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 1 – 2016

Resumo

O EP1 tem como objetivo exercitar os conceitos básicos de orientação a objetos ao implementar um *Visualizador Científico* simples. O sistema deverá atender a especificação definida neste documento.

1 Introdução

Deseja-se criar um *Visualizador Científico*, que permita apresentar de forma gráfica os dados obtidos por um microcontrolador, mais especificamente o disponível no Kit Freescale Freedom FDRM-KL25Z, usado na disciplina "Introdução à Engenharia Elétrica" (323100). O usuário poderá analisar os dados obtidos pelos sensores (do kit e externos) e também os dados de controle produzidos.

Esse projeto será desenvolvido incrementalmente e em dupla nos três Exercícios Programas de PCS 3111. Por simplicidade, a interface com usuário será feita incialmente em console. Para este primeiro EP o *Visualizador Científico* deve permitir gerar um gráfico simples, com duas séries.

A interface com o usuário e a interface com o microcontrolador já estão implementadas, devendo ser apenas usadas. Por enquanto está disponível uma implementação que funciona apenas para sistema operacional Windows. *Simule* funcionamento da interface serial (descrito na seção 5.3) caso se deseje desenvolver em outro sistema operacional.

A solução deve empregar adequadamente conceitos de orientação a objetos apresentados na disciplina: classe, atributo, método, encapsulamento, coesão e acoplamento.

2. Especificação

O Visualizador Científico deve possuir as seguintes funcionalidades:

- Deve ser possível desenhar um **Gráfico**, o qual possui dois **Eixos** e usa duas **Séries**. Como a **Tela** em que o **Gráfico** será desenhado possui um espaço limitado, será necessário mapear os valores das **Séries** para as coordenadas da **Tela**.
- O **Gráfico** possui dois **Eixos**: um das abscissas (horizontal) e um das ordenadas (vertical). Cada **Eixo** deve possuir um título (string), uma unidade (string), uma escala mínima (double), uma escala máxima (double), um número de divisões (int) e o valor do incremento da divisão (double). O número de divisões deve ser calculado a partir do número de pontos disponíveis pela **Tela** e o incremento da divisão deve ser calculado a partir do máximo, mínimo e do número de divisões.
- O **Gráfico** é responsável por desenhar **Séries**, considerando cada **Eixo**. A **Série** será obtida na placa (ou gerada no caso de simulações). **Series** representam um conjunto de valores sequenciais. Por exemplo, pode-se ter uma **Série** com os valores 1, 2, 3, 5, 7, 11 e 13. Uma **Série**

possui um nome (string, que não necessariamente é igual ao do **Eixo**), um tamanho (int) e uma sequência de valores (cada valor é um double).

- Deve ser possível adicionar um valor à **Série**, por enquanto até uma determinada quantidade. Depois dessa quantidade os valores adicionais devem ser desprezados. Deve ser possível obter um valor em uma posição da **Série**, saber o tamanho da **Série** (quantidade de elementos) e se ela está vazia. Deve ser possível calcular o máximo e mínimo da **Série**.
- O **Gráfico** deve ser desenhado na **Tela** (já implementada), a qual representa os elementos gráficos através de caracteres de forma a simplificar a interface.
- Será possível obter valores das **Séries** através de um microcontrolador. A interface com esse microcontrolador (**InterfaceSerial**) já está implementada.

Como exemplo, na Figura 1 é apresentado um Gráfico da Série Tempo (no eixo das abscissas) com a Série Espaço (no eixo das ordenadas). O título do eixo X é "Tempo" e a unidade é "s"; o do eixo y é "Espaco" e a unidade é "m". O Eixo das ordenadas tem 3 divisões, com um incremento de divisão 10 (ou seja, a cada 10 unidades é colocado um marcador). Com isso se tem marcadores em 10, 20, 30, além do marcador na origem (que no caso é no 0). Da mesma maneira, o eixo das abscissas tem 6 divisões, além da origem, com incremento 10. No gráfico são apresentados os pontos (0, 13), (10, 20), (20, 26) e (50, 40). Detalhes do mapeamento das **Series** e **Eixos** na **Tela** serão discutidos na Seção 4.

