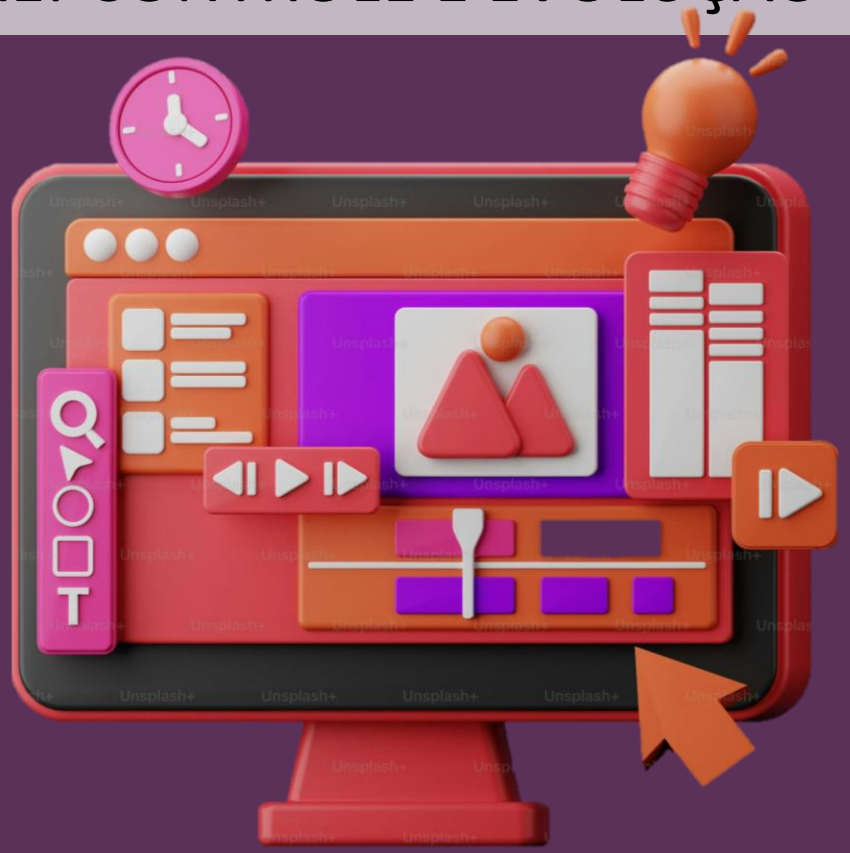
ENGENHARIA DE SOFTWARE

AULA 11 –SUSTENTABILIDADE DO SOFTWARE: CONTROLE E EVOLUÇÃO



WALTER TRAVASSOS SARINHO

@PROF.WALTERTRAVASSOS

WALTER.TRAVASSOS@SECTRAS.EDU.BR

Aula de Hoje

Gerenciamento de Mudanças e Configurações

- Entender como **gerenciar alterações no software** e **configurar o ambiente** para manter a **integridade do sistema** ao longo de seu ciclo de vida.

Manutenção e Evolução de Software

- Compreender a **importância da manutenção contínua** e **evolução do software**, abordando **práticas e desafios**.

Gerenciamento de Mudanças e Configurações do Software

SCM, SCCM

Gerenciamento de Mudanças e Configurações (SCM)

- **Definição**: É o processo de **gerenciar mudanças sistemáticas e documentadas** em software, garantindo que **apenas alterações autorizadas** sejam implementadas.

- **Importância**: O SCM (*software configuration management*) controla a **integridade** dos produtos de software ao longo de seu ciclo de vida, essencial para a estabilidade, rastreabilidade e organização do desenvolvimento.

Principais Componentes do SCM

1. **Identificação de Configuração**: Identificação dos componentes e artefatos de software (código, documentação, etc.) a serem controlados.
2. **Controle de Versão**: Ferramenta e técnica para gerenciar versões diferentes de arquivos de código, configuração e documentação.
3. **Controle de Mudança**: Processos para gerenciar alterações e garantir que sejam apropriadamente analisadas e aprovadas.
4. **Auditoria de Configuração**: Revisão periódica para garantir que o sistema segue os padrões e processos estabelecidos.

