ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO
Departamento de Engenharia de Computação e Sistemas Digitais

PCS3111 - LABORATÓRIO DE PROGRAMAÇÃO ORIENTADA A OBJETOS PARA A ENGENHARIA ELÉTRICA

EXERCÍCIO PROGRAMA 2 – 2º SEMESTRE DE 2019

Resumo

Os EPs de PCS3111 têm como objetivo exercitar os conceitos de Orientação a Objetos aprendidos em aula ao implementar um software para gerenciar competições esportivas. O software permite cadastrar diversas competições; cada competição envolve várias equipes e várias modalidades esportivas.

1 Introdução

Deseja-se criar um software para gerenciar competições esportivas (por exemplo, o InterUSP, o Engenharíadas e a Copa Zeus). Neste segundo EP será possível cadastrar dois tipos de competições: as com uma única modalidade ou as multimodalidades. Além disso, será possível salvar competições em arquivos e carrega-las.

Para implementar essas novas funcionalidades, o programa desenvolvido para o EP1 deve ser evoluído. Deve-se manter a mesma dupla do EP1 (será apenas possível desfazer a dupla, mas não formar uma nova).

A solução deve empregar adequadamente conceitos de orientação a objetos apresentados na disciplina: classe, objeto, atributo, método, encapsulamento, construtor e destrutor, herança, classe abstrata, membros com escopo de classe, programação defensiva, persistência em arquivo e os containers da STL. A qualidade do código também será avaliada (nome de atributos/métodos, nome das classes, duplicação de código etc.).

2. Projeto

Deve-se implementar em C++ as classes **Competicao**, **CompeticaoMultimodalidades, CompeticaoSimples**, **Equipe**, **Modalidade**, **Tabela**, **TabelaComOrdem** e **TabelaComPontos** além de criar um `main` que permita o funcionamento do programa como desejado.

Cada uma das classes deve ter um arquivo de definição (".h") e um arquivo de implementação (".cpp"). Os arquivos devem ter exatamente o nome da classe. Por exemplo, deve-se ter os arquivos "Equipe.cpp" e "Equipe.h". Note que você deve criar os arquivos necessários. Não se esqueça de configurar o Code::Blocks para o uso do C++11 (veja a apresentação da Aula 03 para mais detalhes).

Em relação às exceções, todas as especificadas são da biblioteca padrão (não se esqueça de fazer `#include <stdexcept>`). O texto usado como motivo da exceção não é especificado e não será avaliado.

Atenção:

- O nome das classes e a assinatura dos métodos **devem seguir exatamente** o especificado neste documento. As classes não devem possuir outros membros (atributos ou métodos) **públicos** além dos especificados**, a menos dos métodos definidos na superclasse e que precisaram ser redefinidos**.
- As classes podem possuir atributos e métodos privados ou protegidos, conforme necessário.
- Não é permitida a criação de outras classes além dessas.
- Não faça `#define` para constantes. Você pode (e deve) fazer `#define` para permitir a inclusão adequada de arquivos.

O não atendimento a esses pontos pode resultar erro de compilação na correção e, portanto, **nota `0` na correção automática**.

2.1 Classe Equipe

Uma **Equipe** representa uma equipe participante de uma competição. Cada **Equipe** tem um nome, como por exemplo, Poli, FEA e ESALQ.

Essa classe não teve mudanças em relação ao EP1: a única diferença foi a inclusão da palavra `virtual`. Com isso, ela deve possuir os seguintes métodos **públicos**:

```
Equipe(string nome);
virtual ~Equipe();

virtual string getNome();
virtual void imprimir();
```

O método `getNome` deve retornar o nome da equipe informado pelo construtor.

O método `imprimir` deve jogar na saída padrão (`cout`) os dados da **Equipe** no seguinte formato (o texto entre "<" e ">" representa o valor):

```
Equipe <nome>
```

Por exemplo, a chamada do método `imprimir` para a **Equipe** com nome *Poli* seria:

```
Equipe Poli
```

2.2 Classe Modalidade

Uma **Modalidade** é uma modalidade esportiva realizada durante uma competição. Cada **Modalidade** possui um nome e diversas **Equipes** participantes. Diferentemente do EP1, a **Modalidade** não calcula mais as posições: isso foi delegado para a **TabelaComOrdem** (um dos tipos de **Tabela**). Isso melhora o reaproveitamento de código e melhora a coesão da classe **Modalidade**. A seguir são apresentados os métodos **públicos** dessa classe:

```
Modalidade(string nome, Equipe** participantes, int quantidade);
virtual ~Modalidade();
```

```
virtual string getNome();
virtual Equipe** getEquipes();
virtual int getQuantidadeDeEquipes();

virtual void setResultado(Equipe** ordem);
virtual bool temResultado();
virtual TabelaComOrdem* getTabela();

virtual void imprimir();
```

A assinatura do construtor se manteve igual à do EP1, mas agora ele deve jogar uma exceção do tipo **invalid_argument** caso a quantidade de equipes seja menor do que 2. Os valores recebidos no construtor devem ser retornados pelos métodos `getNome`, `getEquipes` e `getQuantidadeDeEquipes`, respectivamente. Não destrua as **Equipes** no destrutor.

