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

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

EXERCÍCIO PROGRAMA 3 – 2017

Resumo

O objetivo do EP3 é exercitar os conceitos de Orientação a Objetos ao melhorar o *Visualizador Científico* implementado no EP1 e EP2 para permitir séries sem limitação de tamanho, gráficos com várias séries, a persistência das séries e melhorias no tratamento de erros.

1 Introdução

Deseja-se adicionar novas funcionalidades ao *Visualizador Científico*, desenvolvido nos EPs 1 e 2. Deve ser possível `desenhar` mais de uma **Serie** em um **Grafico**. Também deve ser possível salvar uma **Serie** em um arquivo texto e carrega-la posteriormente para gerar um **Grafico**. A limitação de que uma **Serie** tem um `NUMERO_MAXIMO_VALORES` será removida, devendo-se usar um **vector** para armazenar os pontos da **Serie**.

Serão também corrigidos alguns problemas do projeto e da implementação do EP2. O método `adicionar(double x, double y)` não era necessário para uma **SerieTemporal**, mas era herdado da classe **Serie**. Para resolver esse problema, a classe **Serie** se tornou abstrata e se criou um novo tipo de **Serie**, a **SerieNormal**. Uma outra mudança no projeto foi tornar a classe **Eixo** abstrata, criando uma classe **EixoEstatico**[1].

Uma outra melhoria é o aviso de erros: ao invés de ignorar problemas, o código agora gerará exceções – as quais serão tratadas e apresentadas para o usuário.

Como o projeto é incremental, é importante corrigir os problemas identificados na correção dos EPs anteriores. A solução deve empregar adequadamente todos os conceitos de Orientação a Objetos apresentados na disciplina. Assim como nos EPs anteriores, a qualidade de código também será avaliada.

2. Projeto

Deve-se implementar em C++ as classes **PersistenciaDeSerie**, **ErroDeArquivo, EixoEstatico** e **SerieNormal**, além de corrigir as classes **Serie**, **SerieTemporal**, **Eixo**, **EixoDinamico** e **Grafico** considerando a nova especificação. Note que a classe **Ponto** não sofreu alterações, mas os defeitos identificados na última correção devem ser corrigidos e ela também deve ser entregue. Também será necessário alterar o `main` de forma a criar o programa da forma desejada.

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 "EixoDinamico.cpp" e "EixoDinamico.h". Assim como no EP2, não será entregue um código junto com o enunciado; você deve criar os arquivos necessários.

[1] A solução anterior não apresentava problemas. Essa mudança foi criada para exercitar melhor o conceito de classe abstrata.

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 classe pai e que precisaram ser redefinidos. Note que você poderá definir atributos e métodos protegidos e privados, conforme necessário.
- Não é permitida a criação de outras classes além dessas.
- Não use `#define` para definir constantes.

Caso alguma dessas regras não seja seguida, o EP pode *não compilar* na correção feita pelo Judge.

A interface gráfica continua sendo gerada pela classe **Tela**, a qual não sofreu alterações. Assim como no EP2, não use outros recursos disponíveis pelo *framework* Qt. A **InterfaceSerial** também não foi alterada.

2.1 Classe Ponto

A classe **Ponto** não foi alterada e, portanto, deve seguir a mesma especificação do EP2.