Ciclo de Vida das Mudanças

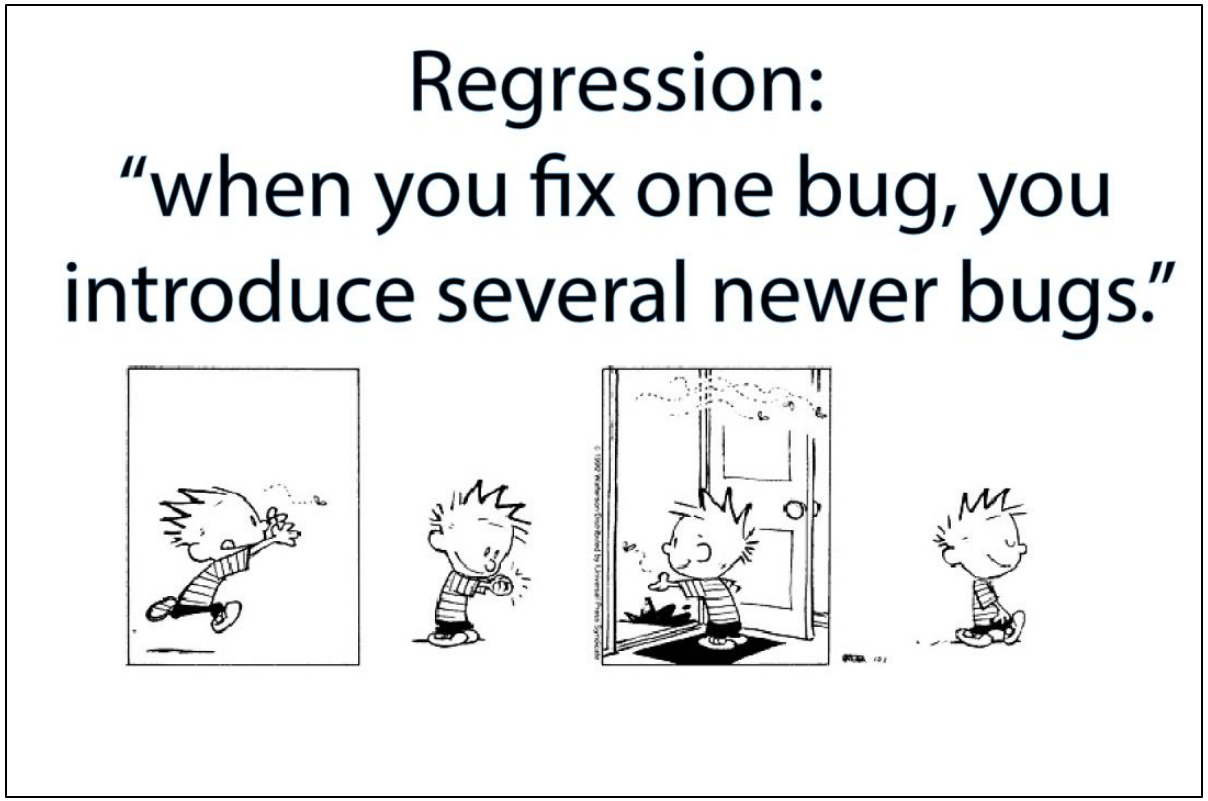
1. Solicitação
2. Avaliação
3. Aprovação
4. Implementação

Podemos utilizar ferramentas, como por exemplo, **Jira** ou **Trello** para gerenciar esse ciclo.

Práticas Recomendadas para Controle de Mudanças

Aprovações Formais

Testes de Regressão



Antes de iniciar a etapa de implementação, temos que ter estabelecido um VCS para o gerenciar as mudanças no software

Version Control System

Sistemas de Controle de Versão (VCS)