O método `setResultado` recebe a ordem de colocação das **Equipes** participantes e deve repassar para um objeto **TabelaComOrdem**. A **Equipe** na posição 0 do vetor é a campeã (1ª colocada); a **Equipe** na posição 1 do vetor é a 2ª colocada; e assim por diante. Assim como no EP1, considere que o vetor `ordem` passado como parâmetro possui a mesma quantidade de **Equipes** informada no construtor (*por simplicidade não é necessário verificar*). Para facilitar a persistência, o método `temResultado` deve retornar um booleano informando se o método `setResultado` foi chamado *pelo menos uma vez*. Ou seja, o método `temResultado` deve retornar false enquanto o método `setResultado` não for chamado. Após o método `setResultado` ser chamado, o método `temResultado` deve retornar true.

O método `imprimir` deve jogar na saída padrão (`cout`) os dados da **Modalidade**. O formato deve ser (os textos entre "<" e ">" representam valores e chamadas):

```
Modalidade: <nome>
<Chamada do método imprimir da TabelaComOrdem>
```

Veja na classe **TabelaComOrdem** a especificação de seu método `imprimir`. Um exemplo de impressão da modalidade "futebol", em que participam as equipes ESALQ, Poli e FEA, e a qual ainda não tem um resultado é apresentado a seguir:

```
Modalidade: futebol
      Poli
      FEA
      ESALQ
```

2.3 Classe Tabela

Uma **Tabela** cuida da ordem das **Equipes**. Existem dois tipos de tabela: **TabelaComOrdem** e **TabelaComPontos**. Com isso, **Tabela** deve ser uma classe **abstrata**. Escolha o(s) método(s) mais adequado(s) para serem **abstrato(s)**. A seguir são apresentados os métodos **públicos** dessa classe:

```
Tabela(Equipe** participantes, int quantidade);
virtual ~Tabela();

virtual int getPosicao (Equipe* participante);
virtual Equipe** getEquipesEmOrdem();
virtual int getQuantidadeDeEquipes();
virtual void imprimir();
```

O construtor da **Tabela** deve receber as **Equipes** participantes e a quantidade delas. O construtor deve jogar uma exceção do tipo **invalid_argument** caso a quantidade de equipes seja menor do que 2. Não destrua as **Equipes** no destrutor.

O método `getQuantidadeDeEquipes` deve retornar o número de **Equipes**, conforme informado no construtor.

O método `getPosicao` deve informar a ordem (posição) da **Equipe** na **Tabela**. Essa ordem depende se a **Tabela** é uma **TabelaComOrdem** ou **TabelaComPontos**:

- Se for uma **TabelaComOrdem**, o método deve retornar 1 para a **Equipe** que estiver na posição 0 do vetor com o resultado, 2 para a **Equipe** que estiver na posição 1 do vetor com o resultado e assim por diante.
- Se for uma **TabelaComPontos**, o método deve retornar 1 para a **Equipe** com o maior número de pontos, 2 para a **Equipe** com a segunda maior pontuação, 3 para a **Equipe** com a terceira maior pontuação e assim por diante. Caso diversas **Equipes** estejam com a mesma pontuação, deve-se considerar que as **Equipes** estão empatadas na mesma posição – que corresponde à menor posição. Por exemplo, considere as **Equipes** ESALQ, Poli e FEA. Caso a pontuação seja [8, 23, 0], o método `getPosicao` deve retornar 1 para a Poli, 2 para a ESALQ e 3 para a FEA. Caso a pontuação seja [8, 23, 23], o método `getPosicao` deve retornar 1 para a Poli, 1 para a FEA e 3 para a ESALQ. Com a pontuação [8, 23, 8], o método `getPosicao` deve retornar 1 para a Poli, 2 para a FEA e 2 para a ESALQ.

Independente do tipo de **Tabela**, caso a **Equipe** não seja um dos participantes (informados no construtor), deve-se jogar uma exceção do tipo **invalid_argument**. Além disso, caso nenhuma **Equipe** tenha pontos (no caso de **TabelaComPontos**) ou caso ainda não se tenha definido o resultado (no caso de **TabelaComOrdem**), esse método deve jogar uma exceção do tipo **logic_error**.

O método `getEquipesEmOrdem` deve retornar um vetor com as **Equipes** ordenadas, da **Equipe** melhor colocada para a **Equipe** com a pior colocação. No caso de **TabelaComOrdem**, o vetor é o mesmo vetor informado como resultado; no caso de **TabelaComPontos**, a equipe com maior pontuação deve ficar na posição 0, a equipe com a segunda maior pontuação deve ficar na posição 1 e assim por diante. No caso de equipes empatadas na mesma posição, coloque-as na ordem que for mais conveniente para você. Por exemplo, caso a equipe Poli tenha pontuação 23, a ESALQ 23 e a FEA 8, o método deve retornar ou o vetor de **Equipes** {Poli, ESALQ, FEA} ou {ESALQ, Poli, FEA}. Assim como no método `getPosicao`, caso nenhuma **Equipe** tenha pontos (no caso de **TabelaComPontos**) ou caso ainda não se tenha definido o resultado (no caso de **TabelaComOrdem**), o método `getEquipesEmOrdem` deve jogar uma exceção do tipo **logic_error**.

