



Boas práticas de programação e codificação

As boas práticas nos auxiliam na hora de programar!

1- Primeira Boa Prática - Indentação / Organizar os códigos.

Todo programador inicia sua trajetória com Algoritmos e Lógica de Programação, e é aqui que ele deve ser orientado sobre as melhores práticas que deverão acompanhá-lo para o resto da sua vida. O primeiro ensinamento a ser passado é também o mais simples de todos, mas que se não for usado de forma adequada, dificultará muito o aprendizado e a leitura do código, é a indentação.

É muito comum vermos iniciantes em programação criando seus algoritmos de forma desorganizada e desalinhada. Chamamos isso de código não indentado, como mostra a Imagem 1. É visível a dificuldade do entendimento do mesmo, comprometendo inclusive o processo de aprendizado. Até um profissional precisa analisar com calma e muita paciência para entender o algoritmo, mesmo que ele tenha poucas linhas. Agora imagine um sistema com centenas delas. Fica praticamente impossível o desenvolvimento dessa maneira.

```
1 package br.com.javamagazine;
2 class Vendas implements InterfaceVendas{
3     public String nomeCliente;
4     protected String descProduto;
5     private double valor=50;
6     public double altera_valor(double valor){
7         this.valor = valor;
8         return valor;
9     }
10    public void imprime(double valor){
11        if (valor>100)
12            JOptionPane.showMessageDialog(null,"Valor acima do permitido");
13        else JOptionPane.showMessageDialog(null,"Valor="+valor);
14    }
15 }
```

-Imagem 1.

Já na Imagem 2, vemos que o mesmo código, agora endentado, fica muito mais legível, limpo e esteticamente mais bonito. Mesmo que você programe sozinho, deve adotar esta boa prática, e se compartilhar a fonte com outras pessoas, é obrigatório para que o projeto não empaque nesse pequeno grande detalhe.

```
1 package br.com.javamagazine;
2
3 import javax.swing.JOptionPane;
4
5 class Vendas implements InterfaceVendas {
6
7     public String nomeCliente;
8     protected String descProduto;
9     private double valor = 50;
10
11     public double alteraValor(double valor){
12         this.valor = valor;
13         return valor;
14     }
15
16     public void imprimeValor(double valor){
17         if (valor > 100)
18             JOptionPane.showMessageDialog(null,"Valor acima do permitido");
19         else
20             JOptionPane.showMessageDialog(null,"Valor="+valor);
21     }
22
23 }
```

-Imagem 2.

Segunda Boa Prática - Comentários e Documentação.

Atire a primeira pedra quem nunca deixou de comentar um código e depois não se lembrava mais da funcionalidade do algoritmo que você mesmo escreveu. No momento em que se está programando, para você e para Deus fica muito clara a finalidade daquela classe, do método ou da variável, porém, passado algum tempo, somente Deus saberá a lógica daquele código, e você poderá não fazer a mínima ideia. Muitos programadores ignoram a importância dos comentários no seu arquivo fonte, geralmente por falta de conscientização dessa boa prática para que o mesmo não gere dor de cabeça no futuro. Você pode usar comentários com duas finalidades:

1- Explicar o algoritmo ou a lógica usada, mostrando o objetivo de uma variável, método, classe...;

2- Documentar o projeto, descrevendo a especificação do código. Desta maneira, qualquer pessoa poderá analisar um arquivo de documentação, mesmo que este não apresente o código fonte. Através de uma ferramenta chamada javadoc é possível gerar documentação baseada em tags específicas que você coloca no seu código fonte.

O primeiro comentário deve ser colocado no início do arquivo indicando o objetivo do mesmo através de um pequeno resumo. Ao criar um método, procure sempre informar qual é a finalidade do mesmo e se ele irá retornar ou não alguma informação. Em muitas variáveis criadas, somente através de seu nome nem sempre é possível saber que informação ela irá armazenar ou para que ela servirá, por isso, não deixe de comentá-la nesses casos.

Na Imagem 3 vemos um código com comentários. Observe que logo no começo do arquivo foi colocado um bloco de comentários usando os delimitadores `/*..*/`. Depois do import, foi inserido `/**..*/`. Esse delimitador indica que o comentário presente dentro dele será usado pelo javadoc para gerar documentação no formato HTML. Também é possível colocar um comentário de linha utilizando `//`, como mostrado para explicar a variável `média`.

Não é objetivo do artigo ensinar a gerar documentação e nem as tags usadas. Queremos apenas conscientizar de sua importância e lembrar que as pessoas tendem a evitar a leitura de comentários muito extensos. Deste modo, coloque somente o que importa e em poucas linhas.

```
1  /*
2  * Procure sempre comentar no início do arquivo fonte colocando informações como
3  * Nome da Classe, Data da Criação
4  * Todos os direitos reservados para Nome da Empresa e o endereço desta completo
5  * Outras informações que achar necessário.
6  * Se você não for gerar documentação ou se um comentário não é adequado para isso,
7  * também pode colocá-lo aqui
8  */
9
10 package br.com.devmedia;
11
12 import javax.swing.JOptionPane;
13
14 /**
15  * A informação que você colocar aqui irá para a documentação gerada através do javadoc.
16  * @version 1.0
17  * @author Neri Aldoir Neitzke - Universidade Ulbra
18  */
19
20 public class CalcMedia {
21     public static void main(String args[]){
22         float nota1, nota2, media;
23         nota1 = 5;
24         nota2 = 7;
25         media = (nota1 + nota2) / 2; //calcula a media do aluno
26         if (media >= 6)
27             JOptionPane.showMessageDialog(null,"Aprovado com média " + media);
28         else
29             JOptionPane.showMessageDialog(null,"Reprovado com média " + media);
30     }
31 }
```

-Imagem 3.