2.2 Classe Serie

A classe **Serie** se tornou *abstrata* e o método `adicionar` foi retirado. Além disso, foi removida a restrição de número de elementos. Para isso, a **Serie** deve ser implementada usando um **vector** (da biblioteca padrão). Foi adicionado um novo método, que retorna o vector de pontos da **Serie**[2]. Com isso, os únicos métodos públicos que a classe deve possuir são:

```
/**
 * Cria uma Serie informando o nome dela e o nome dos canais X e Y.
 */
Serie(string nome, string nomeDoCanalX, string nomeDoCanalY);
// Faça o destrutor ser abstrato (puramente virtual)
virtual ~Serie();

// Permite obter o nome, o nomeDoCanalX e o nomeDoCanalY.
virtual string getNome();
virtual string getNomeDoCanalX();
virtual string getNomeDoCanalY();

/**
 * Informa a quantidade de pontos que a Serie possui.
 */
virtual int getQuantidade();

/**
 * Informa se a Serie está vazia.
 */
virtual bool estaVazia();

/**
 * Obtém todos os pontos da Serie.
 */
virtual vector<Ponto*>* getPontos();

/**
 * Imprime na saída padrão (cout) o nome da Serie e seus pontos
 * seguindo o formato definido.
 */
virtual void imprimir();
```

[2] Esse método foi adicionado para facilitar a correção.

```
/**
 * Obtêm um ponto representando o limite superior da Serie.
 * A coordenada x desse ponto deve ser o máximo valor horizontal
 * existente na Serie e a coordenada y deve ser o máximo valor
 * vertical existente na Serie.
 *
 * Por exemplo, para a Serie {(2, 3), (5, 1), (1, 2)} o limite
 * superior é (5, 3).
 * @throw runtime_error Caso a Serie não tenha valores.
 */
virtual Ponto* getLimiteSuperior();
/**
 * Obtém um ponto representando o limite inferior da Serie.
 * A coordenada x desse ponto deve ser o mínimo valor horizontal
 * existente na Serie e a coordenada y deve ser o mínimo valor
 * vertical existente na Serie.
 *
 * Por exemplo, para a Serie {(2, 3), (5, 1), (1, 2)} o limite
 * inferior eh (1, 1).
 * @throw runtime_error Caso a Serie não tenha valores.
 */
virtual Ponto* getLimiteInferior();

/**
 * Obtém o ponto que está na posição definida da Serie. A contagem de
 * posições começa em 0.
 *
 * Por exemplo, para a Serie {(2, 3), (5, 1), (1, 2)}, getPosicao(0)
 * deve retornar (2, 3) e getPosicao(2) deve retornar (1, 2).
 * @throw out_of_range Caso a posição seja inválida.
 */
virtual Ponto* getPosicao(int posicao);
```

Para a classe ser abstrata, apenas o *destrutor* da **Serie** deve ser definido como abstrato (puramente virtual). Note que o funcionamento dos métodos deve seguir o especificado no EP2. Um detalhe interessante de destrutores abstratos é que, apesar de serem abstratos, você precisa implementá-los.

Alguns métodos devem ser alterados para jogar exceções em algumas situações inválidas:

- `getLimiteSuperior()` e `getLimiteInferior()`: devem jogar uma exceção do tipo `runtime_error` (da biblioteca padrão) caso a **Serie** esteja vazia.
- `getPosicao (int posicao)`: deve jogar uma exceção do tipo `out_of_range` (da biblioteca padrão) caso a posição passada seja inválida.

2.3 Classe SerieNormal

A classe **SerieNormal** representa uma **Serie** que permite adicionar valores para as coordenadas x e y. Com isso ela deve ser uma subclasse de **Serie** e os únicos métodos públicos específicos que a classe deve possuir são (note que a classe pode ter que redefinir métodos da classe pai):

```
SerieNormal(string nome, string nomeDoCanalX, string nomeDoCanalY);
virtual ~SerieNormal();

/**
 * Adiciona um novo ponto à Serie, informando sua coordenada x e y.
 */
virtual void adicionar(double x, double y);
```

2.4 Classe SerieTemporal

Assim como definido no EP2, a classe **SerieTemporal** representa uma **Serie** em que o tempo é a coordenada horizonal. Para implementar isso, a classe **SerieTemporal** deve ser subclasse da **Serie**. A

seguir são apresentados os métodos públicos específicos a essa classe (note que a classe pode ter que redefinir métodos da classe pai):

```
/**
 * Cria uma SerieTemporal informando o nome da Serie e o nome do
 * canalY. O nome do canal X deve ser obrigatoriamente "Tempo".
 */
SerieTemporal(string nome, string nomeDoCanalY);
virtual ~SerieTemporal();

/**
 * Adiciona um novo Ponto à Serie, no instante seguinte ao do
 * ponto anterior. O primeiro ponto deve ser adicionado no instante 1.
 */
virtual void adicionar(double valor);
```