O método `imprimir` deve jogar na saída padrão (`cout`) os dados da **Tabela**. *Coloque sempre uma tabulação* (`'\t'`) *no início de cada linha*, para formata-la na apresentação. O formato depende do tipo da **Tabela** e se já há uma ordem:

- Caso nenhuma **Equipe** tenha pontos (no caso de **TabelaComPontos**) ou caso ainda não se tenha definido o resultado (no caso de **TabelaComOrdem**), o formato deve ser o seguinte (os textos entre "<" e ">" representam valores):

```
<nome da Equipe 1>
<nome da Equipe 2>
...
<nome da Equipe n>
```

Use a ordem das **Equipes** informada no construtor. Por exemplo, para as **Equipes** ESALQ, Poli e FEA, teríamos a saída:

```
ESALQ
Poli
FEA
```

- Caso seja uma **TabelaComOrdem**, apresente na saída no seguinte formato:

```
1o <nome da Equipe 1>
2o <nome da Equipe 2>
...
no <nome da Equipe n>
```

Por exemplo, para um resultado {Poli, ESALQ, FEA}, a saída seria:

```
1o Poli
2o ESALQ
3o FEA
```

- Caso seja uma **TabelaComPontos**, o formato é o seguinte:

```
1o <nome da Equipe 1> (<pontos da Equipe 1> pontos)
2o <nome da Equipe 2> (<pontos da Equipe 2> pontos)
...
no <nome da Equipe n> (<pontos da Equipe n> pontos)
```

Use "pontos" no plural mesmo se a Equipe possuir somente 1 ponto. Por exemplo, caso a **Equipe** Poli tenha 13 pontos, a FEA 10 pontos e a ESALQ 8 pontos, a saída seria:

```
1o Poli (13 pontos)
2o FEA (10 pontos)
3o ESALQ (8 pontos)
```

2.4 Classe TabelaComOrdem

A **TabelaComOrdem** é um subtipo de **Tabela** no qual a colocação das **Equipes** é definida ao informar a ordem delas. Com isso, ela deve ter os seguintes métodos públicos específicos a essa classe:

```
TabelaComOrdem (Equipe** participantes, int quantidade);
virtual ~TabelaComOrdem();

void setResultado (Equipe** ordem);
```

Assim como na **Tabela**, o construtor da **TabelaComOrdem** deve receber as **Equipes** participantes e a quantidade delas. O construtor deve jogar uma exceção do tipo **invalid_argument** caso a quantidade de equipes seja menor do que 2. Não destrua as **Equipes** no destrutor.

O método `setResultado` recebe a ordem de colocação das **Equipes** participantes e deve armazená-la. A **Equipe** na posição 0 do vetor é a campeã (1ª colocada); a **Equipe** na posição 1 do vetor é a 2ª colocada; e assim por diante. Considere que o vetor `ordem` passado como parâmetro possui a mesma quantidade de **Equipes** informada no construtor.

2.5 Classe TabelaComPontos

A **TabelaComPontos** é um subtipo de **Tabela** no qual a colocação das **Equipes** é definida por pontos: quanto maior o número de pontos, melhor é a colocação da equipe. Com isso, ela deve ter os seguintes métodos públicos específicos a essa classe:

```
TabelaComPontos (Equipe** participantes, int quantidade);
virtual ~TabelaComPontos();

int getPontos (Equipe* participante);
void pontuar (Equipe* participante, int pontos);
```

Assim como na **Tabela**, o construtor da **TabelaComPontos** deve receber as **Equipes** participantes e a quantidade delas. O construtor deve jogar uma exceção do tipo **invalid_argument** caso a quantidade de equipes seja menor do que 2. Não destrua as **Equipes** no destrutor.

Seguindo o EP1, o método pontuar deve *adicionar* aos pontos atuais da **Equipe** informada como parâmetro (`participante`) o valor do parâmetro `pontos`. Inicialmente a pontuação das **Equipes** deve ser 0. Caso a **Equipe** informada não seja um participante, o método deve jogar uma exceção do tipo **invalid_argument**.

O método `getPontos` deve retornar a pontuação da **Equipe** passada como parâmetro. Caso a **Equipe** passada como parâmetro não seja um participante, o método deve jogar uma exceção do tipo **invalid_argument**.

2.6 Classe Competicao

Uma **Competicao** é uma disputa realizada entre diversas **Equipes**. Existem dois tipos de **Competicao**: a **CompeticaoSimples** e a **CompeticaoMultimodalidades**. Na **CompeticaoSimples**, só existe uma **Modalidade**; na **CompeticaoMultimodalidades** existem várias **Modalidades.** Por causa disso a classe **Competicao** deve ser **abstrata**. Escolha o(s) método(s) mais adequado(s) para serem **abstrato(s)**. A seguir são apresentados os métodos **públicos** dessa classe:

```
Competicao(string nome, Equipe** equipes, int quantidade);
virtual ~Competicao();

string getNome();
Equipe** getEquipes();
int getQuantidadeDeEquipes();

Tabela* getTabela();
void imprimir();
```

O construtor recebe o nome, o vetor de **Equipes** e a quantidade de equipes. O construtor deve jogar uma exceção do tipo **invalid_argument** caso a quantidade de equipes seja menor que 2. Os métodos `getNome`,

`getEquipes` e `getQuantidadeDeEquipes` devem retornar o nome, o vetor de **Equipes** e a quantidade de **Equipes** conforme informados no construtor.

