



PCS 3111 - LABORATÓRIO DE PROGRAMAÇÃO ORIENTADA A OBJETOS PARA A ENGENHARIA ELÉTRICA

EXERCÍCIO PROGRAMA 2 – 1º SEMESTRE DE 2020

Resumo

Os EPs de PCS3111 têm como objetivo exercitar os conceitos de Orientação a Objetos aprendidos em aula ao implementar uma *Rede Social* simples, similar ao Twitter. Neste segundo EP serão feitas algumas melhorias na rede social implementada no primeiro EP.

1 Introdução

Deseja-se criar uma *Rede Social* simples, no estilo do Twitter. Neste segundo EP serão feitas alterações no programa desenvolvido anteriormente para melhorar o projeto, considerando outros conceitos da orientação a objetos e permitir novas funcionalidades. As principais novas funcionalidades são: permitir a criação de um número ilimitado de perfis, a persistência da rede social e a possibilidade de se remover um seguidor.

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 **Aluno**, **Disciplina**, **Evento**, **PerfilInexistente**, **Mensagem**, **Perfil**, **PersistenciaDaRede**, **Pessoa**, **Professor**, **Publicacao** e **RedeSocial**, além de criar um main que permita o funcionamento do programa como desejado.

Por simplicidade, assuma que os nomes dos perfis, o departamento do professor e a sigla da disciplina não possuem espaços em branco. Isso facilitará a persistência.

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 "Perfil.cpp" e "Perfil.h". Note que você deve criar os arquivos necessários.

Apesar de a especificação ser longa, as classes possuem um comportamento muito similar ao especificado no EP1 e algumas outras funcionalidades desenvolvidas em aula.

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.
- Pode-se definir atributos e métodos protegidos e privados, conforme necessário.
- Não é permitida a criação de outras classes além dessas.
- Não faça #defines de constantes.

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

2.1 Classe Perfil

Um **Perfil** é o elemento básico da rede social. Essa classe foi alterada para evitar um problema no EP1: o fato de um dos tipos de **Perfil** – a **Disciplina** – não precisar de algumas das informações. Com isso, um **Perfil** é uma classe **abstrata** e possui apenas um nome e um identificador (id), além de seguidores e publicações. Escolha o método mais adequado para ser **abstrato**.

Para permitir um número ilimitado de seguidores e de publicações, a classe deve usar os containers da STL e. Os seguidores deverão ser guardados em um vector e as publicações feitas e as recebidas devem ser armazenadas em lists. Por causa dessa mudança, não é mais necessário o define.

A classe **Perfil** deve possuir os seguintes métodos **públicos**:

```
Perfil(string nome);
Perfil(int id, string nome);
virtual ~Perfil();

static void setUltimoId(int ultimoId);
static int getUltimoId();

int getId();
string getNome();

virtual void adicionarSeguidor(Perfil* seguidor);

virtual void publicar(string texto);
virtual void publicar(string texto, string data);

virtual void receber(Publicacao* p);
virtual list<Publicacao*> getPublicacoesFeitas();
virtual list<Publicacao*> getPublicacoesRecebidas();

virtual vector<Perfil*> getSeguidores();

virtual void imprimir();
```

O Perfil tem dois construtores: um que recebe apenas um nome e outro que recebe um nome e um id. O que recebe o nome e o id deve ser usado **apenas pela persistência** (discutida na seção 3). A criação de um **Perfil** pela interface com o usuário deve chamar o construtor que recebe apenas o nome e deve gerar automaticamente um identificador (id). O primeiro **Perfil** criado deve ter id = 1, o segundo perfil deve ter

id = 2 e assim por diante (note que isso independe do tipo dele). Para isso você deverá usar um atributo com escopo de classe (static).

Os métodos `getUltimoId` e `setUltimoId` servem para a persistência. Eles permitem obter e definir o valor do último id gerado e, portanto, tem escopo de classe. O método `getUltimoId` deve retornar 0 para o caso de nenhum **Perfil** ter sido criado. Apenas a persistência deve usar o método `setUltimoId`.

Os métodos `getId` e `getNome` devem retornar os valores do id e do nome.

O método `adicionarSeguidor` deve adicionar um seguidor ao **Perfil**. Note que diferentemente do EP1, este método é void. Caso se tente adicionar o próprio **Perfil** como seguidor dele mesmo ou se o **Perfil** informado já seja um seguidor, esse método deve jogar uma exceção do tipo `invalid_argument`.