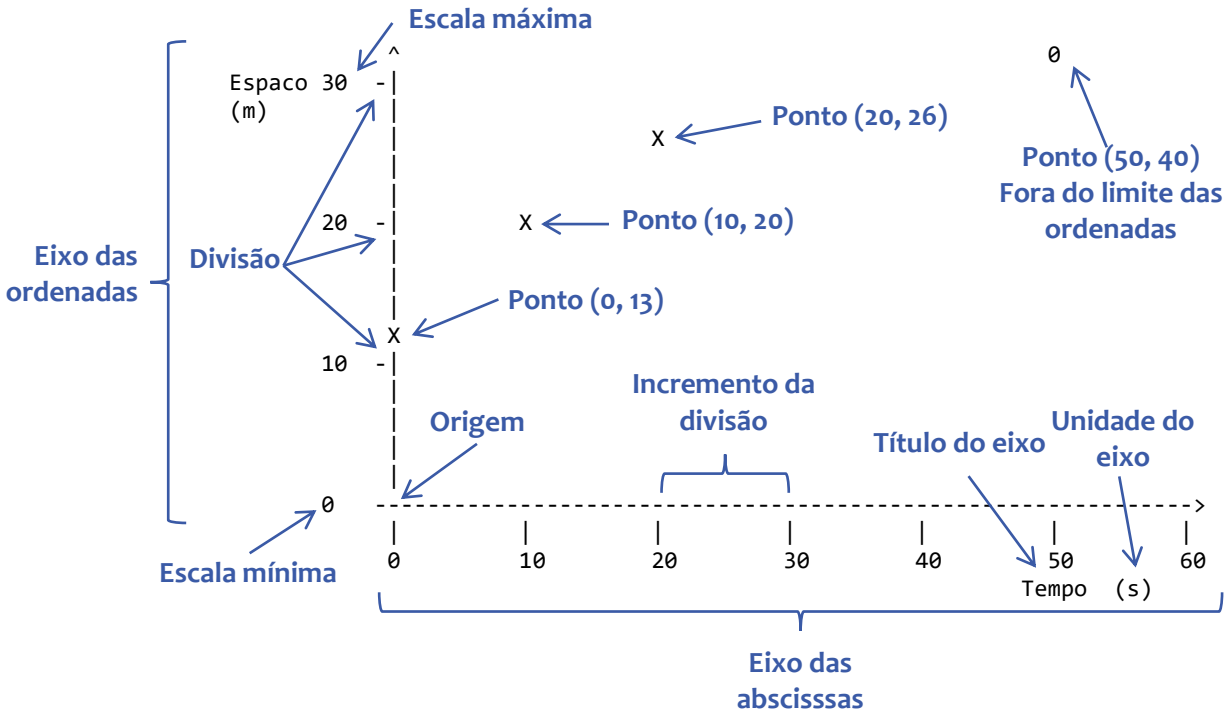


Figura 1: Conceitos exemplificados em um gráfico de Tempo X Espaço.

3 Projeto

Para implementar essas funcionalidades você deve criar um conjunto de classes em C++ para uma solução orientada a objetos. Cada classe deve ter um arquivo de definição (".h") e um arquivo de implementação (".cpp"). Siga a convenção de nomes de classes explicada em aula.

A seguir são explicitados os métodos públicos que as classes devem obrigatoriamente possuir, com a assinatura exatamente conforme definido. As classes não devem possuir outros membros (atributos ou métodos) públicos. Mas você pode (e precisará) adicionar outros membros (atributos e métodos)

privados conforme desejado. Para facilitar o trabalho, os arquivos ".h" dessas classes são entregues junto com enunciado. Altere-os conforme necessário. Todos os arquivos são comentados, descrevendo os métodos.

