

# Testes e Qualidade em Jogos

5º período

Professora: Michelle Hanne

# TÉCNICAS DE TESTE ESTRUTURAIS

A técnica estrutural (ou caixa-branca ou ainda teste caixa de vidro) estabelece **os requisitos de teste** com base na implementação fornecida de um programa, requerendo a **execução de partes ou de componentes do programa**.

Os **caminhos lógicos do software são testados**, fornecendo-se casos de teste que colocam à prova tanto conjuntos específicos de condições e/ou laços, quanto pares de definições e usos de variáveis.

# TÉCNICAS DE TESTE ESTRUTURAIS

Em linhas gerais, usando métodos de teste de caixa-branca, o engenheiro de software pode criar casos de teste que:

- garantam que todos os caminhos independentes de um módulo foram exercitados pelo menos uma vez;
- exercitam todas as decisões lógicas nos seus estados verdadeiro e falso;
- executam todos os ciclos em seus limites e dentro de suas fronteiras operacionais;
- exercitam estruturas de dados internas para assegurar a sua validade (PRESSMAN; MAXIM, 2016).

# TÉCNICAS DE TESTE ESTRUTURAIS

Os critérios pertencentes à técnica estrutural são classificados com base na complexidade, no fluxo de controle e no fluxo de dados. A técnica estrutural apresenta uma série de limitações e desvantagens decorrentes das limitações inerentes à atividade de teste de programa como estratégia de validação.

Tais aspectos introduzem sérios problemas na automatização do processo de validação de software (DELAMARO et al., 2016):

Não existe um procedimento de teste de propósito geral que possa ser usado para provar a correção de um programa;

- Dados dois programas, **é indecidível se eles computam a mesma função**;
- **É indecidível, em geral, se dois caminhos de um programa**, ou de programas diferentes, computam a mesma função;
- **É indecidível, em geral, se um dado caminho é executável**, ou seja, se existe um conjunto de dados de entrada que leve à execução desse caminho.

# TÉCNICAS DE TESTE ESTRUTURAIS

Existem  
também  
limitações que  
são inerentes  
à técnica  
estrutural:

- **Caminhos ausentes:** se o programa não implementa algumas funções, não existirá um caminho que corresponda a essas mesmas funções; conseqüentemente, nenhum dado de teste será requerido para exercitá-las.
- **Correção coincidente:** o programa pode apresentar, coincidentemente, um resultado correto para um dado em particular de entrada, satisfazendo um requisito de teste e não revelando a presença de um defeito; entretanto, se escolhido outro dado de entrada, o resultado obtido poderia ser incorreto.

# CRITÉRIOS DO TESTE ESTRUTURAL

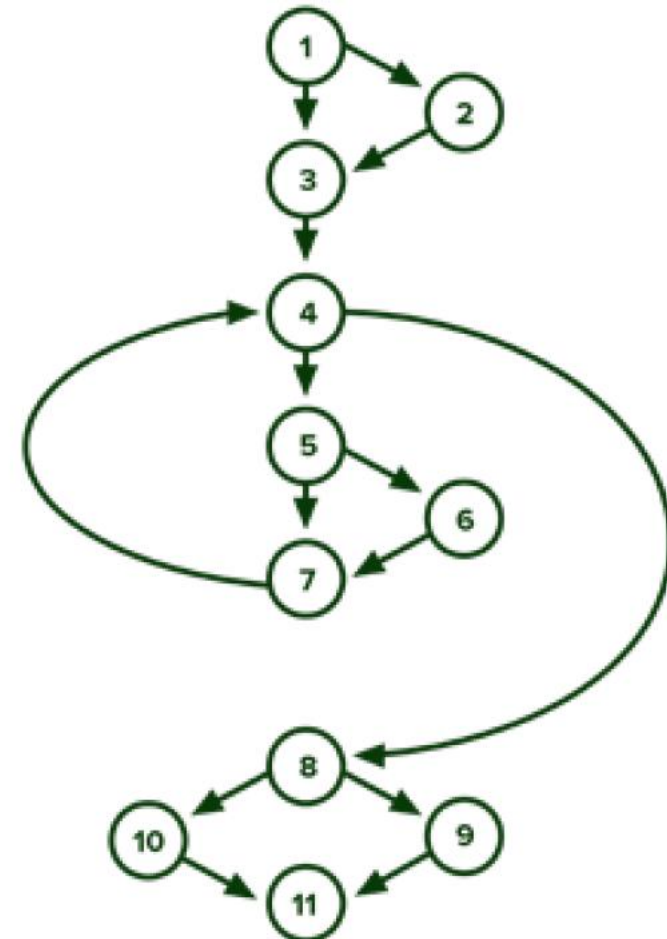
Os critérios estruturais são, em geral, classificados em:

- Critérios baseados na complexidade;
- Critérios baseados em fluxo de controle;
- Critérios baseados em fluxo de dados.

# CRITÉRIOS Baseados na Complexidade

## A complexidade ciclomática

- é uma **métrica de software** que **proporciona uma medida quantitativa da complexidade lógica** de um programa. Utilizado no contexto do **teste de caminho básico**, o **valor da complexidade ciclomática** estabelece o **número de caminhos linearmente independentes do conjunto básico de um programa**, oferecendo um **limite máximo** para o número de casos de teste que devem ser **derivados a fim de garantir que todas as instruções sejam executadas pelo menos uma vez** (DELMARARO et al., 2016).

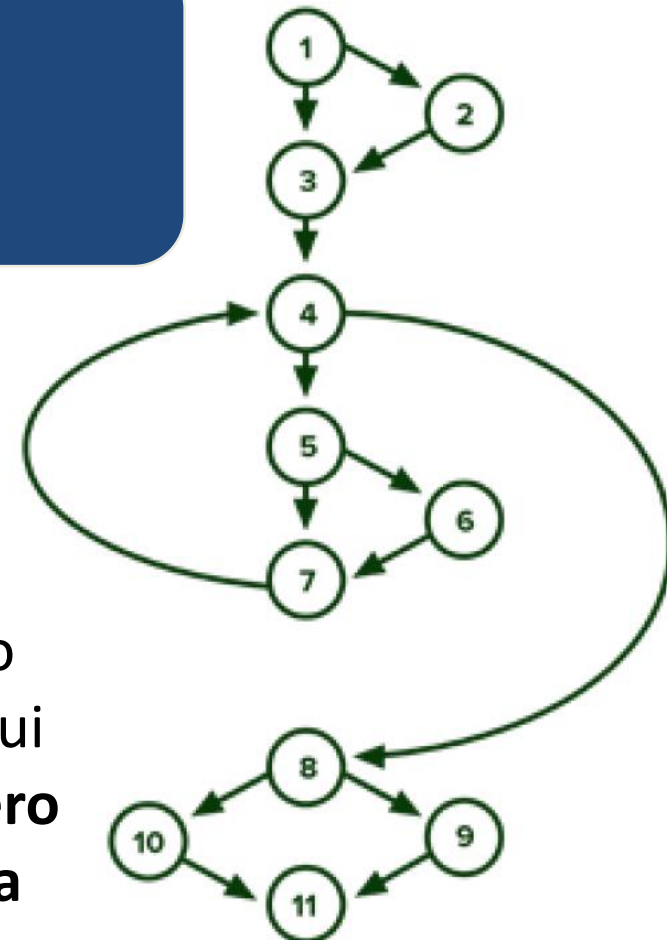


# CRITÉRIOS Baseados na Complexidade

$V(G) = (E - N) + 2$ , tal que  $E$  é o número de arcos e  $N$  é o número de nós do GFC  $G$ ;

$$V(G) = (14 - 11) + 2 = 5$$

O **valor da complexidade ciclomática  $V(G)$**  oferece um limite máximo para o **número de caminhos linearmente independentes** que constitui o conjunto básico e, conseqüentemente, um **limite máximo do número de casos de teste que deve ser projetado e executado** para garantir a cobertura de todas as instruções de programa.





