Desenvolvimento de Sistemas

Aula 04 - WorkFlow e GitHub

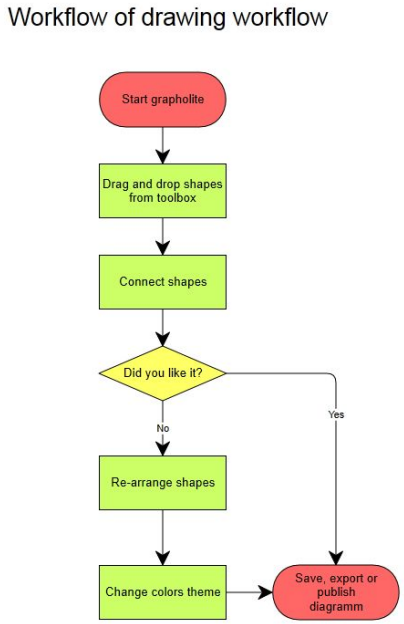
Lorrany B A Marim

SENAI

WorkFlow em TI

Workflow = o fluxo de trabalho “oficial” para transformar uma ideia em software entregue

- Em empresas, software não nasce “pronto”: ele passa por **etapas controladas** para reduzir erro e aumentar qualidade.
- Workflow define **como o time trabalha**, por exemplo:
  - Onde o código fica (repositório Git)
  - Como mudanças entram (branch/PR)
    Como validamos qualidade (revisão + testes)
  - Como liberamos versões (homologação → produção)
    O objetivo é garantir **controle, rastreabilidade, qualidade e segurança**.

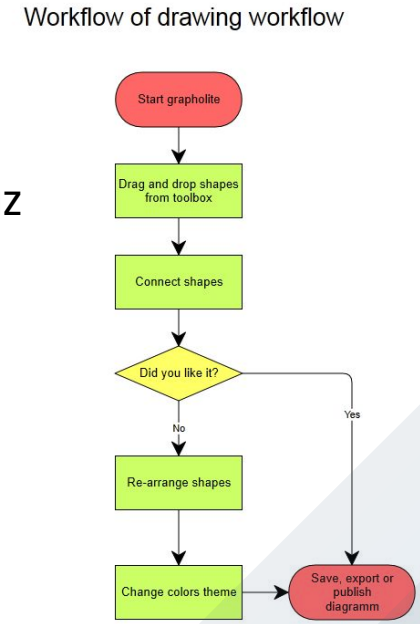


WorkFlow Focado em Construção

Construção → Validação → Teste → Homologação

Um fluxo típico (simplificado)

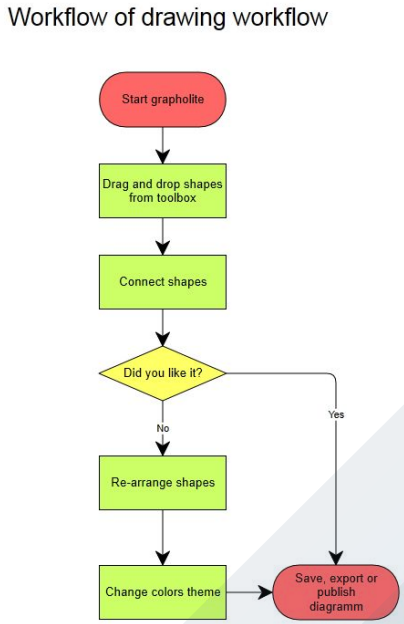
- **Construção (Build/Dev):**
  - Desenvolvedor cria uma branch, implementa, faz commits.
- **Validação (Quality Gate):**
  - Revisão de código (code review) + padrões do projeto + lint/format.
- **Teste (Automação + Manual quando necessário):**
  - Testes unitários, integração, e2e; relatórios de falhas.



WorkFlow Focado em Construção

- **Homologação (HML/Staging):**
  - Versão candidata é testada “como se fosse produção”.
  - Se aprovado, vira release para produção.

Ideia-chave: **Git é a “linha do tempo” que liga todas essas etapas**, porque registra mudanças e permite auditar o caminho do código.



