Ramificação e Manutenção

Aula 08 — Mexer sem quebrar o que funciona

Aula 08 · 16/11/2026 · Indicadores 2, 3

UniSenac Centro Universitário RS
Senac
Fecomércio
IFEP
Sesc
Sindicatos Empresariais

O mapa de hoje

- **Bloco 1** — duas mudanças ao mesmo tempo: `branch` e `merge`
- **Bloco 2** — o conflito, e quem decide uma mudança
- **Bloco 3** — a versão estável, a vida do sistema e a política de recuperação
- **Bloco 4** — mão na massa: um defeito corrigido do jeito certo, e a versão 1.0

> Abram o repositório do grupo no VS Code e o `locadora-historico` da aula passada. Os dois vão ser usados hoje.

Retomando a Aula 07

- O `git revert` apaga o commit errado?
- A senha foi apagada no commit seguinte. Por que não basta?
- O que o `.gitignore` tem de ter **antes** do primeiro commit?

Bloco 1

Duas mudanças ao mesmo tempo

A Ana conserta, o Bruno inventa

No `multa.py` da locadora, na mesma semana:

- A Ana vai consertar a fronteira — o `>` que voltou no commit do desconto.
- O Bruno vai criar o **teto da multa**: nunca mais que R$ 50,00. Leva três dias, e no meio do caminho o código dele não passa nos testes.

Se os dois gravam direto na versão principal, o que acontece com o sistema do balcão na terça, com a regra do Bruno pela metade?

`branch` — uma linha de desenvolvimento separada

- A `main` é a versão que **funciona**. Ninguém trabalha nela pela metade.
- Cada mudança ganha uma **branch**: uma cópia da linha de desenvolvimento, que anda sozinha.
- Quando a mudança está pronta e a suíte está verde, ela volta para a `main`: o **merge**.

O que se quer	Painel do VS Code	Git Bash
criar e ir para uma branch	clicar no nome da branch, embaixo à esquerda → **Create new branch**	`git switch -c corrige-fronteira`
voltar para a `main`	clicar no nome → `main`	`git switch main`
trazer a branch para a `main`	`...` → **Branch** → **Merge...**	`git merge corrige-fronteira`

Fazendo, no `locadora-historico`

1. Crie a branch `corrige-fronteira`
2. No `multa.py`, troque o `> 3` por `>= 3`, rode a suíte — verde — e faça o commit
3. Volte para a `main` e olhe o `multa.py`: o `>` ainda está lá
4. Faça o merge da `corrige-fronteira` na `main`, e rode a suíte de novo

GLOSSÁRIO

`branch`	ramificação — uma linha de desenvolvimento separada
`merge`	mesclagem — juntar duas linhas numa só
`main`	a linha principal — a versão que funciona

O merge que ninguém percebe

- A Ana mexeu numa linha do `multa.py`. O Bruno, em outra.
- No merge, o Git junta as duas mudanças sozinho, porque elas não se tocam.
- É o caso mais comum. E é por isso que a mensagem do commit importa: depois do merge, ela é o que conta quem fez o quê.

Checagem do Bloco 1

- Por que a regra do Bruno, pela metade, não pode estar na `main`?
- Escreva os comandos para: criar a branch `teto-da-multa`, voltar para a `main`, e trazer a `teto-da-multa` para ela
- Quando é a hora certa de fazer o merge na `main`?



`forms.gle/s49i8pcYJvQ3VFcc6`

Intervalo

10 minutos

Bloco 2

O conflito, e quem decide

O dono disse duas coisas

Na segunda, o dono da locadora pediu um reajuste da multa:

- para a Ana, no balcão: *"passa a ser R$ 6,00 por dia"*
- para o Bruno, por mensagem: *"sobe para R$ 5,50"*

Cada um criou a sua branch e mudou a mesma linha. Façam: as duas branches no `locadora-historico`, o merge da da Ana, e depois o da do Bruno.

> Palpite, antes do segundo merge: o Git escolhe qual valor fica?

O conflito

```
<<<<<<< HEAD
MULTA_POR_DIA = 6.00
=======
MULTA_POR_DIA = 5.50
>>>>>>> reajuste-bruno
```

- O Git **não** escolhe. As duas mudanças tocam a mesma linha, e só uma pessoa sabe qual é a certa.
- No VS Code: **Accept Current**, **Accept Incoming** ou **Accept Both** — ou editar à mão. Depois, rodar a suíte e fazer o commit.
- Aqui, nenhum dos dois sabe a resposta. Ela está com o dono.

A solicitação de mudança

Antes de mudar, a mudança é **registrada** e **decidida** — não combinada de boca:

1. **Registrar:** quem pediu, quando, o que mudar e por quê
2. **Analisar:** que partes do sistema mudam, quanto trabalho dá
3. **Decidir:** quem responde pelo sistema aprova, adia ou recusa
4. **Fazer:** numa branch, com teste
5. **Fechar:** o merge cita a solicitação

No GitHub, a solicitação é uma **Issue**, com número. O commit `Reajusta multa para R$ 6 (fixes #4)` fecha a issue 4 sozinho quando chega na `main`.

⚠ Erro comum — resolver o conflito no chute

> O conflito some do arquivo com um clique em **Accept Current**. O Git fica satisfeito — e a decisão de negócio foi tomada por quem estava com o teclado, sem perguntar a ninguém.

- Conflito é **pergunta**, não erro do Git. Quem sabe a resposta, decide.
- Depois de resolver: a suíte **inteira**, antes do commit. Um conflito mal resolvido é o jeito mais rápido de criar defeito novo.

