

Programação e Desenvolvimento de Software 2

Boas práticas de desenvolvimento

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

Introdução

- Código afeta diversas etapas
 - Implementação, teste, manutenção, evolução, ...
- Várias pessoas vão usar o código (até você!)
 - Fácil de entender, não de escrever!

“Any fool can write code that a
computer can understand.
Good programmers write code
that *humans* can understand.”

— Martin Fowler



Introdução

- Boas práticas de programação
 - Regras gerais para melhorar a qualidade

Escrever um código que funciona é diferente de escrever um bom código!

Introdução

- Fácil de entender
- Fácil de manter e modificar
- Fácil de depurar
- Reusabilidade / Robustez

**Código
BOM**



**Código
RUIM**

- Confuso de entender
- Difícil de manter e modificar
- Difícil de depurar
- Não reusável / Frágil



Introdução

- Dificuldades de se ter um código limpo
 - Prazos apertados
 - Cronograma extenso
- Problemas vão se acumulando
 - É só um Ctrl+C/Ctrl+V
 - Amanhã resolvo isso
 - Por que eu fiz isso ontem? 🤔

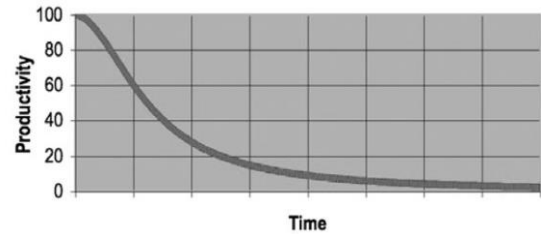


Figure 1-1
Productivity vs. time

BROKEN WINDOWS THEORY



“visible signs of crime, anti-social behavior, and civil disorder create an urban environment that encourages further crime and disorder, including serious crimes.”

Onde podemos ter “janelas quebradas” no código?

THE BOY SCOUT RULE

Leave the campground cleaner than you found it.



Atenção aos detalhes!

Não basta apenas escrever bem o código inicial. Mas ele deve ser **mantido** certo (limpo) ao longo do tempo!


Regras gerais

- Pense no agora e no futuro
 - Modele antes de implementar (CRC, UML)
 - Se possível, utilize Padrões de Projeto
- Defina tipos específicos para o problema
- Considere possíveis situações excepcionais
- Evite o aninhamento de vários “ifs”
 - Reduzir a complexidade ciclomática
 - Por que? → Difícil de entender, testar, modificar, ...

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;
    }
}
```




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

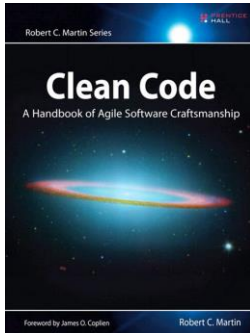
    if (!arq.is_open())
        return false;

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

    return encontrou;
}
```



Regras gerais



“You are reading this book for two reasons. First, you are a programmer. Second, you want to be a better programmer. Good. We need better programmers.”

Nomes

- Nomes estão em todos os lugares!
- Underline
 - `int num_clientes;`
 - `struct list *lista_alunos;`
- CamelCase
 - `int numClientes;`
 - `struct lista *listaAlunos;`
- Escolha um padrão e atenha-se a ele!



Nomes

- Structs/Classes (TADs)
 - Substantivos no singular
- Atributos
 - Substantivos no singular
 - Plural se for uma coleção
- Métodos
 - Ação
 - Verbo no infinitivo

Cliente, Esporte, Aluno, ...

nome, numJogadores, curso, alunos, ...

enviarPagamento, fazerJogada, calcularRSG, ...

Nomes

- Escolha nomes
 - Significativos/pronunciáveis → Comunicação
 - Fáceis de serem pesquisados/encontrados

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



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



Nomes

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

Formatação

- Formatação de código → Comunicação

Vertical	Horizontal
<ul style="list-style-type: none"> • The Newspaper Metaphor <ul style="list-style-type: none"> • Leitura top-down / Geral → Detalhes • Abertura e Densidade <ul style="list-style-type: none"> • Separação de conceitos diferentes • Proximidade implica associação • Distância <ul style="list-style-type: none"> • Variáveis declaradas próximas ao uso • Instâncias declaradas no topo 	<ul style="list-style-type: none"> • Esforçar para manter linhas curtas <ul style="list-style-type: none"> • 80/100/120 caracteres • Nunca fazer scroll para a direita • Abertura e Densidade <ul style="list-style-type: none"> • Espaço em branco para associar coisas • Alinhamento <ul style="list-style-type: none"> • Indentação • Hierarquia