Assim como no EP1, para fazer uma publicação existem dois métodos com o mesmo nome: `publicar`. O que recebe apenas um texto deve publicar uma **Mensagem**; o que recebe um texto e uma data deve publicar um **Evento**. O comportamento é o mesmo do EP1, mas note que não há mais uma limitação de espaço e, portanto, esses métodos são void.

Um outro método é o `receber`, que recebe uma **Publicação** feita por alguém que o **Perfil** segue. Esse método deve ter o mesmo comportamento do EP1.

Os métodos `getSeguidores`, `getPublicacoesFeitas` e `getPublicacoesRecebidas` permitem obter, respectivamente, os **Perfis** seguidores deste **Perfil** e as **Publicações** feitas e recebidas pelo **Perfil**. O método `getSeguidores` deve retornar um ponteiro para um vector de ponteiros de **Perfis** (ou um vector vazio caso o **Perfil** não tenha seguidores). Os métodos `getPublicacoesFeitas` e `getPublicacoesRecebidas` devem retornar um ponteiro para uma list de ponteiros de **Publicações** feitas e as recebidas, respectivamente. Caso o **Perfil** não tenha publicações feitas ou recebidas, o método respectivo deve retornar uma list vazia.

Por fim, foi criado um método `imprimir`. O comportamento dele não é especificado e não será testado. Use-o conforme for conveniente.

2.2 Classe Pessoa

Dada a similaridade de um **Aluno** e um **Professor**, criou-se a classe **Pessoa**. Essa classe deve ser subclasse de **Perfil**, mas também é abstrata. Ela deve possuir os seguintes métodos públicos específicos a essa classe:

```
Pessoa(int numeroUSP, string nome, string email);
Pessoa(int id, int numeroUSP, string nome, string email);
virtual ~Pessoa();

virtual int getNumeroUSP();
virtual string getEmail();
```

Note que você pode redefinir métodos da superclasse, **Perfil**, se necessário, além de criar métodos privados.

Assim como no **Perfil**, a classe **Pessoa** possui dois construtores. O que recebe o id deverá ser usado apenas pela persistência. Uma **Pessoa** possui número USP e e-mail, além do nome. Com isso, essas informações são passadas no construtor e existem métodos de acesso para obtê-los.

Qualquer tipo de **Pessoa** deve ter o mesmo comportamento quando se adiciona um seguidor: ele deve ter uma **Mensagem** adicionada à sua lista de publicações recebidas com o seguinte texto:

Novo seguidor: <nome do seguidor>

O funcionamento disso é o mesmo do especificado no EP1. Assim sendo, essa **Mensagem não deve** ser enviada aos seguidores da **Pessoa**. Ela deve apenas aparecer como uma **Publicação** na lista de publicações recebidas. Esta **Publicação** deve ter a própria **Pessoa** como autor.

2.3 Classe Aluno

Um **Aluno** é uma subclasse de **Pessoa**. Ele não possui nenhum comportamento adicional à **Pessoa**, mas é uma classe concreta. Com isso, ela deve ter os seguintes métodos públicos específicos a essa classe:

```
Aluno(int numeroUSP, string nome, string email);
Aluno(int id, int numeroUSP, string nome, string email);
virtual ~Aluno();
```

Assim como nas classes **Perfil** e **Pessoa**, o construtor que recebe um id deve ser usado apenas pela persistência.

2.4 Classe Professor

Um **Professor** é uma subclasse de **Pessoa** que possui adicionalmente a informação do seu departamento. A seguir são apresentados os métodos públicos específicos a essa classe:

```
Professor(int numeroUSP, string nome, string email, string departamento);
Professor(int id, int numeroUSP, string nome, string email, string departamento);
virtual ~Professor();

string getDepartamento();
```

Assim como nas classes **Perfil**, **Pessoa** e **Aluno**, o construtor que recebe um id deve ser usado apenas pela persistência. O departamento informado no construtor deve ser obtido pelo método de acesso `getDepartamento`.