Como empresas usam o Git no dia a dia

Git não é só “salvar código”: é coordenação do time

- **Branches** para separar trabalho (feature, bugfix, hotfix).
- **Pull Request (PR)** para pedir revisão e discutir mudanças.
- **Commits pequenos e claros** para rastrear o que mudou e por quê.
- **Tags/Releases** para marcar versões (ex.: v1.2.0).
- **Integração com pipelines (CI/CD):**
  - Ao abrir PR ou dar push, rodam testes automaticamente.

- **Resultado:** menos “mistero” no código e mais previsibilidade de entrega.

Controle de Versão

O fim do
“Trabalho_Final_AgoraVai_V7_de_verdade.zip”

- Guardar cópias manuais cria:
  - Confusão (qual é a versão certa?)
  - Risco de perda (pendrive, e-mail, pasta errada)
  - Retrabalho (misturar arquivos vira caos)

- Controle de versão resolve porque:
  - Mantém **histórico completo**
  - Permite **voltar no tempo**
  - Facilita **colaboração sem sobrescrever**
    Registra **quem fez o quê** (autoria)

O que é Git e GitHub

Dois nomes parecidos, funções diferentes

- **Git:** ferramenta que controla o histórico do projeto (no seu computador).
- **GitHub:** plataforma na nuvem para hospedar repositórios Git.
- **Local vs Remoto**
  - **Local:** sua máquina (onde você codifica).
  - **Remoto:** nuvem (backup, colaboração, portfólio).

A Ideia de Commit (Checkpoint)

Commit é um ponto de salvamento com mensagem

- Um commit registra:
  - O estado dos arquivos selecionados
    A data/hora, autor e mensagem

- Por que isso importa:
  - Você consegue rastrear evolução
  - Consegue desfazer mudanças com segurança
  - Consegue explicar o “motivo” (mensagem)

Configuração Inicial

Git precisa saber quem está assinando as mudanças

- Configure uma vez:
  - git config --global user.name "Seu Nome"
  - git config --global user.email "seu@email.com"
- Isso aparece no histórico e no GitHub (autoria).
- No mercado, isso é parte de **rastreabilidade** e **responsabilidade técnica**.

As 3 Áreas do Git

O ciclo de vida do arquivo

- **Working Directory:** onde você edita (arquivo pode estar “untracked”).
- **Staging Area:** área de preparação (o que vai entrar no próximo commit).
- **Repository (.git):** histórico definitivo (onde o commit mora).
- Fluxo mental:
  - **Editar** → **selecionar** (staging) → **salvar no histórico** (commit)

Comandos Essenciais (1)

Começando e entendendo o estado do projeto

- git init
  - Transforma uma pasta em repositório Git (cria a pasta .git).
- git status
  - Mostra o que está modificado, novo, pronto para commit, etc.
- git add .
  - Envia **tudo** para staging (preparar para o commit).
- git add nome_do_arquivo
  - Envia **apenas um arquivo** para staging.

Comandos Essenciais (2)

Salvando e vendo a linha do tempo

- git commit -m "mensagem clara"
  - Cria o checkpoint do que está no staging.

- Como escrever mensagem boa:
  - Diga o que mudou: **“Adiciona tela de login”**
  - Evite: **“ajustes”, “coisas”, “teste”**

- git log
  - Mostra histórico: commits, autores, datas e mensagens.

Enviando um Projeto para o GitHub

Backup + compartilhamento + portfólio

- Criar repositório no GitHub (vazio, sem “bagunçar” o começo).
- Conectar local ao remoto:
  - git remote add origin LINK_DO_REPOSITORIO
- Enviar commits para a nuvem:
  - git push -u origin main
- Padrão atual de branch principal: **main**
  - Se necessário: git branch -M main

GitHub Desktop

Git sem trauma de terminal (principalmente em laboratório/aula)

- Interface visual para:
  - Ver mudanças por arquivo/linha
  - Fazer commit com segurança
  - Dar push/pull sem decorar comandos

- Ideal para:
  - Iniciantes
  - Ambientes com restrição de terminal (labs, rede, permissões)
  - Aulas com foco em fluxo e prática

Instalação do GitHub Desktop Windows

Passo a passo (simples e direto)

