

Universidade São Judas Tadeu

Usabilidade, Desenvolvimento Web, Mobile e Jogos

Aula 03: Controle de versão com Git & Github

1 Introdução

Quando lidamos com arquivos, independentemente de seu tipo (ou seja, estamos falando de arquivos textuais, imagens e qualquer outro tipo), alguns problemas surgem naturalmente. Um bastante comum tem a ver com a versão com que estamos trabalhando. Quando fazemos alguma alteração em um arquivo e a tornamos persistente (ou seja, armazenada em disco, por exemplo) a princípio não dá para recuperar versões anteriores daquele arquivo. Um **Sistema de Controle de Versão**, como o Git, resolve esse e muitos outros problemas. Além do mais, nos dias atuais, o uso do Git está intimamente relacionado a processos de **Devops**, razão pela qual dominar os fundamentos dessa ferramenta é de grande interesse. Neste material iremos estudar as principais funcionalidades oferecidas pelo Git, que é o sistema de controle de versão mais utilizado nos dias atuais.

2 Instalação

2.1 Certifique-se de que você possui uma versão do Git corretamente instalada. Visite o link 2.1.1 para fazer o download de uma versão atual.

Link 2.1.1

<https://git-scm.com/>

Nota: O Git é um sistema de controle de versão **local**. Todas as suas operações podem ser realizadas localmente, em diretórios locais do sistema operacional, sem o uso de serviços remotos, como Github e Gitlab. O interesse no uso de servidores Git como esses surge no momento em que desejamos simplificar o compartilhamento de arquivos, cooperação entre desenvolvedores, garantir que os dados terão menor probabilidade de serem perdidos etc. Caso ainda não saiba, enquanto **Git** é um software que opera localmente no seu computador, **Github** é um serviço Git remoto que permite que seu diretório Git local seja armazenado na nuvem, provendo as vantagens mencionadas entre muitas outras.

3 Exemplo prático (local)

Muitos IDEs (como Eclipse, Android Studio etc) possuem plugins para uso do Git, permitindo seu uso por meio de interfaces gráficas. Evidentemente, isso tende a facilitar a vida do desenvolvedor bem como lhe proporcionar maior produtividade.

Porém, é de fundamental importância conhecer detalhes sobre o funcionamento da ferramenta Git. Assim, quando algo de errado acontecer com a interface gráfica que estiver utilizando, ele terá condições de interagir com o Git mais diretamente, com maiores chances de resolver o problema. Além disso, grande parte dessas ferramentas implementa somente um subconjunto parcial das funcionalidades do Git, o que faz com que o acesso completo a elas seja possível somente por meio da linha de comando.

Repositórios locais

Um repositório Git é uma pasta em que você colocará arquivos e subpastas cuja versão deseja controlar. Quando você inicializa um repositório Git em um diretório existente, uma **pasta oculta chamada .git** é criada dentro dele. Ela armazena informações sobre as versões dos arquivos que você cria e atualiza ao longo do tempo. Além de inicializar um repositório em um diretório existente, também é possível obter a cópia de um que esteja armazenado remotamente, como veremos.

3.1 Inicializando um repositório Git em uma pasta existente

Passo 3.1.1 (Criando um diretório) Crie um diretório chamado **dev** (o nome é arbitrário, pode ser qualquer diretório) para simularmos o desenvolvimento de um projeto composto por arquivos cuja versão controlaremos com o Git. Agora é necessário abrir um prompt de comando e navegar até o diretório dev. Neste exemplo, o diretório dev foi criado na pasta Documents do usuário. Para abrir o prompt e navegar até lá, pressione a tecla Windows e a tecla R simultaneamente e digite cmd. Depois use o seguinte comando. Note que o diretório pode ser diferente na sua máquina.

```
cd C:\Users\rodri\Documents\dev
```

Passo 3.1.2 (Inicializando um repositório Git) Para inicializar um repositório git, digite

```
git init
```

A partir deste instante, os arquivos criados no diretório dev poderão ter sua versão controlada pelo Git.

Nota: Mesmo que o Git esteja instalado, é possível que ela não seja encontrada pelo prompt de comando por não estar configurado nas variáveis de ambiente do sistema operacional. O seguinte comando pode resolver esse problema. Ele somente é necessário caso o comando **git** gere uma mensagem “comando não encontrado” quando digitado no prompt.

setx path=\$path;C:\Program Files\Git\cmd

Passo 3.1.3 (Criando a classe Empregado) Neste guia iremos desenvolver um projeto Java simples que envolve o processamento da folha de pagamento de empregados de uma empresa. Começaremos criando a classe da Listagem 3.1.3.1, cuja finalidade é representar todos os empregados existentes.

Listagem 3.1.3.1

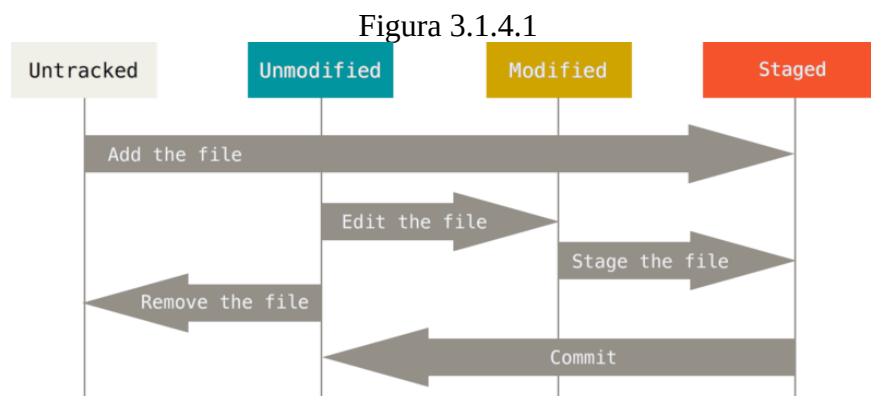
```
public class Empregado{
    private String nome;
    private int idade;

    public String getNome (){
        return this.nome;
    }
    public void setNome (String nome){
        this.nome = nome;
    }
    public int getIdade (){
        return this.idade;
    }
    public void setIdade (int idade){
        this.idade = idade;
    }
}
```