Para representar uma série nesse sistema você deve criar uma classe **Serie**. Essa classe deve possuir os seguintes métodos públicos e a seguinte constante:

```
#define NUMERO_MAXIMO_VALORES 10

void setNome(string nome);
string getNome();

/**
 * Informa se a serie esta vazia.
 */
bool estaVazia();

/**
 * Informa o numero de valores que a serie possui.
 * Por exemplo, a serie (1, 2, 3, 5) possui tamanho 4.
 */
int getTamanho();

/**
 * Obtem o valor que esta na posicao definida da serie. A contagem de
 * posicoes comeca em 0.
 *
 * Em caso de posicoes invalidas, retorne 0.
 *
 * Por exemplo, considere a serie (1, 2, 3, 5, 7, 11, 13). O metodo
 * getValor(0) deve retornar 1; getValor(6) deve retornar 13.
 */
double getValor(int posicao);

/**
 * Obtem o maior valor da serie.
 */
double getMaximo();

/**
 * Obtem o menor valor da serie.
 */
double getMinimo();

/**
 * Adiciona o valor a serie. Ignora o valor passado caso o
 * NUMERO_MAXIMO_VALORES tenha sido ultrapassado.
 *
 * Por exemplo, considere a serie com os valores (1, 2, 3, 5, 7, 11, 13).
 * Ao chamar adicionar(17), a serie ficara (1, 2, 3, 5, 7, 11, 13, 17).
 */
 void adicionar(double valor);
```

O método `adicionar` deve adicionar valores até o `NUMERO_MAXIMO_VALORES`, uma constante que deve ser definida na classe, como apresentado. Caso se tente adicionar mais valores, eles devem ser ignorados (ou seja, não deve ser feito nada – não deve ser emitida nenhuma mensagem de erro ou algo similar: ele só não deve ser adicionado). O método `getValor` deve retornar o valor na posição definida na série, considerando que a contagem começa em 0.

Para representar um eixo, crie uma classe **Eixo**. Essa classe deve ter os seguintes métodos públicos:

```
void setTitulo(string titulo);
void setUnidade(string unidade);
void setNumeroDeDivisoes(int numero);
void setEscalaMinima(double escalaMinima);
void setEscalaMaxima(double escalaMaxima);

string getTitulo();
string getUnidade();
int getNumeroDeDivisoes();

/**
 * Calcule o incremento da divisao a partir do maximo, minimo
 * e numero de divisoes.
 *
 * Deve-se assumir que os valores necessarios ja foram
 * definidos antes de chamar este metodo.
 */
double getIncrementoDaDivisao();
double getEscalaMinima();
double getEscalaMaxima();
```

Os valores do título, unidade, número de divisões, escala mínima e escala máxima devem ser informados ao eixo através dos métodos "set". O incremento da divisão deve ser calculado a partir da escala máxima, escala mínima e número de divisões. Para este EP, a escala máxima e mínima deverão ser informados pelo usuário; o número de divisões deve ser calculado com as informações da tela (veja os detalhes da **Tela** na seção 4).

Um gráfico é representado através da classe **Grafico**. Ela deve possuir os seguintes membros públicos:

```
void setSerieNasAbscissas(Serie* x);
void setSerieNasOrdenadas(Serie* y);

void setEixoDasAbscissas(string titulo, string unidade, double minimo, double maximo);
void setEixoDasOrdenadas(string titulo, string unidade, double minimo, double maximo);

/**
 * Desenha o grafico, colocando na tela os eixos e os pontos.
 */
void desenhar();

/**
 * Apaga a tela.
 */
void reset();

/**
 * Define a tela a ser usada
 */
void setTela(Tela* t);

Serie* getSerieNasAbscissas();
Serie* getSerieNasOrdenadas();
Eixo* getEixoDasAbscissas();
Eixo* getEixoDasOrdenadas();
```

Existem métodos para definir as séries (`setSerieNasAbscissas` e `setSerieNasOrdenadas`) e para criar os eixos (`setEixoDasAbscissas` e `setEixoDasOrdenadas`). Ou seja, as **Séries** devem ser criadas em um outro lugar e passadas para o **Gráfico**; os **Eixos** devem ser criados pelo **Gráfico**. Da mesma forma que com as séries, o método `setTela` deve receber uma **Tela** já criada. O método que plota o gráfico é o método

`desenhar`. O funcionamento desse método é discutido na Seção 4. O método `reset` deve apagar a tela, simplesmente chamando o método apropriado na **Tela**.

4 Interface com o usuário

Por simplicidade, a interface com o usuário será feita em console neste EP. Ela já está implementada na classe **Tela**, a qual não deve ser alterada.

