

Projeto Final - Jogos de Tabuleiro

Valor: 20 pontos

Data de entrega: 23/JAN/2025

O trabalho deve ser feito em grupos de 3 a 5 pessoas.

1. Introdução

O objetivo deste trabalho é desenvolver um sistema usando o paradigma de orientação a objetos para implementar alguns jogos de tabuleiro, especialmente o Reversi, o Lig 4 e o jogo da velha. Você deverá implementar uma hierarquia de classes que permita a criação de diferentes jogos, bem como um sistema de cadastro de jogadores que permita a manutenção de estatísticas. Além da modelagem e implementação, você também deverá usar ferramentas que permitam o controle de versão e desenvolvimento em conjunto, a realização de testes e criação da documentação

Lembre-se, o objetivo não é apenas escrever um programa funcional, mas desenvolver um **sistema confiável, reutilizável e de fácil manutenção e extensão!** Logo, tente aplicar todos os conceitos de POO, modularidade e corretude vistos em sala de aula. Também serão avaliados critérios como criatividade na solução, assim como a possível implementação de funcionalidades extras.

2. Descrição Geral

Basicamente, o seu sistema vai ter 3 partes principais: uma hierarquia de classes que implementam os jogos, um módulo que cuida do cadastro de jogadores e um módulo para executar as partidas entre os jogadores.

Hierarquia de Classes para os Jogos

Você deverá criar uma classe abstrata para representar jogos de tabuleiro em geral. Essa classe deve ser capaz de representar um tabuleiro e possuir algumas funcionalidades básicas comuns aos jogos e outras que deverão ser redefinidas e especializadas nas classes herdeiras. Por exemplo, métodos para ler uma jogada, testar se ela é válida, testar condições de vitória, imprimir o tabuleiro, etc.

Além da classe base, você deverá implementar pelo menos 3 classes herdeiras para implementar 3 jogos: Reversi / Othello (<https://en.wikipedia.org/wiki/Reversi>, <https://cardgames.io/reversi/>), Lig4 (https://en.wikipedia.org/wiki/Connect_Four, <https://connect4.gamesolver.org/pt/>) e o jogo da velha (https://pt.wikipedia.org/wiki/Jogo_da_velha). Em suas classes herdeiras você deverá especializar os métodos da classe base além de criar os métodos acessórios que considerar necessários.

A modelagem específica das classes, atributos e métodos na hierarquia faz parte do trabalho.

Cadastro de Clientes

Você deverá implementar um módulo para cuidar do cadastro de jogadores. Cada jogador tem um nome, um apelido único, e estatísticas com o número de vitórias e derrotas nos jogos. Você deve possuir métodos para cadastrar e remover jogadores bem como imprimir uma listagem dos jogadores cadastrados. O cadastro deverá ser mantido entre as diferentes execuções do sistema, logo você deverá implementar mecanismos para salvar e ler o cadastro a partir de um arquivo. Os comandos básicos para o cadastro de jogadores estão especificados na Seção 2.

Execução de Partidas

O seu sistema deverá ter um mecanismo para executar as partidas entre dois jogadores. Basicamente você deverá entrar com o nome do jogo e os apelidos dos dois jogadores, e o sistema irá executar uma partida. Durante a execução da partida, você deverá ler uma jogada, testar se ela é válida, atualizar o estado do jogo, testar a condição de vitória e imprimir o tabuleiro atualizado. Ao final da partida o programa deve indicar claramente que o jogo foi finalizado e deve atualizar as estatísticas dos jogadores.

Observe que cada jogo possui dimensões apropriadas para o tabuleiro, e em particular Lig4 possui alguns tamanhos distintos aceitáveis. O grupo deve especificar esses tamanhos em seu programa.

Extras

Além da implementação básica, você poderá (deverá?) criar funcionalidades extras. Você pode querer permitir que o usuário escolha o tamanho do tabuleiro, implementar outros jogos, desenvolver uma interface gráfica usando alguma biblioteca do C++, ou mesmo desenvolver um agente inteligente (IA) para substituir um dos jogadores (dica: pesquise um algoritmo chamado Minimax). Essas funcionalidades poderão valer pontos extras... Seja criativo!