Passo 3.1.4 (Passos para fazer um commit) O simples fato de o arquivo Empregado.java estar no diretório dev não faz com que o git controle suas versões automaticamente. É preciso dizer ao git que desejamos que ele faça isso. Para tal, é preciso conhecer os possíveis estados em que um arquivo pode estar do ponto de vista do Git. São eles:

- **Untracked:** arquivos neste estado não estão sob controle do Git.
- **Tracked:** arquivos neste estado estão sob controle do Git e eles podem estar nos seguintes sub-estados:
- **Unmodified:** arquivos cuja última versão já é permanente no Git.
- **Modified:** arquivos que possuem alterações que não foram tornadas permanentes no Git.
- **Staged:** arquivos que foram modificados e que indicamos ao Git que desejamos que sejam incluídos no próximo commit.

A Figura 3.1.4.1 mostra os estados bem como as transições possíveis entre eles.



No momento nosso arquivo se encontra Untracked. Vamos dizer ao Git que desejamos controlar sua versão com o comando

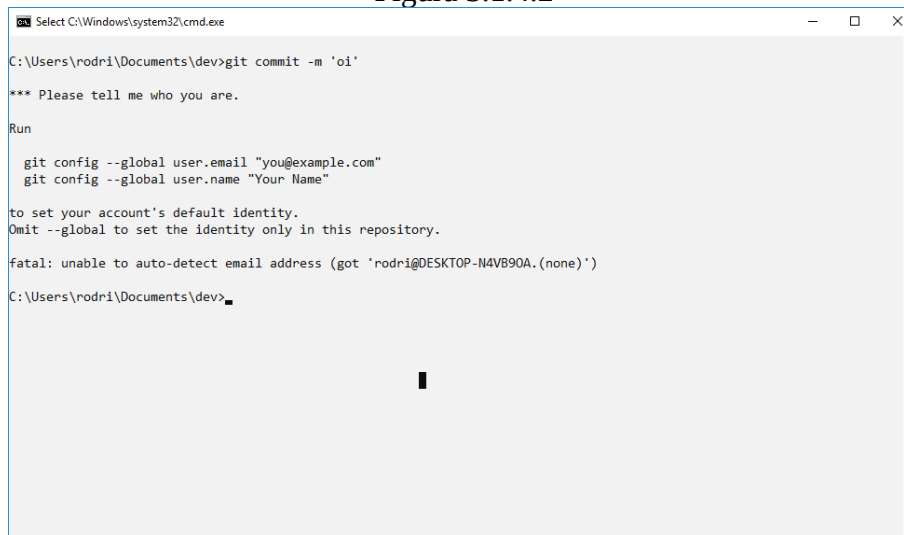
git add Empregado.java

Agora o arquivo Empregado.java está no estado Staged. Isso quer dizer que ele participará do próximo commit. Cada commit tem uma mensagem associada, que idealmente descreve as alterações realizadas desde a última versão. Para fazer um commit (ou seja, tornar as alterações permanentes), usamos o comando

git commit -m “Commit inicial. Classe Empregado criada”

O parâmetro -m do comando significa “message” e o valor a seguir fica associado ao commit, como mensagem que o descreve. Você deve ter obtido uma mensagem parecida com aquela exibida na Figura 3.1.4.2.

Figura 3.1.4.2



```
Select C:\Windows\system32\cmd.exe

C:\Users\rodri\Documents\dev>git commit -m 'oi'

*** Please tell me who you are.

Run

  git config --global user.email "you@example.com"
  git config --global user.name "Your Name"

to set your account's default identity.
Omit --global to set the identity only in this repository.

fatal: unable to auto-detect email address (got 'rodri@DESKTOP-N4VB90A.(none)')

C:\Users\rodri\Documents\dev>
```

Isso ocorre pois o git mantém (lembre-se, localmente) informações sobre quem realizou cada commit. Essas informações devem ser armazenadas em variáveis próprias do Git. Há diversas variáveis que podem ser configuradas. As duas indispensáveis, como mostra a Figura 3.1.4.2, são **user.email** e **user.name**. Elas podem ser definidas globalmente (indicando que têm o mesmo valor para qualquer usuário da máquina), de forma válida para o usuário logado (assim essas informações ficam armazenadas em sua pasta pessoal) e de forma que sejam válidas somente para o repositório atual. Para simplificar vamos usar a opção global. Execute os comandos a seguir, evidentemente ajustando seus valores de nome e email.

```
git config --global user.email 'seuemail@email.com'
git config -- global user.name 'Seu nome'
```

Nota: Caso esteja utilizando ou vá utilizar algum servidor Git como o Github, saiba que essas informações nada têm a ver com as informações do servidor. O e-mail, por exemplo, não necessariamente é o mesmo usado na conta do Github. Essas informações são requeridas pelo Git, que opera somente localmente, para que cada operação realizada seja associada a algum usuário (muitos podem estar trabalhando com o mesmo diretório) e para que seja possível saber qual usuário foi responsável por quais alterações realizadas.

Nota: o parâmetro --global indica que os valores configurados serão válidos para o usuário atualmente logado no sistema operacional. Se você estiver usando uma máquina pública, por exemplo, talvez isso não seja interessante. Por isso, é possível especificar que se deseja fazer as configurações de identificação de modo que elas sejam válidas somente para o diretório atual.

Também é possível fazer a configuração para todos os usuários do sistema operacional de uma única vez, caso deseje. Veja a Tabela 3.1.4.1.

