CENTRO DE TREINAMENTO DA TECNOLOGIA DA INFORMAÇÃO
SERVIÇO NACIONAL DE APRENDIZAGEM INDUSTRIAL
DESENVOLVIMENTO DE SISTEMAS

ELOIZA SIMÕES
LUIZ FILIPE
MATHEUS FERNANDES
RAFAEL SILAS

SA - MANUTENÇÃO DE SISTEMAS

2025
BELO HORIZONTE

ELOIZA SIMÕES
LUIZ FILIPE
MATHEUS FERNANDES
RAFAEL SILAS

SA - MANUTENÇÃO DE SISTEMAS

Trabalho apresentado ao SENAI – CTTI, como requisito para a conclusão da atividade/projeto "SA Manuntenção de Sistemas", da unidade curricular de Manuntenção de Sistemas. Instrutor: Rodolfo Clepf de Carvalho

2025
BELO HORIZONTE

RESUMO

O Documento aborda a importância da **manutenção de sistemas**, destacando os motivos pelos quais sistemas precisam de atualização constante, os impactos da falta de manutenção e um estudo de caso de manutenção corretiva.

Em seguida, apresenta um **checklist completo de atualização de sistemas** desenvolvidos com Visual Studio e Git, abordando desde o commit até a entrega do executável ao cliente. Também detalha o conceito de **controle de versão com Git**, explicando comandos práticos e a estrutura de branches usando o **modelo Git Flow** para organizar o desenvolvimento colaborativo de forma segura e eficiente.

Palavras-chave: Manutenção; Versionamento; Git Flow; Deploy; Correção.

1. Manutenção Corretiva

O que é:
É o processo de reparar ou substituir componentes de um sistema de computação após a ocorrência de uma falha, com o objetivo de restaurar sua funcionalidade

Quando é realizada:
A manutenção corretiva é realizada **após a entrega do sistema**, quando falhas começam a surgir durante o uso. Seu objetivo é **identificar, corrigir e documentar os erros** para que o sistema continue funcionando de forma adequada e confiável.

Levantamento do Bug:

Relato do Usuário: “Não consigo realizar login no seu sistema e não sei o motivo”.
Identificação do erro: sem retorno do erro no login.

1.1 Objetivo

Corrigir problemas detectados após a implantação, garantindo o funcionamento estável do sistema sem comprometer a experiência do usuário.

1.2 Práticas Recomendadas Após a Implantação

Monitoramento Contínuo

- Utilizar logs de erro e ferramentas de monitoramento para detectar falhas assim que ocorrerem.
- Observar comportamentos anormais como travamentos, lentidão ou falhas de login.

Registro de Erros

- Criar um sistema de chamados ou relatórios onde usuários possam registrar problemas encontrados.
- Coletar dados como horário, ação realizada e mensagens de erro.

Análise Técnica

- Investigar a causa raiz do erro antes de aplicar qualquer correção.
- Testar em ambiente isolado (homologação) para evitar novos problemas.

Correção do Erro

- Corrigir o código ou configuração responsável pela falha.
- Garantir que a correção não afete outras partes do sistema.

Testes Pós-Correção

- Executar testes automatizados ou manuais para verificar se o erro foi resolvido.
- Validar se outras funcionalidades não foram impactadas.

Atualização do Sistema

- Publicar a correção em ambiente de produção com segurança.
- Realizar backup antes da atualização, caso algo dê errado.

Documentação

- Registrar o erro, a solução aplicada, a data da correção e o responsável.
- Alimentar um histórico de manutenção para futuras consultas.

Feedback ao Usuário

- Informar ao usuário que relatou o problema que ele foi resolvido.
- Reforça a confiança no sistema e na equipe de manutenção.

2. Controle de Versão

O controle de versão é uma prática essencial no desenvolvimento de software, usada para registrar e acompanhar todas as alterações feitas no código de um projeto. Ele permite identificar quem fez cada modificação, quando foi feita e o que foi alterado, facilitando o trabalho em equipe, a correção de erros e o resgate de versões anteriores quando necessário.

Essa prática é especialmente útil em projetos em constante evolução, pois evita a perda de informações e melhora a organização do time. O sistema de controle de versão mais utilizado atualmente é o Git, que permite criar branches (ramificações) para desenvolver novas funcionalidades ou corrigir problemas sem afetar o código principal. Comandos como commit, push e pull permitem registrar mudanças e sincronizar com repositórios remotos, como GitHub, GitLab ou Bitbucket, promovendo uma colaboração eficiente entre os desenvolvedores.

3. Modelo de Versionamento com Git

Para organizar um projeto com controle de versões de forma eficiente, podemos utilizar um modelo de versionamento baseado em branches (ramificações). Um dos modelos mais utilizados é o Git Flow, que define uma estrutura clara para o desenvolvimento colaborativo e a entrega contínua de software.

Estrutura do Git
Na área de trabalho apertamos com o botão direito em > novo > pasta e colocamos o nome como gitflow-teste.

