Atividade Prática: Code Review e Colaboração com Git & GitHub

Esta atividade simula um fluxo de trabalho de desenvolvimento de software em equipe, focando na colaboração através do Git, na prática de revisão de código (Code Review) e na responsabilidade compartilhada pela qualidade do código.

Objetivo Principal: Desenvolver habilidades técnicas e interpessoais essenciais para o trabalho em equipe, como:

- **Fluxo de Trabalho com Git:** Utilizar branches, commits, e Pull Requests (PRs) de forma eficaz.
- **Revisão de código:** Aprender a fornecer e receber feedback construtivo sobre o código de colegas.
- **Qualidade de código:** Entender a importância de escrever um código claro, eficiente e legível.
- **Comunicação técnica:** Praticar a descrição de mudanças e a argumentação de decisões técnicas.

Formato da Equipe:

- Até 4 integrantes.
- **1 Mantenedor(a) da `main`:** Responsável por criar o repositório, revisar todos os Pull Requests e garantir a qualidade do código que entra na branch principal (`main`).
- **Até 3 Desenvolvedores(as):** Cada um responsável por resolver um problema lógico em uma branch separada.

Explicação da atividade

O ciclo de vida desta atividade espelha o que acontece em empresas de tecnologia todos os dias. Um desenvolvedor não simplesmente escreve um código e o envia para a versão final do produto. Em vez disso, ele trabalha em uma "cópia" isolada do projeto (uma branch) e quando termina, ele propõe que sua alteração seja incluída na base de código principal através de um Pull Request (PR).

Esse PR é um pedido formal para "puxar" (pull) as suas alterações para a branch `main`. Antes que isso seja aprovado, outros membros da equipe (especialmente o líder técnico ou mantenedor) revisam o código. Eles procuram por:

- **Bugs:** O código funciona como esperado? Cobre casos extremos?
- **Clareza:** O código é fácil de entender? Os nomes das variáveis e funções são bons?
- **Boas Práticas:** O código segue as convenções do projeto/linguagem? Existe uma forma mais simples ou eficiente de fazer o mesmo?

Esse processo, chamado Code Review, é fundamental para manter a alta qualidade do software, compartilhar conhecimento entre a equipe e encontrar erros antes que eles cheguem aos usuários.

Problemas lógicos a serem resolvidos

Aqui estão três problemas de dificuldade intermediária. Eles exigem lógica, mas não estruturas de dados complexas.

Desenvolvedor 1: Verificador de anagramas

- **Tarefa:** Crie uma função `sao_anagramas(string1, string2)` que receba duas strings e retorne `True` se elas forem anagramas uma da outra, e `False` caso contrário.
- **Definição:** Um anagrama é uma palavra ou frase formada pelo rearranjo das letras de outra palavra ou frase. Espaços e diferenças entre maiúsculas e minúsculas devem ser ignorados.
- **Exemplos:**
  - `sao_anagramas("amor", "roma")` → `True`
  - `sao_anagramas("A gentleman", "Elegant man")` → `True`
  - `sao_anagramas("gato", "cabra")` → `False`

Desenvolvedor 2: Cifra de César

- **Tarefa:** Crie uma função `cifra_de_cesar(texto, deslocamento)` que aplique a Cifra de César a uma string.
- **Definição:** A Cifra de César é uma técnica de criptografia simples onde cada letra do texto original é substituída por outra letra que se encontra a um número fixo de posições (deslocamento) à frente no alfabeto. A função deve preservar maiúsculas/minúsculas e não alterar caracteres que não sejam letras (números, espaços, pontuação). O alfabeto deve ser considerado circular (depois de 'z' vem 'a').
- **Exemplos:**
  - `cifra_de_cesar("abc", 2)` → `"cde"`
  - `cifra_de_cesar("xyz", 3)` → `"abc"`
  - `cifra_de_cesar("Ataque ao Amanhecer!", 5)` → `"Fyfvzj ft Frfsmjhmjw!"`

Desenvolvedor 3: Encontrar a maior palavra em uma frase

- **Tarefa:** Crie uma função `encontrar_maior_palavra(frase)` que receba uma string contendo uma frase e retorne a maior palavra encontrada nela.
- **Definição:** As palavras são separadas por espaços. Se houver duas ou mais palavras com o mesmo comprimento máximo, a função deve retornar a primeira que aparecer. A pontuação anexada às palavras (como vírgulas ou pontos finais) deve ser ignorada na contagem do comprimento.
- **Exemplos:**
  - `encontrar_maior_palavra("O rato roeu a roupa do rei de Roma")` → `"roupa"`
  - `encontrar_maior_palavra("A jornada de mil milhas começa com um único passo.")` → `"jornada"`
  - `encontrar_maior_palavra("Seja forte e corajoso")` → `"forte"` (retorna a primeira de tamanho 5)

Roteiro da Atividade: Passo a Passo

Fase 1: Preparação (Responsabilidade do Mantenedor)