A classe **Grafico**, a ser implementada, deverá chamar os métodos adequados da **Tela** para que ela desenhe o gráfico esperado. Para facilitar isso, o desenho dos eixos já está implementado[1]. Portanto, você deve apenas adicionar os pontos adequadamente na **Tela**, mapeando os pontos para posições da área disponível na **Tela**. Ao chamar o método `desenhar` da **Tela**, ela apresentará em console os eixos e os pontos adicionados. Para apagar o texto do console deve-se usar o método `apagar` da **Tela** (esse método usa detalhes específicos do sistema operacional).

A tela de desenho é dividida em três áreas, como mostra a Figura 2 (a qual tem um exemplo de um gráfico). A área em vermelho é a área dos eixos, na qual os eixos são desenhados. Essa área é de total responsabilidade da classe **Tela**; só é possível adicionar informações nessa área através dos métodos `setEixoDasAbscissas` e `setEixoDasOrdenadas`. As áreas em branco e em cinza são as que devem ser gerenciadas.

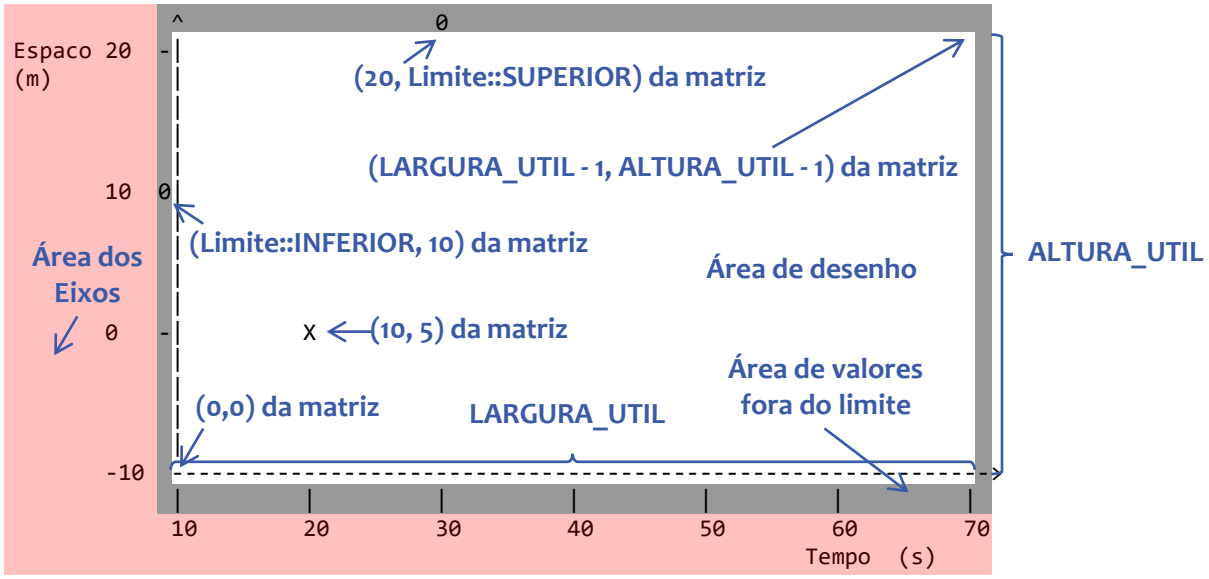


Figura 2: áreas da tela de desenho, de responsabilidade da classe Tela.

A área branca é a área útil para desenho. Ela funciona como uma matriz de tamanho `LARGURA_UTIL` (uma constante da **Tela** com valor 61) por `ALTURA_UTIL` (uma constante da **Tela** com valor 16). O elemento `(0, 0)` dessa matriz é no encontro dos eixos, enquanto que o elemento `(LARGURA_UTIL - 1, ALTURA_UTIL - 1)` é na posição superior direita. A **Tela** possui duas outras constantes: `DIVISAO_ABSCISSA` e `DIVISAO_ORDENADA`. Essas constantes informam a cada quantos elementos nos eixos há uma divisão. No caso, a `DIVISAO_ABSCISSA` tem valor 10, indicando que a cada 10 elementos há uma divisão; a `DIVISAO_ORDENADA` é 5, indicando que a cada 5 elementos há uma divisão. Você deve usar esses valores, juntamente com a `LARGURA_UTIL` e a `ALTURA_UTIL` para definir o número de divisões do **Eixo**.