Terceira Boa Prática: Convenções de nomes para classes, métodos, variáveis...

Usando as boas práticas já citadas e acrescentando a isso convenções de nomes, você irá tornar a leitura do programa muito mais fácil. Colocando nomes adequados e padronizados, você passará informações que ajudarão na compreensão do código, indicando, por exemplo, o que uma variável irá armazenar ou o que um determinado método irá fazer. Se você agregar a isso uma boa convenção de nomes, o ajudará muito no ciclo de desenvolvimento do sistema, como podemos ver abaixo:

-Pacotes: a primeira convenção de nomes que será mostrada é para pacotes. Imagine que você irá desenvolver um sistema para uma instituição de ensino, por exemplo a Universidade Ulbra. Durante o desenvolvimento, devemos agrupar as classes de acordo com seus objetivos. Por exemplo, para classes responsáveis pela interface com o usuário, podemos definir o pacote `br.com.ulbra.view`. Observando-o, facilmente identificamos que ele deve ser escrito de forma semelhante a um endereço web, só que de trás para frente. E ao final, indicamos um nome (ou um conjunto de nomes, preferencialmente separados por “.”), que classifica as classes agrupadas;

Classes e Interfaces: ainda analisando a **Figura 1**, observe a classe chamada `Conexao` e perceba que os nomes das classes iniciam com uma letra maiúscula, sendo simples e descritivo. Quando a classe possuir um nome composto, como `MenuPrincipal`, o primeiro caractere de cada palavra deve ser sempre maiúsculo. Essas regras também são aplicadas para Interfaces;

Métodos: o que difere a convenção de nomes de classes para nomes de métodos é que para os métodos a primeira letra deve estar em minúsculo. Como os métodos são criados para executar algum procedimento, procure usar verbos para seus nomes, por exemplo: `imprimeValor()`;

Variáveis: use a mesma convenção adotada para Métodos, criando sempre nomes curtos (`nota`, `primeiroNome`, `mediaAluno`) e significativos, em que ao bater o olho no nome, o programador já saiba que informação a variável vai armazenar. Evite nomes de variáveis de apenas um caractere (`i`, `j`, `x`, `y`), a não ser para aquelas usadas para índices em laços de repetição onde o algoritmo percorre vetores. Em variáveis finais (constantes), isto é, que possuem um valor fixo, todas as letras devem estar em maiúsculo e com as palavras separadas por sublinhado (“_”), por exemplo: `TAXA_IMPOSTO`;

Quarta Boa Prática - Tratamento de erros (try/catch).

O fantástico do ser humano é o fato de que quando achamos que todas as possibilidades já se esgotaram, vem um camarada e descobre algo novo, ou um problema novo. Nunca podemos desenvolver um sistema pensando que os usuários serão bons programadores ou feras em informática. Durante o desenvolvimento, temos que pensar nos usuários que podem inserir dados de forma incorreta, fazer coisas inimagináveis e que por isso seu código pode apresentar erros.

A realidade do dia a dia nas empresas pode mostrar a você um mundo novo, em que um usuário, em vez de seguir um manual ou suas orientações, pode querer usar o sistema à sua própria maneira, e se o sistema não prever todas essas possibilidades, se prepare para uma grande dor de cabeça. Além destes, existem erros externos que podem ocorrer, como uma impressora desligada, queda de conexão com a internet, um disco encher ou o computador ficar sem memória. Independente do problema ocorrido, seu sistema deve (pelo menos) avisar o usuário o que aconteceu, evitar automaticamente a perda de dados (salvar) e permitir, de forma elegante, que ele possa continuar sendo usado sem maiores traumas.

Todas as linguagens de programação têm suporte ao tratamento de erros, mas nem sempre seu uso é obrigatório; não se iluda por isso, é aí que mora o perigo. Java é considerado uma linguagem robusta, e um dos motivos é o fato dela exigir que você use o try..catch em determinadas situações, como no acesso ao banco de dados ou para a seleção de um arquivo externo qualquer. Isso já garante a você (mesmo não sabendo) o uso de uma boa prática. Para outros casos, é altamente recomendável o tratamento de erros, oferecendo assim maior segurança e confiabilidade ao sistema.

O básico do tratamento de erros é o sistema mostrar uma mensagem quando uma exceção ocorrer, o que deve ser feito tanto na fase de desenvolvimento, para o programador, quanto para o usuário final, quando o sistema estiver em produção (isto é, sendo utilizado).

Ao acontecer algum imprevisto, procure fazer com que o sistema comunique o usuário do problema ocorrido e permita que a operação aconteça novamente. O Java permite tratar os principais erros com bibliotecas específicas, por isso evite usar apenas a classe `Exception`, pois ela é muito genérica e dificultará a localização do erro real. Além disso, procure não usar tratamento de erros em blocos muito grandes, pois dificulta a localização do problema – a não ser que seja extremamente necessário. Quando isso acontecer, tente melhorar o código para que ele não fique tão longo. Uma dica para isso pode ser a implementação de novos métodos. A Imagem 4 mostra um exemplo do uso de `try..catch`

```
1 public boolean conecta(){
2
3     boolean result = true;
4     try { //tenta conectar
5         Class.forName(driver); //carrega o driver do banco
6         conexao = DriverManager.getConnection(url, usuario, senha);
7         JOptionPane.showMessageDialog(null, "Conexao com sucesso");
8     }
9     catch(ClassNotFoundException erroClass) {
10         JOptionPane.showMessageDialog(null, "Driver não Localizado: "+erroClass);
11         result = false;
12     }
13     catch(SQLException erroBanco){
14         JOptionPane.showMessageDialog(null, "Problemas na comunicação com o banco: "+erroBanco);
15         result = false;
16     }
17
18     return result;
19 }
```

-Imagem 4.