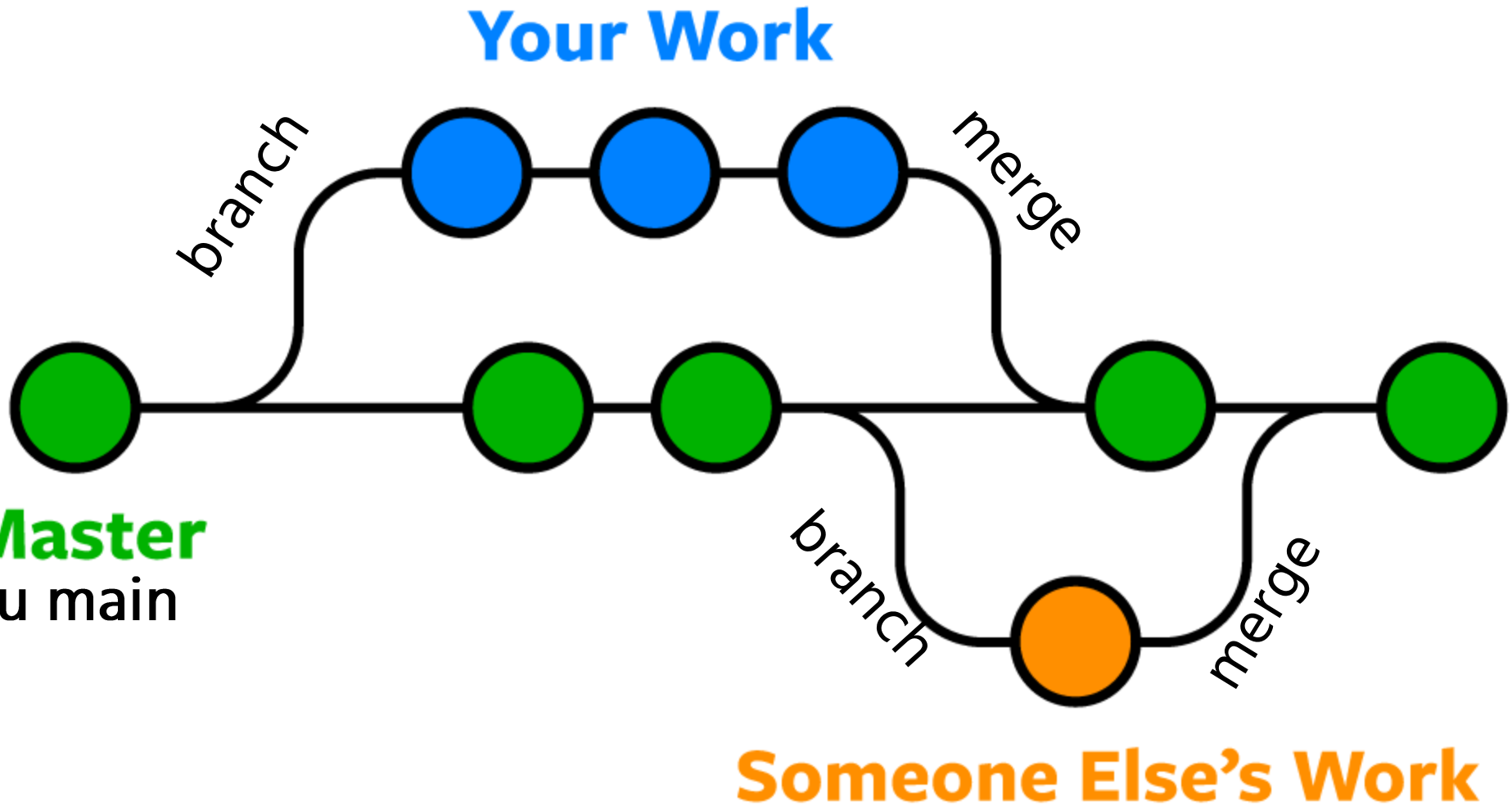
Principais ferramentas: Git, SVN (Subversion).

- Git é o VCS (*version control system*) mais popular e permite o controle distribuído, onde cada desenvolvedor tem uma cópia completa do repositório.

Fluxo de Trabalho em Git

- **Branching e Merging**: Processos essenciais para que vários desenvolvedores trabalhem simultaneamente em diferentes partes do código.
- **Exemplo**: O uso de **branches** para desenvolvimento de funcionalidades, correções e **merges** ao final do desenvolvimento para consolidar as mudanças na versão principal do software.

Exemplo


Your Work
branch
merge
Master
ou main
branch
merge
Someone Else's Work

Como Baixar e Instalar o Git para Windows

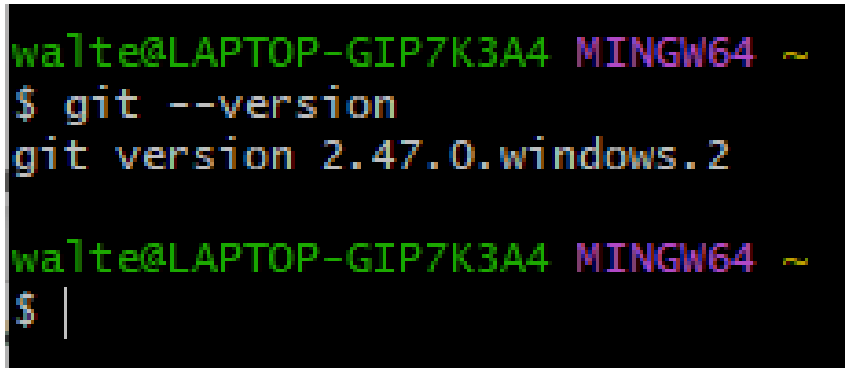
Baixe o Instalador:

- Acesse o site oficial do Git: https://git-scm.com/

Verifique a Instalação:

Após a instalação, abra o **Git Bash** e digite o seguinte comando para verificar se a instalação foi bem-sucedida:

- **git --version**



Configurar usuário e e-mail

Depois de instalar o Git, você pode configurar seu nome de usuário e e-mail (essencial para commits) com os seguintes comandos:

- **git config --global user.name "Seu Nome"**
- **git config --global user.email** seuemail@example.com

Para verificar se seu nome de usuário e e-mail foram configurados corretamente no Git, você pode usar o seguinte

- **git config --global --list**

```
walte@LAPTOP-GIP7K3A4 MINGW64 ~
$ git config --global --list
user.name=walter
user.email=travassoswalter@gmail.com
```

Criar um Repositório e Iniciar o Git

Navegue até a pasta que você deseja transformar em repositório e rode o comando:

- **git init**

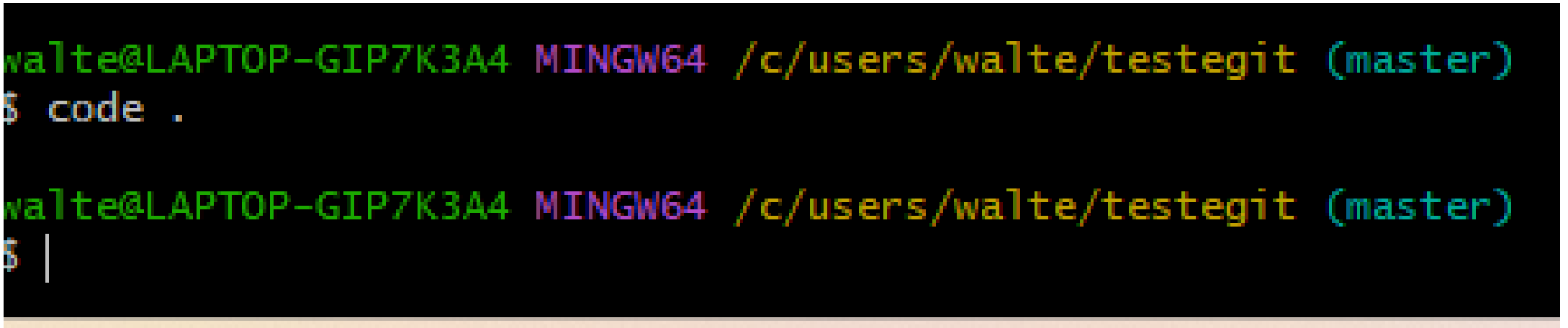
```
walte@LAPTOP-GIP7K3A4 MINGW64 ~
$ git init
Initialized empty Git repository in C:/Users/walte/.git/
```

Use os comandos **cd**, **mkdir** para navegar e criar pastas.