2.5 Classe Disciplina

Uma **Disciplina** é um subtipo de **Perfil** (note que não é uma **Pessoa**) com alguns comportamentos específicos. A seguir são apresentados os métodos públicos específicos a essa classe:

```
Disciplina(string sigla, string nome, Professor* responsavel);
Disciplina(int id, string sigla, string nome, Professor* responsavel);
virtual ~Disciplina();

virtual string getSigla();
virtual Professor* getResponsavel();
```

Assim como no EP1, uma **Disciplina** possui uma sigla, um nome e um **Professor** responsável. Como os demais tipos de **Perfil**, a **Disciplina** possui um construtor que recebe um id e que deve ser usado apenas pela persistência. Ao criar a **Disciplina**, considerando o construtor sem *id*, o **Professor** responsável deve ser automaticamente adicionado como seguidor. No construtor que recebe *id*, não faça isso (para simplificar a persistência).

Os métodos `getSigla` e `getResponsavel` devem retornar o valor respectivo definido no construtor.

Conforme especificado no EP1, a lista de publicações de uma **Disciplina** deve ser limitada a apenas **Publicações** feitas pela própria **Disciplina**. Quando um seguidor é adicionado à **Disciplina** não se deve adicionar a mensagem de aviso. Além disso, como não faz sentido uma disciplina seguir outros **Perfis**, ela não deve fazer nada ao receber uma **Publicação**.

2.6 Classe Publicação

A classe **Publicacao** se tornou abstrata no EP2. Com isso ela terá duas subclasses: **Mensagem** e **Evento**. A seguir são apresentados os métodos **públicos** dessa classe:

```
Publicacao(Perfil* autor, string texto);  
virtual ~Publicacao();  
  
Perfil* getAutor();  
string getTexto();  
  
virtual void curtir(Perfil* quemCurtiu);  
virtual int getCurtidas();  
virtual void imprimir();
```

Essa classe segue a mesma especificação definida no EP1. A única diferença é que o método `curtir` deve ser abstrato, já que ele terá comportamento diferente para cada tipo de **Publicacao**.

2.7 Classe Mensagem

A **Mensagem** é uma subclasse de **Publicacao** que representa uma publicação de um texto. A seguir são apresentados os métodos públicos específicos a essa classe:

```
Mensagem(Perfil* autor, string texto);  
virtual ~Mensagem();
```

O construtor apenas recebe os dados básicos da **Mensagem**, o autor e o texto.

Nessa classe o método `curtir` deve impedir que o próprio autor curta a **Mensagem**. Caso isso aconteça, jogue uma exceção do tipo `invalid_argument`.

2.8 Classe Evento

Assim como no EP1, o **Evento** é uma subclasse de **Publicação** que possui uma data. A seguir são apresentados os métodos públicos específicos a essa classe:

```
Evento(Perfil* autor, string texto, string data);  
virtual ~Evento();  
  
string getData();
```

O construtor deve receber o autor, o texto e a data do **Evento** (o formato da data não é relevante, já que ela será armazenada como uma string). O método `getData` deve retornar a data definida no construtor.

No **Evento** é possível que o próprio autor curta o **Evento**.

2.9 Classe RedeSocial

Assim como no EP1, a **RedeSocial** é a classe responsável por ter a lista de **Perfis** existentes na rede. Ela é bastante similar ao definido no EP1; a diferença é o uso de um vector para guardar os **Perfis** e dois novos métodos de apoio. Com isso ela deve possuir os seguintes métodos **públicos**:

```
RedeSocial();  
virtual ~RedeSocial();  
  
vector<Perfil*> getPerfis();  
Perfil* getPerfil(int id);  
void adicionar(Perfil* perfil);  
  
void imprimir();
```

O construtor da **RedeSocial** não recebe mais parâmetros, já que será usado um vector para armazenar os **Perfis**. Para adicionar um **Perfil** à rede deve ser usado o método `adicionar`. Caso o **Perfil** já exista, a classe deve-se jogar uma exceção `invalid_argument`. Para facilitar, use a função `find` para saber se o **Perfil** já está no vector (<http://www.cplusplus.com/reference/algorithm/find/>).

Para obter todos os **Perfis** existentes deve ser usado o método `getPerfis`, o qual retorna um ponteiro para o vector que possui todos os **Perfis**.

Também há um método `getPerfil` que obtém o **Perfil** a partir do `id` dele. Esse método é útil para a interface com o usuário (note que ela é agora baseada em ids) e para a persistência. Caso não exista o **Perfil** com o `id` informado deve-se jogar uma exceção do tipo **PerfilInexistente**.