O método `getTabela` deve retornar um objeto **Tabela**. Caso seja uma **CompeticaoSimples**, a **Tabela** deve ser a mesma **Tabela** da **Modalidade** – já que a **Competicao** só tem essa **Modalidade**. Caso seja uma **CompeticaoMultimodalidades**, a tabela deverá considerar as pontuações atuais (no momento da chamada do método) das **Equipes** em todas as **Modalidades** que foram adicionadas. Para isso deve-se acumular na **Tabela** a pontuação obtida por cada **Equipe** em cada uma das **Modalidades** da **Competicao**. **Modalidades** sem resultado não devem ser contabilizadas (devendo ser ignoradas). Para mais detalhes da pontuação, veja os detalhes da classe **CompeticaoMultimodalidades**. No caso de uma **CompeticaoMultimodalidades**, caso ainda nenhuma **Modalidade** tenha sido adicionada, o método deve jogar uma exceção do tipo **invalid_argument**.

O método `imprimir` deve jogar na saída padrão (`cout`) os dados da **Competicao** no seguinte formato (os textos entre "<" e ">" representam valores):

```
<Nome da competição>
<Chamada do método imprimir da Tabela ou vazio caso ainda não haja Tabela>
```

Por exemplo, para a competição InterUSP, multimodalidades, com uma tabela com ESALQ: 21, Poli: 23 e FEA: 10, a impressão deve ser:

```
InterUSP
	1o Poli (23 pontos)
	2o ESALQ (21 pontos)
	3o FEA (18 pontos)
```

Para uma competição simples, de Truco, sem resultados, a impressão deve ser:

```
Truco
	ESALQ
	Poli
	FEA
```

Para uma competição multimodalidades, InterUSP, sem modalidades adicionadas, a impressão deve ser:

```
InterUSP
```

2.7 Classe CompeticaoSimples

A **CompeticaoSimples** é um subtipo de **Competicao** em que só há uma **Modalidade**. Com isso, ela deve ter os seguintes métodos públicos específicos a essa classe:

```
CompeticaoSimples(string nome, Equipe** equipes, int quantidade, Modalidade* m);
virtual ~CompeticaoSimples();

Modalidade* getModalidade();
```

O construtor deve receber, além do nome, das equipes e da quantidade de equipes, a **Modalidade** tratada por essa competição. O método `getModalidade` deve retornar a **Modalidade** informada no construtor.

2.8 Classe CompeticaoMultimodalidades

A **CompeticaoMultimodalidades** é um subtipo de **Competicao** em que é possível ter várias **Modalidades**. Para permitir a adição das modalidades, será usado um `list`, da biblioteca padrão. Diferentemente do EP1, a pontuação por colocação poderá ser alterada usando um método com escopo de classe. Com isso, a **CompeticaoMultimodalidades** deve ter os seguintes métodos públicos específicos a essa classe:

```
CompeticaoMultimodalidades(string nome, Equipe** equipes, int quantidade);
virtual ~CompeticaoMultimodalidades();

void adicionar(Modalidade* m);
list<Modalidade*>* getModalidades();

static void setPontuacao(vector<int>* pontos);
static int getPontoPorPosicao(int posicao);
```

O construtor deve receber apenas o nome, as equipes e a quantidade de equipes. Para adicionar uma **Modalidade** deve-se usar o método `adicionar`, o qual colocará a Modalidade passada como parâmetro em um `list`. O `list` com todas as **Modalidades** adicionadas deve ser retornado pelo método `getModalidades`.

Para definir a pontuação por posição da **Equipe** na **Competicao** criou-se o método `setPontuacao`. Esse método é estático, ou seja, ao alterar a pontuação, todas as **CompeticoesMultimodalidades** devem ter o critério de pontos por posição alterado. O método recebe como parâmetro um `vector`, da biblioteca padrão, com a pontuação por posição. A posição 0 do `vector` representa a pontuação do 1º colocado, a da posição 1 representa a do 2º colocado e assim por diante. O `vector` deve ter no mínimo 3 elementos; caso seu tamanho seja menor que 3, deve-se jogar uma exceção do tipo **invalid_argument**. Como valor inicial para o `vector` considere a seguinte pontuação:

1º lugar: 13 pontos
2º lugar: 10 pontos
3º lugar: 8 pontos
4º lugar: 7 pontos
5º lugar: 5 pontos
6º lugar: 4 pontos
7º lugar: 3 pontos
8º lugar: 2 pontos
9º lugar: 1 ponto

Dica: para criar um `vector` com esses valores pode-se fazer:

```
new vector<int>({13, 10, 8, 7, 5, 4, 3, 2, 1})
```

Os valores da pontuação devem ser obtidos pelo método `getPontoPorPosicao`, que também é estático. O método recebe uma posição (começando em 1) e deve retornar o valor da pontuação que se ganha por essa colocação. Caso a colocação não tenha um valor definido, ou a colocação seja um valor menor ou igual a zero, o método deve retornar 0. Por exemplo, considerando o valor inicial de pontuação, o método deve retornar 13 para um parâmetro 1; deve retornar 2 para um parâmetro 8 e retornar 0 para um parâmetro 11.