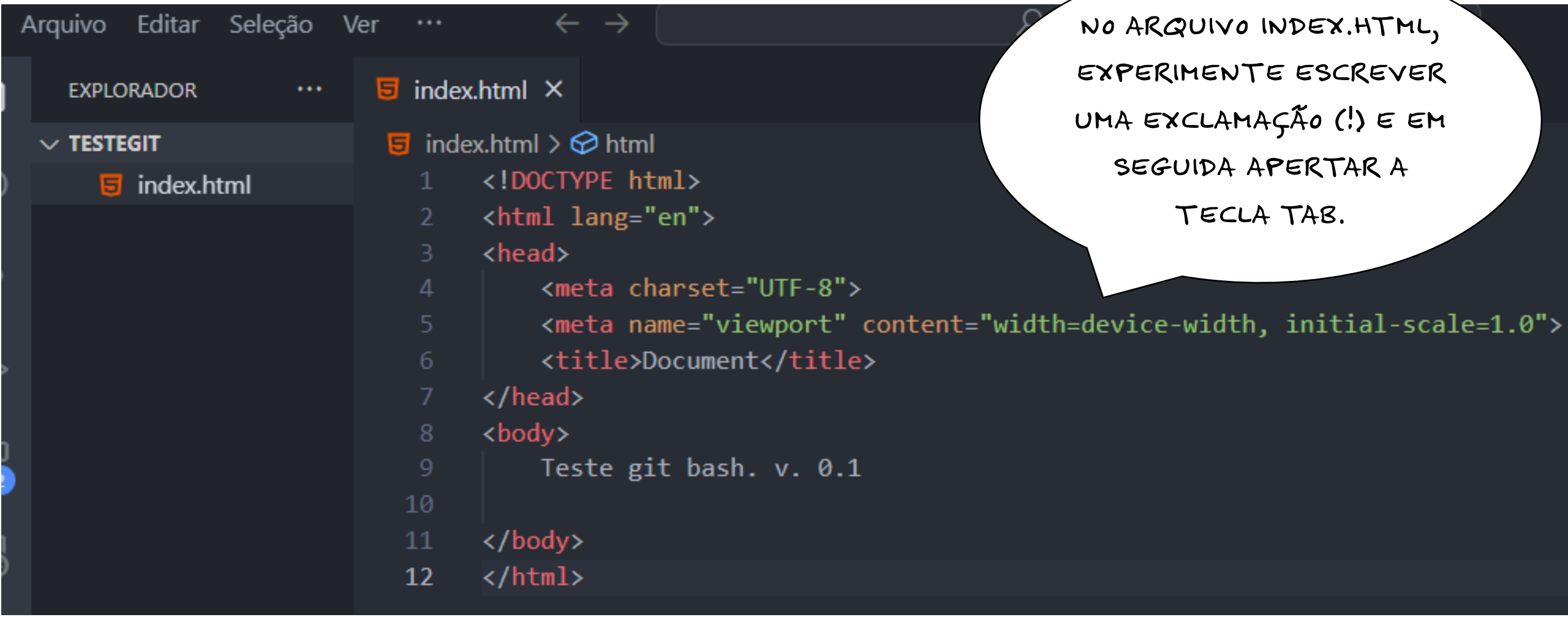
Abrindo o VSCode na pasta atual

Digite o comando a seguir para abrir o VSCode na pasta atual:

- code .



Crie uma página index.html no VS Code



A partir de agora, você pode utilizar o terminal do VS Code (ou continuar no Git Bash)

- **Control + j** – abre o terminal no VS Code.

No Terminal ou no Git Bash...

Depois de fazer alterações no arquivo index.html, você deve adicionar essas alterações ao Git e fazer um **commit**.

Aqui estão os passos:

- **git add index.html**
- // ou para adicionar todos os arquivos: **git add .**
- **git commit -m "Adiciona página HTML inicial"**

```
walte@LAPTOP-GIP7K3A4 MINGW64 /c/users/walte/testegit (master)
$ git add index.html

walte@LAPTOP-GIP7K3A4 MINGW64 /c/users/walte/testegit (master)
$ git commit -m "v0.1"
[master (root-commit) 591558d] v0.1
 1 file changed, 12 insertions(+)
 create mode 100644 index.html
```

Repetir o processo...

Sempre que você fizer alterações nos seus arquivos, siga esses passos para adicionar e fazer commit das mudanças.

Isso cria um **histórico de versões** do seu projeto.

Verificando o Status do Repositório

Você pode usar o comando a seguir para verificar o status do seu repositório e ver quais arquivos foram modificados:

- **git status**

```
walte@LAPTOP-GIP7K3A4 MINGW64 /c/users/walte/testegit (master)
$ git status
On branch master
nothing to commit, working tree clean

walte@LAPTOP-GIP7K3A4 MINGW64 /c/users/walte/testegit (master)
$ git status
]On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   index.html

no changes added to commit (use "git add" and/or "git commit -a")
```

Nesse ponto eu fiz uma alteração no arquivo index.html. Quando executamos novamente o **git status**, aparece a mensagem abaixo.

Criando um Branch (Ramificação)

Os branches permitem que você trabalhe em alterações isoladas do código sem afetar a versão principal.

Verificando os Branches Existentes

Para listar todos os branches:

- **git branch**

O **branch principal** padrão geralmente se chama **main** ou **master**.

Criando um Novo Branch

Vamos criar um novo branch chamado feature-exemplo:

- **git branch** feature-exemplo

Esse comando cria o branch, mas ainda não muda para ele.