Por fim, foi criado um método `imprimir`. O comportamento dele não é especificado e não será testado. Use-o conforme for conveniente.

2.10 Classe PerfilInexistente

A classe **PerfilInexistente** representa que não existe um **Perfil** com o id informado na rede social. Ela deve ser subclasse de **logic_error** (da biblioteca padrão) e os métodos públicos específicos a essa classe são apresentados a seguir.

```
PerfilInexistente();  
virtual ~PerfilInexistente();
```

A mensagem de erro informada pelo método `what` da exceção deve ser "Perfil Inexistente".

3 Persistência

A persistência, por simplicidade, se limitará aos **Perfis** e suas relações. Ou seja, não se persistirá as mensagens publicadas pelos **Perfis**. Uma outra simplificação é que o nome, departamento e sigla não deverão conter espaços.

A seguir é apresentado o formato do arquivo, exemplos de arquivos e a especificação das classes.

3.1 Formato do arquivo

A persistência da rede social deve seguir o formato de arquivo especificado a seguir. Entre "<" e ">" são especificados os valores esperados. Por simplicidade, será utilizado um caractere de nova linha ('\n') como delimitador e *assuma* que não existem espaços nos campos que sejam string.

```
<último id do perfil>
<quantidade de alunos>
<id do Aluno 1> <numeroUSP> <nome> <email>
<id do Aluno 2> <numeroUSP> <nome> <email>
...
<quantidade de professores>
<id do Professor 1> <numerousp> <nome> <email> <departamento>
<id do Professor 2> <numerousp> <nome> <email> <departamento>
...
<quantidade de disciplinas>
<id da Disciplina 1> <sigla> <nome> <id do responsável>
<id da Disciplina 2> <sigla> <nome> <id do responsável>
...
<id de um Perfil> <id do seguidor do Perfil>
<id de um Perfil> <id do seguidor do Perfil>
...
```

Esse formato permite que sejam definidos inúmeros **Perfis**, independentemente do tipo deles. No começo é guardado o id do último **Perfil** cadastrado, obtido por `getUltimoId`. Isso é importante para evitar que existam ids repetidos (não se preocupe em verificar essa consistência).


O arquivo sempre apresenta a quantidade de um tipo de **Perfil** e então o dado de cada um daqueles **Perfis**. Primeiramente são os **Alunos**, depois os **Professores** e por fim as **Disciplinas**. Após apresentar as **Disciplinas**, são apresentados os seguidores dos **Perfis**. A ordem dos **Perfis** e dos seguidores não é relevante, mas essa lista deve apresentar todos os **Perfis**, inclusive o responsável pela **Disciplina** seguindo a **Disciplina** (por simplicidade).

3.2 Exemplo

Um exemplo desse arquivo, com quatro **Perfis** é apresentado a seguir. Existem dois alunos: Maria (id 1) e Jose (id 3). Também existe um professor (Claudio, com id 2) e uma disciplina (PCS3111, com id 4).

```
4
2
1 1111 Maria maria@usp.br
3 2222 Jose jose@usp.br
1
2 3333 Claudio claudio@usp.br PCS
1
4 PCS3111 Lab00 2
1 2
1 3
3 1
4 2
```

As relações entre os Perfis são as seguintes:

- Maria é seguida por Cláudio e por José;  Corrigido
- José é seguido por Maria;
- PCS3111 é seguida por Cláudio;

Um outro exemplo é de um arquivo com apenas um aluno.

```
1
1
1 1234 Paula paula@usp.br
0
0
```

3.3 Classe PersistenciaDaRede

A classe **PersistenciaDaRede** é a classe responsável pela persistência da **RedeSocial** em um arquivo texto. Ela deve permitir salvar uma rede e carregá-la. Os únicos métodos públicos que a classe deve possuir são:

```
PersistenciaDaRede(string arquivo);
virtual ~PersistenciaDaRede();

void salvar(RedeSocial* r);
RedeSocial* carregar();
```

O construtor deve receber o nome do arquivo que será usado para carregar e salvar a **RedeSocial**. O método carregar deve obter a **RedeSocial** que está no arquivo texto, seguindo o formato especificado anteriormente. Caso o arquivo não exista, retorne uma **RedeSocial** vazia. Ao carregar, atualize o último id definido pelo **Perfil**. Caso haja algum problema de leitura (erro de formato ou outro problema), jogue uma exceção do tipo `logic_error`.