Quinta Boa Prática - Testar e Depurar.

Não muito tempo atrás, praticamente não existia testes de software, a não ser o que o próprio programador fazia durante o desenvolvimento do sistema, ou seja, validação dos dados e testes com alguns registros – o que sempre foi muito pouco. No entanto, devido ao surgimento de novas tecnologias, o aumento da complexidade dos programas e a busca por qualidade, os testes de software ganharam um espaço muito importante dentro do ciclo de desenvolvimento de sistemas.

Assim, procure realizar muitos testes no seu sistema e também aprenda a usar um depurador para analisar o código em tempo de execução. Desta forma, será facilitada a localização de defeitos que muitas vezes não aparecem na compilação por serem erros de lógica. Muitos programadores desconhecem a depuração fornecida pelas evoluídas IDEs, que oferecem a possibilidade da execução passo a passo de um programa, mas usam a famosa impressão (`System.out.println()`) para depurar sua classe. No entanto, essa não é uma forma prática e elegante de depurar erros;

Sexta Boa Prática - Backup e Tamanho.

Essa dica é tão antiga quanto importante, mas que muitos só percebem sua relevância quando perdem códigos e têm que refazê-los. Portanto, não deixe de fazer backups com determinada frequência (ao final do dia ou semanalmente, por exemplo);

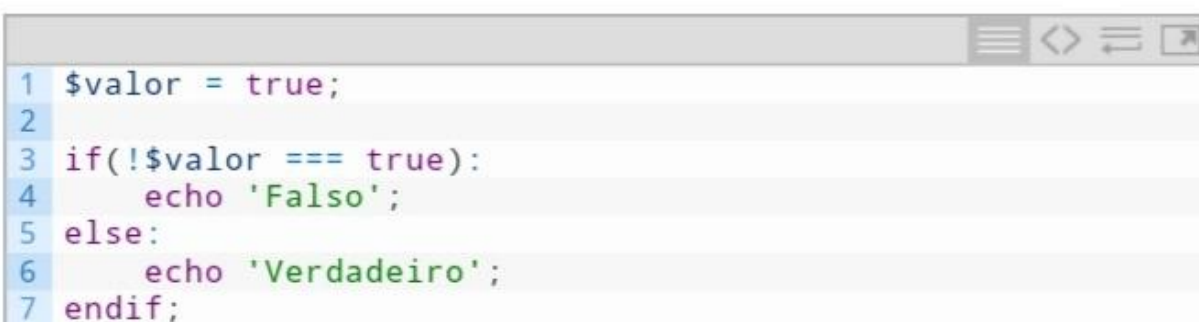
Evite classes muito longas; com mais de 1000 linhas já fica muito complicado sua leitura e depuração, por isso devem ser evitadas. Se o código estiver muito grande, é possível que seu algoritmo precise ser revisado, pois pode estar fazendo mais do que necessita.



Sétima Boa Prática - Evitar condição de negação no IF.

Praticamente não existe programação sem o controle de fluxo usando condições “IF”, a ideia é sempre avaliar se uma condição é verdadeira dentro da condição para executar determinado código.

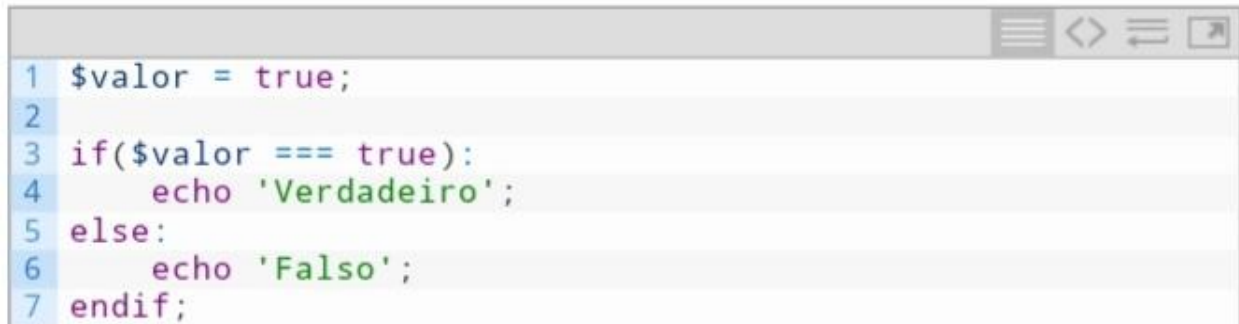
Por esse motivo evite sempre que possível usar verificações de negação na condição, sempre que possível avalie primeiro a condição verdadeira e caso falso execute o código do “ELSE”, isso também torna o código mais legível e menos confuso quando temos “IFs” aninhados.

A screenshot of a code editor window. The editor has a light gray background and a dark gray title bar. The code is written in PHP and is as follows:

```
1 $valor = true;
2
3 if(!$valor === true):
4     echo 'Falso';
5 else:
6     echo 'Verdadeiro';
7 endif;
```

The code is color-coded: keywords like 'if', 'else', and 'endif' are in purple, variables and operators are in blue, and string literals are in green. The editor also shows line numbers on the left side of the code.