Mudando para o Novo Branch:

Agora, vamos mudar para o branch feature-exemplo:

- **git checkout** feature-exemplo

Alternativamente, você pode **criar e mudar** para o novo branch em um único comando:

- **git checkout -b** feature-exemplo

Fazendo Alterações no Novo Branch

Faça as modificações que desejar nos arquivos.

Depois, adicione e faça o commit das mudanças:

- **git add .**
- **git commit -m "Adiciona nova funcionalidade no branch feature-exemplo"**

Voltando para o Branch Principal

Quando você terminar de trabalhar no **branch feature-exemplo**, volte para o **branch principal** (main ou master):

- **git checkout** master

Fazendo o Merge (Mesclagem) dos Branches

Para trazer as mudanças do **feature-exemplo** para o **main**, você precisa estar no **branch main**:

- **git merge** feature-exemplo

Esse comando aplica todas as alterações de feature-exemplo ao branch main.

Se as alterações não tiverem conflito, o merge será automático.

Lidando com Conflitos de Merge

Se houver conflitos (alterações incompatíveis em ambos os branches), Git vai te mostrar os arquivos que precisam ser resolvidos. Você pode abrir esses arquivos e escolher como resolver o conflito.

Após resolver, adicione as alterações e finalize o merge:

- **git add .**
- **git commit -m "Resolve conflitos e finaliza o merge"**


$ git commit -m “🙏”

Deletando o Branch

Após o merge, se você não precisar mais do **branch feature-exemplo**, pode deletá-lo para manter o repositório limpo:

- **git branch -d** feature-exemplo

Git push

- Veja as orientações para enviar seu projeto para nuvem

```
walte@LAPTOP-GIP7K3A4 MINGW64 ~/pastateste (master)
$ git push
fatal: No configured push destination.
Either specify the URL from the command-line or configure a remote repository using

    git remote add <name> <url>

and then push using the remote name

    git push <name>
```

Boas Práticas e Desafios

Case PIX – Banco Central do Brasil (2020)

Resumo do Caso

- Falha: Problemas após o lançamento do sistema PIX, como **duplicação de transações e falhas** de atualização de saldo.

Causas

- **Testes insuficientes** em cenários de **alta carga**.
- **Monitoramento inadequado**.
- Erros de configuração na comunicação entre sistemas.

Impacto: A confiança no PIX foi afetada logo no início, exigindo ajustes urgentes para estabilizar o sistema.

pix
powered by Banco Central

Manutenção e Evolução de Software

O que é manutenção de software?

- É o processo contínuo de **aprimorar, corrigir, adaptar** e **otimizar** um **software após sua entrega inicial**.
- O objetivo é garantir que ele continue funcionando corretamente, atendendo a **novas necessidades e evoluindo com mudanças no ambiente tecnológico e nos requisitos dos usuários**.
- Ela não envolve apenas a correção de problemas, mas também **ajustes para melhorar o desempenho** e adaptá-lo a novos contextos, mantendo o software relevante e confiável ao longo do tempo.
- Esse processo é essencial para que o software continue útil e eficiente no longo prazo.

Tipos de Manutenção de Software

1. **Manutenção Corretiva**: Corrigir **erros ou falhas** detectadas durante o uso do software. **Exemplo**: Resolver bugs que afetam a funcionalidade do sistema.
2. **Manutenção Adaptativa**: Ajustar o software para **mudanças no ambiente externo**, como atualizações de hardware ou sistemas operacionais. **Exemplo**: Adaptar o sistema para uma nova versão do banco de dados.
3. **Manutenção Perfectiva (ou evolutiva)**: Melhorar ou **otimizar o desempenho**, mantendo a funcionalidade. **Exemplo**: Otimizar um algoritmo para torná-lo mais eficiente.
4. **Manutenção Preventiva**: **Reduzir a probabilidade de falhas futuras** por meio de melhorias na estrutura interna do software. **Exemplo**: Refatorar código para facilitar futuras manutenções e reduzir possíveis falhas.

Por que a manutenção é inevitável? Para manter...

- **Confiabilidade**: Correções e melhorias contínuas reduzem erros e aumentam a confiança no sistema.
- **Adaptabilidade**: O ambiente tecnológico muda constantemente (como atualizações de sistemas operacionais, hardware), e a manutenção permite que o software acompanhe essas mudanças.
- **Performance**: Otimizações ajudam o software a operar de forma mais rápida e eficiente.
- **Longevidade**: Quanto mais mantido e atualizado, maior a vida útil do software, o que economiza recursos a longo prazo.