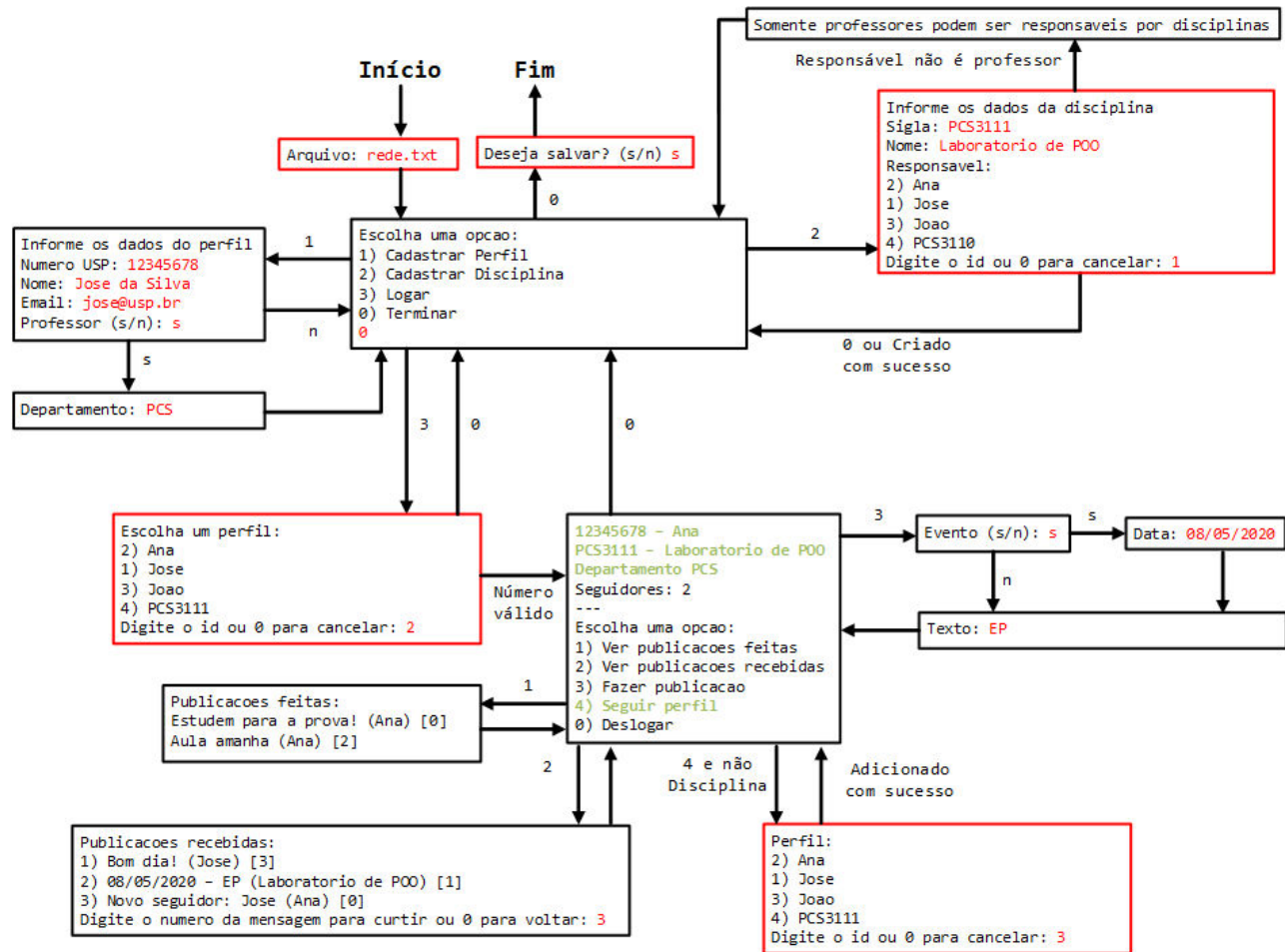
O método salvar salva a **RedeSocial** informada no arquivo informado no construtor.