A **Tabela** gerada pelo método `getTabela` (definido na classe pai) deve representar os valores da pontuação no momento da chamada do método. Por exemplo, suponha uma competição com 3 **Equipes**: ESALQ, Poli e FEA. Essa competição possui 2 **Modalidades**: futebol e basquete. Suponha que a **Modalidade** basquete ainda não teve resultado e a ordem das **Equipes** no futebol foi [Poli, FEA, ESALQ]. Considerando os valores inicias de pontuação, a tabela obtida pelo método `getTabela` deve possuir os valores ESALQ: 8, Poli: 13 e FEA: 10. Caso a modalidade basquete tenha como resultado [ESALQ, Poli] (ou seja, a FEA não participou), ao chamar novamente o método `getTabela` deve-se obter os valores ESALQ: 21, Poli: 23 e FEA: 10.

3 Persistência

O software permitirá salvar e carregar projetos salvos em disco. Para isso deve ser implementada a classe **PersistenciaDeCompeticao**. A seguir é apresentado o formato do arquivo, exemplos de arquivos e a especificação da classe.

3.1 Formato do arquivo

A persistência da competição deve seguir o formato de arquivo especificado a seguir. Cada competição será salva em um arquivo separado. Entre "<" e ">" são especificados os valores esperados. Por simplicidade, será utilizado um caractere de nova linha ('\n') ou espaço (' ') como delimitador (pode ser qualquer um deles) e *assuma* que não existem espaços nos campos que sejam `string`. O formato do arquivo depende se a competição é uma **CompeticaoSimples** ou **CompeticaoMultiModalidades**.

3.1.1 Formato para CompeticaoSimples

O formato para **CompeticaoSimples** é o seguinte:

```
<quantidade de equipes>
<nome da equipe 1>
<nome da equipe 2>
...
<nome da competição>
0
<nome da modalidade>
<1 se tem resultado ou 0 caso contrário>
<quantidade de participantes>
<nomes das equipes em ordem de colocação, ou em ordem qualquer se sem resultado>
...
```

Note que há uma linha vazia no final do arquivo. Por exemplo, a seguir é apresentada uma **CompeticaoSimples** chamada de "InterTruco" com 4 equipes: FEA, ESALQ, Poli e FOUSP. O resultado foi {Poli, FOUSP, FEA, ESALQ}.

```
4 FEA ESALQ Poli FOUSP
InterTruco
0
Truco
1 4 Poli FOUSP FEA ESALQ
```

3.1.2 Formato para CompeticaoMultiModalidades

O formato para **CompeticaoMultiModalidades** é similar à **CompeticaoSimples**; o diferente é que podem existir várias **Modalidades**:

```
<quantidade de equipes>
<nome da equipe 1>
<nome da equipe 2>
...
<nome da competição>
1
<número de modalidades>
<nome da modalidade 1>
<1 se a modalidade 1 tem resultado ou 0 caso contrário>
<quantidade de participantes na modalidade 1>
<nomes das equipes em ordem de colocação, ou em ordem qualquer se sem resultado>
...
<nome da modalidade 2>
<1 se a modalidade 2 tem resultado ou 0 caso contrário>
<quantidade de participantes na modalidade 2>
<nomes das equipes em ordem de colocação, ou em ordem qualquer se sem resultado>
...
```

Por exemplo, para a **CompeticaoMultiModalidades** InterUSP, se tem 3 competidores: Poli, FEA e ESALQ. Existem 4 modalidades: Futebol, Volei, Basquete e Natacao. Apenas para Futebol e Volei há resultado (em Futebol foi {FEA, Poli, ESALQ} e Volei foi {Poli, ESALQ, FEA}).

```
3 Poli FEA ESALQ
InterUSP
1
4
Futebol
1 3 FEA Poli ESALQ
Basquete
0 3 Poli FEA ESALQ
Volei
1 3 Poli ESALQ FEA
Natacao
0 3 Poli FEA ESALQ
```

Um outro exemplo é da **CompeticaoMultiModalidades** X com equipes A, B e C, apenas uma **Modalidade**, Y, sem resultado, e que só participaram A e B.

```
3 A B C
X
1
1
Y
0 2 A B
```

3.2 Classe PersistenciaDeCompeticao

A classe **PersistenciaDeCompeticao** é a classe responsável pela persistência da **Competicao** em um arquivo texto. Ela deve permitir salvar um projeto e carregá-lo. Os únicos métodos públicos que a classe deve possuir são:

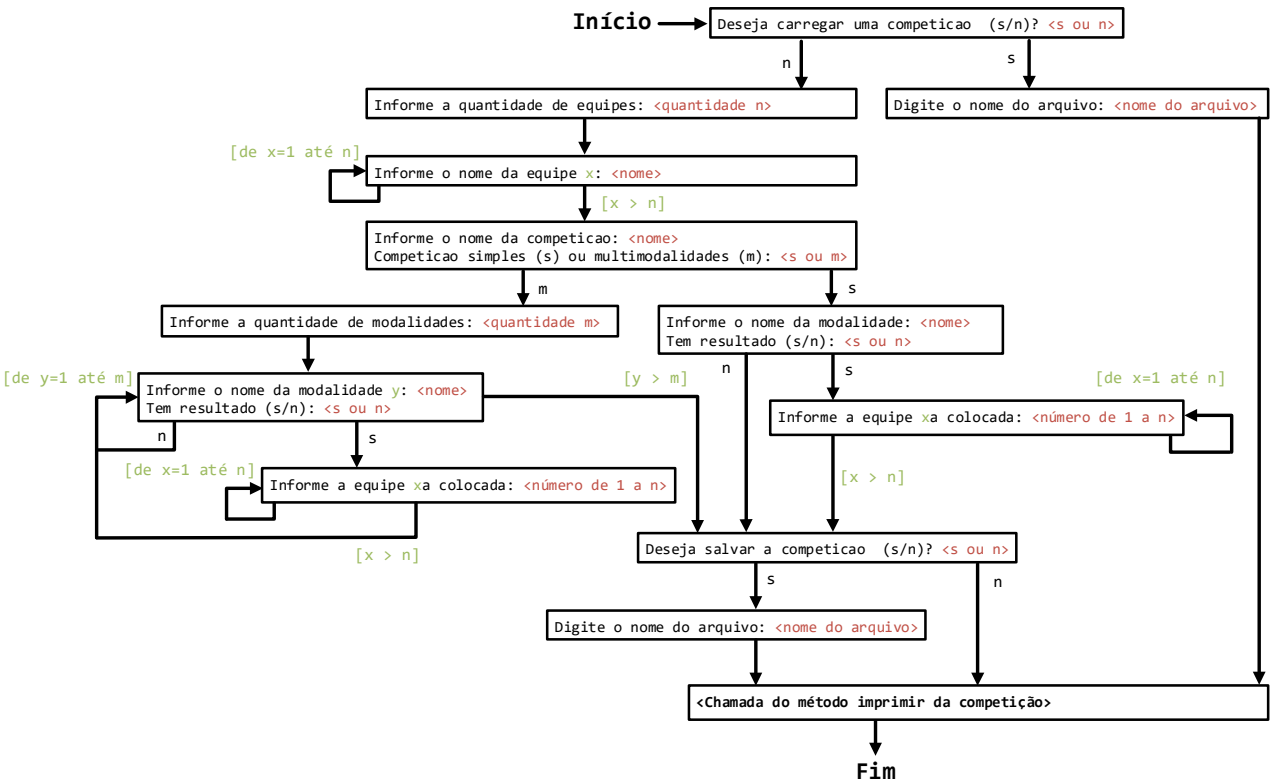
```
PersistenciaDeCompeticao();
virtual ~PersistenciaDeCompeticao();

Competicao* carregar(string arquivo);
void salvar(string arquivo, Competicao* c);
```

O método `carregar` deve criar uma nova **Competicao** (**CompeticaoSimples** ou **CompeticaoMultimodalidades**) a partir dos dados do arquivo informado como parâmetro e retorná-lo. Caso o arquivo não exista ou caso haja algum problema de leitura (erro de formato ou outro problema), jogue uma exceção do tipo **invalid_argument**. Ao carregar, compare as equipes pelo nome e use apenas 1 objeto por Equipe (não crie vários objetos para uma mesma **Equipe**).

O método `salvar` deve persistir em disco, no arquivo passado como parâmetro, os dados da **Competicao** informada. Caso o arquivo não possa ser escrito, jogue uma exceção do tipo **invalid_argument**.

4 Interface com o usuário

Coloque o main em um arquivo em separado, chamado `main.cpp`. O comportamento do main é apresentado esquematicamente no diagrama abaixo. Entre "<" e "> são apresentadas meta-informações: em **vermelho** as entradas do usuário; em **verde** os valores que dependem da quantidade de equipes (n) ou da quantidade de modalidades (m); as reticências (...) representam informações que devem ser repetidas dependendo da quantidade de equipes ou de modalidades. Ao final deve ser chamado o método `imprimir` do objeto competição.



Atenção: a saída do método `imprimir` deve seguir exatamente o especificado no enunciado.

Não se preocupe com erros de digitação: considere que o usuário sempre digita informações corretas. Por exemplo, não verifique se o usuário digitou um número de **Equipe** válida ou se informou a mesma **Equipe** repetidamente ao informar a colocação em uma **Modalidade**. Também considere que todas as **Equipes** participam de todas as **Modalidades**. Por simplicidade, considere que os nomes da **Competicao**, das **Equipes** e das **Modalidades** **não possuem espaço**.

Em caso de exceções, imprima a mensagem da exceção e termine o programa.