Tabela 3.1.4.1

Opção	Resultado	Exemplo
--local	Configuração válida somente para o diretório git atual	git config --local user.name 'nome'
--global	Configuração válida para o usuário do sistema operacional atualmente logado	git config --global user.name 'nome'
--system	Configuração válida para todos os usuários do sistema operacional	git config --system user.name 'nome'

Nota: Caso nenhuma opção dessas seja especificada, **a padrão é --local.**

Agora execute o comando a seguir novamente para fazer seu primeiro commit.

git commit -m “Commit inicial. Classe Empregado criada”

Execute também o seguinte comando para verificar o status de seu repositório.

git status

Passo 3.1.5 (Adicionando tipo e método de cálculo de salário) Agora iremos adicionar uma variável que permite diferenciar empregados pelo seu tipo e também um método que calcula seu salário de acordo com seu tipo. Também serão adicionadas variáveis para armazenar os valores envolvidos nos cálculos. As regras de cálculo são exibidas pela Tabela 3.1.5.1.

Tabela 3.1.5.1

Tipo	Descrição	Cálculo
1	Comum	Salário
2	Comissionado	Salário + Salário * Comissão
3	Bonificado	Salário + Bônus

Depois de alterada, a classe Empregado fica como mostra a Listagem 3.1.5.2.

Listagem 3.1.5.2

```
public class Empregado{
    private String nome;
    private int idade;
    private int tipo;
    private double salario;
    private double comissao;
    private double bonus;
    public double calculaSalario (){
        if (tipo == 1){
            return salario;
        }
        else if (tipo == 2){
            //comissão varia de 0 a 1, é um percentual sobre o salário
            return salario + salario * comissao;
        }
        else if (tipo == 3){
            return salario + bonus;
        }
        else{
            return 0.;
        }
    }
    public String getNome (){
        return this.nome;
    }
    public void setNome (String nome){
        this.nome = nome;
    }
    public int getIdade (){
        return this.idade;
    }
    public void setIdade (int idade){
        this.idade = idade;
    }
}
```

Passo 3.1.5.6 (Um novo commit requer add antes) Feitas as alterações, faremos um novo commit, com o seguinte comando:

git commit -m “Cálculo de salário efetuado de acordo com o tipo”

Porém, note que a mensagem indica que o commit não foi realizado. Isso ocorre pois, antes de fazer um novo commit, precisamos dizer ao Git quais arquivos modificados desejamos que ele inclua nesse commit. Caso execute o comando a seguir você verá isso

git status

A resposta desse comando indica que devemos utilizar o comando add para indicar quais arquivos serão incluídos no próximo commit:

git add Empregado.java

Assim, perceba que **o comando add desempenha dois papéis diferentes:**

1. Ele altera o status de arquivos Untracked para Tracked.
2. Ele altera o status de arquivos Modified para Staged.

Dica: Neste momento, vale a pena checar novamente a Figura 3.1.4.1.

A execução do comando add, portanto, não quer dizer algo como “adicione esse arquivo ao git”, mas sim, “inclua esse arquivo ao próximo, e somente ao próximo, commit”.

Resumindo: Com o comando add indicamos quais arquivos devem participar do próximo commit.

Agora podemos verificar o status do repositório:

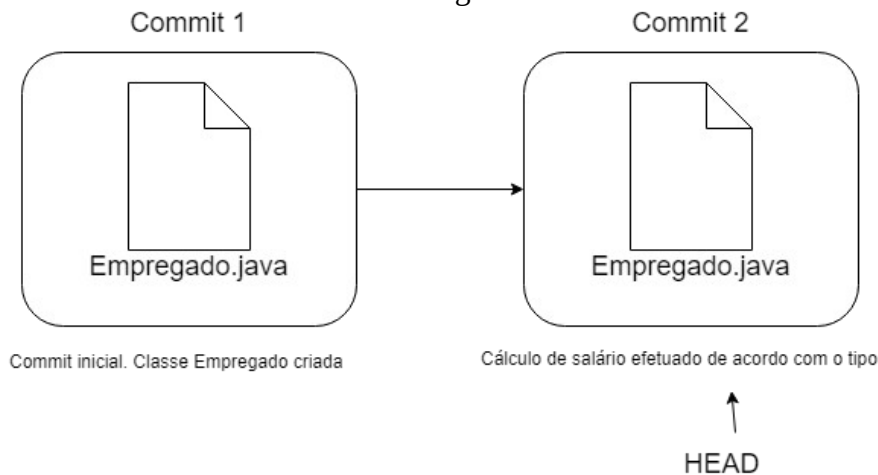
git status

E fazer o commit:

git commit -m “Cálculo de salário efetuado de acordo com o tipo”

Nesse instante, temos dois commits. O que isso significa? Significa que temos duas versões salvas do arquivo e que podemos navegar entre elas conforme a necessidade. Se a última alteração realizada contiver algum erro e for necessário voltar para a versão anterior, por exemplo, o Git resolve isso muito facilmente. A Figura 2.3 ilustra a história do repositório. Note que há um ponteiro chamado **HEAD**. É um nome usado pelo Git para mostrar qual o commit atual.

Figura 2.3



Passo 3.1.7 (Classe para teste) Agora iremos testar a classe Empregado. Para tal, criaremos uma outra classe com essa única finalidade. Ela possui um método main que instancia um empregado de cada tipo, atribui valores a suas variáveis e exibe os valores calculados. Ela se chama TesteEmpregado e o começo de sua implementação é exibido na Listagem 3.1.7.1. Evidentemente ela deve ser criada num arquivo chamado TesteEmpregado já que é pública.