-Avaliando a condição de Negação



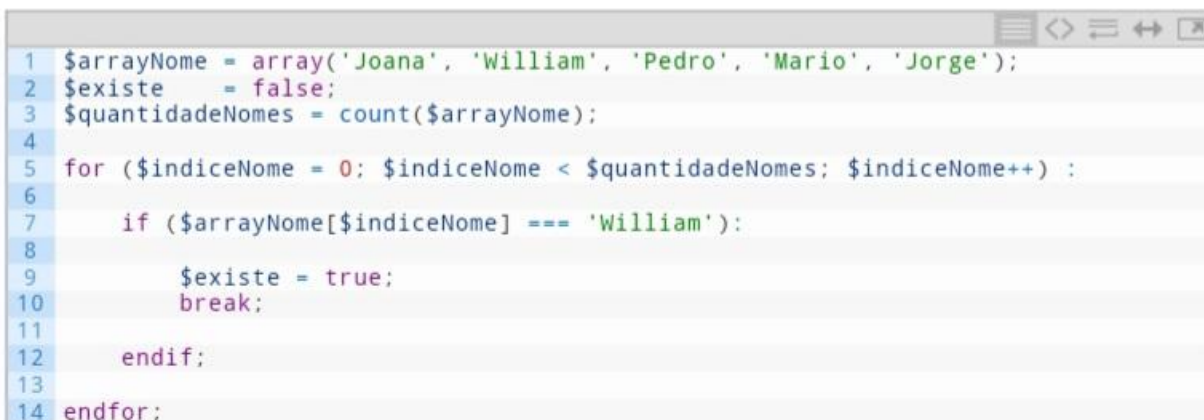
```
1 $valor = true;
2
3 if($valor === true):
4     echo 'Verdadeiro';
5 else:
6     echo 'Falso';
7 endif;
```

-Avaliando a condição Verdadeira

Oitava Boa Prática - Percorrer loops somente o necessário.

Às vezes precisamos percorrer um loop procurando por apenas uma combinação e mesmo após encontrar essa combinação ainda deixamos o loop executando por uma infinidade de vezes.

No exemplo abaixo estou percorrendo a “\$arrayNome” procurando pelo valor ‘William’, caso seja encontrado atribuo o valor TRUE para variável “\$existe” e mesmo assim o loop continua, uma boa prática seria encerrar a execução do loop usando “break”.



```
1 $arrayNome = array('Joana', 'William', 'Pedro', 'Mario', 'Jorge');
2 $existe = false;
3 $quantidadeNomes = count($arrayNome);
4
5 for ($indiceNome = 0; $indiceNome < $quantidadeNomes; $indiceNome++) :
6
7     if ($arrayNome[$indiceNome] === 'William'):
8
9         $existe = true;
10        break;
11
12    endif;
13
14 endfor;
```

-Percorrendo Array até o final mesmo tendo encontrado a combinação

```
1 $arrayNome = array('Joana', 'William', 'Pedro', 'Mario', 'Jorge');
2 $existe    = false;
3 $quantidadeNomes = count($arrayNome);
4
5 for ($indiceNome = 0; $indiceNome < $quantidadeNomes; $indiceNome++) :
6
7     if ($arrayNome[$indiceNome] === 'William'):
8
9         $existe = true;
10
11     endif;
12
13 endfor;
```

-Finalizando loop após encontrar combinação

Nona Boa Prática - Não usar valor padrão em argumentos de funções.

Quando escrevemos uma função que necessita receber argumentos de entrada as vezes sentimos a necessidade de deixar um valor padrão nesse argumento, mas garanto que nem mesmo quem escreveu a função vai lembrar desse valor padrão quando precisar debugar em erro, imagine outro programador.

De preferência em tratar um valor de argumento inválido do que usar um valor padrão:



```
1 function dataBrToEng($data){
2     if (empty($data)):
3         throw new \InvalidArgumentException("Argumento inválido!");
4     else:
5         $data = explode("/", $data);
6         return $data[2].'-'.$data[1].'-'.$data[0];
7     endif;
8 }
```

Décima Boa Prática - Nomear funções de maneira intuitiva.

Nomear funções é outra tarefa que aparentemente é simples, mas se soubermos escolher nomes mais intuitivos e ligados ao objetivo da função, também podemos tornar nosso código mais legível.

Por exemplo, nos meus scripts orientado a objetos ou não, tenho o costume de nomear funções que retornam valores do tipo BOOLEAN sempre iniciando com “is”, essa prática deixa a chamada da função dentro de um “IF” mais intuitivo.

No código abaixo coloquei uma função básica para validar e-mails “isEmailValid()”, quando chamo a função para validar um e-mail o entendimento da condição “IF” fica algo semelhante “SE é um e-mail válido ENTÃO”:



```
1 function isEmailValid($email){
2
3     $conta = "[a-zA-Z0-9\._-]+@";
4     $domino = "[a-zA-Z0-9\._-]+.";
5     $extensao = "([a-zA-Z]{2,4})$/";
6     $pattern = $conta.$domino.$extensao;
7
8     if (preg_match($pattern, $email))
9         return true;
10    else
11        return false;
12 }
13
14 if (isEmailValid('wllfl@ig.com.br')):
15     echo 'Válido';
16 else:
17     echo 'Inválido';
18 endif;
```

Bibliografias (Fontes)

<http://www.devwilliam.com.br/extra/profissional/10-boas-praticas-de-programacao>

<https://www.devmedia.com.br/boas-praticas-de-programacao/21137#Endenta>



VERSIONAMENTO

MANUAL DE USO

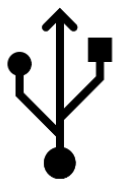
APRENDA A USAR | ELIMINE DÚVIDAS



O QUE É VERSIONAMENTO

O versionamento de software é o processo de atribuir um nome único ou uma numeração única para indicar o estado de um programa de computador. Esses números são geralmente atribuídos de forma crescente e indicam o desenvolvimento de melhorias ou correção de falhas no software.