Alguns exemplos de saída são apresentados a seguir; em **vermelho** são apresentadas as entradas. No primeiro exemplo, carrega-se o arquivo `ex1.txt`, o qual possui a **CompeticaoSimples** *InterTruco* (exemplificada na Seção 3.1.1).

```
Deseja carregar uma competicao (s/n)? s
Digite o nome do arquivo: ex1.txt

InterTruco
        1o Poli
        2o FOUSP
        3o FEA
        4o ESALQ
```

Um exemplo de criação de uma **CompeticaoSimples** é apresentado a seguir**:**

```
Deseja carregar uma competicao (s/n)? n

Informe a quantidade de equipes: 4
Informe o nome da equipe 1: CEE
Informe o nome da equipe 2: CAEP
Informe o nome da equipe 3: AEQ
Informe o nome da equipe 4: CAM

Informe o nome da competicao: IntegraPoli
Competicao simples (s) ou multimodalidades (m)? s

Informe o nome da modalidade: Solidario
Tem resultado (s/n): n

Deseja salvar a competicao (s/n)? n

IntegraPoli
        CEE
        CAEP
        AEQ
        CAM
```

Por fim, é apresentada uma **CompeticaoMultimodalidades** que é salva: a competição *InterUSP*, apresentada na seção 3.1.2.

```
Deseja carregar uma competicao (s/n)? n

Informe a quantidade de equipes: 3
Informe o nome da equipe 1: Poli
Informe o nome da equipe 2: FEA
Informe o nome da equipe 3: ESALQ

Informe o nome da competicao: InterUSP
Competicao simples (s) ou multimodalidades (m)? m

Informe a quantidade de modalidades: 4
Informe o nome da modalidade 1: Futebol
Tem resultado (s/n): s
Informe a equipe 1a colocada: 2
Informe a equipe 2a colocada: 1
Informe a equipe 3a colocada: 3

Informe o nome da modalidade 2: Basquete
Tem resultado (s/n): n

Informe o nome da modalidade 3: Volei
Tem resultado (s/n): s
Informe a equipe 1a colocada: 1
Informe a equipe 2a colocada: 3
Informe a equipe 3a colocada: 2

Informe o nome da modalidade 4: Natacao
Tem resultado (s/n): n

Deseja salvar a competicao (s/n)? s
Digite o nome do arquivo: ex2.txt

InterUSP
        1o Poli (23 pontos)
        2o FEA (21 pontos)
        3o ESALQ (18 pontos)
```

5 Entrega

O projeto deverá ser entregue até dia **19/11** no Judge em <http://judge.pcs.usp.br/pcs3111/ep/>.

Atenção

- Deve ser mantida a mesma dupla do EP1. É possível apenas *desfazer* a dupla. Com isso, cada aluno deve fazer uma entrega diferente (e em separado). Caso você deseje fazer isso, envie um e-mail para levy.siqueira@usp.br até dia **14/11**.
- Não copie código de um outro grupo. Qualquer tipo de cópia será considerada plágio e os grupos envolvidos terão **nota 0 no EP**. Portanto, **não envie** o seu código para um colega de outro grupo!

Entregue todos os arquivos, inclusive o `main` (que deve **obrigatoriamente** ficar em um arquivo "main.cpp"), em um arquivo comprimido no formato ZIP (outros formatos, como RAR e 7Z, podem não ser reconhecidos e acarretar **nota 0**). Os códigos fonte não devem ser colocados em pastas.

Atenção: faça a submissão do mesmo arquivo nos 3 problemas (Parte 1, Parte 2 e Parte 3). Isso é necessário por uma limitação do Judge. Caso isso não seja feito, parte do seu EP não será corrigido – impactando a nota.

Siga a convenção de nomes para os arquivos ".h" e ".cpp". O não atendimento disso pode levar a erros de compilação (e, consequentemente, **nota zero**).

Ao submeter os arquivos no Judge será feita **apenas** uma verificação básica de modo a evitar erros de digitação no nome das classes e dos métodos públicos. **Você poderá submeter quantas vezes você desejar, sem desconto de nota**. Mas note que a nota dada **não é a nota final**: não são executados testes – o Judge apenas tenta chamar todos os métodos definidos neste documento para todas as classes.

6 Dicas

- Implemente a solução aos poucos – não deixe para implementar tudo no final.
- A **CompeticaoMultimodalidades** envolve conceitos vistos apenas na aula 11. Porém, você pode implementar o carregar da classe de persistência sem ter a implementação completa dessa classe (você só não conseguirá testar adequadamente).
- Separe o `main` em várias funções, para melhorar a organização e reaproveitar código.
- Faça métodos auxiliares em `PersistenciaDeCompeticao` para reaproveitar código.
- Para testar o programa faça o `main` chamar uma função de teste que cria objetos com valores interessantes para testar, sem pedir entrada para o usuário. Não se esqueça de remover a função de teste ao entregar a versão final do EP.
- Caso o método `setResultado` em **Modalidade** apenas armazene a *referência ao vetor* (o que é o mais simples), cuidado com o reuso no `main` do mesmo vetor para informar resultados em diferentes modalidades. Note que nesse caso todas as **Modalidades** apontariam para o mesmo vetor de resultados. Para evitar isso, crie um vetor para o resultado de cada modalidade no main.
- Submeta no Judge o código com antecedência para descobrir problemas na sua implementação. É normal acontecerem *RuntimeErrors* e outros tipos de erros.
- Use o "Fórum de dúvidas do EP" para esclarecer dúvidas no enunciado ou problemas de submissão no Judge.
- **Evite submeter nos últimos minutos do prazo de entrega.** É normal o Judge ficar sobrecarregado com várias submissões e demorar para compilar.