1ª Lei de Lehman (1970)
Evolução do Software

- **Princípio**: "O software deve evoluir continuamente para não se tornar obsoleto.“
- **Motivo**: Sistemas de software atendem a necessidades que mudam ao longo do tempo. Para se manterem relevantes e úteis, eles precisam acompanhar essas mudanças.
- **Consequência da Inércia**: Sem atualizações, **o software pode se tornar inadequado, caro de manter** e **arriscado para os negócios.**
- **Prática**: Manutenção constante é essencial para adaptar, corrigir e melhorar o sistema, prolongando sua vida útil e seu valor.

As 8 Leis de Lehman

1. Lei da Mudança Contínua	Um sistema de software em um ambiente real deve evoluir continuamente ou se tornará obsoleto, pois o ambiente e as necessidades dos usuários estão em constante mudança.
2. Lei da Complexidade Crescente	Conforme o software evolui, sua complexidade aumenta, a menos que se realize um esforço contínuo para reduzir essa complexidade.
3. Lei da Auto Regulação	A evolução de um sistema de software tende a ser autocontrolada e tem um padrão estatístico previsível.
4. Lei da Conservação da Estabilidade Organizacional	A quantidade de trabalho ou recursos que podem ser aplicados à manutenção do software é limitada. Assim, a taxa de evolução do sistema permanece aproximadamente constante ao longo do tempo.
5. Lei da Conservação da Familiaridade	Durante a evolução, os desenvolvedores devem manter uma certa familiaridade com o sistema para que as mudanças não tornem o sistema irreconhecível. Isso evita que o sistema se torne muito complexo ou diferente da estrutura original.
6. Lei do Crescimento Contínuo	Para manter a satisfação dos usuários, o sistema deve ser continuamente expandido com novas funcionalidades.
7. Lei da Declinação da Qualidade	A qualidade percebida do software tende a decair ao longo do tempo, a menos que sejam feitos ajustes e melhorias proativas para lidar com o desgaste natural e a evolução das necessidades.
8. Lei do Feedback Organizado	Um sistema de software precisa de um feedback contínuo de seus usuários e desenvolvedores para guiar e refinar o processo de evolução.

Ciclo de Vida da Manutenção

1. Identificação do problema
2. Análise
3. Projeto
4. Implementação
5. Teste
6. Documentação

Fluxo de Manutenção Corretiva Exemplo Prático

1. **Identificação do Bug. Fonte**: Usuários, monitoramento ou testes detectam um problema em produção. **Análise Inicial**: Confirmar o erro e registrar o bug em um sistema de acompanhamento (ex.: Jira).
2. **Diagnóstico e Análise de Causa**. Investigar a causa raiz para entender o impacto e definir a prioridade da correção.
3. **Desenvolvimento da Correção**. Implementar a solução no ambiente de desenvolvimento. Testar a correção com foco no bug e nos possíveis impactos colaterais.
4. **Revisão e Aprovação**. Code Review e aprovação da correção por um revisor ou equipe.
5. **Implantação em Produção**. Realizar o deploy da correção no ambiente de produção, com monitoramento ativo.
6. **Monitoramento Pós-Deploy**. Verificar se a correção solucionou o problema e se não há novos problemas relacionados.

Importância da Refatoração para a Evolução do Sistema

O que é Refatoração?

- Processo de melhorar o código sem alterar seu comportamento externo, tornando-o mais legível, eficiente e fácil de manter.

Por que é Essencial?

- Facilita a Evolução, Reduz a Complexidade, Melhora a Performance, Aumenta a Qualidade.

Em Resumo: Refatorar é um investimento na saúde e na longevidade do software.

Ferramentas e Práticas para Análise e Qualidade de Código

SonarQube

- Ferramenta de análise de código que identifica problemas de qualidade, como bugs, vulnerabilidades, débitos técnicos e complexidade. **Benefícios**: Ajuda a manter o código limpo e seguro, fornecendo métricas e insights para refatoração e melhorias.

Práticas Comuns

- **Code Review**: Revisão entre pares para encontrar e corrigir problemas antes do deploy.
- **Automated Testing**: Testes automatizados para garantir que novas alterações não introduzam erros.
- **Linting**: Verificação automática de padrões de codificação, que promove legibilidade e consistência.