Software modernos são constantemente construídos usando dois esquemas de versionamento distintos: um número interno de versão, que pode ser incrementado diversas vezes por dia (para controlar o número de revisões), e uma versão de lançamento, que usualmente se altera com menos frequência (como um versionamento semântico ou um codinome).



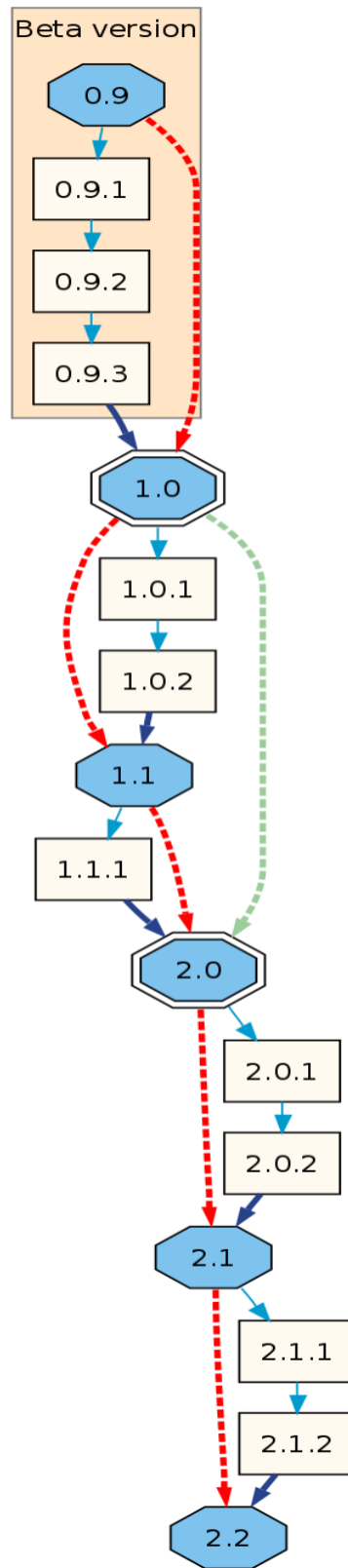
COMO FUNCIONA?

O versionamento é realizado sempre em ordem crescente com a inclusão de novos recursos e de novas ferramentas que melhoram o seu desempenho. Existem softwares nos quais os números de versionamento são internos para permitir o controle pelos seus desenvolvedores e para diferenciar da numeração da versão dos produtos, sendo que esses são do interesse dos usuários finais.

Os números iniciais são referentes às alterações mais significativas e os demais apontam pequenas modificações. Por exemplo, um software com versão 1.2 atualizado para 1.3 indica uma mudança estrutural que pode ser a exclusão e inclusão de ferramentas. Já se a versão for 1.01 e mudar para 1.02 significa que foi realizada a correção de uma falha operacional.

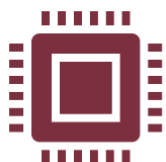


Os métodos de diferenciação são diversos, podendo ser utilizados os números ou a separação de sequências por caracteres. Também são aplicados códigos aleatórios ou a inserção de datas diferenciadas. A finalidade desses registros dos números pode ser a representação de status, liberação, lançamento ou compilação.





ESQUEMAS DE VERSIONAMENTO



Diversas formas diferentes de versionamento são atualmente utilizadas para distinguir diferentes versões de um software. A presença constante da computação no dia-a-dia das pessoas, levou esses esquemas a serem utilizados em contextos fora da computação (como ocorre em Escola **2.0** e Indústria **4.0**, geralmente sugerindo uma progressão tecnológica).

IDENTIFICAÇÃO SEQUENCIAL



No esquema de identificação sequencial, cada versão lançada é associada a um identificador único que consiste em uma ou mais sequências de números ou letras. Este é o único ponto em comum dessa forma de organização, pois a quantidade de sequências utilizadas, o significado de cada sequência e a forma de incremento das sequências varia constantemente de software para software.

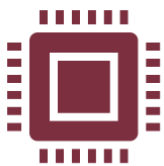
Baseado na significatividade da mudança

Em alguns esquemas, os identificadores sequenciais são utilizados para informar a significância das mudanças entre as versões lançadas. As mudanças são classificadas por nível de significatividade e a decisão de qual sequência deve se alterar é baseada na significatividade das mudanças em relação à versão anterior. Dessa forma, a primeira



sequência é alterada apenas quando ocorrer mudanças significativas, enquanto as sequências subsequentes representam mudanças com significatividade decrescente.

Dependendo da forma de organização, a significatividade da mudança pode ser determinada pelo número de linhas de código alteradas, quantidade de pontos de função adicionados ou removidos, potencial impacto a consumidores (em termos do trabalho requerido para se adaptar à próxima versão), risco de falhas ou de mudanças de comportamento inesperadas, nível de mudanças no leiaute visual, quantidade de novos recursos ou qualquer outro atributo que os desenvolvedores do produto julgue significativo, incluindo eventuais necessidade de associar uma impressão relativa de evolução do produto como estratégia de marketing.