Assim como no EP2, o construtor da **SerieTemporal** não deve receber o nome do canal X. Ele obrigatoriamente deve ser "Tempo". Além disso, o tempo deve ser contado a partir do 1. O método `adicionar(double valor)` deve adicionar um novo **Ponto** à **Serie**, no instante seguinte ao ponto anterior adicionado (tempo + 1). Uma vez que não há um outro método `adicionar` nessa classe, como no EP2 (já que ele não era necessário), não existe o problema de dois pontos em uma mesma coordenada x ou casos de coordenada x inválida.

2.5 Classe Grafico

A classe **Grafico** é a responsável por desenhar um gráfico na **Tela**. Para permitir que o **Grafico** desenhe várias **Series**, o construtor da classe agora recebe um **vector** de **Series** ao invés de uma única **Serie**, como no EP2. Os únicos métodos públicos que a classe deve possuir são:

```
/**
 * Cria um Grafico informando os Eixos e as Series.
 * @throw logic_error Caso o eixo x não tenha orientação horizontal ou
 * o eixo y não tenha orientação vertical.
 */
Grafico(Eixo* x, Eixo* y, vector<Serie*>* series);
virtual ~Grafico();

Eixo* getEixoX();
Eixo* getEixoY();
vector<Serie*>* getSeries();

/**
 * Desenha o Grafico na Tela.
 */
void desenhar();
```

Os métodos `getEixoX`, `getEixoY` e `getSeries` devem apenas retornar os valores informados no construtor.

O construtor deve jogar uma exceção do tipo `logic_error` (da biblioteca padrão) caso o eixo x não tenha orientação horizontal ou o eixo y não tenha orientação vertical. Também deve ser jogado um `logic_error` caso o `vector` com as **Series** seja nulo. Porém, não deve ser jogada uma exceção se o `vector` for vazio.

Assim como no EP2, o método `desenhar` deve criar uma **Tela**, definindo seus eixos, e plotar todos os pontos de todas as **Series** (a ordem em que os pontos são plotados não é relevante). A **Tela**, que não foi alterada para este EP, deve ser então mostrada e depois destruída, assim como no EP2.

2.6 Classe Eixo

Um gráfico deve possuir dois **Eixos:** um das abscissas (horizontal, o "x") e um das ordenadas (vertical, o "y"). Para evitar problemas lógicos (um gráfico com dois eixos horizontais), o construtor dessa classe agora recebe um booleano que informa se a orientação é horizontal (assim como era feito na classe EixoDinamico do EP2). Além disso, essa classe se tornou *abstrata*. Cabe a você decidir quais serão os métodos abstratos – mas note que a classe tem que ser abstrata.

A seguir são apresentados os métodos públicos dessa classe:

```
/**
 * Cria um Eixo informando o título, o mínimo, o máximo e
 * a orientação (true se for horizontal e false se for vertical).
 * @throw runtime_error Caso o minimo >= maximo.
 */
Eixo(string titulo, double minimo, double maximo, bool orientacaoHorizontal);
virtual ~Eixo();

virtual string getTitulo();
virtual double getMinimo();
virtual double getMaximo();
virtual bool temOrientacaoHorizontal();
```

Note que o construtor de **Eixo** deve jogar uma exceção do tipo `runtime_error` (da biblioteca padrão) caso `minimo >= maximo`.

2.7 Classe EixoEstatico

O **EixoEstatico** deve ser uma subclasse de **Eixo** e deve possuir um título, uma orientação, um valor mínimo e um valor máximo da escala, os quais não são mudados depois do objeto ser criado. Por exemplo, um **EixoEstatico** horizontal pode apresentar valores de 0 a 5 e um **EixoEstatico** vertical de -5 a 5.