7 Testes do Judge

Ao submeter o Judge só testará se as classes **possuem** todos os métodos especificados. Ele **não** testará se os métodos são corretos. **Você poderá submeter quantas vezes você desejar, sem desconto de nota**. Após o fim do prazo, os seguintes testes serão executados:

Parte 1

Modalidade: construtor destrutor getNome getEquipes e getQuantidade
Modalidade: nao tem classe base
Modalidade: construtor com quantidade menor que 2
Modalidade: funcionamento do temResultado
Modalidade: setResultado e getTabela
Modalidade: setResultado e getTabela alterando o resultado
Modalidade: imprimir
Equipe: Construtor destrutor e getNome
Equipe: nao tem classe base
Equipe: imprimir
CompeticaoSimples: construtor destrutor getNome getEquipes e getQuantidadeDeEquipes
CompeticaoSimples: construtor com quantidade menor que 2
CompeticaoSimples: getModalidade
CompeticaoSimples: getTabela
CompeticaoSimples: imprimir
CompeticaoMultimodalidades: construtor destrutor getNome getEquipes e getQuantidadeDeEquipes
CompeticaoMultimodalidades: construtor com quantidade menor que 2
CompeticaoMultimodalidades: adicionar e getModalidades
CompeticaoMultimodalidades: getTabela sem resultado
CompeticaoMultimodalidades: getTabela com so uma modalidade com resultado
CompeticaoMultimodalidades: getTabela com varias modalidades com resultado
CompeticaoMultimodalidades: getTabela com varias modalidades alterando resultado
CompeticaoMultimodalidades: getPontoPorPosicao inicial
CompeticaoMultimodalidades: getPontoPorPosicao colocacao invalida
CompeticaoMultimodalidades: getPontoPorPosicao e setPontuacao
CompeticaoMultimodalidades: setPontuacao com menos de 3 valores
CompeticaoMultimodalidades: getTabela com varias modalidades com resultado alterando pontuacao
CompeticaoMultimodalidades: getTabela sem modalidades
CompeticaoMultimodalidades: imprimir
Competicao: eh abstrata
Competicao: nao tem classe base
Competicao: CompeticaoSimples e CompeticaoMultimodalidades sao filhas

Parte 2

TabelaComPontos: construtor destrutor e getQuantidadeDeEquipes
TabelaComPontos: quantidade de equipes menor que 2
TabelaComPontos: getPontos sem pontuar
TabelaComPontos: getPontos equipe nao participante
TabelaComPontos: pontuar equipe nao participante
TabelaComPontos: pontuar e getPontos
TabelaComPontos: pontuar e getPontos chamados varias vezes
TabelaComPontos: getPosicao sem resultado definido
TabelaComPontos: getPosicao nao participante
TabelaComPontos: getPosicao sem empate
TabelaComPontos: getPosicao com empate
TabelaComPontos: getPosicao com empate e mudando pontuacao
TabelaComPontos: getEquipesEmOrdem sem resultado definido
TabelaComPontos: getEquipesEmOrdem com resultado definido
TabelaComPontos: getEquipesEmOrdem com resultado definido e empate
TabelaComPontos: getEquipesEmOrdem com resultado definido alterando pontuacao
TabelaComPontos: imprimir sem ordem
TabelaComPontos: imprimir com pontos
TabelaComOrdem: construtor destrutor e getQuantidadeDeEquipes
TabelaComOrdem: quantidade de equipes menor que 2
TabelaComOrdem: getPosicao sem resultado definido
TabelaComOrdem: getPosicao nao participante
TabelaComOrdem: getPosicao equipes validas 1
TabelaComOrdem: getPosicao equipes validas 2
TabelaComOrdem: getPosicao equipes validas mudando resultado
TabelaComOrdem: getEquipesEmOrdem sem resultado definido
TabelaComOrdem: getEquipesEmOrdem com resultado definido 1
TabelaComOrdem: getEquipesEmOrdem com resultado definido 2
TabelaComOrdem: getEquipesEmOrdem com mudanca de resultado
TabelaComOrdem: imprimir sem ordem
TabelaComOrdem: imprimir com ordem
Tabela: eh abstrata
Tabela: nao tem classe base
Tabela: TabelaComPontos e TabelaComOrdem sao filhas

Parte 3

PersistenciaDeCompeticao: carregar arquivo inexistente
PersistenciaDeCompeticao: carregar arquivo invalido
PersistenciaDeCompeticao: carregar CompeticaoSimples sem resultados
PersistenciaDeCompeticao: carregar CompeticaoSimples com resultados
PersistenciaDeCompeticao: carregar CompeticaoMultimodalidades so 1 modalidade
PersistenciaDeCompeticao: carregar CompeticaoMultimodalidades sem resultados
PersistenciaDeCompeticao: carregar CompeticaoMultimodalidades com resultados
PersistenciaDeCompeticao: carregar CompeticaoMultimodalidades com e sem resultados
PersistenciaDeCompeticao: salvar CompeticaoSimples sem resultados
PersistenciaDeCompeticao: salvar CompeticaoSimples com resultados
PersistenciaDeCompeticao: salvar CompeticaoMultimodalidades so 1 modalidade
PersistenciaDeCompeticao: salvar CompeticaoMultimodalidades sem resultados
PersistenciaDeCompeticao: salvar CompeticaoMultimodalidades com resultados
PersistenciaDeCompeticao: salvar CompeticaoMultimodalidades com e sem resultados