# CRITÉRIOS Baseados em Fluxo de Controle

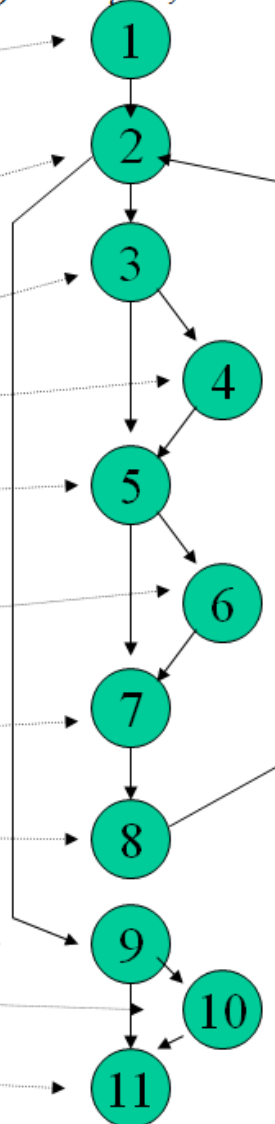
Os critérios baseados em fluxo de controle utilizam apenas características de controle da execução do programa, como comandos ou desvios, para determinar quais estruturas são necessárias. Os critérios mais conhecidos dessa classe são:

- **Todos-nós:** exige que a execução do programa passe, ao menos uma vez, em cada vértice do GFC (Grafo Fluxo de Controle), ou seja, que cada comando do programa seja executado pelo menos uma vez.
- **Todas-arestas (ou Todos-arcos):** requer que cada aresta do grafo, isto é, cada desvio de fluxo de controle do programa, seja exercitado pelo menos uma vez;
- **Todos-caminhos:** requer que todos os caminhos possíveis do programa sejam executados. Executar todos os caminhos de um programa é, na maioria dos casos, uma tarefa impraticável.

# CRITÉRIOS Baseados em Fluxo de Controle

```
function percorreAluno(nomeAluno : str40; mostra : boolean) : integer;
var i : integer; achou : boolean; p : pRegDisp;
begin
```

```
  i:=0;
  achou:= false;
  while((NOT(achou))AND(i<=nAlunos))do
  begin
    i:=i+1;
    if(nomeAluno=vetAlunos[i].nome)then
      achou:=true;
      if(mostra)then
        begin
          writeln('Nome: ',vetAlunos[i].nome);
          writeln('RA: ',vetAlunos[i].RA);
          p:=percorreDisciplina("i,true);
        end;
      end;
      percorreAluno:=i;
      if(NOT(achou))then
        percorreAluno:=(-1);
      end;
    end;
```



Exemplo de Grafo de Fluxo (VILELA C,2005).

# CRITÉRIOS Baseados em Fluxo de Dados

Os critérios baseados em fluxo de dados utilizam a análise de fluxo de dados **como fonte de informação para derivar os requisitos de teste**. Uma característica comum aos critérios dessa categoria é que eles requerem o teste das interações que envolvam **definições de variáveis e referências posteriores a essas definições**.

Uso do **Grafo Def-Uso**, que consiste em uma **extensão de um CFG (Grafo Fluxo de Controle)**. Nesse grafo estendido, são adicionadas informações a respeito do fluxo de dados do programa, caracterizando associações entre pontos nos quais esse valor é utilizado (**referência ou uso de variável**).

# CRITÉRIOS Baseados em Fluxo de Dados

- Definição (Def): um valor é atribuído a uma variável (diferente de declaração da variável)

`int x = 10; a = 5;`

- Uso Computacional (c-uso): uso da variável em computações

`a = b * 1; (c-uso de b)`

- Uso Predicativo (p-uso): uso da variável em expressões predicativas em estruturas de decisão e repetição

`if (a >= b) (p-uso de a e b)`

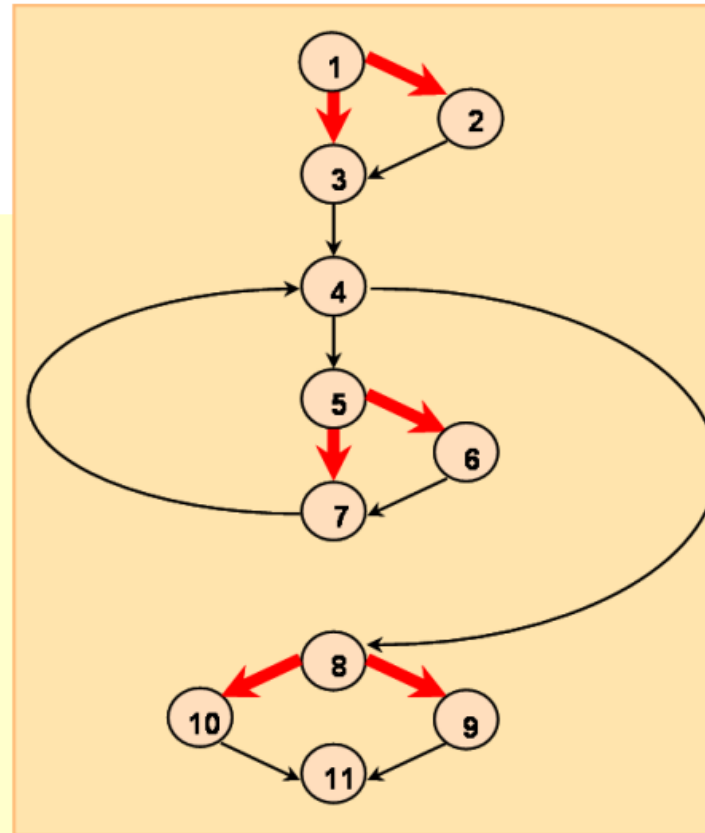
# CRITÉRIOS Baseados em Fluxo de Dados

## GRAFO DEF-USO: EXEMPLO

```

/* 01 */      {
/* 01 */      char  achar;
/* 01 */      int length, valid_id;
/* 01 */      length = 0;
/* 01 */      printf ("Identificador: ");
/* 01 */      achar = fgetc (stdin);
/* 01 */      valid_id = valid_s(achar);
/* 01 */      if (valid_id)
/* 02 */          length = 1;
/* 03 */      achar = fgetc (stdin);
/* 04 */      while (achar != '\n')
/* 05 */      {
/* 05 */          if (!(valid_f(achar)))
/* 06 */              valid_id = 0;
/* 07 */          length++;
/* 07 */          achar = fgetc (stdin);
/* 07 */      }
/* 08 */      if (valid_id && (length >= 1) && (length < 6))
/* 09 */          printf ("Valido\n");
/* 10 */      else
/* 10 */          printf ("Invalido\n");
/* 11 */      }