Os métodos públicos específicos a essa classe são os seguintes (note que a classe pode ter que redefinir métodos da classe pai):

```
/**
 * Cria um EixoEstatico informando o título, o mínimo, o máximo e
 * a orientação (true se for horizontal e false se for vertical).
 * @throw runtime_error Caso o minimo >= maximo.
 */
EixoEstatico(string titulo, double minimo, double maximo,
             bool orientacaoHorizontal);
virtual ~EixoEstatico();
```

Os métodos `getTitulo`, `temOrientacaoHorizontal`, `getMinimo` e `getMaximo` devem apenas retornar os valores definidos no construtor.

Assim como na classe pai, no construtor caso o `minimo >= maximo` deve-se jogar uma exceção do tipo `runtime_error` (da biblioteca padrão).

2.8 Classe EixoDinamico

Um outro tipo de **Eixo** é o **EixoDinamico** (essa classe deve ser subclasse da classe **Eixo**). Como um **Grafico** pode possuir diversas **Series**, o **EixoDinamico** também deve considerar diversas **Series**, recebendo uma **list** de **Series**, da biblioteca padrão, em seu construtor. Com isso, o mínimo e o máximo do **Eixo** devem ser calculados considerando todas as **Series** – ou seja, o mínimo deve ser o mínimo daquela orientação em todas as **Series** da lista (idem para o máximo). Caso *os valores mínimo e máximo sejam suficientemente iguais*, os métodos `getMinimo` e `getMaximo` do **EixoDinamico** devem retornar os

valores padrão, os quais são informados no construtor. Caso todas as **Series** estejam vazias, os métodos `getMinimo` e `getMaximo` também devem retornar os valores padrão. Diferentemente do EP2, o título do **Eixo** deve ser informado no construtor (uma vez que os títulos são potencialmente diferentes).

A seguir são apresentados os métodos públicos específicos a essa classe (note que a classe pode ter que redefinir métodos da classe pai):

```
/**
 * Cria um EixoDinamico informando o título, o mínimo e o máximo
 * padrão, a lista de Series que devem ser consideradas e a orientação
 * (true se for horizontal e false se for vertical).
 * @throw runtime_error Caso o minimoPadrao >= maximoPadrao.
 */
EixoDinamico(string titulo, double minimoPadrao, double maximoPadrao,
             list<Serie*>* series, bool orientacaoHorizontal);
virtual ~EixoDinamico();
```

Assim como na classe pai, no construtor caso o `minimoPadrao >= maximoPadrao` deve-se jogar uma exceção do tipo `runtime_error` (da biblioteca padrão). Caso a `list` com as **Series** seja nula deve-se jogar um `logic_error` (da biblioteca padrão). Porém, não deve ser jogada uma exceção se ela for vazia.

2.9 Classe ErroDeArquivo

A classe **ErroDeArquivo** representa um erro com o processamento do arquivo. Ela deve ser filha de **logic_error** (da biblioteca padrão) e os métodos públicos específicos a essa classe são (note que a classe pode ter que redefinir métodos da classe pai):

```
/**
 * Cria um erro de arquivo com a mensagem descritiva do erro.
 */
ErroDeArquivo(string mensagem);
virtual ~ErroDeArquivo();
```