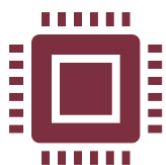
O *versionamento semântico* (em inglês Semantic versioning, ou SemVer) é atualmente o esquema de versionamento mais conhecido e utilizado nessa categoria e utiliza uma sequência de três números (Major.Minor.Patch) e opcionalmente um rótulo de pré-lançamento ou metadados. Nesse esquema, risco e funcionalidade são as medidas de significância. Mudanças impactantes são sinalizadas com o incremento no número majoritário (Major), indicando alto risco; novos recursos não impactantes são sinalizados com o incremento do número secundário (Minor), indicando risco médio; e todas as outras alterações são sinalizadas incrementando o número do ajuste (Patch), indicando baixo risco.





ENTENDA O POR QUE O VERSIONAMENTO É IMPORTANTE

A metodologia de versionamento é importante para impedir a ocorrência de confusão sobre as versões dos softwares que estão em uso e evitar ataques cibernéticos. Ela é uma questão estratégica no desenvolvimento dos programas, sendo que os códigos de segurança necessitam de patches ou remendos para corrigir vulnerabilidades.



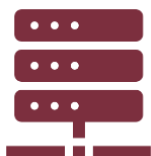
REALIZAÇÃO DO ACOMPANHAMENTO DE VERSÕES

As alterações realizadas na fonte são identificadas apontando o problema solucionado ou o aperfeiçoamento efetuado, com identificação das razões das mudanças e dos profissionais responsáveis. Desse modo, o acompanhamento das versões é um mecanismo que possibilita a tomada de providência nos projetos de softwares. O controle das versões é indispensável para o desenvolvimento colaborativo ou particular.

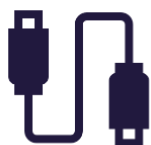


DIMINUIÇÃO DE ERROS E RETRABALHOS

O uso de ferramentas de versionamento de software simplifica as atividades, já que grandes equipes conseguem trabalhar no mesmo projeto sem enfrentar conflitos no desenvolvimento. As mudanças de código são realizadas sem muita preocupação e com mais agilidade.



Os profissionais fazem o trabalho em conjunto dentro da mesma versão para desenvolver as funcionalidades, cooperando uns com os outros. A consequência disso é que erros e retrabalhos são evitados pela execução de tarefas com excelência.



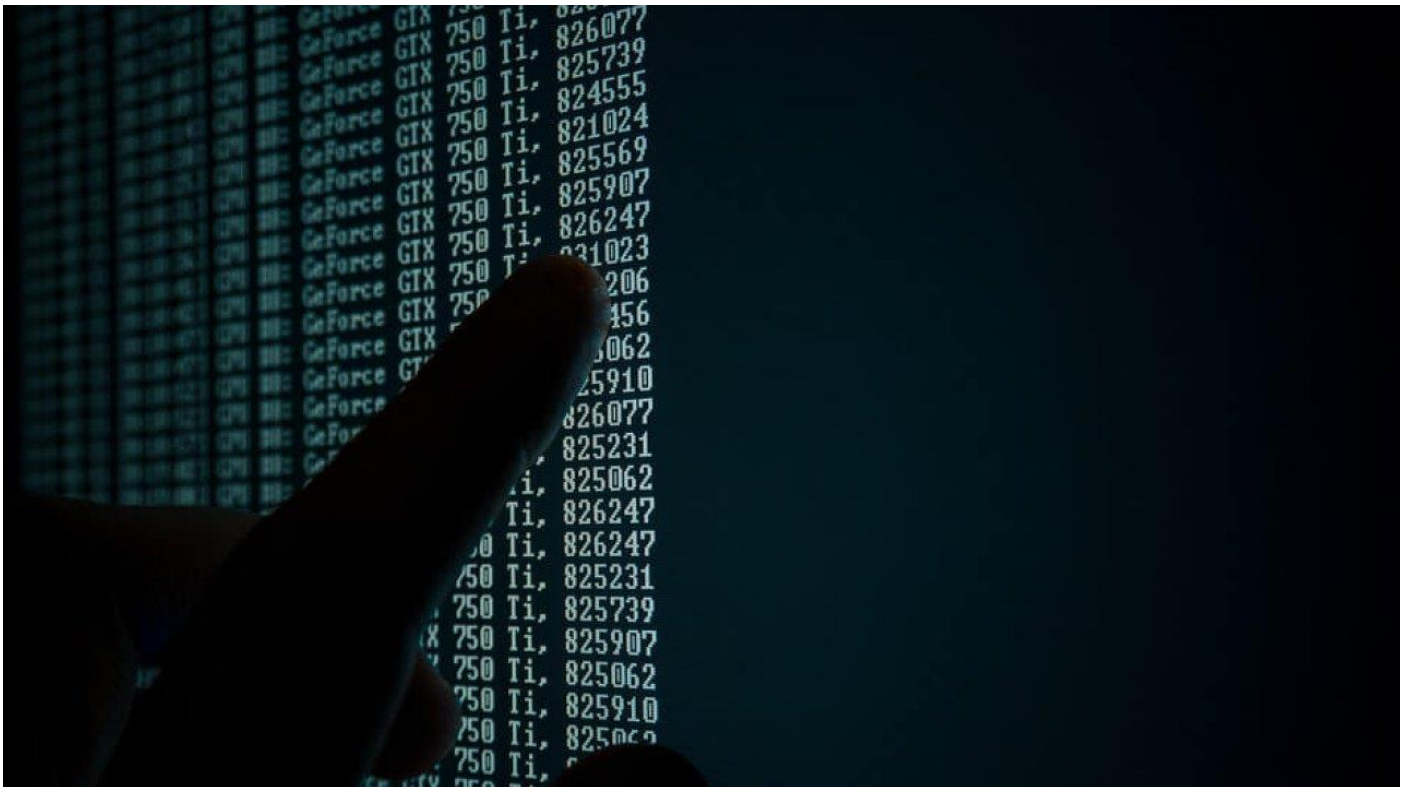
ANÁLISE DO HISTÓRICO DE ALTERAÇÕES

A manutenção do histórico das alterações é primordial para sanar dúvidas futuras sobre a implementação de certas funcionalidades ou sobre modificações em arquivos. Essas informações são recuperadas com facilidade por meio da ferramenta que controla a versão ou pelo versionamento.