2. Entrada e Saída

O seu sistema deverá ler os dados e imprimir as mensagens em formato texto. Basicamente, você deverá ter um loop onde o usuário entra com os comandos e o sistema imprime mensagens. Durante a execução das partidas, você deverá ler as jogadas e imprimir o tabuleiro do jogo bem como eventuais mensagens de erro. Os comandos aceitos com as saídas esperadas ou com os respectivos erros são:

- **Cadastrar Jogador:**

CJ <Apelido> <Nome> (obs. Considere que <Apelido> é composto por uma única palavra)

Jogador <Apelido> cadastrado com sucesso

ERRO: dados incorretos

ERRO: jogador repetido

- **Remover Jogador:**

RJ <Apelido>

Jogador <Apelido> removido com sucesso

ERRO: jogador inexistente

- Listar todos os jogadores ordenados por Apelido ou Nome, seguido pelo número de vitórias e derrotas em cada jogo:

LJ [A|N]

<Apelido> <Nome>

REVERSI - V: <#vitorias> D: <#derrotas>

LIG4 - V: <#vitorias> D: <#derrotas>

VELHA - V: <#vitorias> D: <#derrotas>

- Executar Partida

EP <Jogo: (R|L|V)> <Apelido Jogador 1> <Apelido Jogador 2>

ERRO: dados incorretos

ERRO: jogador inexistente

- Finalizar Sistema

FS

Durante os jogos, a cada turno o sistema deverá solicitar uma jogada de um jogador:

Turno de jogador <Apelido>:

O jogador deverá entrar com a jogada no formato correto e o sistema deverá mostrar alguma mensagem caso a jogada seja inválida.

- Jogada no Reversi

<Linha> <Coluna>

ERRO: formato incorreto

ERRO: jogada inválida

- Jogada no Lig 4

<Coluna>

ERRO: formato incorreto

ERRO: jogada inválida

- Jogada no Jogo da Velha

<Linha> <Coluna>

ERRO: formato incorreto

ERRO: jogada inválida

Você deverá utilizar técnicas de programação defensiva, especialmente **tratamento de exceções** para verificar a consistência dos dados de entrada e se prevenir de possíveis erros de execução.

A impressão do estado do tabuleiro deve ser realizada após cada jogada. O formato a ser utilizado é livre, ficando a critério da criatividade dos desenvolvedores. Segue abaixo uma sugestão para Lig4:

| | | | | |

| O | X | O | | |

| X | O | X | X | |

| X | O | O | X | O |

Note que o tabuleiro está reduzido.

2. Modelagem

Em relação aos procedimentos de análise e elaboração dos requisitos do sistema, para cada uma das três partes descritas acima (que de certa forma funcionam como *User Stories*), você deverá elaborar os cartões de responsabilidade de classes (CRCs), de modo que se possa representar a identificação das classes e suas respectivas responsabilidades bem como as suas colaborações. Você pode usar esse site para lhe auxiliar a fazer o cartão: <https://echeung.me/crcmaker/>

Obs. A aula sobre modelagem está prevista para o dia 03/12.

3. Implementação

O trabalho deve ser feito colaborativamente pelos membros do grupo utilizando a linguagem C++. Utilize como repositório de controle de versão a ferramenta GitHub: crie uma pasta denominada “ProjetoFinal”, que por sua vez irá conter todos os arquivos de código desenvolvidos para compilação, além de adicionais arquivos de configuração e de dados. (em anexo colocamos um breve tutorial com instruções de uso e manipulação do repositório).

4. Testes

Para validar os métodos implementados de modo que você possa atestar o seu correto funcionamento, utilize a biblioteca *doctest* da linguagem C++, que permite que se possa criar testes unitários para cada método de sua respectiva classe.

Veja um exemplo abaixo, que implementa um teste unitário para um método que realiza a operação aritmética de adição entre dois números inteiros:

```
// arquivo de origem (programa_teste.cpp)
int adicionar(int a, int b) {
    return a + b;
}

// arquivo de teste (testes.cpp)
#define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
#include "doctest.h"
#include "programa_teste.cpp"

//Método que irá realizar uma sequência testes de acordo com os resultados
esperados
TEST_CASE("Teste de adição") {
    CHECK(adicionar(10, 10) == 20);
    CHECK(adicionar(-1, 1) == 0);
    CHECK(adicionar(0, 0) == 0);
}

int main() {
    doctest::Context context;
    context.addFilter("teste-case", "-----Teste do método de adição
```

```
-----");  
    return context.run();  
}
```

Observe que para execução e implementação do teste unitário, foi referenciado a biblioteca "doctest.h".