2.10 Classe PersistenciaDeSerie

A classe **PersistenciaDeSerie** é a classe responsável pela persistência de **Series** em um arquivo texto, sejam elas **SerieNormal** ou **SerieTemporal**. Essa classe deve permitir obter uma **Serie** anteriormente salva no arquivo, obter os nomes das **Series** no arquivo e inserir uma **Serie** no arquivo. Os únicos métodos públicos que a classe deve possuir são:

```
/**
 * Cria um objeto que persiste (insere e lê) no arquivo informado.
 * @throw ErroDeArquivo Caso o arquivo não siga o formato.
 */
PersistenciaDeSerie(string arquivo);
virtual ~PersistenciaDeSerie();

/**
 * Obtém a Serie com o nome. Caso não haja uma Serie com esse nome,
 * deve-se retornar NULL.
 */
Serie* obter(string nome);

/**
 * Obtém um vector com os nomes das Series persistidas. Os nomes devem estar
 * na mesma ordem que no arquivo texto (ou seja, a primeira Serie no arquivo
 * deve ficar na posição 0 do vector, a segunda na posição 1 etc.).
 */
vector<string>* getNomes();
```

```
/**
 * Persiste a Serie s usando um nome.
 * @throw ErroDeArquivo Caso não seja possível escrever no arquivo ou
 * caso já exista uma Serie com o mesmo nome.
 */
void inserir(string nome, Serie* s);
```

O construtor deve já carregar o arquivo texto, jogando uma exceção do tipo **ErroDeArquivo** (explicada na Seção 2.9) caso o arquivo informado não siga o formato. Note que não deve ser considerado erro de formato caso o arquivo não exista (nesse caso ele deve ser criado ao inserir uma **Serie**).

O método `getNomes` deve retornar um vector (da STL) de strings que representam os nomes encontrados no arquivo. Caso não haja nomes no arquivo, retorne um vector vazio. Os nomes são usados para obter uma **Serie**, o que é feito pelo método `obter`. Esse método deve retornar uma **SerieNormal** ou uma **SerieTemporal** (dependendo do tipo da **Serie**) a partir do nome informado. Caso não exista uma **Serie** com esse nome no arquivo, retorne NULL.

Por fim, o método `inserir` deve adicionar uma **Serie** (**SerieNormal** ou **SerieTemporal**) ao arquivo texto. Caso não seja possível escrever no arquivo (por qualquer motivo) ou caso já exista uma **Serie** com o nome informado, o método deve jogar uma exceção do tipo **ErroDeArquivo**.

Um detalhe: por simplicidade, caso uma **Serie** carregada seja alterada, suas alterações não devem ser automaticamente persistidas. Caso se deseje persisti-la, ela deve ser inserida com um nome diferente (e, portanto, existirá no arquivo a **Serie** antiga e a nova).

2.10.1 Formato do arquivo

A persistência da **Serie** 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.

```
<Nome da Serie 1>
<Tipo: 0 se for uma SerieTemporal ou 1 se for uma SerieNormal>
<Quantidade 'n' de pontos que a Serie possui>
<Se for SerieNormal, nome do canal x>
<Nome do canal y>
<Se for SerieNormal, valor x1>
<Valor y1>
<Se for SerieNormal, valor x2>
<Valor y2>
...
<Se for SerieNormal, valor xn>
<Valor yn>
<Nome da Serie 2>
<Tipo: 0 se for uma SerieTemporal ou 1 se for uma SerieNormal>
<Quantidade 'n' de pontos que a Serie possui>
...
```

Esse formato permite que sejam definidas inúmeras **Series**, independentemente do tipo delas. Note que no final do arquivo deve existir uma linha em branco (para facilitar a escrita do arquivo).

2.10.2 Exemplo

Um exemplo desse arquivo, com duas **Series** é apresentado a seguir. A primeira **Serie** se chama *AceleracaoY* e é uma **SerieTemporal**, com 4 valores para o canal ACCY. A segunda **Serie** se chama

Aceleracao é uma **SerieNormal** cujo canal X tem nome ACCX e o canal Y tem nome ACCY. Essa **Serie** tem 5 pontos: {(0.0002, -0.0684), (0, -0.0688), (-0.0012, -0.0691), (-0.0012, -0.0691), (-0.0005, -0.0684)}.

```
AceleracaoY
0
4
ACCY
0.0183
0.0247
0.022
0.0171
Aceleracao
1
5
ACCX
ACCY
0.0002
-0.0684
0
-0.0688
-0.0012
-0.0691
-0.0012
-0.0691
-0.0005
-0.0684
```