```



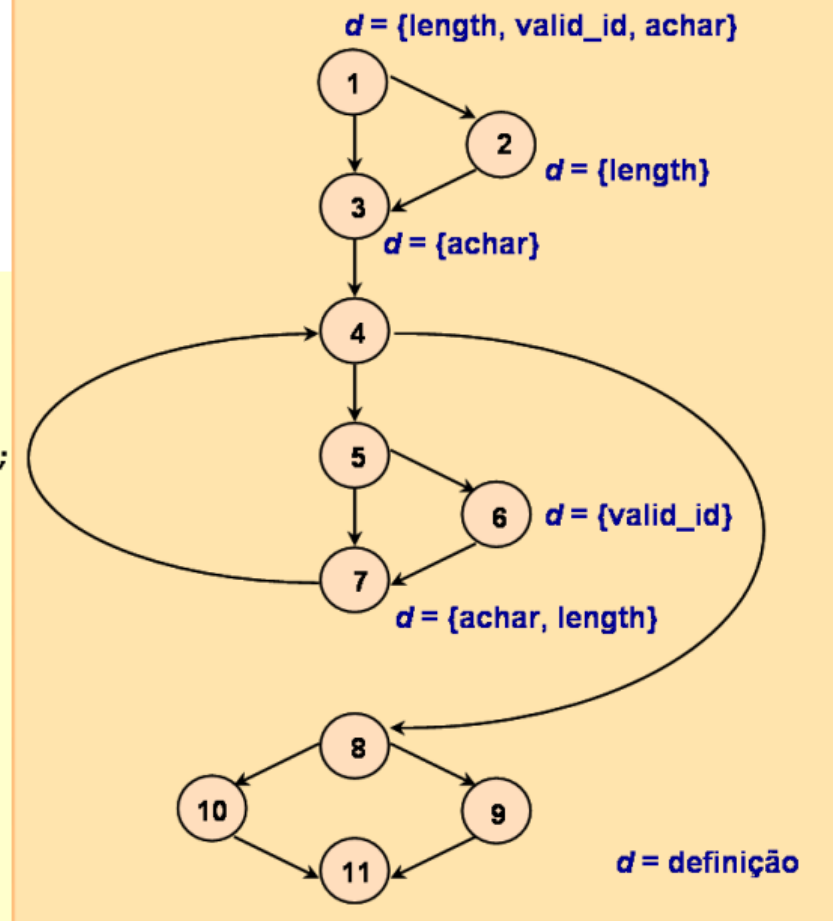
# CRITÉRIOS Baseados em Fluxo de Dados

## GRAFO DEF-USO: EXEMPLO

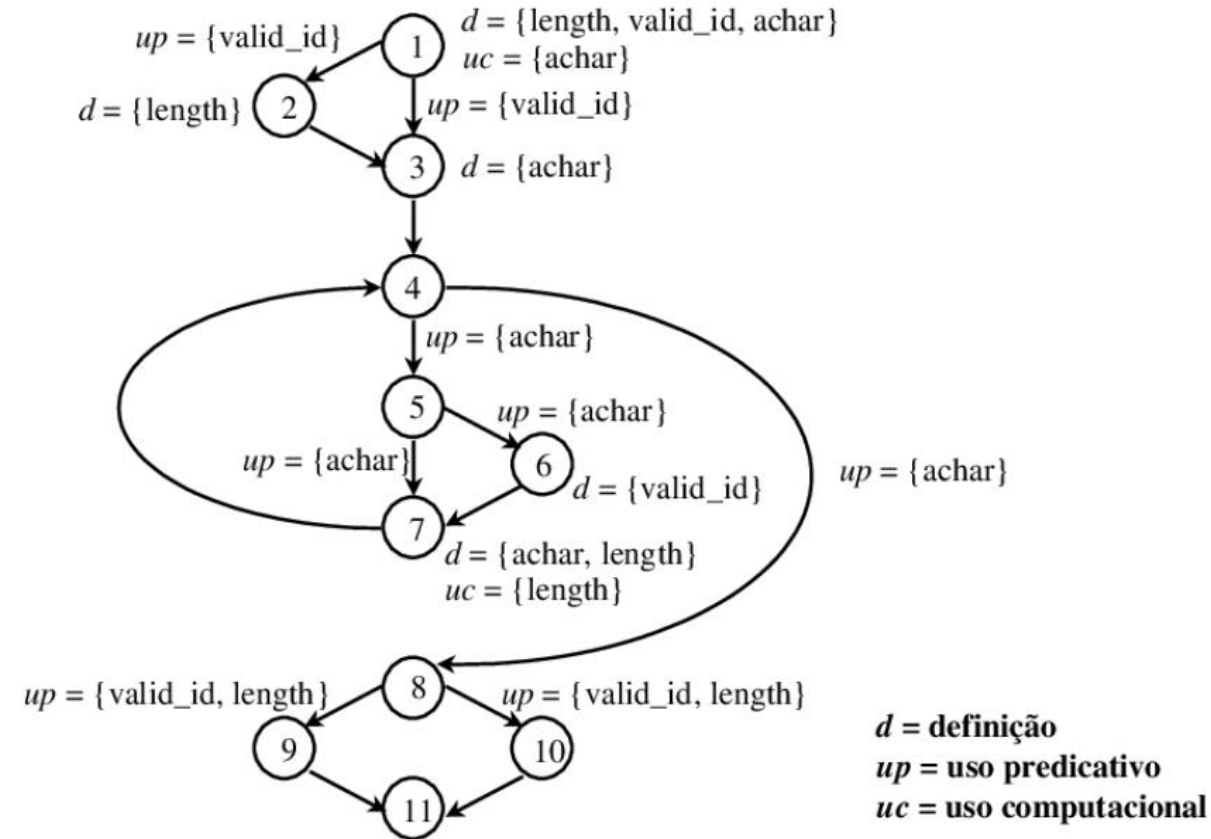
```

/* 01 */ {
/* 01 */     char  achar;
/* 01 */     int length, valid_id;
/* 01 */     length = 0;
/* 01 */     printf ("Identificador: ");
/* 01 */     achar = fgetc (stdin);
/* 01 */     valid_id = valid_s(achar);
/* 01 */     if (valid_id)
/* 02 */         length = 1;
/* 03 */     achar = fgetc (stdin);
/* 04 */     while (achar != '\n')
/* 05 */     {
/* 05 */         if (!(valid_f(achar)))
/* 06 */             valid_id = 0;
/* 07 */         length++;
/* 07 */         achar = fgetc (stdin);
/* 07 */     }
/* 08 */     if (valid_id && (length >= 1) && (length < 6))
/* 09 */         printf ("Valido\n");
/* 10 */     else
/* 10 */         printf ("Invalido\n");
/* 11 */ }

```



# CRITÉRIOS Baseados em Fluxo de Dados



## Grafo Def-Uso do Programa identifier.

Por exemplo, para exercitar a definição da variável **length** definida no nó 1, de acordo com o critério **Todas-Definições, poderiam ser executados um dos seguintes subcaminhos**: (1,3,4,5,7); (1,3,4,8,9); (1,3,4,8,10); e (1,3,4,5,6,7). **O subcaminho (1,3,4,8,9) é não executável**, e qualquer caminho completo que o inclua também é não executável.