No Git Flow, usamos diferentes branches para cada etapa do desenvolvimento. Abaixo estão os principais tipos e suas funções:

3.1 git init

```
Silas@PC MINGW64 ~/OneDrive/Documentos/Area de Trabalho/gitflow-teste (master)
$ git init
Initialized empty Git repository in C:/Users/rafae/OneDrive/Documentos/Area de T
rabalho/gitflow-teste/.git/
```

Esse comando que inicializa um repositório Git vazio no diretório atual. Isso quer dizer que a pasta “gitflow-teste” agora está sendo monitorada pelo Git, e ele criou uma subpasta oculta chamada . git/ que armazena todas as informações de versionamento do projeto.

3.2 git checkout -b main

```
Silas@PC MINGW64 ~/OneDrive/Documentos/Area de Trabalho/gitflow-teste (master)
$ git checkout -b main
Switched to a new branch 'main'
```

Esse comando cria uma nova branch chamada “main” que versiona para ela imediatamente.
Essa branch estava em master (criada automaticamente ao rodar git init), mas decidiu seguir a convenção mais atual usada em muitos projetos, que é usar “main” como a branch principal.

3.3 git checkout -b develop

```
Silas@PC MINGW64 ~/OneDrive/Documentos/Area de Trabalho/gitflow-teste (main)
$ git checkout -b develop
Switched to a new branch 'develop'
```

Esse comando cria uma nova branch chamada “develop” que lá todos os testes e melhorias são desenvolvidas antes de serem integradas à “main” que será feito antes de lançar uma nova versão do projeto.

3.4 git checkout -b feature/teste-login

```
Silas@PC MINGW64 ~/OneDrive/Documentos/Area de Trabalho/gitflow-teste (develop)
$ git checkout -b feature/teste-login
Switched to a new branch 'feature/teste-login'
```

Esse comando criou uma nova branch de funcionalidade (feature) chamada teste-login. Essa branch foi criada a partir da develop, que é onde todas as features são derivadas. Trabalhar isoladamente em uma nova funcionalidade neste caso, o teste de login sem afetar o código da branch develop. Depois que essa feature estiver pronta e testada, ela será integrada de volta na develop.

3.5 echo "Essa é a funcionalidade de login." > login.txt

```
Silas@PC MINGW64 ~/OneDrive/Documentos/Area de Trabalho/gitflow-teste (feature/t
este-login)
$ echo "Essa é a funcionalidade de login." > login.txt
```

Esse comando cria um novo arquivo chamado login.txt contendo o texto: "Essa é a funcionalidade de login." O símbolo “>” serve para redirecionar a saída do comando para um arquivo criado ou modificado. Um novo arquivo login.txt foi criado dentro da branch feature/teste-login, e agora esse arquivo pode agora ser versionado com Git sendo adicionado e commitado.

3.6 git add login.txt

```
Silas@PC MINGW64 ~/OneDrive/Documentos/Area de Trabalho/gitflow-teste (feature/t
este-login)
$ git add login.txt
warning: in the working copy of 'login.txt', LF will be replaced by CRLF the nex
t time Git touches it
```

Esse comando adiciona o arquivo login.txt à "staging area" do Git, ou seja, está dizendo ao Git: "Esse arquivo está pronto para ser commitado." e o aviso é só uma notificação sobre o formato de quebra de linha: CRLF + LF é o padrão no Windows. Como está no Windows, o Git está avisando que vai converter as quebras de linha para o formato Windows no próximo commit.

3.7 git commit -m "feat: adiciona arquivo de teste para a feature de login"

```
Silas@PC MINGW64 ~/OneDrive/Documentos/Area de Trabalho/gitflow-teste (feature/t
este-login)
$ git commit -m "feat: adiciona arquivo de teste para a feature de login"
[feature/teste-login (root-commit) 6a99c41] feat: adiciona arquivo de teste para
 a feature de login
 1 file changed, 1 insertion(+)
 create mode 100644 login.txt
```

Esse comando faz o commit das mudanças que estavam na staging area no caso, o login.txt no histórico do Git. A flag “-m” permite adicionar uma mensagem de commit diretamente na linha de comando. A mensagem é só uma convenção de commits semânticos como no Git Flow e Conventional Commits, só coloquei ela como exemplo de uma descrição. De uma alteração de um commit único (6a99c41) que identifica essa alteração.

3.8 git merge feature/teste-login

```
Silas@PC MINGW64 ~/OneDrive/Documentos/Area de Trabalho/gitflow-teste (feature/t
este-login)
$ git merge feature/teste-login
Already up to date.
```

Esse comando tenta juntar as alterações da feature/teste-login com a branch onde você está, que no caso é a hotfix/corrige-login. Como não tinha nada novo para juntar, o Git mostrou “Already up to date”. Isso significa que tudo que tinha na feature já está nessa branch, então ele não precisou fazer nada.