3 Interface com o usuário

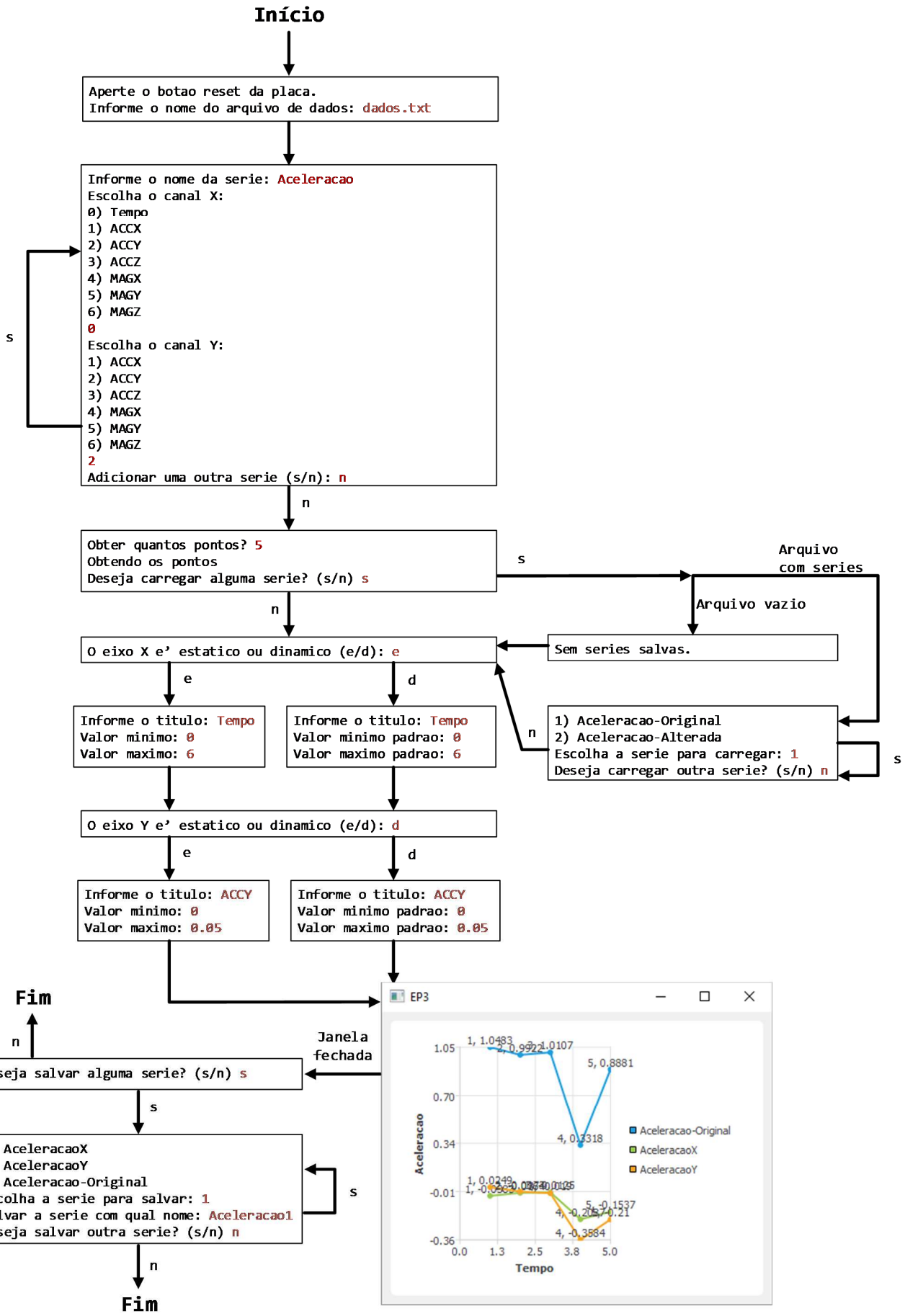
Coloque o main em um arquivo em separado, chamado `main.cpp`. O main deve pedir para o usuário as informações do gráfico, perguntar se o usuário deseja carregar alguma série, desenhar o gráfico e então perguntar se o usuário deseja salvar alguma série. Assim como nos EPs anteriores, **a ordem das mensagens informadas pelo usuário deve seguir exatamente a ordem definida**, assim como o formato das respostas esperadas.