Listagem 3.1.7.1

```

public class TesteEmpregado{
    public static void main (String [] args){

    }
}
  
```

Salve o arquivo e execute o comando a seguir para verificar o status do repositório:

git status

Note que o git identificará a presença de arquivos no estado Untracked. Use o comando a seguir para alterar o status de TesteEmpregado de Untracked para Staged.

git add TesteEmpregado.java

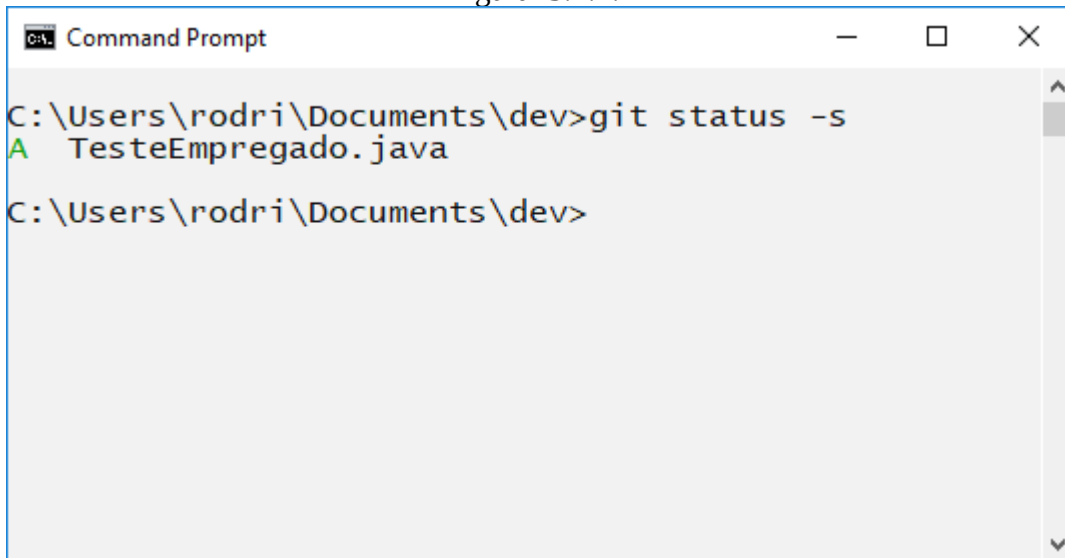
Verifique novamente o status do repositório e veja que o arquivo se encontra pronto para fazer parte do próximo commit, ou seja, ele está no estado Staged. Esse não parece ser um ponto muito importante na história desse projeto, então não faremos commit.

Você também pode usar o seguinte comando para verificar o status do seu repositório. O parâmetro -s significa “**short**” e a saída contém as mesmas informações porém representadas por uma única letra.

git status -s

Você deverá obter algo como o que exibe a Figura 3.1.7.1. A letra A ao lado do nome do arquivo significa que ele foi adicionado ao projeto, passando de Untracked para Tracked e Staged.

Figura 3.1.7.1



```

C:\Users\rodri\Documents\dev>git status -s
A TesteEmpregado.java
C:\Users\rodri\Documents\dev>

```

Continuando o desenvolvimento da classe TesteEmpregado, percebemos que não é possível configurar o tipo de um empregado, pois a classe Empregado não permite isso. Veja a Listagem 3.1.7.2.

Listagem 3.1.7.2

```

public class TesteEmpregado{
    public static void main (String [] args){
        Empregado e1 = new Empregado ();
        Empregado e2 = new Empregado();
        Empregado e3 = new Empregado ();
        //e agora, como configurar o tipo deles?
    }
}

```

Verifique novamente o status de seu repositório:

git status -s

O resultado é exibido na Figura 3.1.7.2.

Figura 3.1.7.2

```
C:\Users\rodri\Documents\dev>git status -s
AM TesteEmpregado.java
C:\Users\rodri\Documents\dev>
```

Note que há duas colunas ao lado esquerdo do nome do arquivo. A coluna do lado esquerdo contém a letra A, que continua com o mesmo significado: Há uma alteração que passou de Untracked para Staged mas que ainda não foi tornada permanente por meio de um commit. A coluna da direita mostra um M. Ele significa que, depois de ter passado para o estado Staged, o arquivo foi modificado novamente. Assim, possuímos duas cópias desse arquivo: uma está no estado Staged e a outra no estado Modified. Se fizermos um commit agora, a versão Staged será tornada permanente. Se fizermos um add e depois um commit, a versão modificada sobrescreverá aquela que está Staged atualmente e será tornada permanente, ou seja, fará parte do commit.

Agora vamos alterar a classe Empregado, adicionando o método setTipo e getTipo. Veja a Listagem 3.1.7.3.

Listagem 3.1.7.3

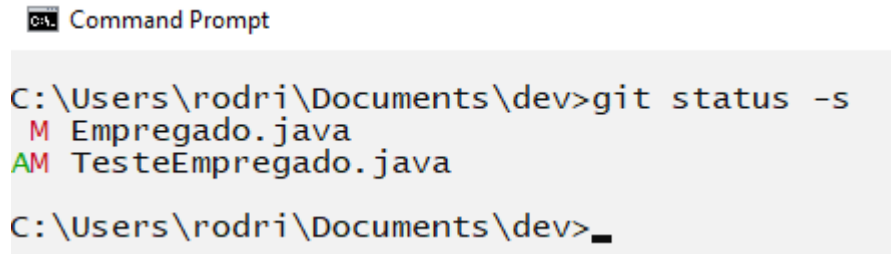
```
public void setTipo (int tipo){
    this.tipo = tipo;
}
public int tipo (){
    return this.tipo;
}
```

Verifique o status do seu repositório:

git status -s

O resultado é exibido na Figura 3.1.7.3.

Figura 3.1.7.3



```

C:\Users\rodri\Documents\dev>git status -s
 M Empregado.java
AM TesteEmpregado.java
C:\Users\rodri\Documents\dev>

```

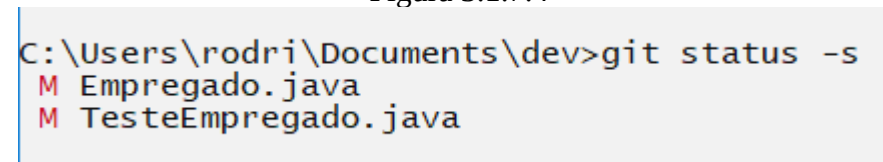
O arquivo Empregado está Modified mas não Staged. Antes de prosseguir, façamos um commit:

git commit -m “Classe de Teste criada. Métodos de acesso para a variável tipo adicionados.”