Se qualquer um dos demais caminhos **for exercitado, o requisito de teste estaria sendo satisfeito**, e para satisfazer o **critério Todas-Definições esta análise teria que ser feita para toda definição que ocorre no programa**. Em relação ao critério Todos-Usos, com respeito à mesma definição, seriam requeridas as seguinte associações: (1,7,length);

# TÉCNICAS DE TESTE BASEADO EM ERROS

- Nessa técnica são utilizados defeitos típicos do processo de implementação de software para que sejam derivados os requisitos de teste.
- Um critério conhecido dentro dessa **técnica é o teste de mutação ou análise de mutante**. No caso do teste de mutação, **o programa que está sendo testado é alterado diversas vezes**, criando-se um **conjunto de programas alternativos conhecidos como mutantes**, como se defeitos estivessem sendo inseridos no programa original.
- O trabalho do **testador é escolher casos de teste que mostrem a diferença de comportamento entre o programa original e os programas mutantes**.



# TÉCNICAS DE TESTE BASEADO EM ERROS

A “hipótese do programador competente” diz que programadores experientes escrevem programas corretos ou bem próximos dos corretos. Já o “efeito de acoplamento” diz que casos de teste capazes de revelar erros simples são tão sensíveis que, implicitamente, também são capazes de revelar erros mais complexos.

O Git é um sistema de controle de versão gratuito e de código aberto criado por Linus Torvalds em 2005. Diferente de outros sistemas de controle de versão centralizados, como SVN e CVS, o **Git é distribuído: todo desenvolvedor tem o histórico completo de seu repositório de códigos local.**

# CVS e SVN

**CVS (Concurrent Versions System):** É um sistema de controle de versões OpenSource e Multiplataforma. É um software de versionamento de código o qual utiliza uma arquitetura de Cliente-Servidor para armazenar as versões do projeto atual. **CVS por ser um sistema centralizado**, precisa de um servidor central ao qual os desenvolvedores precisam sincronizar-se para realizar as suas alterações e não é possível dar "checkouts" reservados.

**SVN: Apache Subversion** é um sistema de controle de versão e revisão de software de código aberto sob a licença Apache. Gerenciou arquivos e pastas que estão presentes no repositório. Ele pode operar através da rede, o que o permite e é usado por pessoas em computadores diferentes.

# Git

É um software de **versionamento de código** o qual também usa uma **arquitetura de cliente-servidor** para armazenar versões do projeto. Diferente do **CVS** o **git** funciona sem um **servidor central**, permitindo aos desenvolvedores terem seus próprios repositórios enquanto há um repositório público o qual podem ser enviadas as alterações.

- **Sendo um sistema de controle de versão descentralizado**, commits podem ser feitos mais frequentemente, permitindo a implementação de funcionalidades através de diversos commits.

# CVS e SVN

GIT	SVN
Git é um sistema de vice-controle distribuído de código aberto desenvolvido por Linus Torvalds em 2005. Ele enfatiza a velocidade e a integridade dos dados	Apache Subversion é uma versão de software de código aberto e um sistema de controle de revisão sob a licença Apache.
Git tem um modelo distribuído.	SVN tem um modelo centralizado.
No git, cada usuário tem sua própria cópia de código em seu local, como sua própria ramificação.	No SVN existe um repositório central com uma cópia de trabalho que também faz alterações e confirma no repositório central.
No git, não exigimos nenhuma rede para executar a operação git.	No SVN, exigimos Rede para executar a operação SVN.
Git é mais difícil de aprender. Tem mais conceitos e comandos.	O SVN é muito mais fácil de aprender em comparação com o git.
O Git lida com um grande número de arquivos, como arquivos binários, que mudam rapidamente, por isso se tornam lentos.	SVN controla facilmente o grande número de arquivos binários.
No git, criamos apenas o diretório .git.	No SVN criamos o diretório .svn em cada pasta.
Não possui uma boa interface do usuário em comparação com o SVN.	SVN tem interface de usuário simples e melhor.

# CVS e SVN

GIT	SVN
<p>Características do GIT:</p> <ul style="list-style-type: none"><li>• Sistema distribuído.</li><li>• Ramificação.</li><li>• Compatibilidade.</li><li>• Desenvolvimento não linear.</li><li>• Leve.</li><li>• Código aberto.</li></ul>	<p>Características do SVN:</p> <ul style="list-style-type: none"><li>• Os diretórios são versionados</li><li>• Copiar, excluir e renomear.</li><li>• Metadados com versão de formato livre.</li><li>• Compromissos atômicos.</li><li>• Ramificação e marcação.</li><li>• Mesclar rastreamento.</li><li>• Bloqueio de arquivo.</li></ul>

# Git - Conceitos

**Repositório:** Lugar onde ficam guardados os arquivos sob controle do CVS que compõem os diversos módulos. Um repositório pode ser local (armazenado na própria estação de trabalho) ou em um servidor remoto.

Aqui está uma visão geral básica de como o Git funciona:

- Crie um "repositório" (projeto) com uma ferramenta de hospedagem de git (como o Bitbucket)
- Copie (ou clone) o repositório na sua máquina local
- Adicione o arquivo ou seu repositório local e faça "commit" (salve) as alterações
- "Coloque" suas alterações na sua ramificação principal
- Faça uma alteração no seu arquivo com uma ferramenta de hospedagem de git e faça commit
- "Puxe" as alterações para a sua máquina local
- Crie uma "ramificação" (versão), faça uma alteração, faça commit da alteração
- Abra uma "solicitação pull" (proponha alterações na ramificação principal)
- "Mescle" sua ramificação com a principal

# Git - Instalação

O primeiro passo será abrir o terminal, se estiverem no Mac ou Linux. Se estiverem utilizando Windows, podem utilizar o Git Bash, que vem junto com o Git.