1. **Criar o Repositório:** O(A) Mantenedor(a) cria um novo repositório no GitHub. Adicione um `README.md` e um `.gitignore` para identificar o grupo e a linguagem escolhida.
2. **Adicionar Colaboradores:** Vá em `Settings > Collaborators` e adicione os outros membros da equipe. Eles precisarão aceitar o convite por e-mail.
3. **Estrutura Inicial:** Crie um arquivo principal (ex: `solucoes.py`) com as assinaturas das três funções, mas sem a implementação.
   Python

```python
def sao_anagramas(string1, string2):
    # TODO: Implementar a lógica
    pass

def cifra_de_cesar(texto, deslocamento):
    # TODO: Implementar a lógica
    pass

def valida_cpf(cpf_string):
    # TODO: Implementar a lógica
    pass
```

4. **Proteger a Branch `main`:** Em `Settings > Branches`, adicione uma "branch protection rule" para a `main`. Marque a opção **"Require a pull request before**

merging". Isso impede que qualquer pessoa (incluindo o mantenedor) envie código diretamente para a `main`, forçando o uso de PRs.

5. **Commit e Push Inicial:** Adicione (`git add .`), comite (`git commit -m "setup inicial do projeto"`) e envie (`git push origin main`) essa estrutura inicial para o GitHub.

Fase 2: Desenvolvimento (Responsabilidade dos Desenvolvedores)

1. **Clonar o Repositório:** Todos os membros da equipe devem clonar o repositório para suas máquinas locais: `git clone <URL_DO_REPOSITORIO>`.
2. **Distribuir Tarefas:** A equipe decide qual desenvolvedor(a) ficará com qual problema.
3. **Criar uma Branch:** Cada desenvolvedor(a) cria sua própria branch para trabalhar. O nome da branch deve ser descritivo.

```
# Exemplo para o desenvolvedor do problema de anagrama
git checkout -b feat/funcao-anagrama
```

4. **Resolver o Problema:** Implemente a lógica da sua função no arquivo, ex: `solucoes.py`.
5. **Commitar o Trabalho:** Faça commits claros e concisos à medida que avança

```
git add solucoes.py
git commit -m "feat: implementa logica da funcao de anagrama"
```

6. **Enviar a Branch para o GitHub:**

```
git push origin feat/funcao-anagrama
```

Fase 3: Code Review (Responsabilidade de TODOS)

1. **Abrir o Pull Request (PR):** O(A) desenvolvedor(a) vai até a página do repositório no GitHub. Um aviso para criar um PR a partir da sua nova branch aparecerá. Clique nele.
    - **Título do PR:** Coloque um título claro (ex: "Implementa a função de verificação de anagramas").
    - **Descrição:** Escreva uma breve descrição do que foi feito, como testar e se há algo que os revisores devam prestar atenção especial.

2. **Revisar o Código:**
    - O Mantenedor deve revisar o PR.
    - Vá na aba "Files changed" do PR para ver o código.
    - Para deixar um comentário, clique no `+` que aparece ao lado de uma linha de código.
    - O que observar?
        - **Funciona?** O código parece resolver o problema corretamente?

        - **Clareza:** Os nomes de variáveis são bons? A lógica é fácil de seguir?
        - **Casos Extremos (Edge Cases):** O que acontece se a função receber uma string vazia? Ou números?
        - **Estilo:** O código está bem formatado?

3. **Discutir e Melhorar:**
    - O autor do PR deve responder aos comentários, explicar suas decisões ou fazer as alterações solicitadas.
    - Para enviar correções, basta fazer novos commits na mesma branch e dar `git push`. O PR será atualizado automaticamente.
    - A discussão continua até que todos estejam satisfeitos com o código.

Fase 4: Integração (Responsabilidade do mantenedor)

1. **Aprovar o PR:** Quando o código estiver bom, o mantenedor deve aprovar formalmente o PR no GitHub.
2. **Fazer o "Merge":** O(A) Mantenedor(a) é o único que pode clicar no botão "Merge pull request".

Fase 5: Sincronização e finalização

1. **Atualizar a `main` Local:** Após cada merge, todos os membros da equipe devem atualizar sua branch `main` local para receber as novas funcionalidades.

```
git checkout main
git pull origin main
```

2. **Repetir o Processo:** Repita as Fases 3 e 4 para os PRs dos outros desenvolvedores.
3. **Debriefing:** Ao final, quando todas as funcionalidades estiverem na `main`, a equipe deve se reunir para discutir:
    a. O que funcionou bem no processo?
    b. Qual foi a maior dificuldade?
    c. Qual feedback de code review foi mais útil?
    d. O que vocês fariam diferente da próxima vez?

Guia Prático: comandos essenciais do git para o dia a dia

Pense no Git como um sistema inteligente para salvar "versões" do seu trabalho. O repositório no GitHub é a "fonte da verdade" central, como uma biblioteca principal. O seu computador guarda uma cópia local, onde você faz sua mágica.

Estes comandos são as ferramentas que você usa para interagir com a sua cópia local e sincronizá-la com a biblioteca principal.

