada())
            celulasMarcadas.push_back(celula);
    return celulasMarcadas;
}
```

## Métodos

### Exemplo

```
list<Celula> pegarCelulasMarcadas(list<Celula> const& tabuleiro) {
    list<Celula> celulasMarcadas;
    for (Celula celula : tabuleiro)
        if (celula.estaMarcada())
            celulasMarcadas.push_back(celula);
    return celulasMarcadas;
}
```

## Métodos

### Passagem de parâmetros

- Use referências (&) sempre que possível, e ponteiros (\*) quando for necessário.
  - Não podem ser NULL, logo, mais seguras
  - Não podem ser *re-assigned* (inicialização)
- **const**
  - Não permite que a variável referenciada seja alterada através da referência na função
    - Sempre utilize se esse é o comportamento esperado
  - Ajuda o compilador e no entendimento (debug)

<https://www.learncpp.com/cpp-tutorial/73-passing-arguments-by-reference/>

## Modularização

- Particionar um programas em *módulos*
- O módulo geralmente é um par de arquivos
  - `modulo.hpp`
    - Contém a declaração das funções e tipos de dados
    - Parte conhecida/importada por outros módulos
  - `modulo.cpp`
    - Contém a implementação das funções

## Modularização

- Organizar os módulos de forma a permitir um melhor reaproveitamento
- Nomear de acordo com as funcionalidades
- Programador não precisa conhecer detalhes da implementação do módulo
  - Usuário do TAD / Programador do TAD

## Comentários

- Às vezes não são vistos com bons olhos
  - Fracasso em se expressar apenas no código
  - Mal necessário, não deve ser comemorado
- Por que essa resistência?
  - Nem sempre condizem com a realidade
  - Códigos mudam e evoluem, e o comentário?
  - Um mau comentário mais atrapalha que ajuda

## Comentários

- Sempre que possível e necessário
- Devem
  - Ser informativos sobre o funcionamento
  - Alertar sobre possíveis consequências
- Não devem
  - Ser redundantes
  - Dizer algo que devia estar claro pelo código

## Comentários

- No início de cada módulo
- Descrever constantes e variáveis globais
- Funções
  - Descrever os parâmetros e retorno
  - Explicar o que a função faz, não como ela faz
- Revisar comentários quando o código mudar

## Controle de versão

- Mantém um registro das modificações
- Permite
  - Desenvolvimento colaborativo
  - Saber quem fez as mudanças e quando
  - Reverter mudanças, voltar a um estado anterior
- Soluções livres
  - Git, Mercurial (HG), SVN, CVS

## Controle de versão

### Git

- Sistema de controle de versões distribuído
- Desenvolvimento do kernel do Linux
  - Linus Torvalds, 2005
- Github
  - Serviço para armazenar repositórios
  - Se você é estudante não precisa pagar
    - <https://education.github.com/pack>

## Controle de versão

### Git – Snapshots

- Armazena dados (código) como snapshots do projeto ao longo do tempo
- Você decide quando fazer um snapshot, e de quais arquivos
- Poder voltar para visitar qualquer snapshot
- Quase todas operações são locais

## Controle de versão

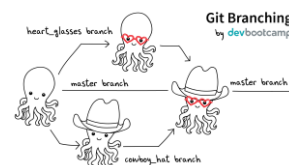
### Git

- Commit
  - Registro de quais arquivos você alterou desde a última vez que você fez um commit (criação de um novo snapshot)
- Repositório
  - Coleção de arquivos e o histórico dos mesmos
  - Máquina local ou servidor remoto (github)
  - O ato de copiar um repositório de um servidor remoto é chamado cloning (clone)
  - O ato de fazer o download de commits que não existem na sua máquina é chamado de pulling (pull)
  - O processo de adicionar as suas mudanças locais no repositório remoto é chamado de pushing (push)

## Controle de versão

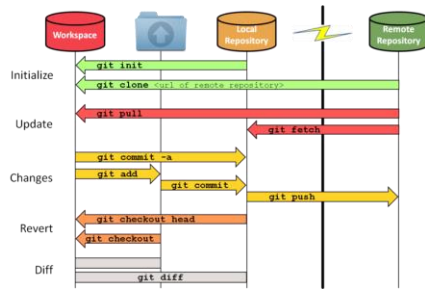
### Git

- Branch
  - Maneira de trabalhar em diferentes versões de um repositório de uma só vez



## Controle de versão

### Git



DCC

Programação Modular - Boas práticas de programação

37

## Controle de versão

### Git – Exemplo

#### ■ Clonando um repositório

- `git clone http://github.com:/pds2-dec-ufmg/repositorio-exemplo`

#### ■ Submetendo alterações

- `git add *`
- `git commit -m "Mensagem"`
- `git push origin master`

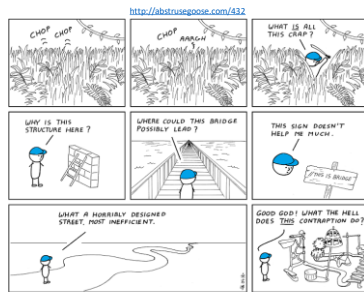
[https://rogerdudler.github.io/git-guide/index.pt\\_BR.html](https://rogerdudler.github.io/git-guide/index.pt_BR.html)  
<https://help.github.com/en/articles/cloning-a-repository>  
<https://help.github.com/en/articles/pushing-to-a-remote>

DCC

Programação Modular - Boas práticas de programação

38

## Considerações finais



DCC

PDS 2 - Boas práticas de desenvolvimento

39

## Considerações finais



DCC

PDS 2 - Boas práticas de desenvolvimento

40