[1] Essa solução pode ser vista como possuindo um problema de coesão, já que o mapeamento do **Eixo** na **Tela** deveria ser responsabilidade do **Grafico** (ou mesmo do **Eixo**). Isso sem falar do acoplamento gerado. Porém isso foi feito para simplificar a implementação.

Para escrever na área branca é necessário chamar na **Tela** o método `adicionarPontoEm (int x, int y)`, o qual recebe os valores x e y da matriz. Cabe à classe **Grafico** chamar esse método com os valores adequados, mapeando o ponto do gráfico ao elemento da matriz da Tela. Por exemplo, no gráfico representado na Figura 2, o ponto `(20, 0)` do gráfico deverá ficar no elemento `(10, 5)` da matriz. Como a matriz trabalha com posições inteiras, caso o mapeamento do gráfico para a área de desenho indique uma posição não inteira, *trunque* o valor ao chamar o método. No exemplo da Figura 2, tanto os pontos `(20, 0)` e `(20, 1)` devem ficar no elemento `(10, 5)` da matriz.

A área em cinza representa a área de valores fora do limite do eixo. Ou seja, caso se deseje adicionar um ponto que ficaria em uma posição fora da área disponível de desenho, é nessa área que será colocado um marcador. Pontos podem ficar fora do limite inferior ou superior da área de desenho. Por causa disso foram definidos os métodos `adicionarForaDoLimiteDasOrdenadasEm` (adiciona um ponto fora do limite do eixo das ordenadas, mas que possui um valor válido para o eixo das abscissas), `adicionarForaDoLimiteDasAbscissasEm` (adiciona um ponto fora do limite do eixo das abscissas, mas que possui um valor válido para o eixo das ordenadas) e `adicionarForaDoLimiteEm` (adiciona um ponto fora do limite das abscissas e das ordenadas). Para informar se o limite ultrapassado no eixo é o inferior ou superior, foi criado o *enumerador* **Limite**, que está definido em "Tela.h". Um *enumerador* é um tipo especial que permite enumerar seus valores, limitando seus valores a apenas os definidos. Dessa forma, o **Limite** permite apenas dois valores: `INFERIOR` e `SUPERIOR`. Para usá-lo é só incluir o "Tela.h" e fazer `Limite::INFERIOR` ou `Limite::SUPERIOR` para usar os valores válidos do limite. Por exemplo, no gráfico representado na Figura 2, caso se deseje desenhar um ponto na posição `(30, 30)` do gráfico, deve ser chamado o método `adicionarForaDoLimiteDasOrdenadasEm(20, Limite::SUPERIOR)`; para desenhar um ponto na posição `(0, 10)` do gráfico deve ser chamado o método `adicionarForaDoLimiteDasAbscissasEm(10, Limite::INFERIOR)` e para adicionar o ponto `(80, -20)` do gráfico deve ser chamado o método `adicionarForaDoLimiteEm(Limite::SUPERIOR, Limite::INFERIOR)`.

4.1 Função main

Assim como nas aulas, coloque o main em um arquivo em separado. O main deve pedir para o usuário as informações do gráfico e então desenhá-lo. Considere que as duas **Series** são obtidas pela placa; peça para o usuário para cada série o nome dela, o canal a ser usado (vide a seção 5.2 sobre a interface com a placa) e para o respectivo eixo o nome, o título e a unidade. Um exemplo da interface é apresentado a seguir:

```
Serie no eixo x: AceleracaoX
Qual o canal a ser usado: ACCX
Titulo do eixo x: Acel.X
Unidade do eixo x: m/s2
Escala minima do eixo x: -10
Escala maxima do eixo x: 10


Serie no eixo y: AceleracaoY
Qual o canal a ser usado: ACCY
Titulo do eixo y: Acel.Y
Unidade do eixo y: m/s2
Escala minima do eixo y: -5
Escala maxima do eixo y: 10
```

Para desenhar o gráfico, a classe IHC (já implementada e entregue) apresenta dois métodos auxiliares. O método `escFoiPressionado` verifica se entre a última chamada desse método e a chamada atual, a tecla ESC foi pressionada. Isso pode ser usado para cancelar o desenho do gráfico. O método `sleep` faz com que o programa espere por 500ms para continuar. Como o microcontrolador envia mensagens a cada 500ms, a ideia é que o programa pegue uma mensagem do microcontrolador, processe-a e então espere até que a próxima mensagem esteja disponível. Por enquanto ambos os métodos são específicos para Windows.