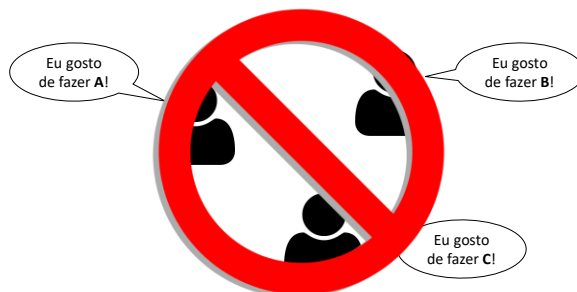
Formatação

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
 - Configure o editor para que tab apareça como espaços
 - Indente com 2 ou 4 espaços

Formatação

- Utilize regras e convenções comuns (não invente!)
 - Nomes, formatação, organização, ...
 - Melhora a legibilidade, facilita compreensão geral



Formatação

Guias de estilo

Formatting

Coding style and formatting are pretty arbitrary, but a project is much easier to follow if everyone uses the same style. Individuals may not agree with every aspect of the formatting rules, and some of the rules may take some getting used to, but it is important that all project contributors follow the style rules so that they can all read and understand everyone's code easily.

To help you format code correctly, we've created a [settings file for emacs](#).

Line Length

Each line of text in your code should be at most 80 characters long.

We recognize that this rule is controversial, but so much existing code already adheres to it, and we feel that consistency is important.

Pros:

Those who favor this rule argue that it is rude to force them to resize their windows and there is no need for anything longer. Some folks are used to having several code windows side-by-side, and thus don't have room to widen their windows in any case. People set up their work environment assuming a particular maximum window width, and 80 columns has been the traditional standard. Why change it?

Cons:

Proponents of change argue that a wider line can make code more readable. The 80-column limit is an hidebound throwback to 1960s mainframes; modern equipment has wide screens that can easily show longer lines.

Decision:

80 characters is the maximum.

A line may exceed 80 characters if it is

- a comment line which is not feasible to split without harming readability, ease of cut and paste or auto-linking – e.g. if a line contains an example command or a literal URL longer than 80 characters.
- a raw string literal with content that exceeds 80 characters. Except for test code, such literals should appear near the top of a file.
- an include statement.
- a header guard.
- a using declaration.

<https://google.github.io/styleguide/cppguide.html>
http://www.stroustrup.com/bs_faq2.html

Formatação

Guias de estilo

Conditionals

The `if` and `else` keywords belong on separate lines. There should be a space between the `if` and the open parenthesis, and between the close parenthesis and the curly brace (if any), but no space between the parentheses and the condition.

```
if (condition) { // no spaces inside parentheses
    ... // 2 space indent.
} else if (...) { // The else goes on the same line as the closing brace.
    ...
} else {
    ...
}
```

```
if(condition) { // Bad - space missing after if.
if ( condition ) { // Bad - space between the parentheses and the condition
if (condition){ // Bad - space missing before {.
if(condition){ // Doubly bad.
```

Mau uso dos espaços horizontais!

```
if (condition) { // Good - proper space after IF and before {.
```

Short conditional statements may be written on one line if this enhances readability. You may use this only when the line is brief and the statement does not use the `else` clause.

```
if (x == kFoo) return new Foo();
if (x == kBar) return new Bar();
```

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 funções que fazem parte do próximo nível de abstração (detalhes)
 - Informações importantes primeiro (newspaper)
 - Métodos dependentes em sequência

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?

Legibilidade != Complexidade

- Perguntas sem respostas
 - 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 (domínio)
- 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;
}
```

O que isso
representa?

- 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

- Referências (&) sempre que possível
 - Não podem ser NULL, logo, mais seguras
 - Não podem ser *re-assigned* (inicialização)
- Ponteiros (*) apenas quando for necessário!
- **const**
 - Variável referenciada não pode ser alterada pela referência
 - Sempre utilize se esse é o comportamento esperado
 - Ajuda o compilador e o programador (entendimento, debug)

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



PDS 2 - Boas práticas de desenvolvimento

27

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!



PDS 2 - Boas práticas de desenvolvimento

28

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 (informações legais, visão geral)
- Descrever constantes e variáveis globais
- “To Do” → Coisas a serem resolvidas no futuro
- Funções
 - Descrever os parâmetros e retorno
 - Explicar o que a função faz, não como ela faz

Sempre revisar os comentários quando o código mudar!

Comentários

```
// Check to see if the employee is eligible for full benefits
if ((employee.flags && HOURLY_FLAG) && (employee.age > 65))
```

Gaste alguns segundos para explicar a maior parte de sua intenção no código. Crie funções que fazem a mesma coisa que o comentário iria descrever.