5. Documentação

Todo o desenvolvimento do projeto deverá ser documentado utilizando-se *Doxygen5* e o arquivo README do repositório. No código, detalhe as estruturas de dados utilizadas e o funcionamento dos métodos. No README, faça uma breve apresentação do problema, visão geral da solução focando na estrutura e funcionamento do programa, e as principais dificuldades encontradas. Liste e explique também os extras que tiver implementado.

Primeiramente, baixe a ferramenta a partir do link: <https://www.doxygen.nl/manual/docblocks.html>

Após instalar a ferramenta, o procedimento de execução e geração da documentação é bem simples e automatizado, utilizando o comando abaixo, via terminal, gera-se a documentação inicial:

```
doxygen -g Doxyfile
```

Ao executar o comando, será gerado o arquivo doxyfile, que por sua vez, permite que você possa customizar a geração da documentação a partir da definição dos seguintes parâmetros:

INPUT: Defina o caminho para o diretório do código-fonte.

OUTPUT_DIRECTORY: Indica o caminho de saída de geração da documentação.

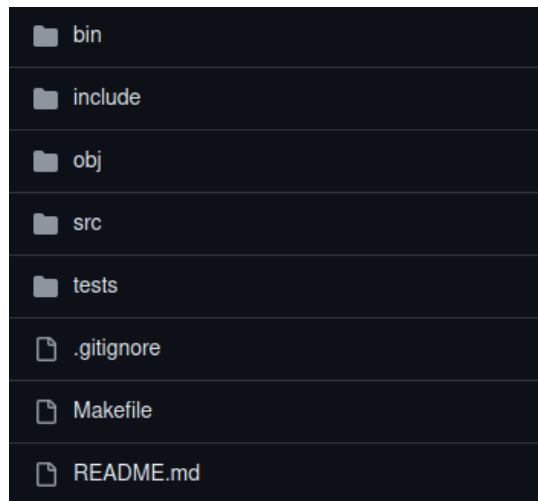
PROJECT_NAME: Atribui nome do projeto.

EXTRACT_ALL: Se definido como "YES", o Doxygen gerará documentação para todo o código, mesmo que não esteja documentado.

Após a definição dos parâmetros no arquivo, você pode executar novamente o comando para que seja gerado novamente a documentação de modo que os parâmetros definidos reflitam na documentação elaborada.

6. Estrutura do Diretório do Projeto

Uma dica para a estrutura de diretórios ao realizar trabalhos em C++ é utilizar uma organização semelhante à estrutura mostrada abaixo. Essa estrutura não é obrigatória no sentido de ser fixa, mas deve possuir organização parecida, podendo-se adicionar novas pastas para bibliotecas de terceiros, por exemplo.



- **bin/**: Armazena os executáveis gerados pela compilação.
- **include/**: Contém arquivos de cabeçalho (.hpp) usados no projeto.
- **obj/**: Guarda arquivos objeto (.o ou .obj) criados durante a compilação.
- **src/**: Onde estão os arquivos de código-fonte (.cpp).
- **tests/**: Abriga arquivos de teste para validação do código.
- **.gitignore**: Lista arquivos e pastas que o Git deve ignorar.
- **Makefile**: Contém instruções para automatizar a compilação do projeto.
- **README.md**: Fornece informações sobre o projeto, como descrição e instruções.

7. Critérios de Avaliação

- Modelagem: 2pts
- Documentação: 3pts
- Implementação Correta, sem bugs: 6pts
- Uso das técnicas de POO (Encapsulamento, herança, polimorfismo, modularidade, etc): 5pts
- Testes de Unidade / Cobertura: 2pts
- Programação defensiva, Tratamento de Exceções: 2pts

(A participação individual dos alunos será avaliada pelos *commits* feitos no github e também durante a entrevista. **Todos os membros do grupo devem estar presentes na entrevista**)

8. Cronograma

O cronograma de atividades do trabalho é mostrado abaixo:

Atividade	Data
Definição do Grupo	Até dia 12/DEZ/24
Entrega (Github)	Até 23/JAN/2025
Apresentação / Entrevista	28 e 30 JAN/2025

As informações sobre os grupos deverão ser submetidas no moodle.

Anexo: Tutorial Github

Passo 1: Criar uma Conta no GitHub

Se você ainda não tem uma conta no GitHub, comece criando uma em <https://github.com>.

Passo 2: Instalar o Git

Realize a instalação do Github em seu computador.

Link para baixá-lo: <https://git-scm.com/downloads>.