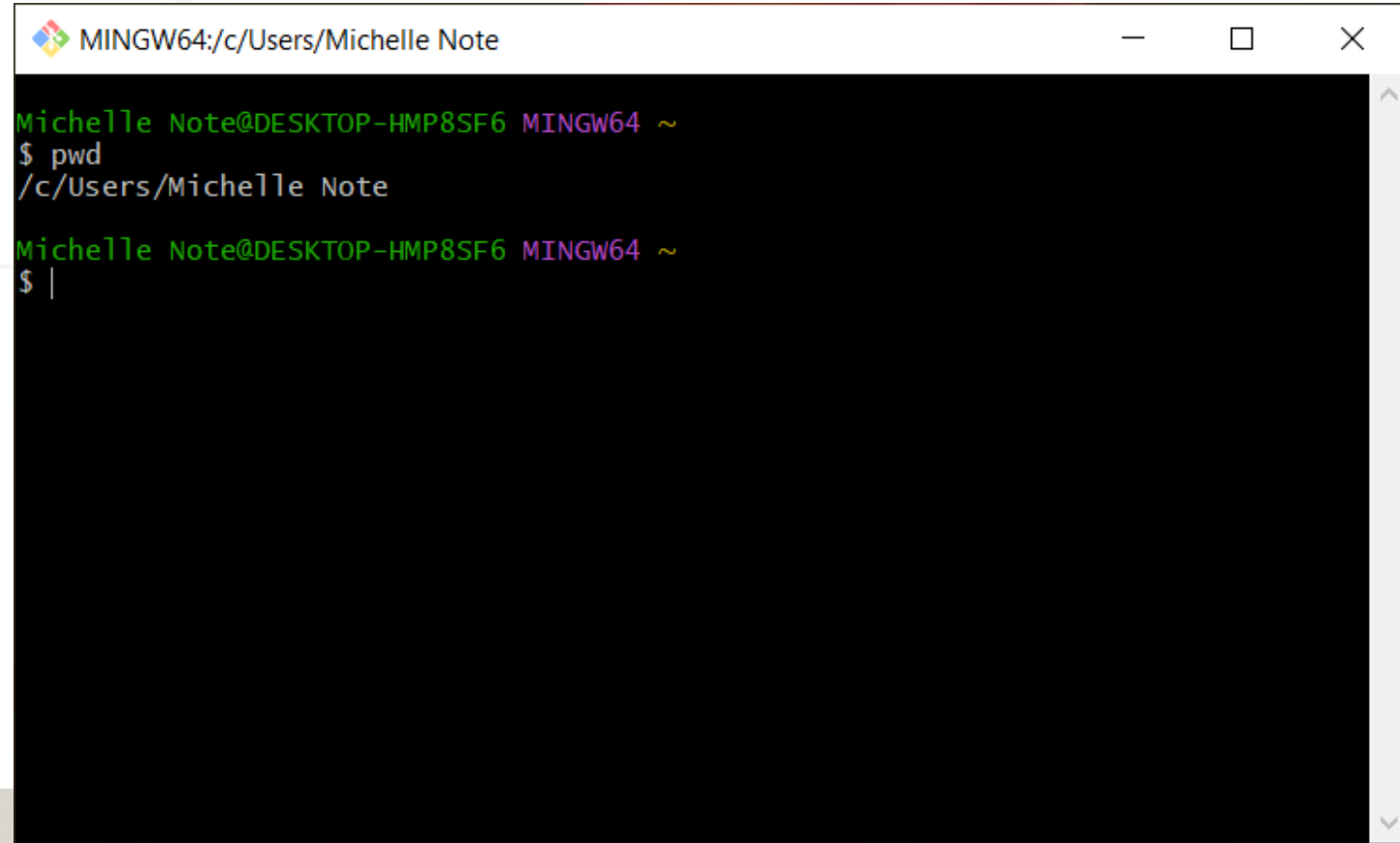
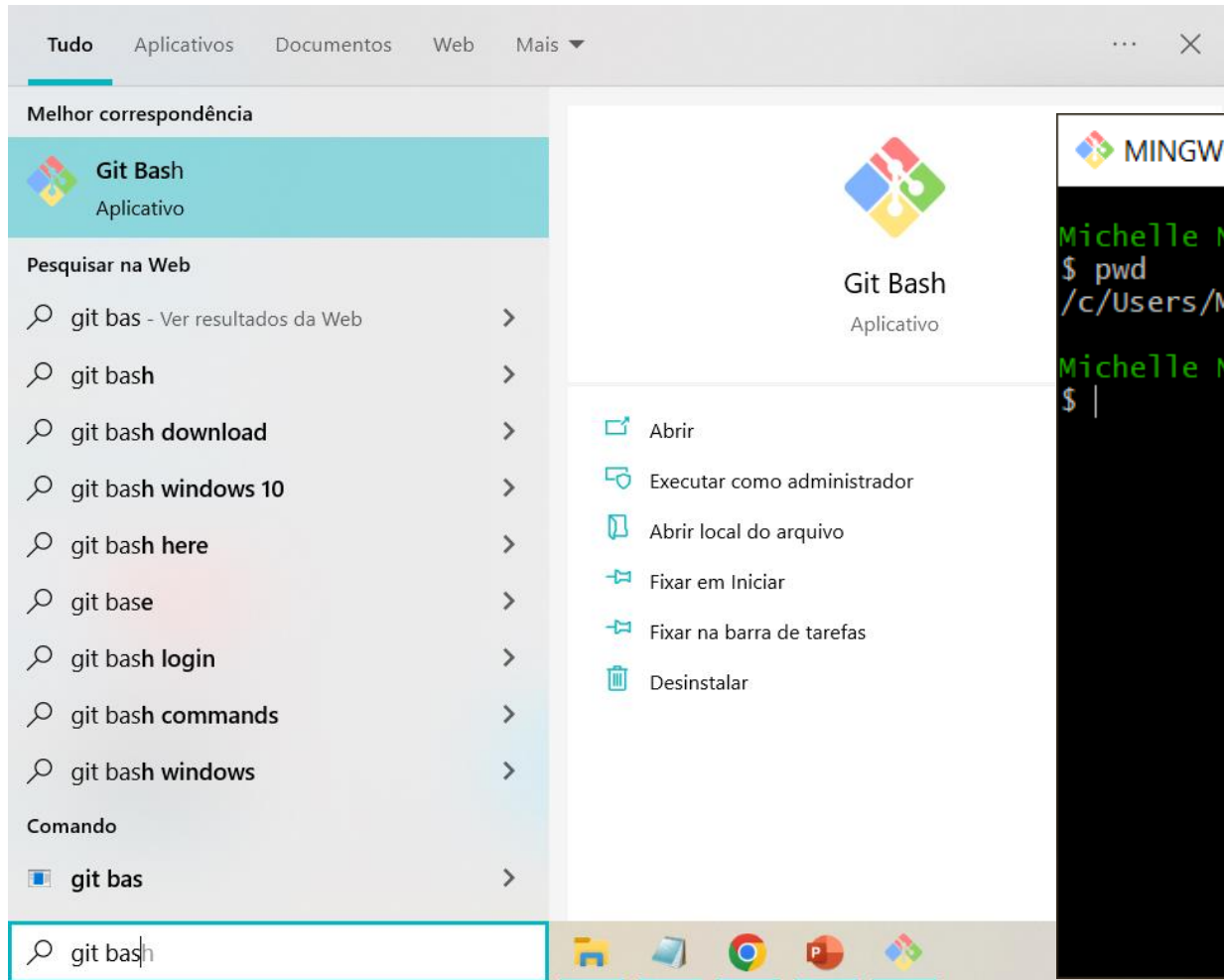
Acessar o site abaixo, faça o download, em seguida proceder a instalação:

<https://gitforwindows.org/>

O Git Bash é o aplicativo para ambientes do **Microsoft Windows** que **oferece a camada de emulação para a experiência de linha de comando Git**. Bash é acrônimo para "*Bourne Again Shell*". Shells são aplicativos terminais usados como interface em sistemas operacionais por meio de comandos gravados.



# Git - Instalação



# Git Bash

O comando do Bash **pwd** é usado para exibir o "**diretório de trabalho atual**". O **pwd** é equivalente ao executar o **cd** em terminais DOS. Essa é a pasta ou o caminho no qual está a sessão de Bash atual.

O comando do Bash **ls** é usado para listar o conteúdo do diretório de trabalho atual. O **ls** é equivalente ao **DIR** em terminais de host do console do Windows.