A interface com o usuário é apresentada esquematicamente no diagrama abaixo. Cada retângulo representa uma "tela", ou seja, o conjunto de informações apresentadas e solicitadas. As setas representam as transições de uma tela para outra – os valores na seta representam o valor que deve ser digitado para ir para a tela destino (quando não há um valor é porque a transição acontece incondicionalmente. Em vermelho são apresentados exemplos de dados inseridos pelo usuário.

> **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:

- Caso uma exceção aconteça, no `main` capture-a e apresente a mensagem dela (use o método `what`) e termine o programa.
- Por simplicidade, uma serie carregada pode ser novamente salva.
- Por simplicidade, uma mesma serie pode ser carregada várias vezes.


Início
Aperte o botao reset da placa.
Informe o nome do arquivo de dados: dados.txt
Informe o nome da serie: Aceleracao
Escolha o canal X:
0) Tempo
1) ACCX
2) ACCY
3) ACCZ
4) MAGX
5) MAGY
6) MAGZ
0
Escolha o canal Y:
1) ACCX
2) ACCY
3) ACCZ
4) MAGX
5) MAGY
6) MAGZ
2
Adicionar uma outra serie (s/n): n
s
n
Obter quantos pontos? 5
Obtendo os pontos
Deseja carregar alguma serie? (s/n) s
s
Arquivo com series
Arquivo vazio
n
Sem series salvas.
O eixo X e' estatico ou dinamico (e/d): e
e
d
Informe o titulo: Tempo
Valor minimo: 0
Valor maximo: 6
Informe o titulo: Tempo
Valor minimo padrao: 0
Valor maximo padrao: 6
n
1) Aceleracao-Original
2) Aceleracao-Alterada
Escolha a serie para carregar: 1
Deseja carregar outra serie? (s/n) n
s
O eixo Y e' estatico ou dinamico (e/d): d
e
d
Informe o titulo: ACCY
Valor minimo: 0
Valor maximo: 0.05
Informe o titulo: ACCY
Valor minimo padrao: 0
Valor maximo padrao: 0.05
EP3
Aceleracao
Tempo
1.05
0.70
0.34
-0.01
-0.36
0.0
1.3
2.5
3.8
5.0
Aceleracao-Original
AceleracaoX
AceleracaoY
Janela fechada
Fim
n
Deseja salvar alguma serie? (s/n) s
s
1) AceleracaoX
2) AceleracaoY
3) Aceleracao-Original
Escolha a serie para salvar: 1
Salvar a serie com qual nome: Aceleracao1
Deseja salvar outra serie? (s/n) n
s
n
Fim