RECONHECIMENTO DA VERSÃO MAIS ATUAL

As ferramentas têm funcionalidades diferentes e, por isso, o versionamento serve para reconhecer as versões atuais, mas também as anteriores. Se acontecer um erro ou problema que comprometa a segurança do software, o usuário pode fazer o downgrade para a versão anterior. Dessa forma, os dados são mantidos seguros se houver falha detectada na produção.



AUMENTO DA SEGURANÇA

O versionamento faz a documentação das mudanças realizadas, possibilitando a recuperação objetiva e simplificada de códigos, informações ou ferramentas deixadas para trás durante o desenvolvimento do software. Se houver um comprometimento do programa, um bug ou qualquer dificuldade, a solução poderá ser buscada no código da versão antecedente.





CORREÇÃO DE PROBLEMAS DE FUNCIONAMENTO

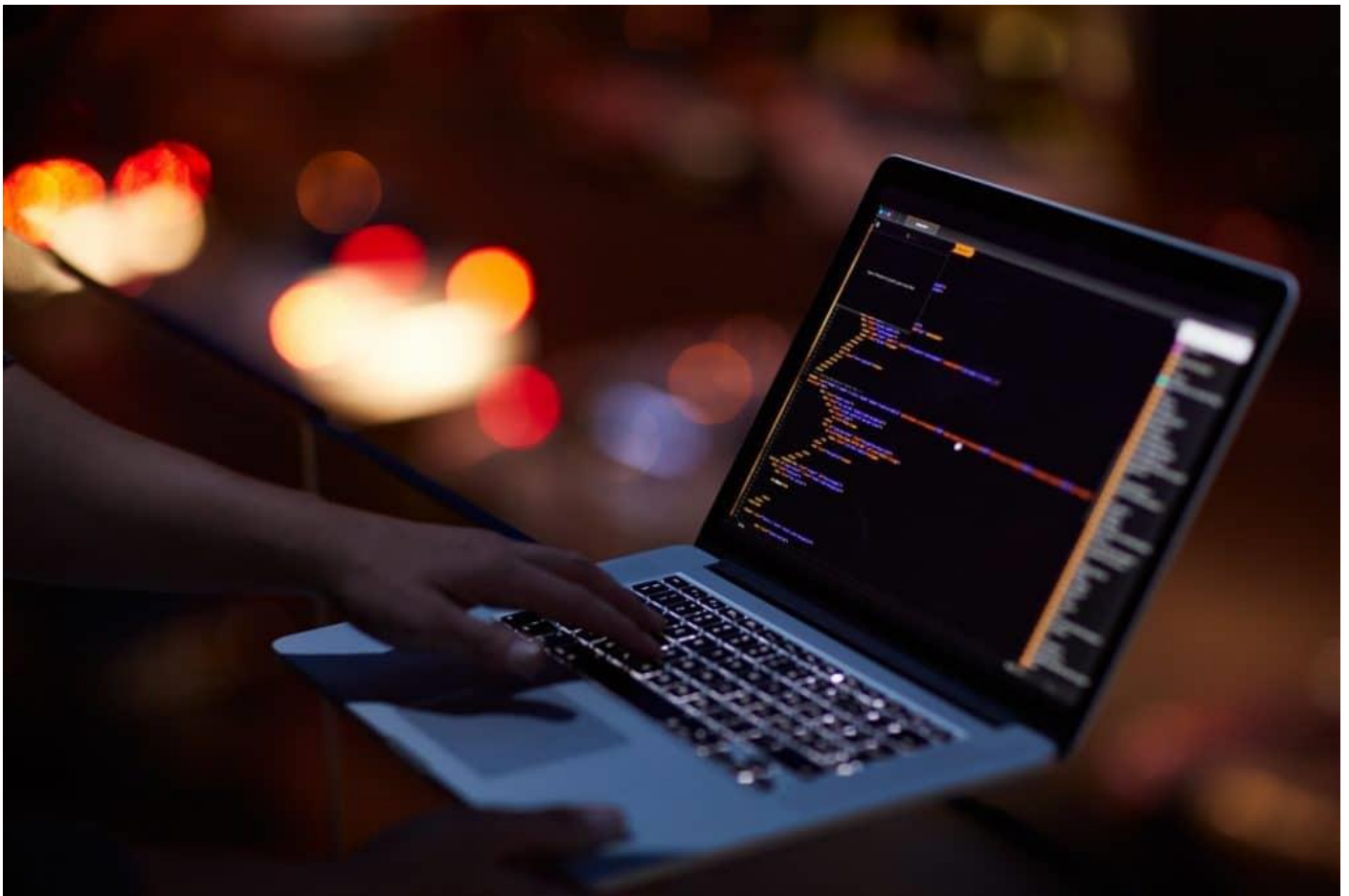
As falhas de segurança podem comprometer as operações, visto que todos os dias surgem novas ameaças que atacam os sistemas conectados à internet. O processo de controle de versão é uma prática que protege o código da empresa, já que a sequência lógica utilizada para o armazenamento das informações do software ajuda a compreender o que ocorreu em desastres.