O Bash e o host do console do Windows têm o comando **cd** (Change Directory). **O cd é chamado junto com o nome do diretório.**

# Git Bash – Configuração Inicial

## Configuração LOCAL:

Acessar o Git Bash e configurar a sua conta de nome e e-mail:

```
git config --global user.name "seu nome"
```

```
git config --global user.email "seu-email@email.com"
```

# Git Bash – Configuração Inicial

## Configuração para conexão com o GitHub

1- Abrir o GIT BASH e digite:

**ssh-keygen -t rsa -C "seu-email@dominio.com"**

ATENÇÃO: será solicitado o usuário e senha do seu GitHub.

2- Será criada a chave **SSH** no diretórios **C:\Users\nome-do-seu-usuario.ssh\id\_rsa.pub**.

**Acessar este arquivo e copiar o conteúdo da chave.**

3- Acessar o **github**, vá em **configurações - SSH Keys - Add SSH Keys**.  
**Colar o SSH copiado** e atribuir um título para a chave.

# Git Bash – Configuração Inicial

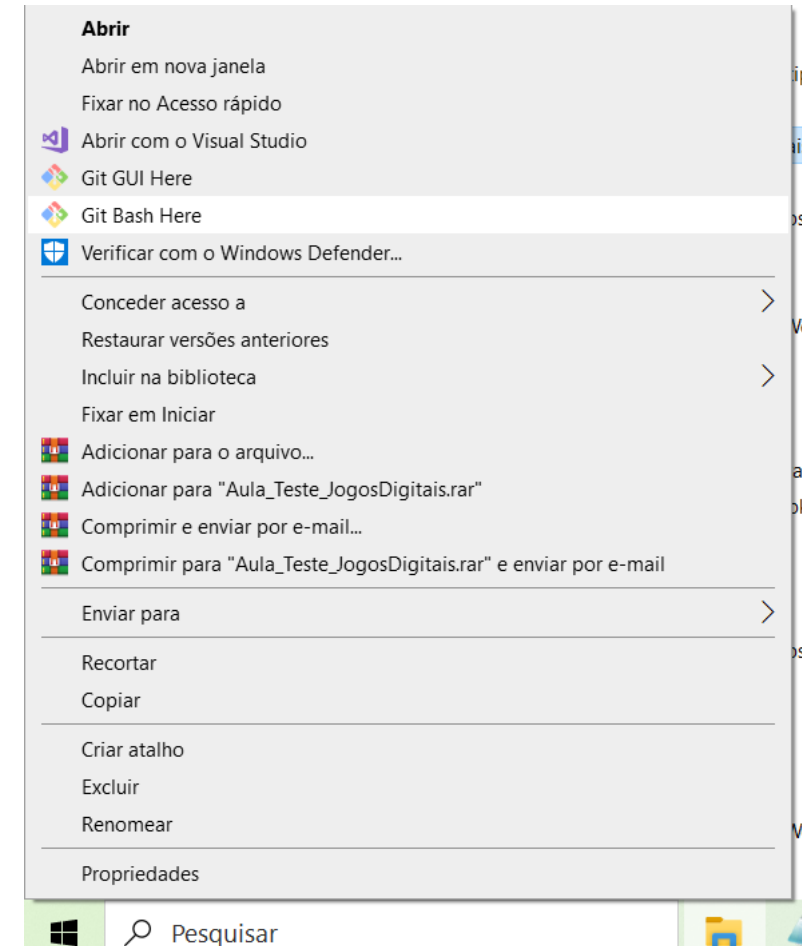
Depois de **instalar o Git...**

Crie uma pasta vazia onde será seu projeto ou clique em uma pasta que já existe para utilizar.

Para indicar que a pasta utilizará o serviço de Git, devemos inicializar:

**git init**

```
Michelle Note@DESKTOP-HMP8SF6 MINGW64 ~/Documents/GitHub/Aula_Testes/JogosDigitais
$ git init
Initialized empty Git repository in C:/Users/Michelle Note/Documents/GitHub/Aula_Testes/JogosDigitais/.git/
Michelle Note@DESKTOP-HMP8SF6 MINGW64 ~/Documents/GitHub/Aula_Testes/JogosDigitais
```



# Clonar Repositório do GitHb

**Git clone:** é uma comando para baixar o código-fonte existente de um repositório remoto (ex: Github). Em outras palavras, **git clone**, basicamente, faz uma cópia idêntica da versão mais recente de um projeto em um repositório e a salva em seu computador.

```
git clone <https://link-com-o-nome-do-repositório>
```

```
git clone https://github.com/mihanne/Exemplos_POO.git
```

# Branch

## 2. Git branch

**Branches (algo como ramificações, em português)** são altamente importantes no mundo do git. Usando as branches, vários desenvolvedores conseguem trabalhar em paralelo no mesmo projeto simultaneamente.

Podemos criar branches:

***git branch <nome-da-branch>***

Esse comando criará uma branch em seu local de trabalho. Para fazer o **push (algo como enviar)** da nova branch para o repositório remoto, você precisa usar o comando a seguir:

***git push -u <local-remoto> <nome-da-branch>***

# Branch

Como ver as branches:

**git branch ou git branch --list**

Como excluir uma branch:

**git branch -d <nome-da-branch>**



# Git Checkout

Para trabalhar em uma **branch**, primeiro, é preciso "**entrar**" nela. Usamos **git checkout**, na maioria dos casos, **para trocar de uma branch para outra**. Também podemos usar o comando **para fazer o checkout de arquivos e commits**.

**git checkout <nome-da-branch>**

Também existe um comando de atalho que permite criar e automaticamente trocar para a branch criada ao mesmo tempo:

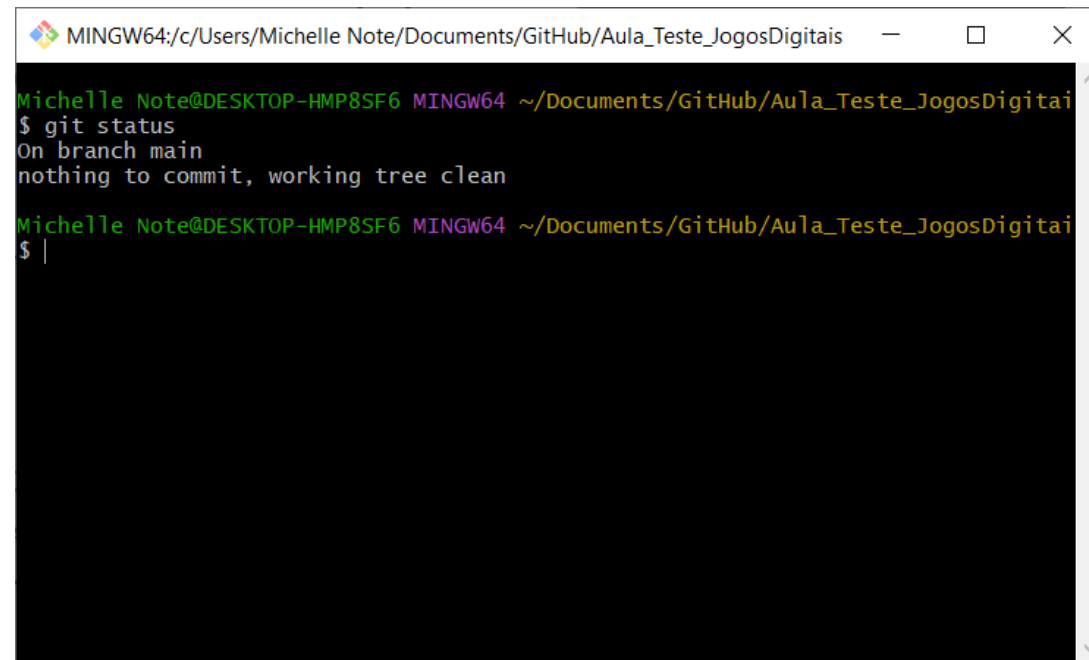
**git checkout -b <nome-da-branch>**

Esse comando cria a branch em seu espaço de trabalho local (a flag -b representa a branch) e faz o checkout na nova branch logo após sua criação.