- Baixe e instale o **GitHub Desktop**
- Abra o programa e faça login na sua conta GitHub
- Configure:
  - Nome e e-mail (o Desktop normalmente já orienta)
    Verifique se o **Git** está disponível:
    - O GitHub Desktop costuma instalar/configurar dependências automaticamente

Usando o do GitHub Desktop

Escolha o que combina com seu cenário

- **A) Clone Repository (clonar)**
  - Quando o repositório já existe no GitHub
  - Baixa para o computador e conecta automaticamente

- **B) Create New Repository (criar do zero)**
  - Cria um repo local e depois publica no GitHub

- **C) Add Existing Repository (adicionar existente)**
  - Quando você já tem uma pasta com projeto e quer “transformar em Git”

Fluxo Básico no GitHub Desktop

O mesmo ciclo do Git, só que visual

- Faça alterações no projeto (código/arquivos).

- No Desktop:
  - Veja a lista de arquivos alterados
  - Revise o diff (o que mudou)
  - Escreva mensagem de commit (clara)
  - Clique em **Commit to main** (ou sua branch)

- Depois:
  - Clique em **Push origin** para enviar ao GitHub

Branch + Pull Request

Jeito mais comum de trabalhar em time

- Crie uma branch (ex.: feature/login).
- Faça commits nessa branch.
- Dê push.
- Abra um **Pull Request** no GitHub:
  - Explica o que foi feito
  - Pede revisão
  - Dispara testes automáticos (quando existe CI)

- Só depois de aprovado, entra na branch principal.

Boas Práticas e Erros Comuns

Para evitar dor de cabeça

- Antes de começar a mexer:
  - **Pull** para trazer atualizações do remoto.

- Commits:
  - Pequenos e frequentes
  - Mensagens objetivas

- Sempre use status (ou o equivalente visual no Desktop) para não “se perder”.

Boas Práticas e Erros Comuns

- Use .gitignore para não enviar:
  - pastas de build, cache, node_modules, arquivos locais, etc.

- Leia erros com calma:
  - Errar faz parte do trabalho — **interpretar erro é habilidade profissional**.

Git e Trabalho em Equipe

Um caminho simples e realista

- **Local (PC do dev):** onde cada membro codifica e testa rápido.
- **Repositório remoto (GitHub):** onde o time integra e revisa mudanças.
- **Ambiente DEV:** versão integrada para “ver funcionando” com frequência.
- **Ambiente TESTE / HML (QA/Staging):** validação mais completa antes de liberar.
- **Produção (PROD):** versão final para o usuário.

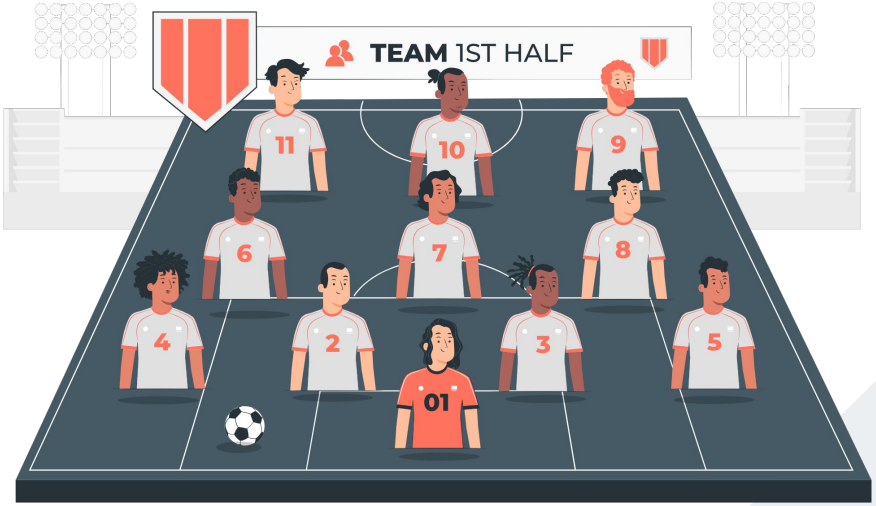
Regra de ouro: ninguém envia direto para produção. O código “sobe” por etapas.

Papéis do Time

Dev (todos): implementa, testa localmente, cria PR e corrige feedback.