Passo 3: Configurar o Git

Depois de instalar o Git, configure-o com seu nome de usuário e endereço de e-mail. Abra um terminal ou prompt de comando e digite os seguintes comandos, substituindo "Seu Nome" e "seu@email.com" pelo seu nome de usuário e endereço de e-mail do GitHub:

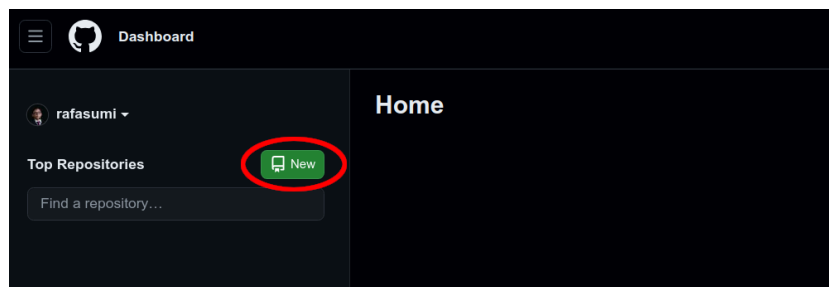
```
git config --global user.name "Seu Nome"
```

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

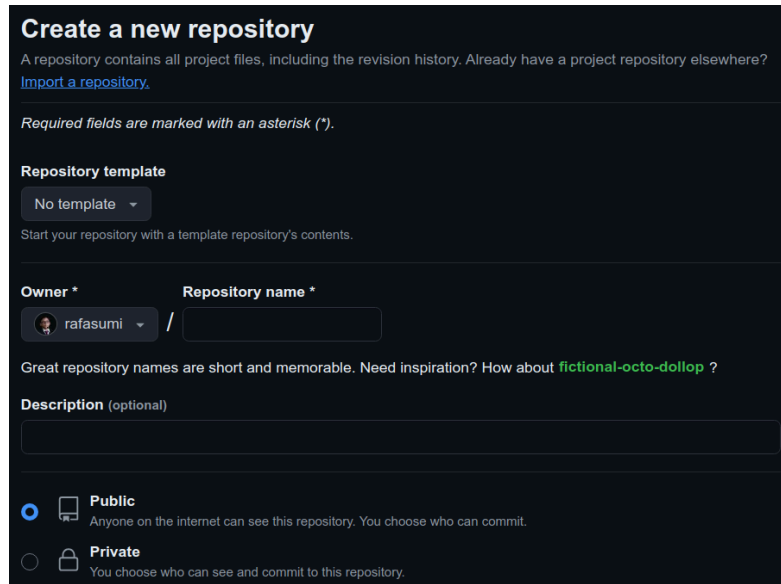
Passo 4: Criar um Novo Repositório no GitHub

Faça login na sua conta do GitHub.

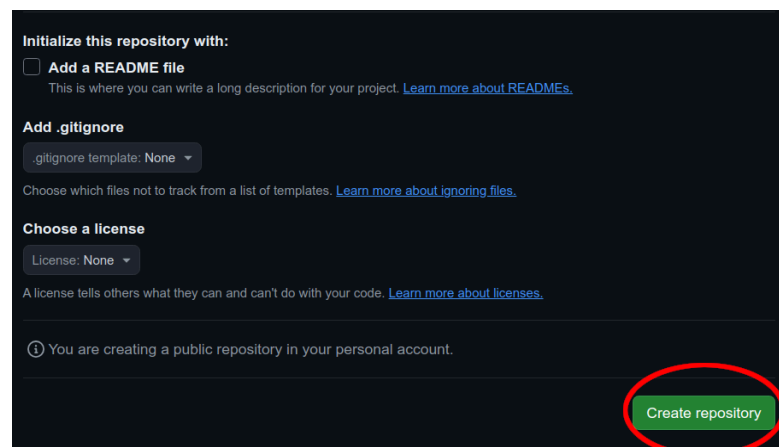
Clique no botão "New" na página inicial do GitHub para criar um novo repositório.



Preencha o nome do repositório, a descrição (opcional), escolha a visibilidade (público ou privado) e selecione as opções adicionais conforme necessário.



Clique em "Create repository" para criar o repositório.