O comando a seguir mostra o status do repositório agora. A Figura 3.1.7.4 mostra o resultado.

git status -s

Figura 3.1.7.4



```

C:\Users\rodri\Documents\dev>git status -s
 M Empregado.java
 M TesteEmpregado.java

```

Note que somente a versão Staged de TesteEmpregado participou do Commit. Isso ocorreu pois deixamos de alterar o estado de Empregado.java para Staged. Vamos alterar o estado de ambos os arquivos para Staged:

git add *
git status -s

Veja o resultado na Figura 3.1.7.5. Veja que ambos arquivos possuem um M na coluna à esquerda. Isso quer dizer que há versões com alterações e que elas já foram colocadas no estado Staged e, portanto, participarão do próximo Commit.

Nota: A coluna à esquerda pode mostrar as letras A ou M indicando que um arquivo está no estado Staged. A letra A é usada para representar que o arquivo saiu do estado Untracked para Staged. A letra M representa sua transição de Modified para Staged.

Figura 3.1.7.5

```
C:\Users\rodri\Documents\dev>git status -s
M  Empregado.java
M  TesteEmpregado.java
C:\Users\rodri\Documents\dev>
```

E fazer um novo commit:

git commit -m “Incluindo arquivos faltantes do commit anterior”

Checamos o status do repositório agora e a Figura 3.1.7.6 o exibe:

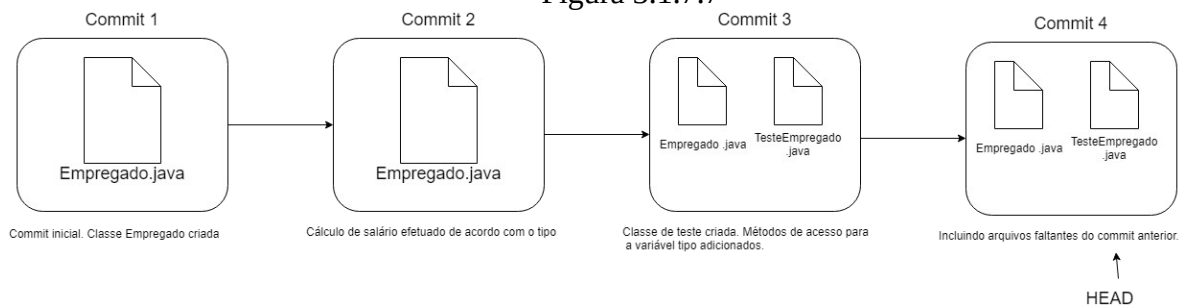
git status -s

Figura 3.1.7.6

```
C:\Users\rodri\Documents\dev>git status
On branch master
nothing to commit, working tree clean
```

Neste instante, o histórico armazenado pelo Git é aquele ilustrado pela Figura 3.1.7.7.

Figura 3.1.7.7



Use o comando a seguir para verificar esse histórico. Note que ele é exibido como uma pilha: o último commit realizado é o primeiro a ser exibido.

git log

Agora vamos concluir o teste inicial de cálculo de salário. Para tal, note que será necessário adicionar métodos de acesso e modificadores para as variáveis que representam os valores recebidos pelos empregados na classe `Empregado` e fazer as contas na classe `TesteEmpregado`. Assim, após as alterações, iremos alterar o estado de cada arquivo para Staged e fazer um novo commit. As alterações são exibidas nas listagens 3.1.7.4 e 3.1.7.5.

Listagem 3.1.7.4

```
//adicionar à classe Empregado.java
public void setSalario (double salario){
    this.salario = salario;
}
public double getSalario (){
    return this.salario;
}
public void setComissao (double comissao){
    this.comissao = comissao;
}
public double getComissao (){
    return this.comissao;
}
public void setBonus (double bonus){
    this.bonus = bonus;
}
public double getBonus (){
    return this.bonus;
}
```

Listagem 3.1.7.5

```
//a classe TesteEmpregado ficou assim
public class TesteEmpregado{
    public static void main (String [] args){
        Empregado e1 = new Empregado ();
        Empregado e2 = new Empregado();
        Empregado e3 = new Empregado ();
        //configurando tipo dos empregados
        e1.setTipo(1);
        e2.setTipo(2);
        e3.setTipo(3);
        //configurando alguns valores
        e1.setSalario(2000);
```

```
e2.setSalario(1700);
e2.setComissao (0.2); //20% de comissão sobre o salário
e3.setSalario(1500);
e3.setBonus(400);
//mostra os valores
System.out.println (e1.calculaSalario());
System.out.println (e2.calculaSalario());
System.out.println (e3.calculaSalario());
}
}
```

Para testar, vamos compilar os arquivos com o seguinte comando:

javac Empregado.java TesteEmpregado.java

Caso queira ver os resultados, use:

java TesteEmpregado

Agora podemos fazer um novo commit. Usando o comando a seguir, veremos que ambos os arquivos estão no estado Modified. Veja a Figura 3.1.7.8 e note que o estado dos arquivos compilados também é exibido. O símbolo ? indica que eles estão no estado Untracked. Lidaremos com eles em breve.

git status -s

Figura 3.1.7.8

```
C:\Users\rodri\Documents\dev>git status -s
 M Empregado.java
 M TesteEmpregado.java
?? Empregado.class
?? TesteEmpregado.class
```

Promovemos os arquivos com código fonte (somente eles, os compilados, com extensão .class, não) ao estado Staged com:

git add Empregado.java TesteEmpregado.java

Verificamos novamente o estado. Eles continuam modificados, porém no estado Staged (o M agora aparece na coluna à esquerda). Veja a Figura 3.1.7.9.

git status -s

Figura 3.1.7.9

```
C:\Users\rodri\Documents\dev>git status -s
M   Empregado.java
M   TesteEmpregado.java
??  Empregado.class
??  TesteEmpregado.class
```

E fazemos o commit:

git commit -m “Primeiro teste concluído com sucesso”