2026

Checagem do Bloco 2

O merge parou com isto no `relatorio.py`:

```
<<<<<<< HEAD
        self.maior = 0.0
=======
        self.maior = float("-inf")
>>>>>>> corrige-relatorio
```

- Qual lado fica? Use o que vocês sabem desde a Aula 01
- O que você roda antes de fazer o commit do merge?
- Escreva o título de uma issue que teria evitado esse conflito

forms.gle/oqtpsm4M7SEFEa8DA

Intervalo

10 minutos

Bloco 3

A versão estável e a vida do sistema

Que versão está no balcão?

- A `main` anda todo dia. O balcão precisa de uma versão que **não muda** e que dá para recriar igual, quando der problema.
- Essa versão ganha um nome: uma **tag**. É o *release* — a versão entregue a quem usa.

```
git tag v1.0.0
git push --tags
```

Posição em v1.0.0	Muda quando…
1	a mudança é grande — quem usa precisa reaprender alguma coisa
0 do meio	entrou funcionalidade nova, e o que existia continua igual
0 do fim	correção ou ajuste — para quem usa, nada novo nem diferente

Três tipos de mudança depois de pronto

Tipo	Na locadora	Do esforço de manutenção
correção de defeitos	o `>` da multa	17%
adaptação ao ambiente	o laboratório troca o PostgreSQL de versão	18%
adição de funcionalidade	o teto da multa, o desconto do cliente fiel	**65%**

- A maior parte da manutenção **não é consertar defeito**: é o sistema mudando porque o negócio mudou.
- No mercado, vocês vão ouvir corretiva, adaptativa e evolutiva — são os mesmos três.
- E **refatorar** — melhorar a estrutura sem mudar o que o sistema faz — é manutenção que previne a próxima.

A vida de um sistema

Evolução
mudanças grandes, a cada versão

Em serviço
só o essencial: defeito e ambiente

Interrupção gradual
ainda usado, ninguém mais muda

- Sistema usado **muda sempre** — ou vai ficando menos útil.
- A cada mudança ele fica **mais complexo**, a não ser que alguém trabalhe para simplificar.
- Versão com muita novidade costuma pedir uma versão **só de correção** logo depois. Planejem as duas.

A política de recuperação

Pergunta	O código	Os dados
o quê	o repositório inteiro	o banco do projeto
como	`git push` para o GitHub	`pg_dump` — a Aula 17 da UC2
quando	a cada commit que vale guardar	todo dia, ou antes de cada mudança grande
onde	fora da máquina: o GitHub	fora da máquina: drive, pendrive, nuvem
quem	quem fez o commit	um nome, não "alguém do grupo"
como sei que funciona	clonar numa pasta nova e rodar a suíte	restaurar num banco vazio e consultar

O GitHub guarda o código. **Não** guarda o banco — e o banco não pode ir para lá: tem dado de cliente.

Checagem do Bloco 3

- A versão `v1.2.3` recebeu a correção da fronteira da multa. Qual é a próxima? E se em vez disso entrasse o teto da multa?
- Classifique: o pgAdmin mudou e o CRUD parou de conectar · o dono quer relatório por cliente · o `elif` do relatório
- Por que "o backup está no GitHub" é uma resposta errada para os dados?



`forms.gle/Hjn5XVRyisTmbPd3A`

Intervalo

10 minutos

MÃO NA MASSA

Um defeito corrigido do jeito certo, e a versão 1.0

No repositório do grupo:

1. Escolham um defeito **aberto** na planilha da Aula 06 e abram uma **Issue** no GitHub com ele
2. Uma branch para o conserto; o teste vermelho, o conserto, a suíte verde
3. Commit citando a issue — `fixes #N` —, merge na `main`, `push`: a issue fecha sozinha
4. Atualizem a planilha: o defeito vira **corrigido**, com a ação corretiva
5. Escrevam o `RECUPERACAO.md` do grupo, com as seis perguntas da política, e façam o commit
6. Suíte verde na `main`? Marquem a `v1.0.0` e mandem a tag para o GitHub

MÃO NA MASSA

O combinado desta entrega

GLOSSÁRIO

o link do repositório — `github.com/<conta>/grupo03-uc3`, com a tag `v1.0.0`

O que eu confiro: Uma issue **fechada pelo commit**, com o conserto feito numa **branch** · A suíte verde na `v1.0.0` · A planilha com o defeito **corrigido** e a ação corretiva · O `RECUPERACAO.md` com **nomes**, não "alguém do grupo" · Nenhum dado do banco no repositório

> Eu passo de mesa em mesa e vou pedir para ver a issue, a branch e o merge — na ordem em que aconteceram.

PARA CASA · OPCIONAL

Antes da próxima aula

Testar em casa: siga o `RECUPERACAO.md` do grupo ao pé da letra, numa pasta nova e num banco vazio. Ele funciona sem ninguém explicar?

Tarefa opcional: abra uma issue para uma funcionalidade nova que o dono pediria, e diga em que versão ela entraria — `v1.0.1` ou `v1.1.0`?

Para treinar: o `locadora-historico.zip`, com as branches e o conflito de hoje.

Para ver acontecer: demonstrações da Aula 08 — resolver o conflito da multa e ver quem decide o valor, e numerar as versões que vêm depois da `v1.0.0` — e oito perguntas com resposta na hora.

Para a próxima aula

Amanhã é a entrega da UC. O sistema de vocês está todo no repositório — testes, plano, registro, a versão 1.0 e o jeito de recuperar tudo?

UniSenac Centro Universitário RS
Senac
Fecomércio
IFEP
Sesc
Sindicatos Empresariais