1. `git clone`

- **O que faz?** Cria uma cópia fiel (um clone) de um repositório remoto (do GitHub, por exemplo) na sua máquina local. É o primeiro passo para começar a trabalhar em um projeto que já existe.
- **Analogia:** É como ir à biblioteca e tirar uma fotocópia completa de um livro para poder fazer anotações e trabalhar nele em casa, sem alterar o original.
- **Como usar?** git clone <URL_DO_REPOSITORIO>
- **Exemplo prático:**

  git clone https://github.com/equipe-incrivel/projeto-final.git

O Ciclo de Salvar e Enviar: `add`, `commit` e `push`

Esses três comandos trabalham juntos. Pense neles como um processo de 3 etapas para enviar suas alterações de casa (seu PC) para a biblioteca (GitHub).

Analogia do Envio:

1. `git add`: Você seleciona os arquivos que terminou de editar e os coloca em uma caixa de envio (chamada de "Staging Area").
2. `git commit`: Você fecha a caixa, lacra e cola uma etiqueta nela, descrevendo o que há dentro. A caixa está pronta para ser enviada, guardada em sua área de saída.
3. `git push`: O caminhão dos correios leva a sua caixa da sua área de saída para o armazém central (o repositório remoto).

2. `git add`

- **O que faz?** Adiciona as alterações que você fez nos arquivos à "Staging Area" (nossa "caixa de envio"). O Git agora está ciente dessas alterações e as está preparando para o próximo "pacote" (commit).
- **Como usar?**
    - Para adicionar um arquivo específico: git add <nome_do_arquivo>
    - Para adicionar todos os arquivos modificados de uma vez (muito comum):

      git add .

Exemplo prático:

git add solucoes.py

git add README.md

Ou, de forma mais rápida:

git add .

3. `git commit -m "sua mensagem"`

- **O que faz?** Pega tudo o que está na "Staging Area" (`git add`) e cria um ponto de salvamento permanente no seu histórico local. É um registro fotográfico do estado do seu projeto naquele momento. A parte `-m "..."` é a mensagem (a etiqueta da caixa), que deve ser clara e descritiva.
- **Como usar?**
  git commit -m "Uma mensagem clara sobre o que você fez"
- **Exemplo prático:**
  git commit -m "implementa a função de verificação de anagramas"

  Dica: Boas mensagens de commit são curtas e começam com um verbo no presente (ex: "corrige", "adiciona", "remove").

4. `git push`

- **O que faz?** Envia os seus commits (as caixas lacradas e etiquetadas) do seu repositório local para o repositório remoto (GitHub), atualizando a "biblioteca principal" com o seu trabalho.
- **Como usar?**
  git push <nome_do_remoto> <nome_da_branch>

  Normalmente, o remoto se chama `origin` e a branch é aquela em que você está trabalhando.

- **Exemplo prático:**
    - Se você está trabalhando na branch `main`: git push origin main
    - Se você está na branch da sua funcionalidade (como na atividade anterior): git push origin feat/funcao-anagrama

O Ciclo de Sincronização: `fetch` e `pull`

Estes comandos servem para você se atualizar com o trabalho que seus colegas enviaram para o repositório remoto.

5. `git fetch`

- **O que faz?** Baixa as "notícias" sobre todas as atualizações do repositório remoto, mas não as aplica ao seu trabalho local ainda. Ele apenas atualiza sua visão do que está acontecendo lá.
- **Analogia:** Você recebe as manchetes do jornal do dia. Você sabe quais são as notícias novas, mas ainda não leu os artigos e nem os incorporou ao seu conhecimento. É uma forma segura de "dar uma olhada" no que mudou.
- **Como usar?**
  git fetch
- **Exemplo prático:**

  Após dar um checkout para a branch que está, utilize o comando: git fetch

6. `git pull`

- **O que faz?** É a combinação de duas ações: ele primeiro faz um `git fetch` (baixa as notícias) e, em seguida, tenta automaticamente mesclar (`merge`) essas atualizações no seu código local. É a forma mais comum de manter seu projeto atualizado.
- **Analogia:** É como receber o jornal (`fetch`) e imediatamente ler todos os artigos e integrá-los às suas anotações (`merge`). É mais direto, mas você deve estar preparado para resolver conflitos se você e seu colega alteraram a mesma linha de código.
- **Como usar?**
  git pull <nome_do_remoto> <nome_da_branch>
- **Exemplo prático:**

  git pull origin main

  Este é o comando que você geralmente executa no início do dia para garantir que está trabalhando na versão mais recente do projeto.

Resumo do fluxo de trabalho diário

1. **Começando o dia:** Pegue as últimas atualizações do projeto.
    - `git pull origin main`
2. **Trabalhando em uma nova tarefa:** Crie sua própria branch.
    - `git checkout -b nome-da-nova-feature`
3. **Durante o trabalho:** Faça suas alterações no código.
4. **Salvando seu progresso:** Adicione e comite suas alterações.
    - `git add .`
    - `git commit -m "Mensagem clara do que foi feito"`
5. **Compartilhando com a equipe:** Envie sua branch para o GitHub para que outros possam ver (e para abrir um Pull Request).
    - `git push origin nome-da-nova-feature`