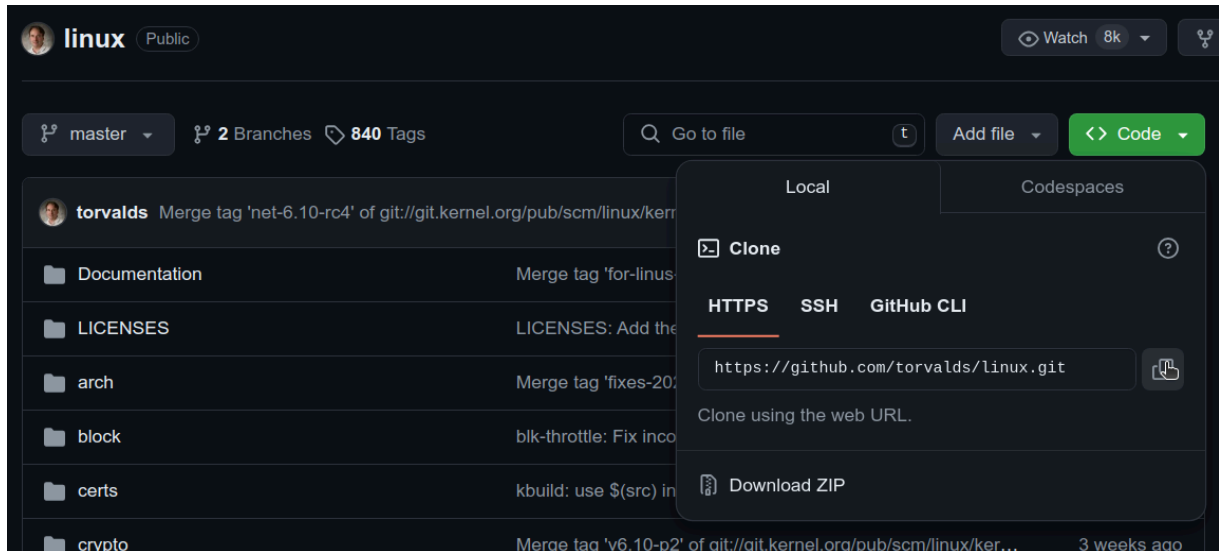
Passo 5: Clonar o Repositório no Seu Computador

Ao criar um repositório na plataforma Github, você precisa cloná-lo para o seu computador.

Para fazer isso, execute o seguinte comando no seu terminal, substituindo seu-nome-de-usuário pelo seu nome de usuário do GitHub e nome-do-repositório pelo nome do repositório que você criou:

```
git clone https://github.com/seu-nome-de-usuário/nome-do-repositório.git
```

Para obter a URL sem precisar digitar, você também pode clicar no botão “Code” na página do seu repositório e copiá-la de lá.



Passo 6: Adicionar Arquivos ao Repositório

Navegue até a pasta do repositório clonado usando o comando `cd nome-do-repositório`.

Crie um arquivo ou adicione arquivos existentes na pasta do repositório.

Use o comando `git add` para adicionar os arquivos ao controle de versão. Por exemplo, para adicionar todos os arquivos, você pode usar `git add ..`

Passo 7: Fazer um Commit

Depois de adicionar os arquivos, você precisa fazer um commit para registrar as alterações. Use o seguinte comando:

```
git commit -m "Adicionar arquivos iniciais"
```

Substitua "Adicionar arquivos iniciais" pela mensagem de commit que descreve as alterações que você fez.

Passo 8: Enviar as Alterações para o GitHub

Agora, você pode enviar as alterações para o GitHub usando o comando `git push`:

```
git push origin main
```

Isso envia suas alterações para o repositório no GitHub.

Criação do arquivo makefile: De forma a automatizar o processo de compilação dos arquivos desenvolvidos, crie um arquivo do tipo "make", para que ao executar o programa desenvolvido o mesmo seja construído de forma ordenada, bem como sejam definidos as dependências do programa a partir da definição de diretivas.

Para isso, utilizando como referência o sistema operacional Linux, crie um arquivo denominado “Makefile” ou “makefile” e defina as suas diretivas de acordo com o seu projeto. Inicialmente você pode trabalhar com a diretiva “all”, que sinaliza quais instruções inicialmente serão executadas. Exemplo:

```
all:meu_programa.o minha_biblioteca.o

gcc -o meu_programa meu_programa.o minha_biblioteca.o

clean:

    $(RM) meu_programa
```

Na representação acima cada diretiva representa as seguintes definições:

all: Irá compilar os arquivos contidos no diretório raiz.

clean: remove os arquivos de compilação do programa.

Você também pode criar suas próprias diretivas. Exemplo:

```
exibeMensagem:

    echo "Exibe mensagem"
```