# Git Status

O comando **git status** nos dá todas as informações necessárias sobre a **branch atual**.

## git status



```
MINGW64:/c/Users/Michelle Note/Documents/GitHub/Aula_Teste_JogosDigitais
Michelle Note@DESKTOP-HMP8SF6 MINGW64 ~/Documents/GitHub/Aula_Teste_JogosDigitais
$ git status
On branch main
nothing to commit, working tree clean

Michelle Note@DESKTOP-HMP8SF6 MINGW64 ~/Documents/GitHub/Aula_Teste_JogosDigitais
$ |
```

# Git Add

Ao criarmos, modificarmos ou excluirmos um arquivo, essas alterações acontecerão em nosso espaço de trabalho local e não serão incluídas no **próximo commit (a menos que alteremos as configurações)**.

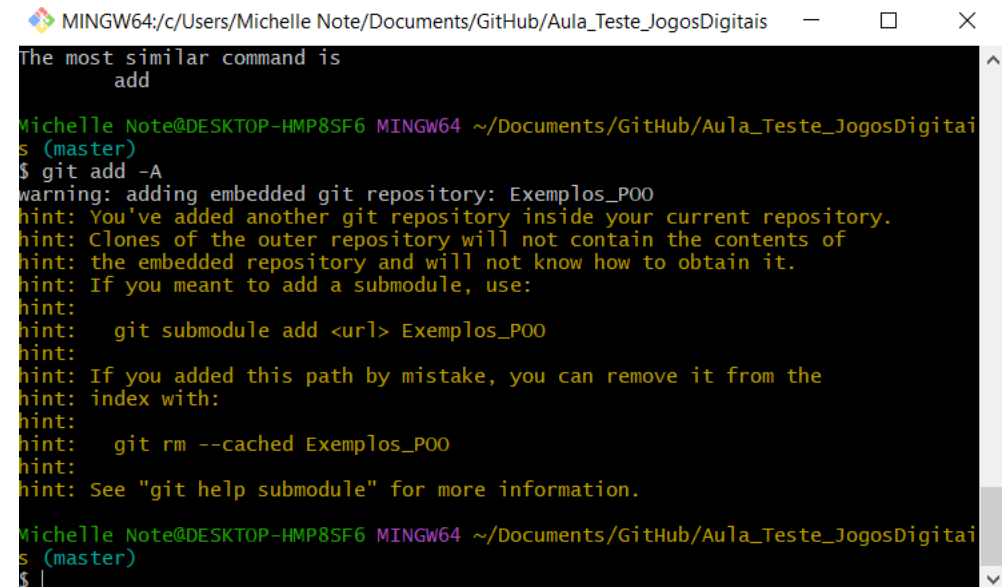
Precisamos usar o comando **git add** para incluir as alterações de um ou vários arquivos em nosso **próximo commit**.

Para adicionar um único arquivo:

**git add <arquivo>**

Para adicionar tudo ao mesmo tempo:

**git add -A**



```
MINGW64/c:/Users/Michelle Note/Documents/GitHub/Aula_Testes_JogosDigitais
The most similar command is
  add

Michelle Note@DESKTOP-HMP8SF6 MINGW64 ~/Documents/GitHub/Aula_Testes_JogosDigitais (master)
$ git add -A
warning: adding embedded git repository: Exemplos_P00
hint: You've added another git repository inside your current repository.
hint: Clones of the outer repository will not contain the contents of
hint: the embedded repository and will not know how to obtain it.
hint: If you meant to add a submodule, use:
hint:   git submodule add <url> Exemplos_P00
hint: If you added this path by mistake, you can remove it from the
hint: index with:
hint:   git rm --cached Exemplos_P00
hint: See "git help submodule" for more information.

Michelle Note@DESKTOP-HMP8SF6 MINGW64 ~/Documents/GitHub/Aula_Testes_JogosDigitais (master)
$
```

# Git Commit

Importante: o comando git add não altera o repositório. As alterações não são salvas até que se use o git commit.

Quando chegamos a determinado ponto em desenvolvimento, queremos salvar nossas alterações (talvez após uma tarefa ou resolução de problema específica). Git commit é como definir um ponto de verificação no processo de desenvolvimento.

```
git commit -m "mensagem do commit"
```

**Importante:** git commit salva suas alterações no espaço de trabalho local.

# Git Push

Após fazer o commit de suas alterações, a próxima coisa a fazer é enviar suas alterações ao servidor remoto. **Git push** faz o upload dos seus **commits** no **repositório remoto**.

```
git push <repositório-remoto> <nome-da-branch>
```

```
git push -u origin main
```

# Enviando arquivos para o GitHub

1- Criar um repositório no GitHub

2- Criar uma pasta local e acessar este diretório no Git Bash. Clicar com o botão direito na pasta e escolher a opção **Git Bash Here**.

3- Inicializar o Git na pasta:

**git init**

4- Criar/copiar os arquivos para esta pasta

5- Adicionar arquivos ao repositório.