O status do repositório ainda inclui os arquivos .class no estado Untracked (Veja a Figura 3.1.7.10):

git status -s

Figura 3.1.7.10

```
C:\Users\rodri\Documents\dev>git status -s
??  Empregado.class
??  TesteEmpregado.class
```

Por um lado, não queremos controlar a versão de arquivos compilados. Por outro, também não queremos que, a cada checagem de status do repositório, o Git nos mostre uma lista de arquivos, pois a saída pode ficar poluída se eles forem muitos. Para resolver esse problema, vamos adicionar um **arquivo** chamado **.gitignore**. Sua finalidade é muito simples: Iremos listar os nomes de arquivos (ou padrões que os identificam) que desejamos que o Git ignore. Eles permanecerão no estado Untracked e o Git não os exibirá na lista quando checarmos o status do repositório. Crie um arquivo chamado .gitignore em seu repositório e inclua o conteúdo da Listagem 3.1.7.6.

Listagem 3.1.7.6

```
# todos os arquivos que tenham .class como sufixo
*.class
```

Nota: No Windows você não conseguirá criar um arquivo chamado .gitignore usando o Windows Explorer. Para tal, na linha de comando, use **echo > .gitignore** e depois abra o arquivo com o seu editor de texto e apague o conteúdo gerado pelo comando echo.

Nota: Certifique-se de que o arquivo está usando o encoding **UTF-8**, caso contrário o Git potencialmente não será capaz de ler seu conteúdo. Use seu editor de texto para fazer essa verificação e altere o encoding, caso necessário.

Execute o comando a seguir novamente e veja o resultado da Figura 3.1.7.11.

git status -s

Figura 3.1.7.11

```
C:\Users\rodri\Documents\dev>git status -s
?? .gitignore
```

Note que o próprio arquivo `.gitignore` poderá evoluir ao longo do tempo e possivelmente iremos controlar sua versão também. Assim, vamos incluí-lo ao projeto com o comando:

git add .gitignore

Verificamos novamente o status, que nos dá como resultado o que a Figura 3.1.7.12 exibe:

git status -s

Figura 3.1.7.12

```
C:\Users\rodri\Documents\dev>git status -s
A .gitignore
```

Finalmente fazemos um novo commit, tornando permanente o arquivo `.gitignore`:

git commit -m “Adicionando o arquivo .gitignore”

4 Repositórios remotos

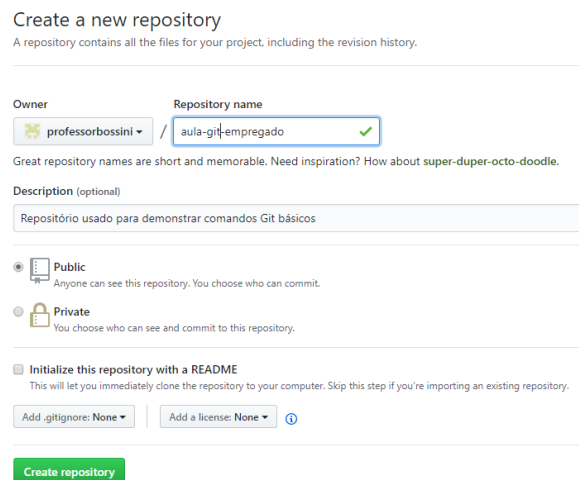
Note que, até então, temos armazenado o histórico de versão de nosso projeto apenas localmente. O Git permite que repositórios locais sejam sincronizados com repositórios remotos. Há diversos servidores Git disponíveis como o GitHub e o GitLab. Neste tutorial, iremos utilizar o GitHub.

Passo 4.1 (Criando uma conta e um repositório no Github) Para criar uma conta no Github, abra o Link 4.1.1 e crie uma conta gratuita para você.

Link 4.1.1
<https://github.com/>

Com a conta criada, crie um novo repositório (clique em Repositories e então em New). Escolha um nome e uma descrição para ele. Veja a Figura 4.1.1.

Figura 4.1.1



Create a new repository

A repository contains all the files for your project, including the revision history.

Owner: professorbossini / Repository name: aula-git-empregado ✓

Great repository names are short and memorable. Need inspiration? How about super-duper-octo-doodle.

Description (optional): Repositório usado para demonstrar comandos Git básicos

☒ Public: Anyone can see this repository. You choose who can commit.

☐ Private: You choose who can see and commit to this repository.

☐ Initialize this repository with a README: This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: None | Add a license: None ⓘ

Create repository

Passo 4.2 (Adicionando um remote) A fim de fazer o upload de nosso código, é preciso adicionar um “remote” a nosso projeto **local**. O Git permite que um repositório local possua referências a diferentes remotes. Por exemplo, iremos agora adicionar (ao repositório local) um remote que representa o repositório que acabamos de criar no Github. Ele terá um nome que podemos escolher e, a partir daí, poderemos fazer uploads. Futuramente, talvez queiramos manter o projeto sincronizado com outro repositório remoto. Talvez um novo repositório no Github, talvez um repositório no Gitlab ou até mesmo um repositório em um servidor Git configurado na rede local. Para isso, basta adicionar um novo remote. O seguinte comando permite verificar os remotes que nosso repositório local já possui:

git remote

Note que a saída ficou vazia. Isso ocorre pois ainda não adicionamos nenhum remote ao nosso repositório. Faremos isso agora, com o seguinte comando:

git remote add origin <https://github.com/professorbossini/aula-git-projeto-empregado.git>

Nota: Ajuste o link de acordo com o seu usuário no Github e o nome do repositório.

Nesse instante, temos um remote em nosso repositório cujo nome é **origin**. Para fazer upload a ele, usaremos o nome **origin**.

Nota: A palavra **origin** não tem nada de especial. Ela é usada por padrão pelo Git quando fazemos a cópia de um repositório remoto para um local. Ou seja, o repositório remoto já fica adicionado ao local como um remote e seu nome é origin. Assim, poderíamos ter usado qualquer outra palavra que nos fosse de interesse.