Um exemplo de entrada e saída é apresentado abaixo, indicando em vermelho os valores informados pelo usuário.

```
Aperte o botao reset da placa.
Informe o nome do arquivo: dados.txt
Informe o nome da serie: AceleracaoX
Escolha o canal X:
0) Tempo
1) ACCX
2) ACCY
3) ACCZ
4) MAGX
5) MAGY
6) MAGZ
0
Escolha o canal Y:
1) ACCX
2) ACCY
3) ACCZ
4) MAGX
5) MAGY
6) MAGZ
1
Adicionar uma outra serie (s/n): s
Informe o nome da serie: AceleracaoY
Escolha o canal X:
0) Tempo
1) ACCX
2) ACCY
3) ACCZ
4) MAGX
5) MAGY
6) MAGZ
0
Escolha o canal Y:
1) ACCX
2) ACCY
3) ACCZ
4) MAGX
5) MAGY
6) MAGZ
2
Adicionar uma outra serie (s/n): n
```

```
Obter quantos pontos? 5
Obtendo os pontos
Deseja carregar alguma serie? (s/n) s
1) Aceleracao-Original
2) Aceleracao-Alterada
Escolha a serie para carregar: 1
Deseja carregar outra serie? (s/n) n
O eixo X e' estatico ou dinamico (e/d): e
Informe o titulo: Tempo
Valor minimo: 0
Valor maximo: 5
O eixo Y e' estatico ou dinamico (e/d): d
Informe o titulo: Aceleracao
Valor minimo padrao: 0
Valor maximo padrao: 0.2
Deseja salvar alguma serie? (s/n) s
1) AceleracaoX
2) AceleracaoY
3) Aceleracao-Original
Escolha a serie para salvar: 1
Salvar a serie com qual nome? AceleracaoX
Deseja salvar outra serie? (s/n) n
```

4 Interface com o microcontrolador

Desde o EP1 a classe **InterfaceSerial** joga exceções do tipo `runtime_error` ou `logic_error` (da biblioteca padrão) nos seguintes métodos:

- `inicializar`: joga `runtime_error` caso não seja possível inicializar a interface serial.
- `getValor`: joga `logic_error` caso a interface não tenha sido inicializada ou um runtime_error caso o canal não tenha sido encontrado.
- `atualizar`: joga `logic_error` caso a interface não tenha sido inicializada.

Agora é possível capturar essas exceções e trata-las. Pela especificação da interface com o usuário, deve-se apenas captura-las, apresenta-las para o usuário e terminar o programa.

5 Entrega

O projeto deverá ser entregue até dia **01/12** no Judge disponível em <http://judge.pcs.usp.br/pcs3111/ep/>

A entrega deve ser feita por cada membro da dupla, assim como nos demais EPs (ou seja, os dois devem submeter o mesmo exercício no Judge do EP). A entrega consiste em três submissões. Veja no Judge os detalhes da submissão.

Atenção

- Deve ser mantida a mesma dupla do EP2. 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 **24/11**.
- Os dois membros da dupla devem submeter o EP. Pode haver desconto na nota caso um dos alunos não entregue o EP.
- Deve-se submeter 3 vezes o EP, para cada entrega (problema) no Judge.
- Submeta arquivos ".zip" apenas. Não submeta arquivos ".rar" ou ".7z".

Cada parte deve ser entregue em um arquivo comprimido. Os fontes não devem ser colocados em pastas. Não submeta arquivos do Qt. O Judge fará uma verificação *básica* do software, verificando se é possível instanciar as classes. Não altere o nome dos arquivos entregues e 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).

O Judge fará uma verificação *básica* do software, verificando se é possível instanciar as classes definidas neste documento e os seus respectivos métodos especificados. Ou seja, essa correção não é a correção final.

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

6 Dicas

- Separe o `main` em vários métodos para reaproveitar código. Planeje isso!
- É muito trabalhoso testar o programa ao executar o `main` *completo*, já que é necessário informar vários dados. Para testar, crie um `main` mais simples, que cria os objetos do jeito que você quer testar. Só não se esqueça de entregar o `main` correto!
- Corrija todos os problemas identificados no EP2 antes de implementar o EP3.
- Implemente a solução aos poucos – não deixe para implementar tudo no final.
- Divida o trabalho entre a dupla. Por exemplo, um pode ser responsável pelas classes **Serie** e outro do **Eixo**; um pode fazer a leitura e o outro a escrita do arquivo.
- Teste em separado a leitura e a escrita de arquivo.
- Cuidado ao fazer o `delete` de séries carregadas. Certifique-se que o `delete` não é feito duas vezes!