A ideia é que o main tenha uma estrutura similar a:

```
InterfaceSerial *is = new InterfaceSerial();
is->inicializar (COMM);

Grafico* g = new Grafico();
g->setTela (new Tela() );

...

IHC* ihc = new IHC();
while (!ihc->escFoiPressionado()) {
  is->atualizar();
  g->desenhar();
  ihc->sleep();
}
```

5 Interface com o microcontrolador

O Gráfico permitirá desenhar valores obtidos de um microcontrolador, o disponível no Kit Freescale Freedom FDRM-KL25Z. Os dados serão obtidos usando a porta serial, conectada por um cabo USB. Para isso é necessário um programa na placa e uma classe no projeto que fará a interface com a placa.

5.1 Programa na placa

Os dados devem ser gerados por um programa na placa que deve enviar uma mensagem com os títulos e, a cada intervalo de tempo, mensagens com os dados. Essas mensagens devem ser na forma de strings. A string tem um cabeçalho ('T', para título, ou 'A', para dados analógicos) e usa vírgula (",") como separador dos dados. O formato de uma mensagem de título é o seguinte:

```
\r\nT: <NOME1>, <NOME2>, <NOME3>,...\r\n
```

Onde <NOME1>, <NOME2>, <NOME3> são nomes dos sensores ou dos dados de controle. Os caracteres "\r\n" marcam o início e o final da mensagem. De forma similar, a mensagem com os dados deve seguir o formato:

```
A: <DADO1>, <DADO2>, <DADO3>,...\r\n
```

Onde <DADO1>, <DADO2>, <DADO3> são valores ponto flutuante (usando "." para separar a parte inteira da decimal). Nessa mensagem não é necessário usar "\r\n" como marcador de início.

Quando o programa na placa começar a ser executado, ele deve enviar uma mensagem de cabeçalho. Depois disso, a cada intervalo de tempo (sugerido 500ms) o programa na placa deve enviar uma mensagem com os dados, que podem representar dados obtidos por sensores ou dados de controle. A seguir é apresentado um exemplo de uma sequência de mensagens enviadas pela placa:

```
T: ACCX, ACCY, ACCZ, MAGX, MAGY, MAGZ
A: -0.0383, 0.0249, 1.0483, 25.3, -3071.9, -2655.7
A: -0.0164, -0.0073, 0.9922, 25.4, -384.1, -3168.1
A: -0.0125, -0.0190, 1.0107, 25.5, 1254.2, -1529.6
A: -0.2087, -0.3584, 0.3318, 24.2, -2434.3, 309.3
```

Esse programa foi um exercício sugerido pela disciplina 0323100, mas não será exigido em PCS3111 (sem ele você *somente* não conseguirá ver o gráfico sendo desenhado com dados obtidos pela placa).

5.2 Interface com a placa

A interface com a placa foi implementada na classe **InterfaceSerial**. Não será necessário conhecer os detalhes de implementação – é necessário apenas usá-la. A definição dessa classe é apresentada a seguir:

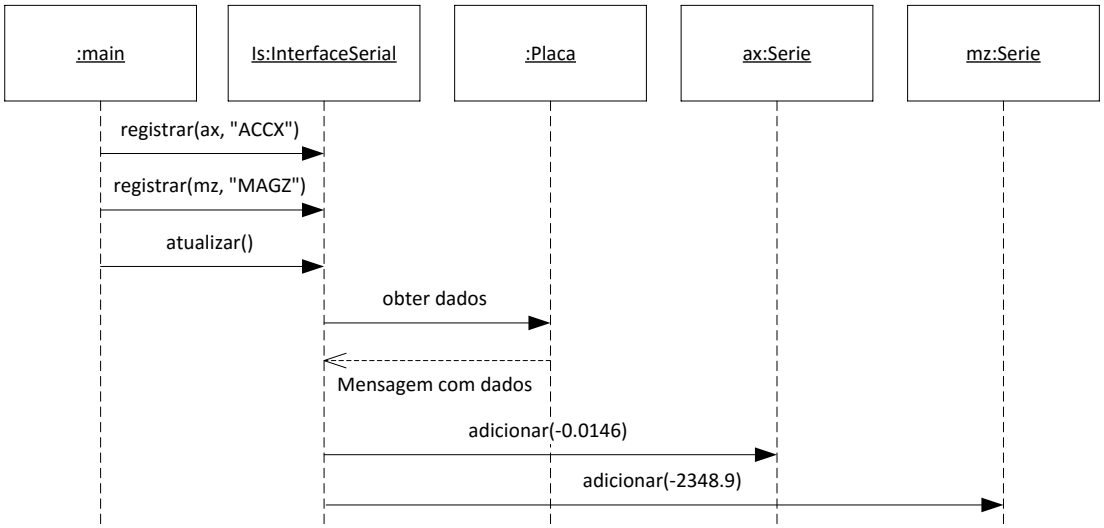
```
/**
 * Inicializa a comunicacao usando uma porta.
 */
void inicializar(string porta);

/**
 * Registra a serie especifica a um canal.
 */
void registrar(string canal, Serie* serie);

/**
 * Atualiza as series registradas com o canal.
 *
 * Em caso de problema de leitura, retorna false. Caso
 * contrario, retorna true.
 */
bool atualizar();
```

O método `inicializar` recebe uma porta de comunicação. No Windows, a porta serial é uma das portas COM, devendo ser passada uma string como, por exemplo, "`\\\\.\\COM4`" (para a porta 4). Siga as instruções da disciplina 323100 para encontrar a porta adequada. Ao chamar esse método, ele verificará se existe uma placa conectada na porta serial informada e pedirá para que o botão reset da placa seja apertado, para que se inicie a execução do programa e a mensagem de título seja passada.

O funcionamento dos dois outros métodos é o seguinte: quando se pede para atualizar os dados (método `atualizar`), a **InterfaceSerial** lê os dados da placa e avisa cada **Serie** registrada os valores lidos, chamando o método `adicionar` da Serie.[2] Portanto, quando for criada uma **Serie** que recebe dados da placa, ela deve ser registrada na **InterfaceSerial**. Um diagrama representando esse funcionamento é apresentado a seguir:



As caixas representam objetos, cujos nomes e tipos, respectivamente, são separados por "`:`". As chamadas de métodos são representadas como setas. No exemplo, o `main` registra a **Serie** `ax`, previamente criada, na **InterfaceSerial** para receber os dados do canal "ACCX"; e a **Serie** `mz`, previamente criada, para receber os dados do canal "MAGZ". Quando for chamado o método `atualizar` da **InterfaceSerial**, ele chama o método `adicionar` de cada **Serie** registrada para cada canal que teve dado

[2] Isso é um padrão de projeto chamado de *Publisher-Subscriber*. Uma visão bem geral desse padrão é apresentada em https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern.

recebido (ou seja, se não foi recebido nenhum dado, não é chamado nenhum `adicionar`). No caso, ele avisa a **Serie** `ax` o valor -0.0146 e para a **Serie** `mz` o valor -2348.9.

Cuidado: só deve haver uma instância do objeto **InterfaceSerial**!

5.3 Teste da interface com a placa

Quando se desenvolve software que se comunica com hardware, nem sempre se quer testar o software usando o hardware *real*, seja porque não se tem o hardware disponível, quer se evitar o desgaste do hardware, a comunicação com o hardware é demorada ou mesmo porque se quer testar condições específicas que são difíceis de acontecer naturalmente no hardware. Nesses casos pode-se criar uma classe que *simula* a interface com o hardware.