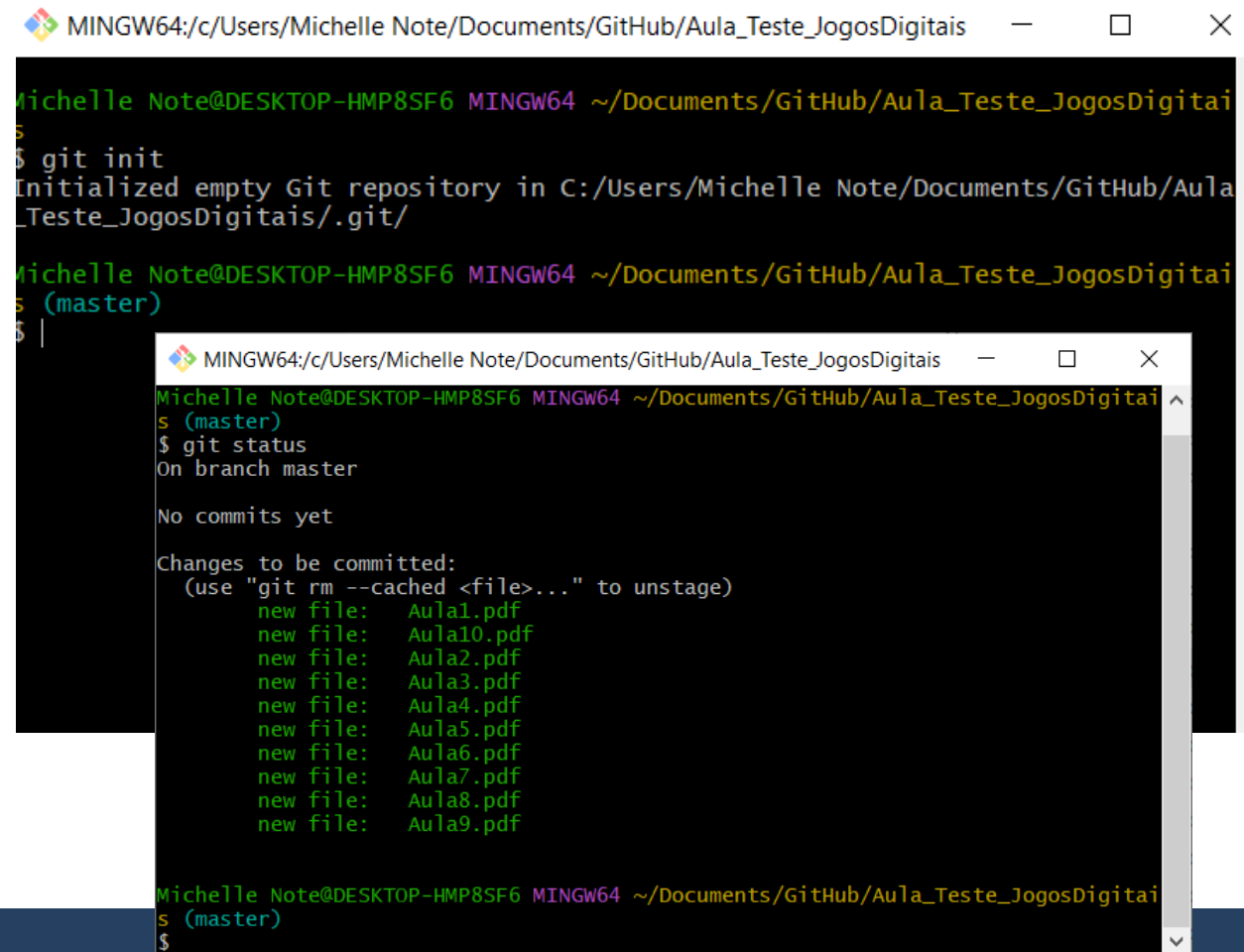
**git add .**

6- Verificar se os arquivos foram adicionados

**git status**

7- Adicionando os arquivos à Branch

**git branch -M main**



```
MINGW64:/c:/Users/Michelle Note/Documents/GitHub/Aula_Teste_JogosDigitais
$ git init
Initialized empty Git repository in C:/Users/Michelle Note/Documents/GitHub/Aula_Teste_JogosDigitais/.git/

MINGW64:/c:/Users/Michelle Note/Documents/GitHub/Aula_Teste_JogosDigitais
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   Aula1.pdf
    new file:   Aula10.pdf
    new file:   Aula2.pdf
    new file:   Aula3.pdf
    new file:   Aula4.pdf
    new file:   Aula5.pdf
    new file:   Aula6.pdf
    new file:   Aula7.pdf
    new file:   Aula8.pdf
    new file:   Aula9.pdf
```

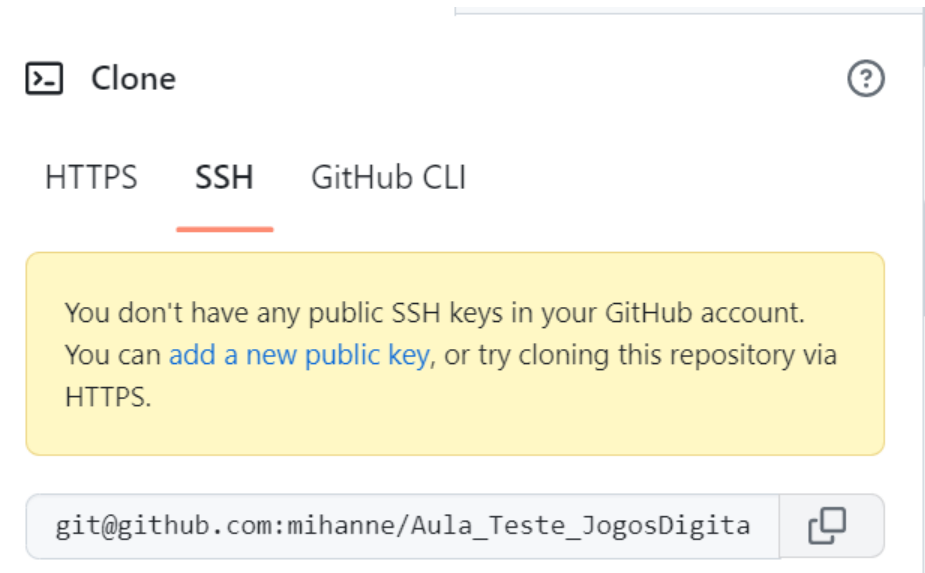
# Enviando arquivos para o GitHub

**8-** Conectando-se com seu repositório no github:  
Acessar o Github e copiar o endereço de SSH

**git remote add origin [git@github.com:mihanne/Aula\\_Testes\\_JogosDigitais.git](https://github.com/mihanne/Aula_Testes_JogosDigitais.git)**

**9-** Enviando os arquivos selecionados para o GitHub

**git push -u origin main**



# Referências

- BARBOSA, Ellen Francine et al. Introdução ao teste de software, 2000. Disponível em: [https://www.researchgate.net/publication/306255146\\_INTRODUCAO\\_AO\\_TESTE\\_DE\\_SOFTWARE](https://www.researchgate.net/publication/306255146_INTRODUCAO_AO_TESTE_DE_SOFTWARE) Acesso em: 3 fev. 2020.
- DELAMARO, Márcio Eduardo; MALDONADO, José Carlos; JINO, Mário. Introdução ao Teste de Software. 2. ed. Rio de Janeiro: Campus - Elsevier, 2016.
- NETO, Arilo. Introdução a teste de software. Engenharia de Software Magazine, v. 1, p. 22, 2007. Disponível em: [https://www.researchgate.net/profile/Arilo\\_Neto/publication/266356473\\_Introducao\\_a\\_Teste\\_de\\_Software/links/5554ee6408ae6fd2d821ba3a/Introducao-a-Teste-de-Software.pdf](https://www.researchgate.net/profile/Arilo_Neto/publication/266356473_Introducao_a_Teste_de_Software/links/5554ee6408ae6fd2d821ba3a/Introducao-a-Teste-de-Software.pdf). Acesso em: 17 dez. 2019.
- PRESSMAN, Roger S.; MAXIM, Bruce R. Engenharia de Software - uma abordagem profissional. 8. ed. Porto Alegre: Amgh Editora, 2016.

<https://www.atlassian.com/br/git>

<https://fullcycle.com.br/git-e-github/>

[https://pt.wikiversity.org/wiki/CVS\\_vs\\_Git](https://pt.wikiversity.org/wiki/CVS_vs_Git)