Listando novamente os remotes:

git remote

Temos a saída da Figura 4.2.1.

Figura 4.2.1

```
C:\Users\rodri\Documents\dev>git remote  
origin
```

Para ver informações mais detalhadas, como na Figura 4.2.2, usamos

git remote -v

Figura 4.2.2

```
C:\Users\rodri\Documents\dev>git remote -v  
origin https://github.com/professorbossini/aula-git-projeto-empregado.g  
it (fetch)  
origin https://github.com/professorbossini/aula-git-projeto-empregado.g  
it (push)
```

Passo 4.3 (Fazendo upload ao repositório remoto (push)) Agora utilizaremos uma operação chamada “push”. Ela envia os arquivos (todas as cópias que participaram do último commit) ao servidor remoto que escolhemos pelo nome (no nosso caso, só temos um, chamado origin).

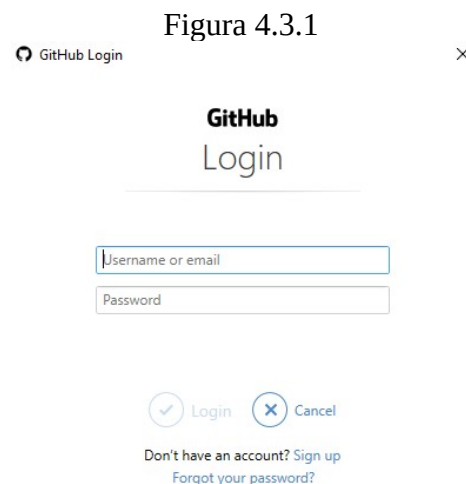
Nota: Commit e push são operações completamente diferentes. Commit torna permanentes alterações realizadas localmente, desde que elas passem pelo estado Staged. Push envia os arquivos do último Commit para um remote escolhido.

Para fazer o push, vamos usar o seguinte comando:

git push origin master

Nota: “master” é o nome da branch em que estamos trabalhando atualmente. Esse assunto será tratado posteriormente. Desde já, saiba que o nome master também é só uma convenção usada pelo Git e não tem nada de especial. O nome poderia ser qualquer outro. Mas ele já criou a branch principal com esse nome, por isso estamos utilizando.

Você verá a tela de login exibida pela Figura 4.3.1. Dependendo de sua instalação, pode ser que o login seja pedido pela linha de comando também.



Entre com suas credenciais para completar o login. Após o termino do upload, a saída deve ser parecida com o que exhibe a Figura 4.3.2.

Figura 4.3.2

```
r
OpenGL Warning: crPixelCopy3D:  simply crMemcpy'ing from srcPtr to dstPt
r
OpenGL Warning: crPixelCopy3D:  simply crMemcpy'ing from srcPtr to dstPt
r
OpenGL Warning: crPixelCopy3D:  simply crMemcpy'ing from srcPtr to dstPt
r
Enumerating objects: 20, done.
Counting objects: 100% (20/20), done.
Delta compression using up to 2 threads
Compressing objects: 100% (18/18), done.
Writing objects: 100% (20/20), 2.43 KiB | 829.00 KiB/s, done.
Total 20 (delta 5), reused 0 (delta 0)
remote: Resolving deltas: 100% (5/5), done.
remote:
remote: Create a pull request for 'master' on GitHub by visiting:
remote:   https://github.com/professorbossini/aula-git-projeto-emprega
ado/pull/new/master
remote:
To https://github.com/professorbossini/aula-git-projeto-empregado.git
* [new branch]      master -> master
```

Visite novamente a página de seu repositório no Github. Você deve ver algo como o que exibe a Figura 4.3.3.

Figura 4.3.3

The screenshot shows the GitHub interface for the repository 'aula-git-projeto-empregado' by 'professorbossini'. At the top, there are buttons for 'Watch', 'Star', and 'Fork', all with a count of 0. Below this is a navigation bar with links for 'Code', 'Issues' (0), 'Pull requests' (0), 'Projects' (0), 'Wiki', 'Insights', and 'Settings'. The main content area starts with the text 'Repositório usado para demonstrar comandos Git básicos' and an 'Edit' button. Below this is a section for repository statistics: '6 commits', '1 branch', '0 releases', and '0 contributors'. A 'Branch: master' dropdown and a 'New pull request' button are also visible. The commit history table shows three commits: one by 'Rodrigo' adding a '.gitignore' file, and two by 'Empregado.java' and 'TesteEmpregado.java' with the message 'Primeiro teste concluído com sucesso'. At the bottom, there is a prompt to 'Add a README'.

Commit	Author	Message	Time
b781fe3	Rodrigo	Adicionando o arquivo .gitignore	an hour ago
	Empregado.java	Primeiro teste concluído com sucesso	2 hours ago
	TesteEmpregado.java	Primeiro teste concluído com sucesso	2 hours ago

Percebeu que o Github mostra até o número de commits feitos até então e permite verificar o conteúdo de cada um deles?

5 Releases

Conforme o desenvolvimento do projeto evolui, é natural chegar a um momento em que temos um commit que representa algo “pronto”, como um produto para ser colocado em produção, um exercício pronto para ser entregue para o professor etc. Ou seja, é natural termos commits que desejamos destacar dos demais, marcar como importantes. Isso pode ser feito com um recurso do git chamado **tag**. Quando adicionamos uma tag a um commit, estamos dizendo que aquele commit é importante, representa algo importante para o projeto. Enquanto localmente esse recurso se chama “tag”, quando feito o upload o Github, ele o exibirá como uma “release”, ou seja, um commit marcado com uma tag é uma “release”.

