Atividade 1: Gerando versões

1: Usaria o esquema de versionamento SemVer (Versionamento Semântico). Porque ele é intuitivo e permite uma comunicação clara sobre a natureza das atualizações, ajudando tanto desenvolvedores quanto usuários a entender rapidamente a importância de uma nova versão. Ele e mais conhecido pelos usuários mais básicos por ser bem intuitivo

2: Nomearia a primeira versão como 1.0.0. A versão 1.0.0 representa uma versão estável e pronta para uso público, indicando que o software passou por testes e está pronto para ser utilizado na pratica

3: Se uma nova funcionalidade foi adicionada sem quebrar a compatibilidade com versões anteriores, a nova versão seria 1.1.0. O incremento do número MINOR (1) indica que há uma nova funcionalidade, mas o software ainda é compatível com a versão anterior.

4: Histórico de versões:

1.0.0: Primeira versão pública do software.

1.1.0: Nova funcionalidade adicionada.

Caso uma correção de bug seja feita posteriormente, a próxima versão seria algo como 1.1.1.

5. Correções e novas funcionalidades a partir da versão 1.1.0

A partir da versão 1.1.0, temos:

Três correções de bugs (incremente o número PATCH):

1.1.1 (primeira correção),

1.1.2 (segunda correção),

1.1.3 (terceira correção).

Duas novas funcionalidades (incremente o número MINOR, e o PATCH volta a zero):

1.2.0 (primeira nova funcionalidade),

1.3.0 (segunda nova funcionalidade).

Duas correções de bugs:

1.3.1 (primeira correção após as funcionalidades),

1.3.2 (segunda correção).

Portanto, a versão mais recente seria 1.3.2.

6. Mudanças críticas na API e sequência de novas versões

a mudança crítica na API não é compatível com a versão anterior, então deve ser incrementado o número MAJOR. Após essa mudança, ocorrem os seguintes eventos:

A mudança crítica leva à versão 2.0.0 (quebra de compatibilidade).

Uma nova funcionalidade é adicionada (incrementa MINOR):

2.1.0.

Duas correções de bugs (incrementa PATCH):

2.1.1 (primeira correção),

2.1.2 (segunda correção).

O histórico ficaria assim:

1.0.0: Primeira versão pública.

1.1.0: Nova funcionalidade.

1.1.1, 1.1.2, 1.1.3: Correções de bugs.

1.2.0, 1.3.0: Novas funcionalidades.

1.3.1, 1.3.2: Correções de bugs.

2.0.0: Mudanças críticas na API.

2.1.0: Nova funcionalidade.

2.1.1, 2.1.2: Correções de bugs.

7. Exemplos reais de software usando o esquema SemVer

Node.js:

v10.0.0: Lançamento de uma versão importante, com mudanças de compatibilidade na API.

v10.1.0: Uma nova funcionalidade foi adicionada.

v10.1.1: Correção de bugs.

Django:

v3.0.0: Mudança crítica na API, com grandes alterações na framework.

v3.1.0: Adição de novas funcionalidades.

v3.1.1 e v3.1.2: Correções de bugs.

React:

v16.0.0: Lançamento de uma nova versão com mudanças na arquitetura.

v16.3.0: Adição de uma nova funcionalidade.

v16.3.1 e v16.3.2: Correções de bugs

Atividade 2: CONFIGURAÇÃO INICIAL DO GIT: exercícios feitos no próprio repositório

Atividade: 3 REPOSITÓRIOS E PRIMEIROS PASSOS

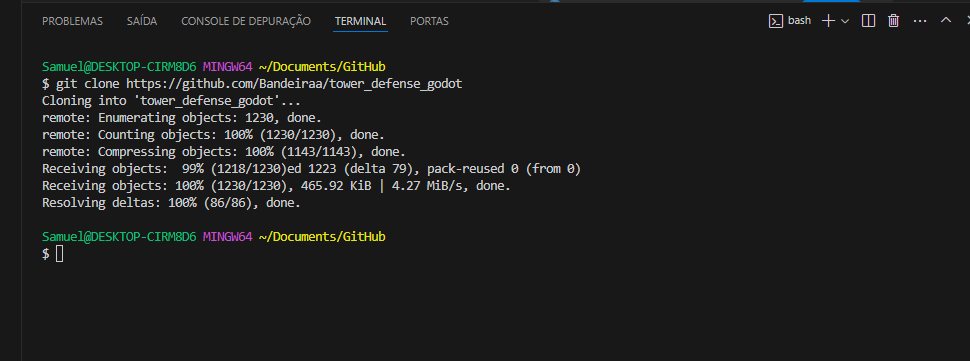
1:

<https://github.com/Bandeiraa/dawn_forest_godot_4.git>

<https://github.com/Bandeiraa/tower_defense_godot>

<https://github.com/Bandeiraa/zelda_clone>

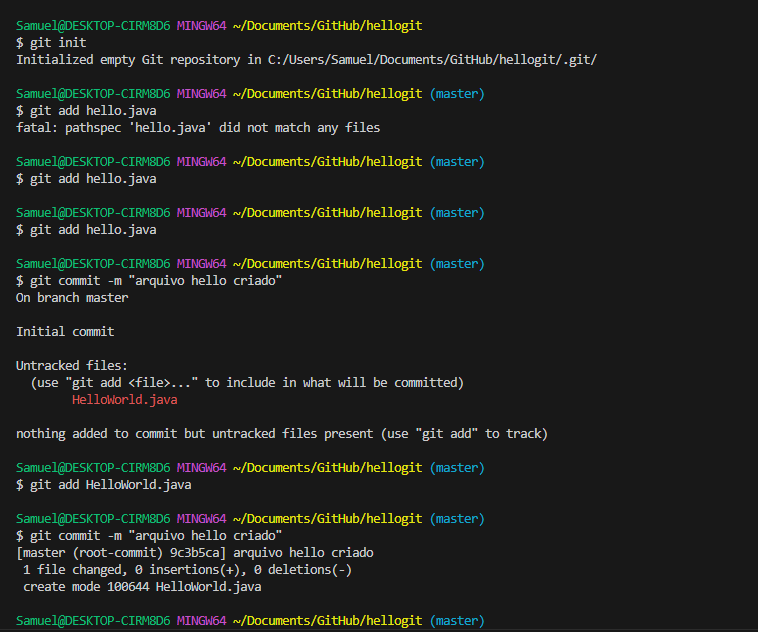
2:



1. **$ git clone** [**https://github.com/Bandeiraa/tower\_defense\_godot**](https://github.com/Bandeiraa/tower_defense_godot): O comando git clone baixa uma cópia completa do repositório tower\_defense\_godot hospedado no GitHub para a máquina local.
2. **Cloning into 'tower\_defense\_godot'...**: O Git cria uma nova pasta chamada tower\_defense\_godot no computador e começa a baixar os arquivos do repositório.
3. **remote: Enumerating objects**: O Git está identificando e listando os objetos (arquivos, commits, etc.) que fazem parte do repositório remoto.
4. **remote: Counting objects**: Ele conta todos os objetos que serão clonados.
5. **remote: Compressing objects**: O Git está comprimindo os objetos para otimizar a transferência dos arquivos.
6. **Receiving objects**: Os arquivos comprimidos estão sendo baixados para a máquina local.
7. **Resolving deltas**: O Git está resolvendo as diferenças entre os arquivos e as versões anteriores (se houver), otimizando o armazenamento local.

No final, o repositório está pronto para uso na máquina local.

3: 4 e 5

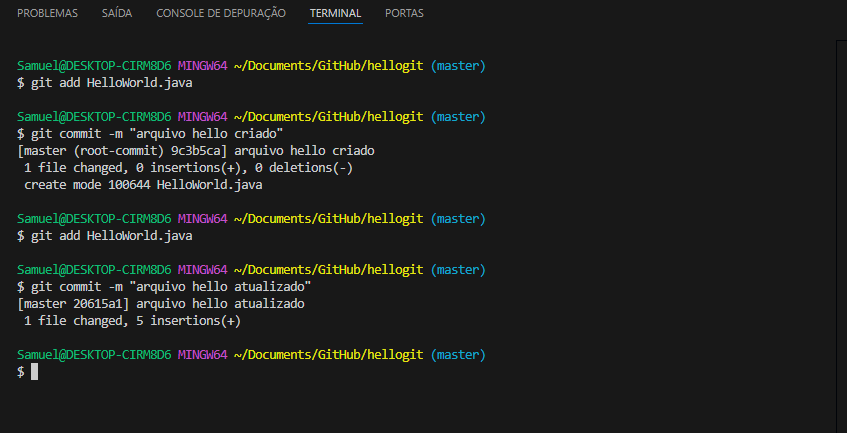


Inicialmente, houve um erro ao tentar adicionar o arquivo porque o nome estava incorreto.