No nosso caso, uma forma de fazer isso é alterar a classe **InterfaceSerial** ao invés de usar a classe entregue, gerando assim uma classe de teste[3]. Contanto que os métodos públicos se mantenham os mesmos, as demais classes que dependem da **InterfaceSerial** não conseguirão ver a diferença entre a classe real e a de teste. Portanto, você pode alterar tanto a definição (por exemplo, colocando alguns atributos de apoio) quanto a implementação. O cuidado é não entregar essa classe como se fosse a classe real!

Cabe ao testador decidir como os métodos funcionarão e como eles devem responder. Para isso ele precisa pensar no que ele quer testar e implementar a classe de forma que ela faça o teste adequado. Por exemplo, segue uma implementação de uma **InterfaceSerial** que registra apenas `2` **Series** e retorna sempre os mesmos valores. No ".h" foram tirados todos os métodos e atributos privados e colocados os seguintes atributos:

```
...
private:
  Serie* s1 = NULL;
  Serie* s2 = NULL;
  int numeroDeChamadas = 0;
```

No ".cpp" foi usada a seguinte implementação:

```
#include "InterfaceSerial.h"

void InterfaceSerial::inicializar (string porta) {
}

void InterfaceSerial::registrar (string canal, Serie* serie) {
  if (s1 == NULL) s1 = serie;
  else if (s2 == NULL) s2 = serie;
}

bool InterfaceSerial::atualizar() {
  int valoresS1[] = {10, 20, 30, 40};
  int valoresS2[] = {10, 15, 20, 25};

  s1->adicionar(valoresS1[numeroDeChamadas]);
  s2->adicionar(valoresS2[numeroDeChamadas]);
  numeroDeChamadas++;
  return true;
}
```

Note que a implementação faz *diversas* simplificações. O `inicializar`, por exemplo, não faz nada, já que isso não importa no teste; o método `registrar` só registra `2` séries, `s1` e `s2`, desprezando o nome do canal;

[3] Existem soluções mais elegantes.

o `atualizar` envia somente 4 possíveis valores, dependendo do `numeroDeChamadas` feito. Cabe ao testador criar a classe que teste o que ele quer da forma mais simples possível.

6 Entrega

O projeto deverá ser entregue até dia **12/09** em um Judge específico, disponível em <https://pcs.usp.br/pcs3111/ep/> (nos próximos dias vocês receberão um login e uma senha). As duplas devem ser formadas por alunos da mesma turma e elas devem ser informadas no Moodle do Stoa até a data de entrega do EP.

> **Atenção**: não será possível alterar as duplas para os próximos EPs. Apenas será possível *desfazer* a dupla (e cada aluno fará uma entrega em separado).

A entrega deve ser feita por cada membro da dupla (ou seja, os dois devem submeter o mesmo exercício no Judge do EP). A entrega consiste em duas partes:

- **Parte 1**: todos os arquivos fonte do projeto (apenas ".cpp" e ".h")
- **Parte 2**: entregue todos os arquivos com exceção aos relativos ao **Eixo** e a **Serie** (não entregue nem o ".h" nem o ".cpp" relativos a essas classes).

Cada parte deve ser entregue em um arquivo comprimido que deve ter como nome o número USP do aluno (por exemplo, para o número USP 123456 o arquivo deve ser "123456.zip"). Os fontes não devem ser colocados em pastas. O Judge fará uma verificação *básica* do software, verificando se é possível instanciar as classes. Não altere o nome dos arquivos entregues: `Tela.h`, `Tela.cpp`, `IHC.h`, `IHC.cpp`, `InterfaceSerial.h` e `InterfaceSerial.cpp`.

7 Dicas

- Não adicione, de forma nenhuma e por motivo nenhum, métodos ou atributos públicos às classes. Você pode (e deve) adicionar atributos e métodos privados.
- As classes **Tela**, **IHC** e **InterfaceSerial** devem ser entregues da mesma forma que vocês a receberam. Isso não impede que você faça alterações para testes – é só não entregar com essas alterações.
- A correção do gráfico verificará se os métodos da **Tela** foram chamados na quantidade certa e com os valores corretos. Portanto, não faça chamadas desnecessárias aos métodos da **Tela**.
- Teste com diversos valores diferentes. Considere também valores que podem causar erros.
- Implemente a solução aos poucos – não deixe para implementar tudo no final.