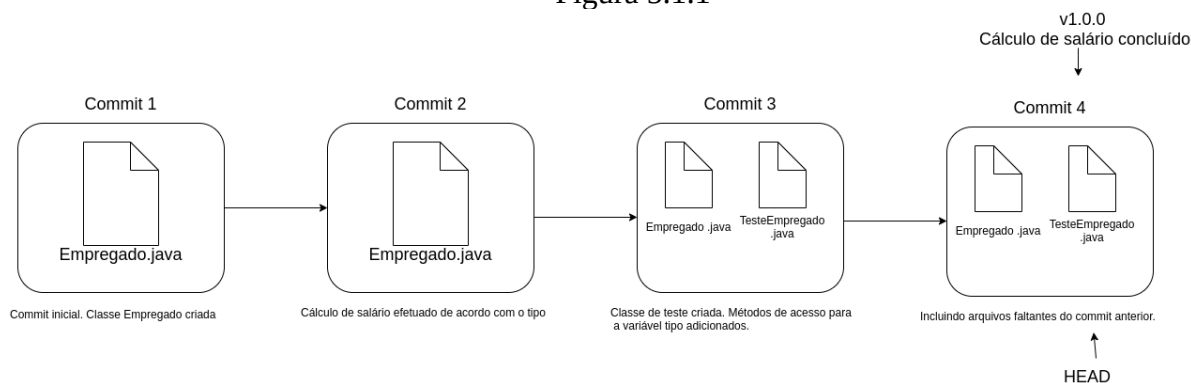
Passo 5.1(Adicionando uma tag) Use o comando a seguir para adicionar uma tag ao commit atual. Vamos marcá-lo como importante.

git tag v1.0.0 -a -m “Cálculo de salário concluído”

Nota: A opção **-a** indica que essa é uma tag **anotada**. Elas incluem o nome e e-mail de seu criador, data da criação, uma mensagem associada. Essas informações podem ser de interesse para futura inspeção do repositório. Sem essa opção, estaríamos criando uma tag **leve**. Uma tag leve é somente um ponteiro para um commit e não armazena nenhuma outra informação como feito pelas tags anotadas.

Neste instante, a situação do repositório é ilustrada pela Figura 5.1.1.

Figura 5.1.1



Nota: O padrão v1.0.0 é inteiramente opcional. Uma tag é simplesmente uma string. O nome dela também poderia ser algo como “EntregaDoPrimeiroExercicio”, “Tag1” ou qualquer coisa que faça sentido de acordo com o contexto.

Passo 5.2 (Novas edições nos arquivos) Vamos fazer novas edições nos arquivos e gerar novos commits para, ao final, marcar o último como um novo ponto importante.

5.2.1 Abra o arquivo **TesteEmpregado.java** e atualize seu conteúdo para aquele exibido pela Listagem 5.2.1.1. Passamos a lidar com os empregados em uma lista.

Listagem 5.2.1.1

```
import java.util.*;
//a classe TesteEmpregado ficou assim
public class TesteEmpregado{
    public static void main (String [] args){
        Empregado e1 = new Empregado ();
        Empregado e2 = new Empregado();
        Empregado e3 = new Empregado ();
        //configurando tipo dos empregados
        e1.setTipo(1);
        e2.setTipo(2);
        e3.setTipo(3);
        //configurando alguns valores
        e1.setSalario(2000);
        e2.setSalario(1700);
        e2.setComissao (0.2); //20% de comissão sobre o salário
        e3.setSalario(1500);
        e3.setBonus(400);
        //adicionando todo mundo em uma lista
        List <Empregado> emps = Arrays.asList(e1, e2, e3);
        //mostra os valores
        /*System.out.println (e1.calculaSalario());
        System.out.println (e2.calculaSalario());
        System.out.println (e3.calculaSalario());*/
        for (Empregado e : emps)
            System.out.println (e.calculaSalario());
    }
}
```

A seguir, faça um novo commit e, por fim, gere uma nova release, adicionando uma nova tag ao projeto. Comece verificando o status do repositório com

git status -s

Altere o status do arquivo TesteEmpregado para Staged, com

git add TesteEmpregado.java

Verifique novamente o status e certifique-se de que o arquivo está Staged (a letra de seus status é verde).

git status -s

Faça o commit

git commit -m “Processamento de empregados usando lista”

Envie para o Github

git push origin master

Agora vamos gerar uma nova tag, indicando que essa é uma nova versão importante

git tag v1.0.1 -a -m “Processamento com listas”

Verifique a lista de tags existentes até o momento com

git tag

Note que as tags existem somente localmente. Elas ainda não foram transportadas para o Github. O comando a seguir fará isso

git push origin --tags

Acesse o link de seu repositório no Github e veja que ele mostra 2 releases. Cada tag é interpretada como uma liberação (release) de versão pelo Github. Veja a Figura 5.2.1.1.

Figura 5.2.1.1

professorbossini / aula-git-projeto-empregado

Watch 0 Star 0 Fork 0

Code Issues 0 Pull requests 0 Actions Projects 0 Wiki Security Insights Settings

Repositório usado para demonstrar comandos Git básicos [Edit](#)

[Manage topics](#)

7 commits 1 branch 0 packages 2 releases 1 contributor

Branch: master New pull request Create new file Upload files Find file Clone or download

professorbossini Processamento de empregados usando lista Latest commit 59327d6 14 minutes ago

.gitignore	Adicionando o arquivo .gitignore	16 months ago
Empregado.java	Primeiro teste concluído com sucesso	16 months ago
TesteEmpregado.java	Processamento de empregados usando lista	14 minutes ago

Help people interested in this repository understand your project by adding a README. [Add a README](#)

Clique em **2 releases** para ver a lista de releases de seu repositório, como na Figura 5.2.1.2.

Figura 5.2.1.2

Code Issues 0 Pull requests 0 Actions Projects 0 Wiki Security Insights Settings

Releases Tags [Draft a new release](#)

13 minutes ago v1.0.1 ...
59327d6 zip tar.gz

1 hour ago v1.0.0 ...
b781fe3 zip tar.gz

Referências

Git. 2020. Disponível em <<https://git-scm.com>> Acesso em agosto de 2020.

GitHub. 2020. Disponível em <<https://github.com>>. Acesso em agosto de 2020.