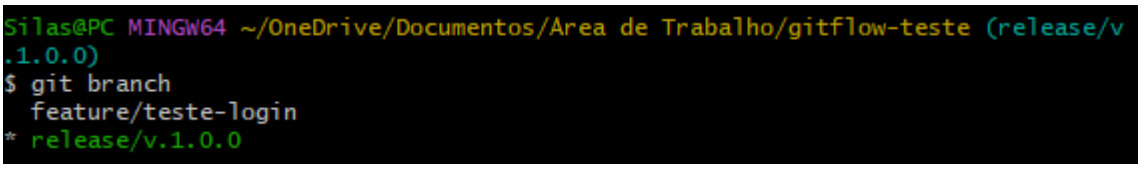
3.9 git checkout -b release/v.1.0.0

```
Silas@PC MINGW64 ~/OneDrive/Documentos/Area de Trabalho/gitflow-teste (feature/t
este-login)
$ git checkout -b release/v.1.0.0
Switched to a new branch 'release/v.1.0.0'
```

```
Silas@PC MINGW64 ~/OneDrive/Documentos/Area de Trabalho/gitflow-teste (release/v
.1.0.0)
$git chekout main git checkour -b hotfix/corrige-login
git: 'chekout' is not a git command. See 'git --help'.
```

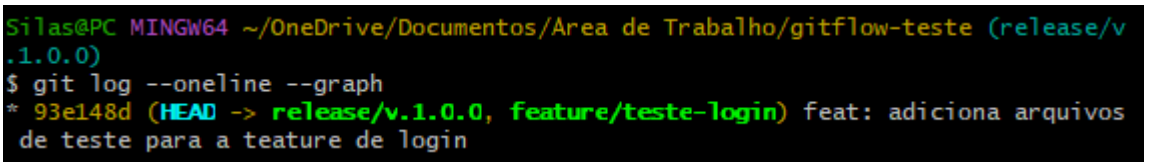
Aqui também são utilizadas as branches release e hotfix. A branch release é criada a partir da develop quando o projeto está pronto para ser lançado, permitindo os ajustes finais antes da publicação. Já a branch hotfix é usada para corrigir erros críticos diretamente na produção e é criada a partir da main. Ambas, ao serem finalizadas, devem ser mescladas de volta tanto na main quanto na develop para manter o projeto sincronizado.

3.10 git branch



Esse comando registra as alterações feitas no projeto de forma clara, usando o Git. Para isso, fazemos commits com mensagens que expliquem exatamente o que foi modificado. Por exemplo: feat: adicionar botão de cadastro, fix: corrigir erro no layout. Usar mensagens descritivas facilita o entendimento do que foi feito em cada etapa e ajuda na organização do projeto.

3.11 git log –oneline –graph



Após configurar o projeto local com o Git Flow, é importante criar um repositório remoto no GitHub para armazenar o código na nuvem e permitir o trabalho em equipe. Por fim, envia-se o conteúdo local com git push -u origin main (ou a branch desejada). Esse processo garante versionamento remoto, backup e colaboração entre os membros do grupo.

4 Checklist de Atualização - Projeto Visual Studio com Git (Commit) e Executável

4.1 Controle de Versão (Git via Visual Studio)

- ☐ O código-fonte foi atualizado e testado localmente antes de qualquer commit?
- ☐ O commit foi feito com **mensagem clara e semântica** (ex: `fix: tratada exceção ao carregar formulário`)?
- ☐ A branch usada para desenvolvimento está correta? (ex: `dev`, `hotfix`, `main`)

- [ ] O commit foi enviado ao GitHub com sucesso (`push`) e está visível no repositório remoto?
- [ ] Foi criada uma **tag de versão** (ex: `v1.2.0`) antes da publicação?

4.2 Geração do Executável

- [ ] O projeto foi publicado via **Visual Studio → Publish** com o modo:
  - [ ] `Self-contained` (sem precisar instalar o .NET no cliente)
  - [ ] Arquitetura correta (`win-x64` ou `win-x86`)
  - [ ] Opção **"Produce single file"** habilitada (um único `.exe`)?
- [ ] A pasta `publish` foi testada localmente antes de empacotar?
- [ ] O `.exe` final está funcional e apresenta as novas correções ou funcionalidades?

4.3 Empacotamento e Entrega

- ☐ A pasta `publish` foi compactada em `.zip` para envio ao cliente?
- ☐ Foi criada uma **pasta separada com a versão anterior** para possível reversão?
- ☐ O nome do arquivo ZIP contém a **versão** (ex: `SistemaTechSolutions_v1.2.0.zip`)?

4.4 Instruções para o Cliente

O cliente foi orientado a:

- ☐ Fazer **backup da pasta do sistema atual**
- ☐ **Excluir a versão antiga**
- ☐ **Descompactar** a nova versão no mesmo local
- ☐ Rodar o novo `.exe` diretamente (sem necessidade de instalar .NET)

4.5 Testes Pós-Atualização

- ☐ O sistema foi testado após a troca de versão?
- ☐ As **mensagens de erro** aparecem corretamente com `MessageBox` em caso de falha?
- ☐ O sistema loga erros (em `.txt` ou outro mecanismo) caso ocorra falha?

4.6 Reversão Rápida (Rollback)

- [ ] A versão anterior está disponível em arquivo `.zip` com identificação (ex: `v1.1.0.zip`)?
- [ ] As instruções para restaurar a versão anterior foram fornecidas ao cliente?
- [ ] O backup foi feito antes da substituição?