Depois de corrigir o nome do arquivo e adicioná-lo corretamente, foi possível criar o commit.

Atualização no arquivo





O arquivo HelloWorld.java foi adicionado ao índice e um commit foi feito para atualizar o arquivo.

A mensagem do commit indica que o arquivo foi atualizado com 5 novas linhas de código.

ATIVIDADE: IGNORANDO ARQUIVOS

1)

!lib.a: Isso indica uma exceção. Se o arquivo lib.a estiver dentro de um diretório ou caminho que seria ignorado por uma regra anterior, esta linha força o Git a rastrear esse arquivo específico. O ! reverte a ignorância de um arquivo.

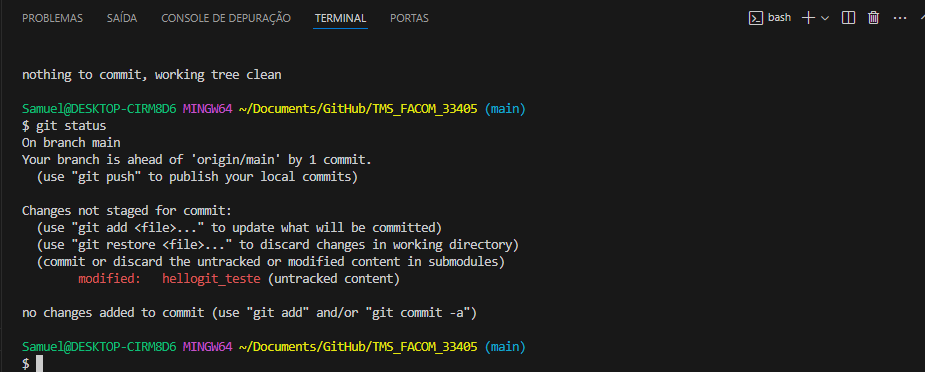
/TODO: Esta linha faz com que o Git ignore um arquivo ou diretório chamado TODO localizado na raiz do repositório (devido ao /). Ele não afeta arquivos ou diretórios com o mesmo nome em subdiretórios.

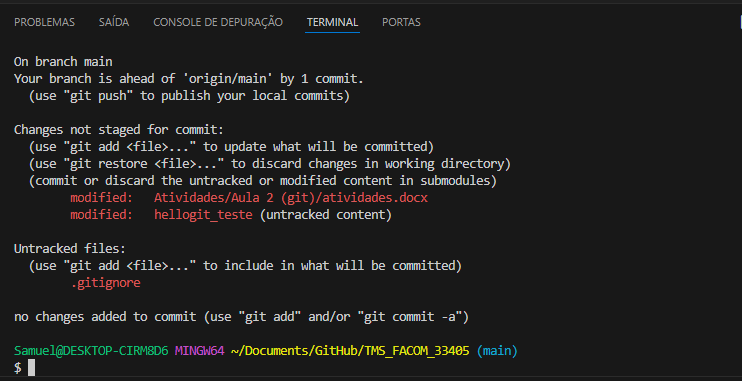
build/: O Git vai ignorar o diretório build e todo o seu conteúdo, independentemente de onde ele estiver no projeto. Tudo dentro de build/ será ignorado.

doc/\*.txt: O Git vai ignorar todos os arquivos .txt que estejam diretamente dentro do diretório doc. Arquivos .txt em subdiretórios de doc não serão ignorados.

doc/\*\*/\*.pdf: O Git vai ignorar todos os arquivos .pdf no diretório doc e em qualquer subdiretório, em qualquer nível de profundidade. A expressão \*\* indica que a busca se aplica recursivamente em todos os subdiretórios.

2)





ATIVIDADE: HISTÓRICO

Teste a consulta a seguir no repositório com o código do próprio Git: git log --pretty="%h - %s" --author=gitster -- since="2008-10-01" --before="2008-11-01" --no- merges -- t/

abc1234 - Mensagem do commit 1

def5678 - Mensagem do commit 2

ghi9101 - Mensagem do commit 3

ATIVIDADE: DESFAZENDO