Também, auxilia a averiguar o que ficou comprometido e o que pode ser feito para a construção de versões mais confiáveis e robustas. Portanto, o versionamento de software vai muito além de registrar uma numeração relacionada ao desenvolvimento. Trata-se de uma prática que aumenta a produtividade, agiliza processos e eleva a segurança da informação corporativa.

Também se torna possível fazer uma regressão nas mudanças. Se você perceber que alguma alteração de melhoramento não funcionou bem, pode simplesmente reverter uma por vez até encontrar qual causou o problema.

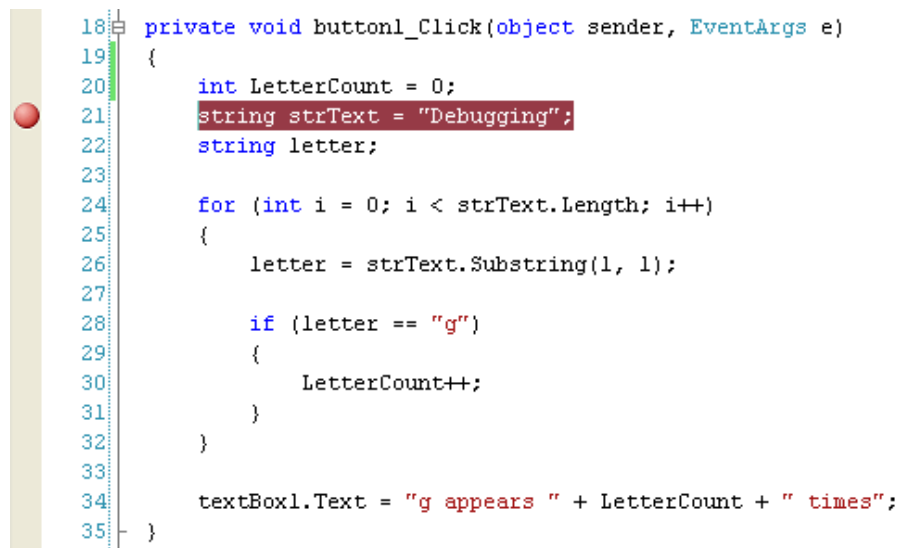
AFINAL POR QUE O VERSIONAMENTO É TÃO IMPORTANTE?

Os softwares lançados no mercado apresentam versões iniciais que sofrem alterações com o passar do tempo. Isso é necessário para o aperfeiçoamento das ferramentas, que necessitam de atualização para cumprir com as suas funções de acordo com as novas tecnologias. Essas versões são atualizadas constantemente com base em numerações.



Manual do Breakpoint

Breakpoint ou ponto de parada é um ponto intencional de pausar o programa um programa no computador durante a depuração. De forma geral , ele representa a habilidade do programa conhecer um erro durante a execução e mostrar onde é o erro

A screenshot of a code editor window. On the left margin, there is a vertical toolbar with a red circle icon representing a breakpoint, which is positioned next to line 21 of the code. The code is written in C# and defines a method named 'button1_Click'. The code includes variable declarations for 'LetterCount', 'strText', and 'letter', a 'for' loop that iterates over the characters of 'strText', an 'if' statement that checks if the current character is 'g', and a final line that updates the text of 'textBox1' based on the count of 'g's. The line numbers 18 through 35 are visible on the left side of the code block.

```
18 private void button1_Click(object sender, EventArgs e)
19 {
20     int LetterCount = 0;
21     string strText = "Debugging";
22     string letter;
23
24     for (int i = 0; i < strText.Length; i++)
25     {
26         letter = strText.Substring(i, i + 1);
27
28         if (letter == "g")
29         {
30             LetterCount++;
31         }
32     }
33
34     textBox1.Text = "g appears " + LetterCount + " times";
35 }
```

Esse é um exemplo de breakpoint sendo utilizado no meio de um código

A forma mais comum de usar o breakpoint é quando a execução do programa é interrompida antes da instrução especificada pelo programador. Entretanto outros tipos de breakpoints podem ser utilizados, como a leitura ou escrita de um endereço ou área de memória específicos,

um ponto específico do tempo, um evento específico como a chegada de uma mensagem em uma rede social.

Como criar um breakpoint: você pode colocar um breakpoint em uma função e a execução irá parar se quando for chamada a função como podemos ver no exemplo:

```
break main
```

Nesse caso o programa irá para no começo do código. Agora para criar em uma linha você tem que colocar o número da linha ao lado do break como podemos ver no exemplo

```
break 10
```

Quando você tiver varias linhas e precisar especificar algo dentro da linha você pode colocar assim

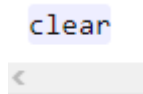
```
break exemplo.c:10
```

O exemplo.c seria o que está dentro da linha.

Agora que aprendemos a criar um breakpoint vamos apagar um breakpoint quando um breakpoint é criado ele vai ganhar um número, você pode apagar um breakpoint usando seu numero como está no exemplo abaixo:

```
delete 3
```

Agora para apagar todos os breakpoints só executar o comando clear igual esta na imagem abaixo

A screenshot of a debugger's command window. The word 'clear' is entered in a text field and is highlighted with a light blue selection. Below the text field is a grey button with a left-pointing arrow, representing the 'Execute' or 'Run' command.

Bom esse é o fim do nosso manual espero que tenha ajudado.