4 Interface com o usuário

Coloque o main em um arquivo em separado, chamado `main.cpp`. Ela é apresentada esquematicamente no diagrama abaixo. Cada retângulo representa uma "tela", ou seja, o conjunto de informações apresentadas e solicitadas. As telas com borda **vermelha** possuem diferenças em relação ao EP1 (são novas ou foram alteradas). As setas representam as transições de uma tela para outra – os textos na seta representam o valor que deve ser digitado para ir para a tela destino ou a condição necessária (quando não há um texto é porque a transição acontece incondicionalmente). Em **vermelho** são apresentados exemplos de dados inseridos pelo usuário. Em **verde** são apresentadas mensagens que acontecem dependendo do tipo do perfil:

- Quando for um **Professor** ou um objeto **Perfil**, deve ser apresentado o texto "<número USP> - <nome>" como, no exemplo, "123456789 - Ana". Quando for uma **Disciplina**, o texto a ser apresentado é "<sigla> - <nome>" como, no exemplo, "PCS3111 – Laboratorio de POO".
- A informação do departamento só deve aparecer se o perfil for de um **Professor** (nos outros casos essa linha não deve existir).
- A opção "4) Seguir perfil" não deve aparecer se o perfil logado for de uma **Disciplina**.

Atenção: A interface com o usuário deve seguir exatamente a ordem definida (e exemplificada). Se a ordem não for seguida, haverá desconto de nota.



Alguns detalhes:

- A interface é bastante similar a do EP1. As diferenças:
 - No início do programa se pergunta o arquivo a ser carregado. Se ele não existir, deve ser criada uma **RedeSocial** vazia (já é o comportamento de **PersistenciaDaRede**).
 - No final do programa se pergunta se deseja salvar a **RedeSocial**. Caso se responda sim, a rede deve ser salva no arquivo informado no início da execução.
 - A seleção de **Perfis** é feita pelo id deles (use o método `getPerfil` da **RedeSocial**). Portanto, é possível que os ids dos **Perfis** não estejam ordenados (após carregar um arquivo).
- Na tela de publicações, o número entre colchetes é a quantidade de curtidas.
- Caso alguma operação jogue uma exceção, apresente a mensagem da exceção e termine o programa.

5 Entrega

O projeto deverá ser entregue até dia **26/06** no Judge disponível em <http://judge.pcs.usp.br/pcs3111/ep/>. Você deverá fazer duas submissões (Entrega 1 e Entrega 2). **Todas as entregas deverão possuir o mesmo conteúdo.**

Atenção: não copie código de um outro colega. Qualquer tipo de cópia será considerada plágio e ambos os alunos terão **nota 0 no EP**. Portanto, **não envie** o seu código para um outro colega! Cópias de trabalhos de anos anteriores também receberão 0.

Entregue todos os arquivos, inclusive o `main` (que deve obrigatoriamente ficar em um arquivo “`main.cpp`”), em um arquivo comprimido. Cada entrega deve ser feita em um arquivo comprimido. Os fontes não devem ser colocados em pastas.

Siga a convenção de nomes para os arquivos “.h” e “.cpp”. O não atendimento disso pode levar a erros de compilação e, conseqüentemente, **nota zero**.

Ao submeter os arquivos no Judge será feita 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. Mas note que a nota gerada por essa verificação **não é a nota final**: não são executados testes – o Judge apenas tenta chamar todos os métodos definidos para todas as classes.

Você pode submeter quantas vezes quiser, sem desconto na nota.

6 Dicas

- Reaproveite o `main` do EP1 para implementar o `main` do EP2. Se você organizou bem o `main` (por exemplo, com uma função específica para selecionar um **Perfil**, como sugerido), a criação do novo `main` será muito rápida.
- Crie métodos auxiliares (privados) para organizar os métodos de persistência.
- Teste o método `carregar` e o `salvar` da persistência separadamente. Use um `main` de teste, aproveitando o método `imprimir` da **RedeSocial**.
- Use os métodos da biblioteca padrão para facilitar o uso de `vectors` e `lists`.
- Implemente a solução aos poucos – não deixe para implementar tudo no final.
- É trabalhoso testar uma classe específica usando o `main com menus`, já que pode ser necessário passar por várias telas até testar o que se deseja. Para simplificar o teste, crie um `main` que cria os objetos do jeito que você quer testar. Só não se esqueça de entregar o `main` correto!
- **Evite submeter nos últimos minutos do prazo de entrega. É normal o Judge ficar sobrecarregado com várias submissões e demorar para compilar.**