Revisor (um colega/lead): revisa PR, pede ajustes, aprova.

Responsável pela release (pode ser o professor ou um aluno líder): decide quando está pronto para ir para HML e PROD.



Estrutura Simples de “Branches”

Branchs com propósito claro

- **main:** sempre estável (o que poderia ir para produção).
- **develop:** integração do dia a dia (o que vai para o ambiente DEV).
- **feature/**nome-curto: novas funcionalidades (um membro por tarefa).
- **hotfix/**nome-curto: correção urgente em produção (se existir PROD).

Em projeto escolar, usar **main + develop + feature/** já resolve 90% do fluxo.

Regra para não dar conflito

Antes de começar a codar

1. Atualize seu projeto:

- git checkout develop
- git pull origin develop

2. Crie sua branch da tarefa:

- git checkout -b feature/login

3. Só trabalhe dentro da sua branch (não em develop/main).

Passo a Passo

“eu terminei minha parte, e agora?”
Quando um membro quer enviar sua tarefa

1. Verifique o estado:
- git status

2. Adicione e commite (mensagens claras):
- git add .
- git commit -m "Adiciona validação do formulário de login"

3. Envie sua branch para o GitHub:
- git push -u origin feature/login

4. Abra um **Pull Request (PR)**:
- **Base:** develop — **Compare:** feature/login
- Descreva: o que fez, como testar, prints se tiver

Passo a Passo

Comportamento do time

- Ninguém “puxa e mistura” código do colega manualmente.

- Os colegas devem:
  - Ler o PR (arquivos alterados + diffs)
  - Rodar o projeto localmente se necessário
  - Comentar pontos de melhoria (clareza, bugs, padrão)
  - Aprovar quando estiver ok

Se o PR quebra o projeto: não aprova. Primeiro corrige.

Passo a Passo

Quando o PR é aprovado

- Faz **merge** do PR em develop (via GitHub).
- Agora develop representa a versão integrada do time.
- O deploy/uso em **ambiente DEV** normalmente vem daqui:
  - “DEV mostra o que está acontecendo agora no time”.

Depois do merge: todos devem atualizar:

- git checkout develop
- git pull origin develop

Atualização da Branches

Se sua tarefa demora alguns dias

- De tempos em tempos, traga as novidades do develop para sua branch:

Opção A (simples, com merge):

- git checkout feature/minha-tarefa
- git merge develop

Opção B (mais avançada, rebase – só se a turma já estiver segura):

- git rebase develop

Ideia: conflito pequeno agora é melhor que conflito gigante no fim.

Ambientes

O que valida cada ambiente

- **Local:** teste rápido do dev (rodar, clicar, console, casos básicos).
- **DEV:** integração do time (ver se as partes se encaixam).
- **TESTE / HML (QA/Staging):**
  - checagem mais rigorosa (fluxos completos, regras, dados)
  - testes manuais + automatizados quando existirem
- **PROD:** só entra o que passou por HML com segurança.

Em Teste/Homologação

Criando um “candidato” a versão estável

- Quando o develop está bom, cria-se uma etapa para estabilizar:
  - pode ser uma branch release/x.y (opcional)
  - ou simplesmente preparar um PR de develop → main

Fluxo simples (turma):

1. Abrir PR: **base main**, compare develop
2. Rodar validações (checklist)
3. Se aprovado: merge em main → pronto para HML/PROD

Em Produção

E se der problema depois de publicado?

- Correção urgente:

1. Criar branch a partir de main:
    - git checkout main
    - git pull origin main
    - git checkout -b hotfix/erro-login
2. Corrigir, commit, push, PR para main
3. Depois de mergear em main, levar a correção para develop também
    - para o time não “perder” a correção no fluxo diário

Checklist do que fazer (SEMPRE)

1. Antes de codar: **pull no develop**
2. Uma tarefa = **uma branch feature/**
3. Sempre integrar via **PR**, nunca “jogar arquivo no colega”
4. PR precisa ter:
   a. descrição do que foi feito
   b. como testar
   c. prints (se for tela)

Após merge em develop: **todo mundo dá pull**
Em caso de conflito: resolver com calma e comunicar o time


Obrigado!
SENAI