```
if (employee.isEligibleForFullBenefits())
```

Veja mais exemplos no livro.



PDS 2 - Boas práticas de desenvolvimento

31

Classes (Structs)

- Devem ser pequenas
 - Métodos → Linhas, Classes → Responsabilidades
- Single Responsibility Principle
 - Classe deve ter apenas uma responsabilidade
 - Apenas um “motivo” para mudar
- Organização

```
// public static constants
// private static variables
// private instance variables
// public functions
// private methods called by public function
```



PDS 2 - Boas práticas de desenvolvimento

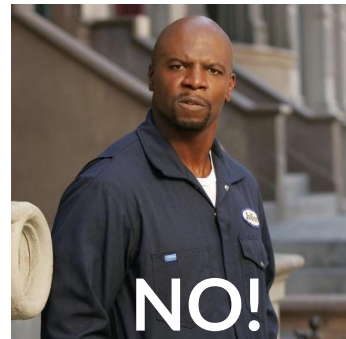
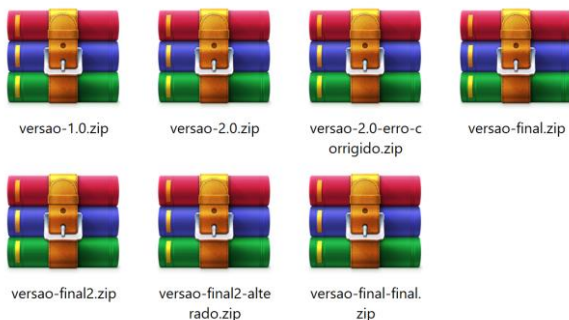
32

Você já...

- Fez uma mudança no código e quis voltar?
 - Perdeu o código certo, backup velho demais
- Quis ter múltiplas versões do mesmo código?
 - Desenvolvimento em paralelo
 - Ver a diferença entre duas (ou mais) versões
- Gostaria de identificar se uma mudança em particular quebrou ou consertou seu código?



Você já...



Controle de versão

Histórico

Colaboração

Variações



- Quem fez o que, quando e onde
- Trabalho simultâneo no mesmo código
- Linhas de evolução em paralelo

Controle de versão

Git

- Sistema de controle de versões distribuído
- Desenvolvimento do kernel do Linux
 - Linus Torvalds, 2005
- Github
 - Serviço para armazenar repositórios
 - Estudantes ganham alguns extras
 - <https://education.github.com/pack>



Controle de versão

Git – Snapshots

- Armazena dados (código) como snapshots
 - Evolução do projeto ao longo do tempo
- Você decide quando realizar um snapshot e os arquivos
 - Quais devem fazer parte “da foto” (estado)
- Pode voltar para verificar qualquer snapshot
- Quase todas as operações são locais
 - Eventualmente sincronizado com o servidor (local/remoto)