Exemplo de linting: ESLint
Software para melhorar código JavaScript

Vamos supor que temos o seguinte código JavaScript, que funciona, mas não segue boas práticas:

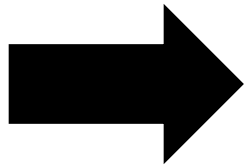
```
function calculateSum(a, b){
  return a + b
}
```

Exemplo de linting: ESLint
Software para melhorar código JavaScript

Ao aplicar o ESLint, recebemos sugestões, como:

- **Ponto e vírgula ausente**: O ESLint recomenda adicionar ponto e vírgula ao final de cada linha.
- **Espaçamento padrão**: Pode sugerir que haja um espaço entre function e as chaves ou parêntese para legibilidade.
- Após aplicar as correções sugeridas:

```
function calculateSum(a, b){
  return a + b
}
```



```
function calculateSum(a, b) {
  return a + b;
}
```

Case Real:
Netflix e a Migração para Microserviços

- **Cenário**: Em 2008, a Netflix tinha uma arquitetura monolítica que não suportava o crescimento rápido de usuários e causava quedas frequentes do sistema.
- **Desafios**: Sistema difícil de escalar. Erros impactavam a plataforma inteira, comprometendo a experiência do usuário.
- **Solução**: A Netflix começou a migrar para uma arquitetura de microsserviços, separando cada função em serviços independentes.

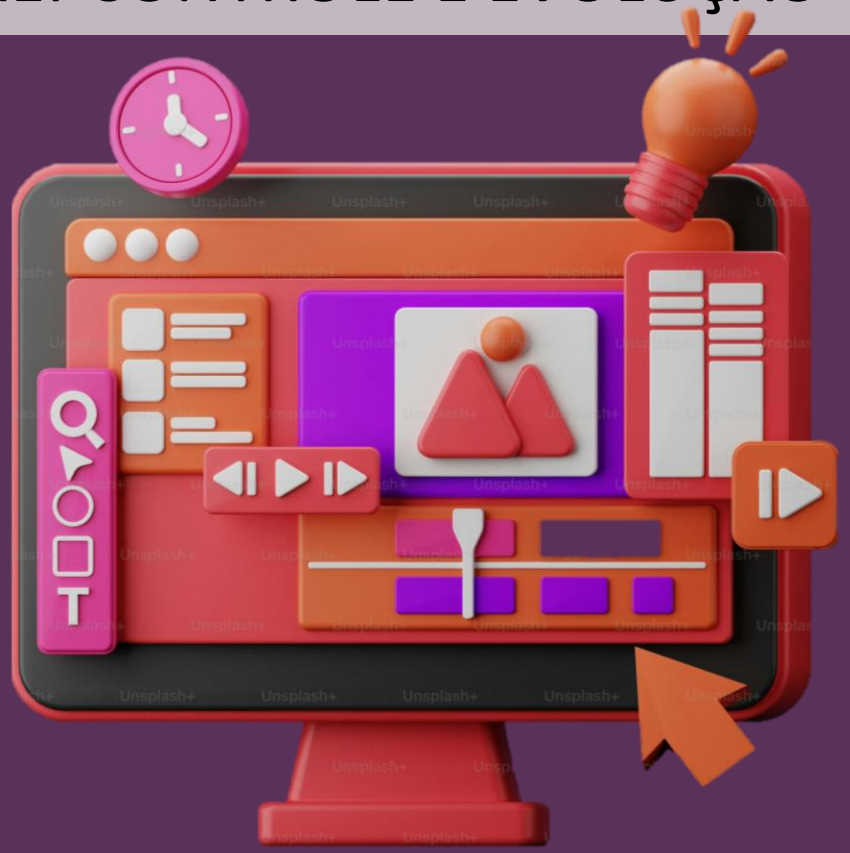
Resultados:

- **Alta Disponibilidade**: Redução significativa de falhas.
- **Escalabilidade**: Suporte ao crescimento global.
- **Inovação Rápida**: Liberdade para desenvolver e atualizar partes da plataforma de forma independente.

Lista de Exercícios

ENGENHARIA DE SOFTWARE

AULA 11 –SUSTENTABILIDADE DO SOFTWARE: CONTROLE E EVOLUÇÃO



WALTER TRAVASSOS SARINHO

@PROF.WALTERTRAVASSOS

WALTER.TRAVASSOS@SECTRAS.EDU.BR