<https://git-scm.com/book/pt-br/v1/Primeiros-passos-No%C3%A7%C3%B5es-B%C3%A1sicas-de-Git>



PDS 2 - Boas práticas de desenvolvimento

37

Controle de versão

Git

- Repositório
 - Coleção de arquivos e o histórico dos mesmos
 - Máquina local ou servidor remoto (github)

Commit

Envia os arquivos selecionados (**add**) e que você alterou desde a última vez que fez um commit (criação de um snapshot / local).

Clone

Ato de copiar inicialmente um repositório de um servidor remoto para o local.

Pull

Ato de fazer o download de commits que não existem na sua máquina (local).

Push

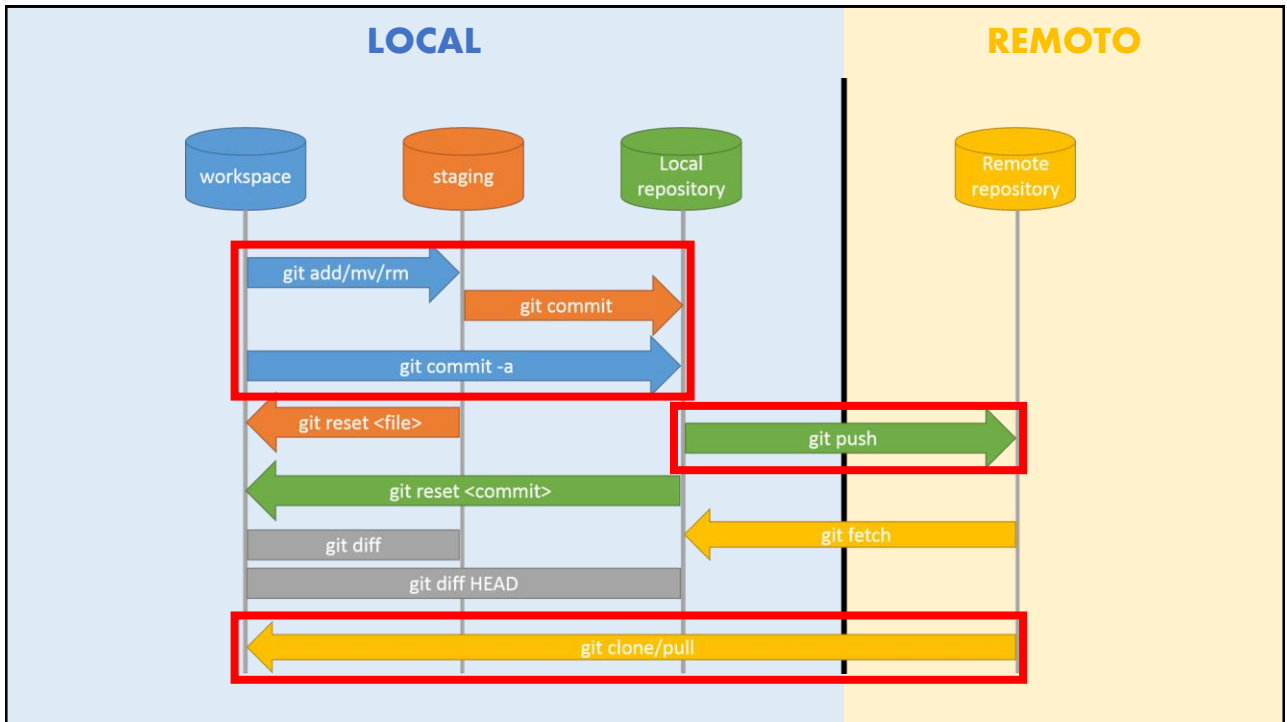
Processo de adicionar as suas mudanças locais no repositório remoto.

<https://guides.github.com/>



PDS 2 - Boas práticas de desenvolvimento

38



Controle de versão

Git – Exemplo

▪ Clonando um repositório

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

▪ Submetendo alterações

- `git add *`
- `git commit -m "Mensagem"`
- `git push <remote> <branch>`

- **Todos arquivos**
- **Local**
- **Permissão!**

origin

master

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 111

PDS 2 - Boas práticas de desenvolvimento

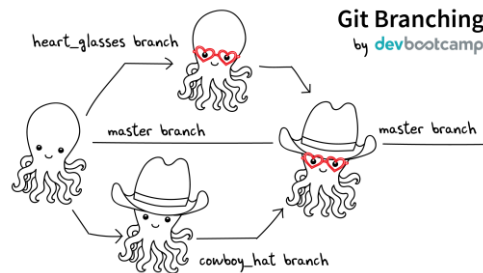
40

Controle de versão

Git

▪ Branch

- Mesmo repositório → Diferentes linhas de desenvolvimento
- Alterações simultâneas e independentes → Merge



<https://git-scm.com/book/pt-br/v1/Ramifica%C3%A7%C3%A3o-Branching-no-Git-O-que-%C3%A9-um-Bran>

DCC 111

PDS 2 - Boas práticas de desenvolvimento

41

Considerações finais

- Código limpo → Requer compromisso
 - Muito mais do que vimos aqui
 - Nunca é tarde para começar!
- Um passo de cada vez
 - Clone o repositório
 - Identifique e corrija os problemas
 - Faça vários commits para praticar
 - Discuta e veja as soluções dos colegas



DCC 111

PDS 2 - Boas práticas de desenvolvimento

42